Sicherheit und Datenschutz in nicht-interaktiven Crowdsourcing Szenarien

Vom Promotionsausschuss der
Technischen Universität Hamburg-Harburg
zur Erlangung des akademischen Grades
Doktor-Ingenieur (Dr.-Ing.)
genehmigte Dissertation

von
Tobias Jeske

aus
Lüneburg

2015

TUHH
Technische Universität Hamburg-Harburg

1. Gutachter:
Prof. Dr. Dieter Gollmann

2. Gutachter:
Prof. Dr.-Ing. Hannes Federrath

Vorsitzende des Promotionsverfahrens:
Prof. Dr. Sibylle Schupp

Tag der mündlichen Prüfung:
03.06.2015

Danksagung

Die vorliegende Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Institut für Sicherheit in verteilten Anwendungen der Technischen Universität Harburg.

Mein besonderer Dank gilt Herrn Prof. Dr. Dieter Gollmann für die Anregung zu dieser Arbeit sowie für seine Unterstützung und Förderung bei der Ausführung. Ebenso möchte ich mich bei Herrn Prof. Dr.-Ing. Hannes Federrath bedanken für die Übernahme des Mitberichts und für die fachliche Unterstützung im Oberseminar.

Gern erinnere ich mich auch an die Zusammenarbeit mit den ehemaligen Studenten Alexander Klein, Ashar Javed, Bradley J. Manning, Christian Feist, Christian Reimann, Denis Graf, Eugen Axt, Felix Kurth, Feng Xu, Florian Sieck, Giuseppe Barbieri, Ivonne Andrea Mantilla-Gonzalez, Jan Habermann, Jörn Heissler, Maren Wendler, Muhammad Yousuf Bin Azhar, Peter Weder, Sahil Sharma, Simon Stoye und Tatiana Uspenskaya. Durch ihre Mitarbeit konnte ich wertvolle Erfahrungen sammeln, die zur Entstehung dieser Arbeit beigetragen haben. Darüber hinaus möchte ich allen Personen meinen Dank aussprechen, die mich bei der Anfertigung dieser Arbeit unterstützt haben.

Diese Arbeit ist Meike Greve sowie meinen Eltern Regina und Michael Jeske gewidmet, die mir während dieser Zeit aufmunternd zur Seite standen.

Inhaltsverzeichnis

1 Motivation

Die Bedeutung von Crowdsourcing-Szenarien nehmen immer weiter zu. In diesem Kapitel werden neben der Einführung in das Thema Crowdsourcing die sich daraus ergebenen Sicherheits- und Datenschutzanforderungen erläutert und anhand von zwei Szenarien veranschaulicht.

1.1 Crowdsourcing

Der Begriff *Crowdsourcing* wurde erstmals 2006 von Jeff Howe in einem Artikel des Wired Magazine verwendet [84]. Er bezeichnet mit dem Begriff die Auslagerung von Tätigkeiten auf eine große Gruppe von Menschen, die Teilaufgaben eines größeren Problems (freiwillig) lösen. Howe verwendet zwei Definitionen für Crowdsourcing [83]:

1. „The Soundbyte Version: The application of Open Source principles to fields outside of software."
2. „The Whitepaper Version: Crowdsourcing is the act of taking a job traditionally performed by a designated agent and outsourcing it to an undefined, generally large group of people in the form of an open call."

Klassische Beispiele für Crowdsourcing sind z. B. die Wikipedia [175] oder das OpenStreetMap Projekt [131]. Viele freiwillige Helfer erstellen dabei gemeinsam eine freie Enzyklopädie bzw. kartografieren die Welt.

Beim *Mobile Crowdsourcing* übermitteln die Benutzer ihre Daten mobil [177]. Viele Japaner sendeten z. B. nach der Reaktorkatastrophe 2011 in Fukushima die aktuellen Strahlungswerte ihrer Geigerzähler einschließlich GPS-Position

an die Website RDTN.org [62]. Die Seite wurde zu einer wichtigen Quelle für Regierungsentscheidungen.
Insbesondere durch eine zunehmende Anzahl verschiedener Sensoren (z. B. GPS, Kompass, Kamera) nimmt die Bedeutung von Smartphones für das Mobile Crowdsourcing zu. Mobile Crowdsourcing ermöglicht z. B. in Katastrophengebieten die Koordinierung von Hilfskräften. Nach dem Erdbeben in Haiti übermittelten Einwohner über E-Mail, SMS und Telefon Informationen über den aktuellen Zustand ihres unmittelbaren Umfelds an die Ushahidi-Plattform. Das erlaubte Hilfsorganisationen eine schnellere Einschätzung der Lage und eine gezieltere Verteilung von Hilfsgütern [121].
Bei anderen Projekten werden Lautstärke und Positionsangaben, die durch Mikrophon und GPS-Sensor des Smartphones ermittelt werden, an einen zentralen Server gesendet. Daraus lässt sich die örtliche innerstädtische Lärmbelastung präziser und schneller dokumentieren [173].

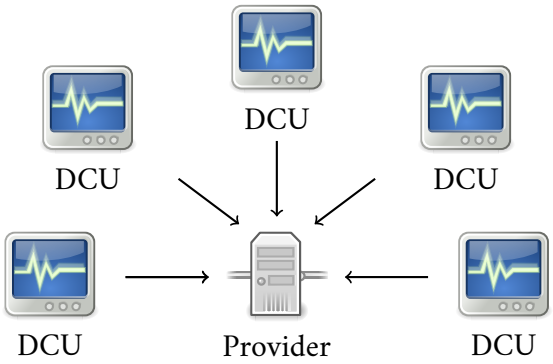


Abbildung 1.1: Nicht-interaktives Crowdsourcing.

Bei allen Projekten dieser Art ist immer eine Benutzerinteraktion zur Datenübertragung notwendig. In dieser Arbeit wird der Begriff *nicht-interaktives Crowdsourcing* erstmals verwendet (siehe Abbildung 1.1). Damit sind Crowdsourcing-Szenarien gemeint, bei denen Daten automatisch, d. h. ohne Einwirken des Benutzers erhoben und übertragen werden. Nicht-interaktives Crowdsourcing ist insbesondere dann wichtig, wenn Daten kontinuierlich und in Echtzeit an eine Zentrale übertragen werden. Eine Messeinrichtung (z. B. ein Smartphone) wird als *DCU (Data Collecting Unit)* und die Zentrale, an die Daten gesendet werden, als *Provider* bezeichnet. Provider bieten Dienste auf Basis der Daten an.
Aufgrund der immer stärkeren Vernetzung von Geräten, der geringen Kosten und der geringen Benutzerinteraktion nimmt in Zukunft das nicht-interaktive Crowdsourcing einen hohen Stellenwert ein. Diese Arbeit beschränkt sich ausschließlich auf das nicht-interaktive Crowdsourcing.

1.2 Sicherheits- und Datenschutzaspekte

Durch die regelmäßige, automatische Übertragung von Messdaten in Echtzeit an den Provider bieten sich Möglichkeiten der Manipulation. Zur Erhöhung der Zuverlässigkeit der Datenbasis muss der Einfluss eines Angreifers möglichst ausgeschlossen werden. Durch den Vergleich der empfangenen Daten mit denen anderer Benutzer lässt sich zwar ihre Plausibilität prüfen, die Maßnahme ist aber wirkungslos, wenn ein Angreifer eine beliebige Anzahl von DCUs simulieren kann. Die Anzahl der fehlerhaften Datenpakete kann die Zahl an korrekten Datenpaketen übertreffen.

Der Einflussbereich eines Angreifers lässt sich durch eine Authentifizierung der DCU eingrenzen. Wird davon ausgegangen, dass es für einen Angreifer schwierig ist, an mehr als eine beschränkte Anzahl von Authentifizierungskennungen zu gelangen, kann jedes Datenpaket einer DCU zugeordnet werden. Durch diese *Verknüpfbarkeit* kann ein bösartiges Verhalten eines Angreifers erkannt werden.

Aus Sicht des Providers ist eine Authentifizierung wünschenswert. Allerdings kollidiert diese Anforderung in manchen Szenarien mit den Datenschutzbedürfnissen der Benutzer. Durch die Verknüpfung der Datensätze sind z. B. Rückschlüsse auf Lebensgewohnheiten oder die Erstellung von Bewegungsprofilen möglich. Insbesondere in nicht-interaktiven Crowdsourcing-Szenarien, in denen Daten kontinuierlich und ohne Benutzerinteraktion übertragen werden, haben Benutzer ein Interesse anonym zu bleiben.

Die folgenden beiden Abschnitte beschreiben zwei unterschiedliche nicht-interaktive Crowdsourcing-Szenarien, bei denen sowohl die Sicherheits- als auch die Datenschutzanforderungen erfüllt sein müssen.

1.3 FCD-Daten von Smartphones

Heutige Navigationsgeräte empfangen Staumeldungen über den *TMC (Traffic Message Channel)* [168]. Quelle der TMC-Nachrichten sind die Polizei, festinstallierte Sensoren wie Verkehrskameras und Induktivschleifen und Meldungen von Freiwilligen. Die Verkehrsmeldungen werden im nicht-hörbaren Bereich des FM-Frequenzbandes durch Radiostationen übertragen. TMC ist weit verbreitet, hat allerdings zwei wesentliche Nachteile. Zum einen sind Verkehrsmeldungen häufig bereits veraltet, wenn sie das Navigationsgerät empfängt[1], zum anderen ist die Übertragungsrate mit 60 bit/s gering, so dass nur ca. 10 Meldungen pro Minute übertragbar sind.

[1] Die Meldungen sind zum Großteil redaktionell aufgearbeitet und werden nicht automatisch ermittelt und versendet.

Im Jahr 2007 wiesen Andrea Barisani und Daniele Bianco in einer Untersuchung nach, wie falsche TMC-Meldungen an Navigationsgeräte gesendet werden können [12]. Ein Angriff ist möglich, weil TMC-Daten unverschlüsselt übertragen werden. Die betroffenen Navigationsgeräte müssen allerdings in Reichweite des Senders sein.

Ebenfalls 2007 erweiterte Google den Kartendienst Google Maps um Google Live Traffic, die Visualisierung von Verkehrsinformationen in Echtzeit [50]. Google benutzt als Berechnungsgrundlage im Gegensatz zu TMC die Positionsdaten von Smartphones mit Android Betriebssystem [6]. Dadurch ergibt sich eine deutlich schnellere Abbildung des Verkehrsflusses. Die Daten werden als *FCD (Floating Car Data)* bezeichnet. Positionsdaten werden durch das Navigationssystem oder wie bei Google Live Traffic durch das Smartphone ermittelt und in der Regel über eine Mobilfunkanbindung an den Dienstanbieter übertragen. Daraus lassen sich im Gegensatz zu TMC Verkehrsinformationen in Echtzeit generieren.
Bei Google Live Traffic handelt es sich um ein nicht-interaktives Crowdsourcing-Szenario. Die DCUs sind die einzelnen Smartphones. Der Provider ist Google.

Der Verkehrsfluss wird nach Aktivierung durch den Benutzer in Google Maps durch die drei Farben rot, orange und grün dargestellt. Ist die Straße rot eingefärbt, wird auf einen Stau bzw. Stop-and-Go Verkehr hingewiesen, orange deutet auf stockenden Verkehr und grüne Straßen weisen auf keinerlei Behinderungen hin. Nachdem Google vorerst nur den Verkehrsfluss einiger Hauptverkehrsstraßen in vereinzelten Städten der USA anzeigen konnte, folgten später weitere Städte und mittlerweile stehen Verkehrsinformationsdaten in einer Vielzahl von Ländern und Städten zur Verfügung.
Die Verkehrsflussdaten werden nicht nur in Google Maps visualisiert, sondern seit 2011 auch zur Optimierung der Routenberechnung in Google Navigation verwendet [148]. Verkehrsstaus können dadurch in Echtzeit erkannt und umfahren werden.

Durch die hohen Wachstumsraten im Smartphone-Markt können Firmen wie Google ihre Systeme weiter ausbauen. Durch mehr Smartphones und damit mehr Sensoren im Feld erreichen sie eine höhere Genauigkeit. Es ist zu erwarten, dass in Zukunft Verkehrsdaten auf nahezu allen Straßen zur Verfügung stehen.
FCD beinhaltet nicht nur Positionsdaten. Ähnliche Konzepte wie Google Live Traffic werden in Fahrzeuge integriert und z. B. um Wetterinformationen (Regen, Temperatur, Nebel etc.) ergänzt [60].

1.3.1 „Google-Protokoll“

Das Protokoll zur Übertragung der FCD vom Smartphone zu Google ist ein auf dem *MASF (Mobile Application Sensing Framework)* basierendes Request/Respon-

se Protokoll [91]. Der komplette Protokollablauf wird über einen TLS-Tunnel abgewickelt, um Vertraulichkeit und Integrität der Daten zu gewährleisten.

In der Request-Nachricht sendet das Smartphone Zustandsinformationen der GPS-, WLAN- und Mobilfunk-Einheit an Google. Der Umfang der Daten hängt erstens von der Ausstattung des Smartphones und zweitens von der Anzahl der aktivierten Einheiten ab. Kann Google aus den Daten den ungefähren Standort berechnen, wird dieser in der Response-Nachricht zurück an das Smartphone gesendet. Eine explizite Positionsbestimmung durch den Benutzer wird beschleunigt, denn die Berechnung über GPS ist zwar genauer, kann aber ohne Kenntnis des ungefähren Standortes mehrere Minuten dauern. Gerade im innerstädtischen Bereich ist aufgrund der hohen Dichte an WLAN-Netzen und unter Berücksichtigung der Signalstärke eine Lokalisierung nur über WLAN (und ohne GPS) bis auf wenige Meter möglich.

1.3.2 Datenschutz

Zur Bewertung, inwieweit ein Tracking durch Google möglich ist, wird ein Rundkurs mit dem Auto abgefahren (siehe Abbildung 1.2). Die Fahrtrichtung ist im Uhrzeigersinn. Die Messungen erfolgen nachts bei geringem Verkehrsaufkommen, um die höchste erlaubte Geschwindigkeit fahren zu können.

In einem ersten Versuch ist WLAN aktiviert und das Smartphone über Bluetooth mit einem externen GPS-Empfänger verbunden, um ein stabiles GPS-Signal zu gewährleisten. Es sind keine Apps auf dem Smartphone aktiviert, die Standortdaten abfragen.
Die gestrichelte, schwarze Linie zeigt die zurückgelegte Strecke. Der Abstand zwischen zwei Strichen ist ein Maß für die Geschwindigkeit (geringere Abstände zeigen eine niedrigere Geschwindigkeit an). Die blaue und rote Kurve sind die Ergebnisse zweier Messfahrten. Sind Messpunkte miteinander verbunden, sind sie Teil einer Nachricht, die an Google übertragen wird.
Generell ist eine hohe Trackinggenauigkeit festzustellen. Größere Lücken zwischen den Messpunkten bei höheren Geschwindigkeiten könnten durch Kenntnis der Streckeneigenschaften und Zeitstempel der Messpunkte interpoliert werden. Bei sinkender Fahrgeschwindigkeit liegen die Messpunkte dichter beieinander. Damit steigt die Genauigkeit.

In einem weiteren Versuch ist der GPS-Empfang deaktiviert. Die aktuelle Position kann nur noch über WLAN und Funk ermittelt werden. Dieser Versuchsaufbau ist in der Regel realistischer. Meist ist der GPS-Empfang des Smartphones deaktiviert bzw. der GPS-Empfang durch Kleidung oder Taschen eingeschränkt.
Abbildung 1.3 zeigt die Visualisierung der Standortinformationen bei inaktivem GPS-Empfang. Die schwarze, gestrichelte Linie gibt die zurückgelegte Strecke

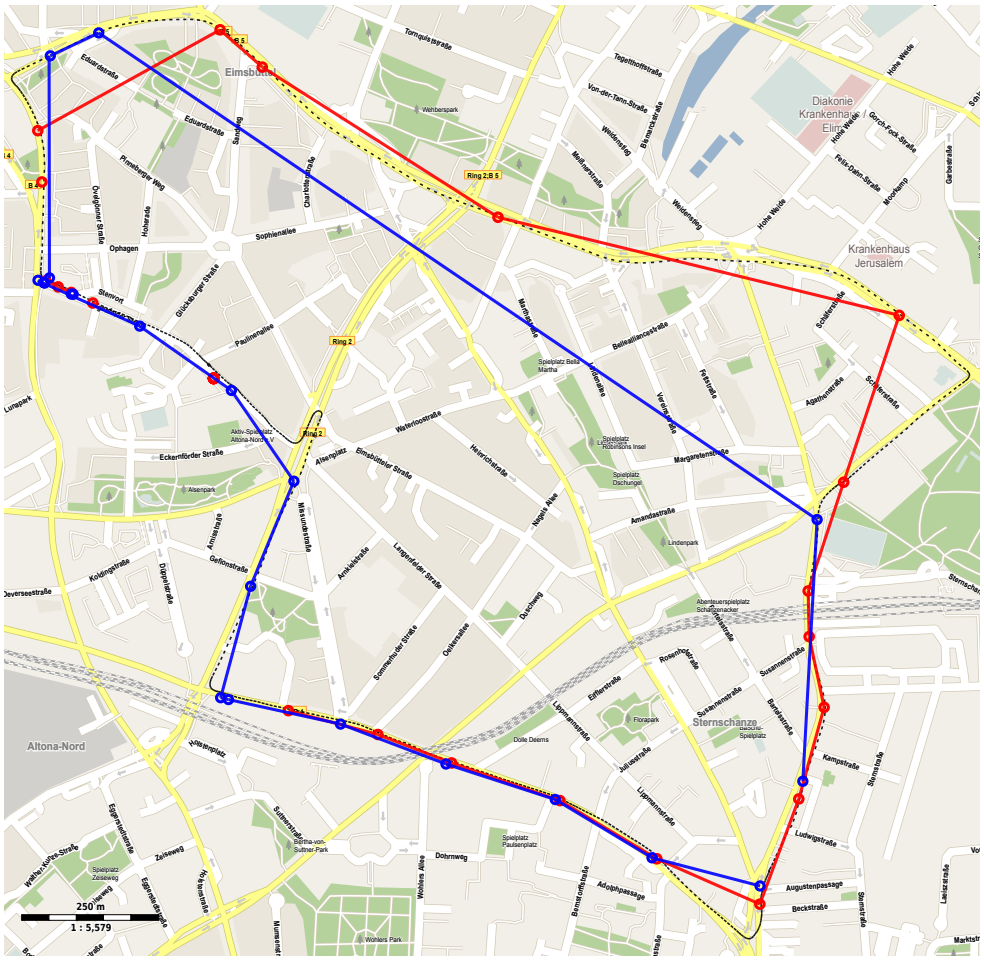


Abbildung 1.2: Evaluierung des „Google-Protokolls“ mit eingeschaltetem GPS, Kartenmaterial © OpenStreetMap und Mitwirkende, CC BY-SA.

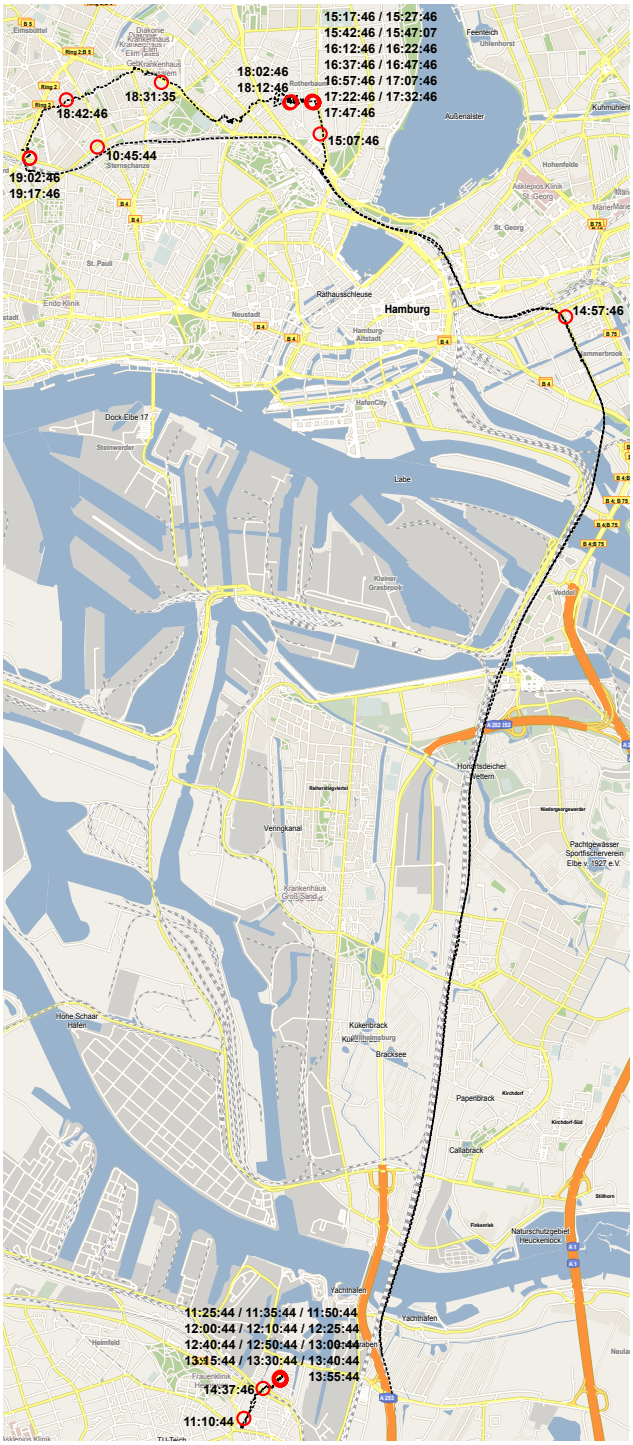


Abbildung 1.3: Evaluierung des „Google-Protokolls“, wenn nur WLAN zur Positionsbestimmung zur Verfügung steht, Kartenmaterial © OpenStreetMap und Mitwirkende, CC BY-SA.

wieder und ein roter Kreis mit Uhrzeit markiert einen an den Provider gesendeten Messpunkt. Zu erkennen ist, dass zwischen 10:44 und 11:07 Uhr sowie zwischen 14:43 und 15:03 Uhr jeweils eine lange Strecke in kurzer Zeit zurückgelegt wird (Fahrt mit der S-Bahn). Im Gegensatz dazu wird die Strecke zwischen ca. 18:15 und 19:00 Uhr sehr langsam zurückgelegt (zu Fuß). Aufgrund der niedrigen Geschwindigkeit ist ein genaueres Tracking möglich.

Aus der Abbildung lässt sich entnehmen, dass ein Android-Smartphone ohne GPS-Empfang in Abständen zwischen 10 und 30 Minuten Standortinformationen sendet. Ein genaues Tracking der zurückgelegten Strecke ist dadurch nicht möglich. Befindet sich der Anwender aber längere Zeit an einem Ort, kann das detektiert werden. Zwischen ca. 11:20 und 14:30 Uhr befand sich das Smartphone im Gebäude des Instituts für Sicherheit in verteilten Anwendungen der TU Hamburg-Harburg. Zwischen 15:10 und 17:50 Uhr kann der Anwender in einer Bibliothek der Uni Hamburg und zwischen 18:00 und 18:15 Uhr in der Uni Mensa lokalisiert werden. Aufgrund der Positionsbestimmung über WLAN und Funkmasten ist eine Lokalisierung des Smartphones auch innerhalb eines Gebäudes mit einer Genauigkeit von wenigen Metern möglich.

Jedes Smartphone verfügt über ein eigenes, individuelles Paar aus *Platform-Key* und *MASF-Cookie*, die in jeder Request-Nachricht des „Google-Protokolls“ enthalten sind [91]. Dadurch ist es möglich, aus einzelnen Nachrichten ein geschlossenes Bewegungsprofil zu erstellen. Selbst nach einem Neustart des Smartphones ändern sich Platform-Key und Cookie nicht.
Verwendet der Benutzer zusätzliche Dienste (z. B. Google Mail) auf seinem Smartphone, lässt sich das Bewegungsprofil über die IP-Adresse mit einer E-Mailadresse verknüpfen.

1.3.3 Korrektheit

Die Standortdaten werden über einen TLS-Tunnel an Google übertragen. Dadurch ist es einem Angreifer nicht möglich, ein fremdes Smartphone zu überwachen oder Daten unbemerkt zu verändern.
Allerdings ist TLS wirkungslos, wenn sich der Angreifer am Anfang des TLS-Tunnels befindet und das komplette Smartphone unter seiner Kontrolle hat. Google kann die Korrektheit der Positionsdaten nicht sicherstellen.
Nach einem Factory-Reset verfügt das Smartphone über keinen Plattform-Key. Ist kein Plattform-Key in der Request-Nachricht enthalten, generiert Google einen neuen Key und sendet ihn in der Response-Nachricht zurück an das Smartphone. Der Angreifer kann den Cookie zufällig wählen [91].
Damit wäre es einem Angreifer möglich, gefälschte Standortinformationen anonym an Google zu senden und die Verkehrsflussanalyse gezielt zu beeinflussen.

Verwenden Autofahrer Navigationssysteme, die auf Grundlage dieser Verkehrsflussanalyse navigieren, lassen sich Autofahrer gezielt in einen Stau lenken oder Straßen frei halten [91].

Fährt ein Angreifer z. B. eine Strecke mit dem Auto ab, und zeichnet die an Google gesendeten Datenpakete auf, kann er sie später mit geändertem Cookie, Platform-Key und Zeitstempeln erneut übertragen. Die Gewichtung der gefälschten Daten lässt sich erhöhen, indem er mehrere zeitversetzte Übertragungen mit unterschiedlichen Cookies und Platform-Keys vornimmt und damit mehrere Verkehrsteilnehmer simuliert. Wird modelliertes Rauschen auf die Messwerte (z. B. Empfangsstärken der WLAN Access Points, Positionsdaten) addiert, ist eine Unterscheidung zwischen realen und gefälschten Standortinformationen nicht mehr möglich. Auch lassen sich mit höherem Aufwand komplett künstliche Daten erzeugen, die für Google nicht von echten Daten zu unterscheiden sind.

Der hier vorgestellte Angriff hat gegenüber dem von Andrea Barisani und Daniele Bianco beschriebenen Angriff auf TMC [12] zwei Vorteile. Erstens lässt sich der beschriebene Angriff weltweit ausführen. Der Angreifer muss nicht vor Ort sein, während beim TMC-Angriff der Angreifer nur lokal Einfluss auf Navigationssysteme nehmen kann. Zweitens ist der beschriebene Angriff mit deutlich geringeren finanziellen Mitteln realisierbar. Der TMC-Angriff ist aufgrund des speziellen Equipments sehr kostenintensiv.

1.4 Smart Metering

Das *Smart Grid* integriert alle Akteure (Energieversorger, Netzbetreiber, Endverbraucher, ...) auf dem Strommarkt zu einem System mit dem Ziel, durch Kommunikation der einzelnen Akteure die Energieversorgung zu optimieren und für eine gleichmäßige Auslastung zu sorgen [16].

Smart Meter erlauben, den aktuellen Stromverbrauch zum VNB (Verteilnetzbetreiber) oder EVU (Energieversorgungsunternehmen) zu senden [122]. Berg Insight schätzt, dass im Jahr 2015 weltweit etwa 302,5 Millionen solcher intelligenten Zähler installiert sein werden [15]. In Deutschland sind z. B. in neuen und renovierten Privathäusern Smart Meter durch das Energiewirtschaftsgesetz vorgeschrieben [26].
Smart Meter können den Stromverbrauch für den Kunden visualisieren und ihn benachrichtigen, wenn etwa der Preis pro kWh günstig ist. Daraus ergeben sich Anreize für den Kunden, die Benutzung elektrischer Geräte wie z. B. Waschmaschine oder Klimaanlage auf Zeiten zu verlagern, in denen der Strompreis aufgrund eines größeren Angebots aus z. B. erneuerbaren Energien niedrig ist. Insbesondere die zunehmende Elektromobilität macht es erforderlich, das Laden der Akkus auf Zeiten mit Stromüberangebot zu verschieben [123].

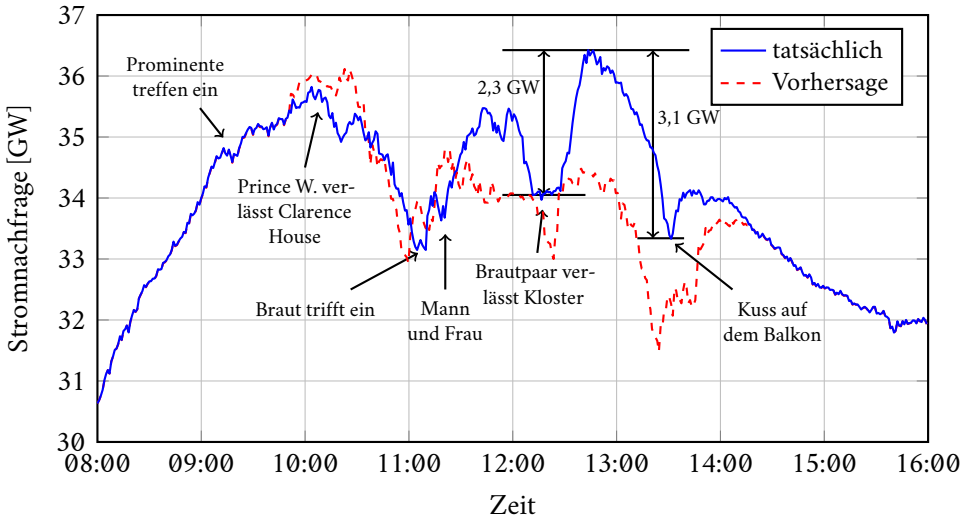


Abbildung 1.4: Stromverbrauch während der königlichen Hochzeit von Kate und William in Großbritannien am 29.04.2011.

Smart Meter ermöglichen auch die Steuerung der elektrischen Geräte beim Endkunden. So kann z. B. das Laden des Akkus eines Elektrofahrzeuges nicht nur dann erfolgen, wenn der Preis pro kWh günstig ist. Angestrebt wird, dass der VNB z. B. den benötigten Energiebedarf vor dem Laden des Akkus mitgeteilt bekommt. Der VNB kann dann diese Daten zur Optimierung seiner Kraftwerksauslastung verwenden.

Abbildung 1.4 veranschaulicht beispielhaft, warum eine Ressourcenoptimierung notwendig ist. Das Diagramm zeigt die Stromnachfrage während der königlichen Hochzeit in Großbritannien am 27.04.2011 [59]. Die blaue Kurve stellt den tatsächlichen Stromverbrauch, die rote Kurve den geschätzten Stromverbrauch dar. VNBs wie das National Grid in Großbritannien schätzen den Stromverbrauch vorweg ab, um Kraftwerke effizienter steuern zu können. Insbesondere bei Großereignissen ist das wichtig, da sich die Stromnachfrage kurzfristig stark ändern kann. Die Abbildung zeigt z. B. zwischen 12 und 14 Uhr einen An- und Abstieg des Stromverbrauches um mehrere GW innerhalb weniger Minuten[2]. Kraftwerke können so kurzfristige Leistungsschwankungen nur schwer ausgleichen. VNBs schätzen deswegen den Stromverbrauch ab, um rechtzeitig auf die geänderte Nachfrage reagieren zu können. Die Abbildung zeigt, dass für diesen Tag die Abschätzung der Stromnachfrage nicht optimal war. Hätte das National Grid nicht Strom aus dem Ausland eingekauft, wäre es zu einem Blackout gekommen. Da diese Lösung in der Regel teuer ist, versucht der VNB das durch eine gezieltere

[2] Viele Engländer haben sich die Trauung vor dem Fernseher angeschaut. Der starke Anstieg um kurz nach 12 Uhr könnte z. B. dadurch begründet sein, dass mehr Menschen als vorhergesagt zu der Zeit gekocht haben. Ein Herd hat einen relativ hohen Stromverbrauch mit einer Leistung von mehreren kW.

Kraftwerkssteuerung zu verhindern.

Eine Möglichkeit zur Vorhersageverbesserung ist die Verwendung von Smart Metern. Smart Meter übertragen den aktuellen Stromverbrauch eines Haushaltes in Echtzeit an den VNB. Die aktuellen Verbrauchsdaten können mit Zusatzinformationen verknüpft werden, wie z. B. der Information, welche Geräte gerade aktiv sind oder Schätzungen über den zukünftigen Verbrauch. Wird z. B. die Batterie eines Elektroautos aufgeladen, berechnet die Ladestation aus der momentanen und maximalen Batteriekapazität und dem Ladestrom den zukünftigen Strombedarf und überträgt ihn über den Smart Meter an den VNB.
Die Kenntnis über den Verbrauch eines einzelnen Haushaltes ist dabei für den VNB nicht von Interesse, sondern nur die Verbräuche einer bestimmten Gruppe von Abnehmern. Die Verbrauchsdaten könnten z. B. regional oder nach Art des Verbrauchers (Privathaushalt, Industriebetrieb) akkumuliert werden.

Bei dem Smart Metering Szenario handelt es sich um ein nicht-interaktives Crowdsourcing-Szenario. Die Smart Meter sind die einzelnen DCUs. Der Provider ist der VNB bzw. das EVU.

1.4.1 Datenschutz

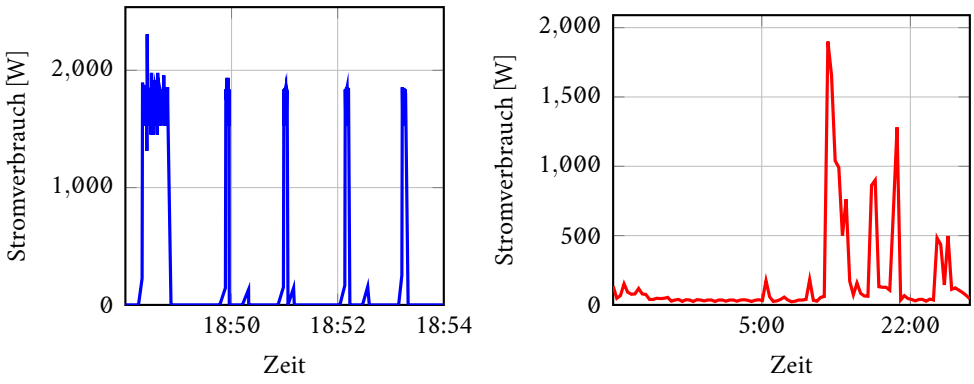


Abbildung 1.5: Stromverbrauch eines Backofens (links) und eines einzelnen Haushaltes über 24 Stunden (rechts).

Über ein Viertel der Befragten gaben bei einer Forsa-Umfrage zu Nachteilen des Smart Metering die Angst vor dem „gläsernen Kunden“ als Hauptnachteil an [66]. Wenn der VNB über die Zeit den exakten Stromverbrauch der Haushalte kennt, kann er Rückschlüsse auf die Lebensgewohnheiten der Kunden ziehen, denn elektrische Geräte lassen sich über ihren Stromverbrauch identifizieren [122].

Abbildung 1.5 (links) zeigt beispielhaft das Energieprofil eines Backofens. Zu Anfang wird der Ofen von Raumtemperatur auf Soll-Temperatur aufgeheizt. Danach werden die Heizelemente von Zeit zu Zeit eingeschaltet, um die Temperatur

zu halten. Dieses Energieprofil ist charakteristisch für die meisten elektrischen Backöfen.
Auch wenn in einem Haushalt mehrere Geräte zur selben Zeit aktiv sind, lassen sich durch Kenntnisse der charakteristischen Stromprofile einzelne Verbraucher identifizieren. Wird die Zeit, in der Geräte aktiv sind, mit einbezogen, ergeben sich weitere wichtige Informationen. Der Stromverbrauch von identifizierten Geräten lässt sich aus dem Gesamtverbrauch herausrechnen, um damit die Identifikation weiterer Geräte zu erleichtern. So haben z. B. Greveler et al. gezeigt, wie sich bei einer hohen Abtastrate von $0{,}5\ s^{-1}$ das aktuelle Fernsehprogramm durch die Helligkeitswechsel identifizieren lassen kann [76].

Abbildung 1.5 (rechts) zeigt den elektrischen Energieverbrauch eines Haushaltes über einen Tag. Trotz der relativ niedrigen Auflösung von nur drei Messpunkten pro Stunde lässt sich erkennen, wann die Kaffeemaschine (10 Uhr), der Backofen (15 Uhr) und die Geschirrspülmaschine (17 - 19:30 Uhr) eingeschaltet waren. Selbst wenn eine Identifikation individueller Geräte nicht möglich ist, kann aus längeren Verläufen entnommen werden, wann sich typischerweise Personen im Haushalt aufhalten. Diese Informationen können z. B. Einbrecher nutzen.

Damit der VNB die Lastprofile für die Auslastungsplanung verarbeiten kann, muss er sie speichern. Hacker oder internes Personal mit böswilligen Absichten schaffen es trotz Zugriffskontrollen, sich unberechtigten Zugang zu sensiblen Daten zu verschaffen. Die Fälle bei Google oder Sony zeigen das eindrucksvoll [179] [128]. Die Folgen sind ein Imageschaden und Vertrauensverlust bei den Kunden.

Zusammenfassend lässt sich sagen: Eine detaillierte Kenntnis über den aktuellen (und zukünftigen) Stromverbrauch ist für den VNB notwendig, um Lastprognosen zu erstellen. Die Verarbeitung und Speicherung einzelner Lastprofile ist aus datenschutzrechtlicher Sicht problematisch.

1.4.2 Korrektheit

Die Korrektheit der Verbrauchsdaten ist insbesondere zu Abrechnungszwecken notwendig. Dazu muss sich der Smart Meter beim EVU authentifizieren und identifizieren. Eine anonyme Übertragung der Daten darf nicht erfolgen.
Eine Übertragung der Verbrauchswerte in schneller zeitlicher Folge ist nicht erforderlich. Der Smart Meter kann in regelmäßigen Abständen eine authentifizierte Abrechnungstabelle erhalten, die den Preis pro KWh für bestimmte Zeitintervalle angibt[3]. Der Smart Meter akkumuliert die Verbrauchswerte und überträgt sie einmal im Monat an das EVU.

[3] Der Preis pro kWh ändert sich dynamisch und der aktuelle Tarif wird vom EVU vorgegeben [114].

Im Gegensatz zu den Abrechnungsdaten müssen für die Prognose der Stromnachfrage die Smart Meter in kürzeren Intervallen (z. B. alle 15 Minuten) Daten an das EVU übertragen. Die Korrektheit der Daten ist auch hier erforderlich. Gleichzeitig darf sich ein Smart Meter nicht identifizieren, um die Anonymität des Kunden zu gewährleisten.

1.5 Anforderungen

Beim Vergleich der beiden beschriebenen nicht-interaktiven Crowdsourcing-Szenarien lassen sich eine Reihe von Anforderungen für neue Protokolle spezifizieren.

1. *Authentifizierung*
   Nicht jede DCU kann Daten an den Provider senden, die der Provider akzeptiert. Eine DCU muss vorher bestimmte Kriterien erfüllen. Dazu kann z. B. im Smart Metering Szenario ein gültiger Vertrag mit dem EVU gehören. Eine DCU muss einen Nachweis besitzen, um sich gegenüber dem Provider als gültige DCU authentifizieren zu können.

2. *Widerrufbarkeit*
   Dem Provider muss die Möglichkeit gegeben werden, eine DCU von der weiteren Übertragung von Daten auszuschließen. Beim Smart Metering Szenario wäre das z. B. der Fall, wenn die Integrität des Smart Meters nicht mehr sichergestellt werden kann[4]. Alle Nachrichten der DCU werden vom Provider abgelehnt.

3. *Integrität*
   Die Manipulation der Messdaten, die von einer DCU zum Provider gesendet werden, kann vom Provider erkannt werden.

4. *Identifizierung*
   In manchen Szenario wie z. B. dem Smart Metering Szenario ist eine Identifizierung der DCU zu Abrechnungszwecken erforderlich.

5. *Anonymität und Nicht-Verknüpfbarkeit*
   Die Privatsphäre des Benutzers bzw. Kunden muss geschützt werden. Die Erstellung von Bewegungs- oder Verhaltensprofilen durch den Provider darf im FCD-Szenario nicht möglich sein. Im Smart Metering Szenario dürfen einzelne Haushaltsgeräte nicht durch das EVU bzw. den VNB identifiziert werden können. Dazu gehört auch die Erkenntnis, ob sich Personen

[4] Der Smart Meter wurde aufgebrochen und z. B. Schlüsselmaterial extrahiert.

im Haushalt aufhalten[5].
Die Verknüpfung mehrerer Nachrichten sollte bis zu einem bestimmten Grad nicht möglich sein, da ansonsten die Gefahr besteht, einzelne DCUs und damit Benutzer/Kunden zu identifizieren.

6. *Robustheit*
Der Provider fordert, dass Daten, die an ihn geschickt werden, korrekt sind. In der Praxis lässt sich diese Anforderung nicht immer durchsetzen. Im FCD-Szenario kann ein Benutzer die GPS-Daten manipulieren und sich (allerdings häufig unter dem Verlust der Gerätegarantie) Vollzugriff auf das Smartphone verschaffen [105]. Im Smart Metering Szenario wird zwar eine Manipulation der Verbrauchswerte erschwert, aber häufig ist eine Verplombung des Smart Meters der einzige Schutz gegen Manipulation [51]. Somit kann die Kompromittierung einer DCU nicht ausgeschlossen werden. Es wird allerdings gefordert, dass der Einfluss eines Angreifers auf die Datenbasis des Providers gering ist[6].

7. *Praktikabilität*
Ein Smart Meter oder Smartphone als DCU besitzt in der Regel nicht die Leistungsfähigkeit eines Standard-PCs. Aus diesem Grund muss der Rechenaufwand an das Protokoll gering sein. Eine DCU muss in der Lage sein, über das Protokoll Daten in einem Zeitintervall von maximal 10 Minuten an den Provider senden zu können.

Die Vertraulichkeit der Messdaten, die von einer DCU zum Provider gesendet werden, wird nicht explizit gefordert. Durch die Forderung nach Anonymität soll eine Verknüpfung zwischen DCU und Daten nicht möglich sein und die Möglichkeiten eines Angreifers, der die Kommunikation abhört, sind damit begrenzt. Soll trotzdem die Vertraulichkeit zwischen DCU und Provider sichergestellt werden, lässt sich dieses Ziel durch Protokolle wie z. B. TLS erreichen, die transparent in einer Netzwerkschicht unterhalb des Crowdsourcing-Protokolls ausgeführt werden [53].

1.6 Ziele der Arbeit

In dieser Arbeit sollen Lösungen auf Basis von Zero-Knowledge Proofs aufgezeigt werden, welche die Anforderungen aus Abschnitt 1.5 in nicht-interaktiven

[5] Steht ein Haushalt leer, ist der Stromverbrauch in der Regel konstant bzw. folgt einem festen Muster (Kühlschränke sorgen durch periodisches An- und Abschalten des Kompressors für einen nicht-konstanten Stromverbrauch).

[6] Unter der Voraussetzung, dass das Verhältnis aus kompromittierten zu nicht-kompromittierten DCUs gering ist.

Crowdsourcing-Szenarien erfüllen. Demonstriert wird die Praktikabilität der entwickelten Protokolle anhand des FCD- und Smart Metering Szenarios.

Aufgrund der Implementierungskomplexität von Protokollen mit Zero-Knowledge Proofs soll ein Compiler entwickelt werden, mit dem sich die Protokolle ohne Programmierkenntnisse auf nahezu jeder beliebigen (Hardware-)Plattform implementieren lassen können.
DCUs verfügen in der Regel über eine begrenzte Rechenkapazität, die beim Entwurf der Algorithmen berücksichtigt werden muss. In Crowdsourcing-Szenarien kommt der Provider aufgrund der zu verarbeitenden Datenmengen häufig an seine Kapazitätsgrenze. Die vorliegende Arbeit soll zeigen, wie sich durch die Entwicklung neuer Algorithmen typische Berechnungen in Zero-Knowledge Proofs beschleunigen lassen können und die Ausführung dieser Algorithmen auf Grafikkarten gegenüber einer CPU-Implementierung effizienter durchgeführt werden kann.

Teile dieser Arbeit wurden bereits auf Fachkonferenzen bzw. im Rahmen von Fachvorträgen veröffentlicht.

Veröffentlichungen:

- Tobias Jeske, „Floating Car Data from Smartphones: What Google And Waze Know About You and How Hackers Can Control Traffic“, Black Hat 2013, Amsterdam, Niederlande, 12. - 15. März 2013
- Tobias Jeske, Felix Kurth, „Big Number Modulo Exponentiations For Zero-Knowledge Protocols On GPUs“, GPU Technology Conference, San Jose, USA, 14. - 17. Mai 2012
- Tobias Jeske, „Privacy-Preserving Smart Metering Without a Trusted-Third-Party“, SECRYPT 2011 - Proceedings of the International Conference on Security and Cryptography, S. 114 - 123, Seville, Spanien, 18. - 21. Juli 2011
- Tobias Jeske, „Datenschutzfreundliches Smart Metering - Ein praktikables Lösungskonzept“, Datenschutz und Datensicherheit - DuD, Vol. 35, No. 8. (1. August 2011), S. 530 - 534

Vorträge:

- Tobias Jeske, „Compiler and Framework for Implementing Cryptographic Protocols“, Central European Conference on Cryptology 2013, Telč, Tschechien, 27.06.2013
- Tobias Jeske, „Sichere & Anonyme Kommunikation im Smart Grid“, Smart Metering & Smart Grid Security 2012, Potsdam, 23.03.2012
- Tobias Jeske, „Privacy-Preserving Smart Metering“, European Smart Grid Cyber Security and Privacy 2011, Amsterdam, Niederlande, 15.11.2011

- Tobias Jeske, „Datenschutzfreundliches Smart Metering“, Energie & Technik - Smart Home & Metering Summit, München, 26.10.2011
- Tobias Jeske, „Smart Meter - der Stromzähler als Spion? oder: Der anonymisierte Strom“, Faszination Strom - Tag der E-Technik 2011, Hamburg, 07.10.2011
- Tobias Jeske, „Datenschutz für Smart Metering“, Smart Grid Symposium, Ettlingen, 02.02.2011
- Tobias Jeske, „A Compiler for Optimised Zero-Knowledge Proofs of Knowledge“, Johannes Kepler Universität, Linz, Österreich, 23.11.2010

2 Zero-Knowledge Proofs

Dieses Kapitel gibt eine Einführung in Zero-Knowledge Proofs und weitere kryptographische Grundbausteine, die für das Verständnis der weiteren Kapitel notwendig sind.

In den folgenden Kapiteln bezeichnet $\{0,1\}^\ell$ für $\ell \in \mathbb{N}^+$ die Menge der ganzen Zahlen aus dem Intervall $[0, 2^\ell - 1]$. In der Menge $\pm\{0,1\}^\ell$ befinden sich alle ganzen Zahlen aus dem Intervall $[-2^\ell + 1, 2^\ell - 1]$.

2.1 Gruppentheorie

Ganze Zahlen werden in dieser Arbeit durch Elemente einer Gruppe repräsentiert [117].

Definition 2.1.1 (Gruppe) *Eine Gruppe $(G, \circ)$ sei eine Menge G mit einer binären Verknüpfung $\circ : G \times G \to G$, für die folgende Axiome erfüllt sind:*

- Abgeschlossenheit
  Es gilt: $\forall g_0, g_1 \in G : g_0 \circ g_1 \in G$
- Assoziativität
  Es gilt: $\forall g_0, g_1, g_2 \in G : (g_0 \circ g_1) \circ g_2 = g_0 \circ (g_1 \circ g_2) = g_0 \circ g_1 \circ g_2$
- Neutrales Element
  Es gibt ein neutrales Element $e \in G$, d. h.
  $\exists e \in G : \forall g \in G : g \circ e = e \circ g = g$

- Inverses Element
  Es gibt zu jedem $g \in G$ ein inverses Element $g^{-1} \in G$, d. h.
  $\forall g \in G : \exists g^{-1} \in G : g \circ g^{-1} = g^{-1} \circ g = e$

Ist zusätzlich folgendes Axiom erfüllt, handelt es sich um eine kommutative *oder* abelsche Gruppe*.*

- Kommutativität
  Es gilt: $\forall g_0, g_1 \in G : g_0 \circ g_1 = g_1 \circ g_0$

Das neutrale Element einer Gruppe und das jeweilige inverse Element eines Gruppenelementes sind eindeutig.
Die Anzahl der Elemente einer Gruppe $(G, \circ)$ ist die *Ordnung* $|G|$ der Gruppe G. Die Ordnung eines Gruppenelementes g ist die kleinste, positive ganze Zahl d für die $g^d = 1$ gilt. Falls eine solche Zahl d nicht existiert, ist die Ordnung von g unendlich.

Eine Gruppe ist endlich, wenn die Ordnung endlich ist, ansonsten ist es eine Gruppe mit *unendlicher Ordnung*.
Grundlage dieser Arbeit sind *additive Gruppen* $(G, +)$ mit neutralem Element 0 und *multiplikative Gruppen* $(G, \cdot)$ mit neutralem Element 1. Alle Gruppen seien abelsche Gruppen. G kennzeichnet eine Gruppe, bei der die Gruppenoperation nicht relevant ist.

Definition 2.1.2 (Untergruppe) *Eine Gruppe $(U, \circ)$ heißt* Untergruppe *der Gruppe $(G, \circ)$, falls $U \subseteq G$ gilt und die Gruppenoperation in beiden Gruppen U und G identisch ist. Zusätzlich erfüllt die Untergruppe alle Axiome aus Definition 2.1.1.*

In dieser Arbeit wird die Untergruppe der quadratischen Reste $\mathrm{QR}\{G\} = \{g^2 \mid g \in G\}$ verwendet, welche alle Elemente enthält, die sich als Quadrat eines Elementes der Gruppe G darstellen lassen können.

Definition 2.1.3 (Zyklische Gruppen) *Eine Gruppe $(G, \circ)$ heißt* zyklisch*, wenn es ein Element $g \in G$ gibt, das alle Elemente der Gruppe G erzeugen kann, d. h. es gilt: $\forall g' \in G : \exists a \in \mathbb{Z} : g^a = g'$. Das Element g ist der* Generator *von G und man schreibt $G = \langle g \rangle$.*

Definition 2.1.4 (Potenz) *Es sei g ein Gruppenelement der Gruppe G, e das neutrale Element der Gruppe und $a \in \mathbb{Z}$. Dann ist die a-te Potenz von g definiert als* $g^a = e \underbrace{\circ g \circ g \cdots \circ g}_{a\text{-mal}}$.

Definition 2.1.5 (Eulersche Phi-Funktion [117]) *Es sei $n \geq 1$, dann ist $\varphi(n)$ die Anzahl von ganzen Zahlen im Intervall $[1, n]$, die teilerfremd zu n sind. Die Funktion φ heißt* Eulersche Phi-Funktion.

Ist p eine Primzahl, dann gilt $\varphi(p) = p - 1$. Lässt sich eine ganze Zahl n zerlegen in $n = n_0 \cdot n_1$ mit $\text{ggT}(n_0, n_1) = 1$, dann gilt $\varphi(n_0 \cdot n_1) = \varphi(n_0) \cdot \varphi(n_1)$. Jede endliche, zyklische Gruppe G hat genau $\varphi(|G|)$ Generatoren [153].

In dieser Arbeit werden insbesondere folgende Gruppen verwendet:

- Die Gruppe $(\mathbb{Z}, +)$ bzw. $\mathbb{Z}$ ist eine zyklische Gruppe der ganzen Zahlen mit unendlicher Ordnung und der Addition als Gruppenoperation. Das inverse Element g^{-1} eines Gruppenelementes g ist $-g$. Die Generatoren der Gruppe sind 1 und -1. Die Potenzoperation in der Gruppe ist die Multiplikation.
- Die Gruppe $(\mathbb{Z}_n, +)$ bzw. $\mathbb{Z}_n$ ist eine zyklische Gruppe der ganzen Zahlen mit der Ordnung $n \in \mathbb{N}^+$ und der modularen Addition als Gruppenoperation. Das inverse Element g^{-1} eines Gruppenelementes g ist $g^{-1} = n - g$. Die Potenzoperation in der Gruppe ist die modulare Multiplikation. $\mathbb{Z}_n$ hat genau $\varphi(n)$ Generatoren, da die Gruppe zyklisch ist. Ist der Modulus prim, ist jedes Element bis auf die Null ein Generator der Gruppe.
- Die Gruppe $(\mathbb{Z}_n^*, \cdot)$ bzw. $\mathbb{Z}_n^*$ ist eine Gruppe der ganzen Zahlen mit dem Modulus $n \in \mathbb{N}^+$ und der modularen Multiplikation als Gruppenoperation. Alle Elemente der Gruppe sind teilerfremd zu n. $\mathbb{Z}_n^*$ ist dann (und nur dann) zyklisch, falls $n = 2, 4, p^a$ oder $n = 2p^a$ mit $a \in \mathbb{N}^+$, $a \geq 1$ und einer Primzahl $p \geq 3$ gilt [117]. Mithilfe des erweiterten euklidischen Algorithmus kann das inverse Element eines Gruppenelementes berechnet werden. Die Potenzoperation in der Gruppe entspricht der modularen Exponentiation.
  Ist n das Produkt zweier Primzahlen $n = p \cdot q$, dann bezeichnet $\mathbb{Z}_n^*$ eine RSA-Gruppe [146].

In der Praxis ist es häufig notwendig, Elemente mit großer Ordnung zu finden, damit bestimmte zahlentheoretische Probleme nicht praktisch lösbar sind (siehe Kapitel 2.2). Der folgende Satz erleichtert die Suche nach entsprechenden Generatorelementen in $\mathbb{Z}_n^*$ [117].

Satz 2.1.1 (Satz von Lagrange) *Es sei G eine endliche Gruppe und H eine Untergruppe von G. Die Ordnung eines Gruppenelementes $g \in G$ und $|H|$ teilen die Gruppenordnung $|G|$.*

Zwei Sonderfälle sind für die Gruppe $\mathbb{Z}_n^*$ von besonderem Interesse.

1. Der Modulus $n = p$ ist eine *sichere Primzahl*, d. h. es gilt $p = 2 \cdot p' + 1$ und p' ist ebenfalls eine Primzahl (Sophie-Germain-Primzahl). Die Gruppe ist zyklisch und es wird die Untergruppe $U = \mathrm{QR}_n \subseteq \mathbb{Z}_n^*$ der quadratischen Reste betrachtet. Die Ordnung von U ist $|\mathrm{QR}_n| = (n-1)/2 = p'$ und damit prim [117]. Jedes Element aus QR_n ist ein Generator der Untergruppe U.

2. Es sei $n = p \cdot q$ das Produkt zweier sicherer Primzahlen $p = 2p' + 1$ und $q = 2q' + 1$. Ist ein Element einer RSA-Gruppe mit großer Gruppenordnung gesucht, wird in der Praxis häufig ein Element aus der Untergruppe $U = \mathrm{QR}_n$ verwendet. Die Gruppe QR_n hat die Ordnung $|U| = \frac{\varphi(n)}{4} = \frac{(p-1)(q-1)}{4} = p' \cdot q'$ [117].
   Die Suche nach einem Generatorelement der Untergruppe U ist leicht durchführbar. Es wird ein Gruppenelement $h \in_R \mathbb{Z}_n^*$ gewählt. Das Element $g = h^2$ ist ein Generator der Gruppe U, falls $\mathrm{ggT}(g \pm 1, n) = 1$ gilt [118].

2.1.1 Gruppenhomomorphismen

Ein Gruppenhomomorphismus ist eine strukturerhaltende Abbildung und ist wie folgt definiert [64].

Definition 2.1.6 (Gruppenhomomorphismus) *Es seien $(G, \circ)$ und $(H, \diamond)$ zwei beliebige Gruppen. Die Abbildung $\phi : G \to H$ heißt Gruppenhomomorphismus, wenn gilt:*

$$\forall g_0, g_1 \in G : \phi(g_0 \circ g_1) = \phi(g_0) \diamond \phi(g_1)$$

Das neutrale Element des Urbildes e_G wird bei einem Gruppenhomomorphismus auf das neutrale Element des Bildes e_H abgebildet ($\forall g \in G$: $\phi(g) = \phi(g \circ e_G) = \phi(g) \diamond \phi(e_G) \Rightarrow \phi(e_G) = e_H$) und inverse Elemente im Urbild $g^{-1} \in G$ werden auf die entsprechenden inversen Elemente im Bild abgebildet ($\forall g \in G : \phi(g) \diamond \phi(g^{-1}) = \phi(g \circ g^{-1}) = \phi(e_G) = e_H = \phi(g) \diamond \phi(g)^{-1} \Rightarrow \phi(g^{-1}) = \phi(g)^{-1}$).

Ein Gruppenhomomorphismus ϕ lässt sich exakter als *Gruppenepimorphismus*, *Gruppenmonomorphismus* und *Gruppenisomorphismus* beschreiben.
Bei einem Gruppenepimorphismus ist die Abbildung ϕ surjektiv (auf jedes Element der Bildmenge wird mindestens einmal abgebildet), bei einem Gruppenmonomorphismus injektiv (auf jedes Element der Bildmenge wird höchstens einmal abgebildet) und bei einem Gruppenisomorphismus bijektiv (auf jedes Element der Bildmenge wird genau einmal abgebildet). Sind die Gruppen G und H identisch, liegt ein *Endomorphismus* vor. Ist zudem die Abbildung ϕ bijektiv, wird von einem *Automorphismus* gesprochen.

Die Berechnung des Bildes ist bei den meisten Homomorphismen einfach, während die Berechnung des Urbilds aus einem Bild schwierig, d. h. nur in nicht polynomialer Zeit möglich ist.

2.2 Schwierige zahlentheoretische Probleme

In dieser Arbeit basiert die Sicherheit der Protokolle auf schwierigen zahlentheoretischen Problemen.

2.2.1 Diskreter-Logarithmus-Problem

Es sei g der Generator einer endlichen zyklischen Gruppe G und h ein weiteres Gruppenelement der Gruppe G. Der *diskrete Logarithmus* ist die kleinste ganze Zahl a, so dass $g^a = h \Leftrightarrow a = \log_g(h)$ gilt [9].

Definition 2.2.1 (Diskreter-Logarithmus-Problem) *Das DLP (Diskreter-Logarithmus-Problem) beschreibt die Aufgabe für ein Element $h \in G$, in einer zyklischen Gruppe G mit Generator $g \in G$ den diskreten Logarithmus $a = \log_g(h)$ zu finden.*

Die Schwierigkeit, das DLP zu lösen, hängt von der Gruppe ab. Bei einer additiven Gruppe $\mathbb{Z}_n$, $n \in \mathbb{Z}$ ist das DLP einfach, d. h. in polynomialer Zeit lösbar. Für eine multiplikative Gruppe $\mathbb{Z}_p^*$ mit primem Modulus p ist dagegen zurzeit kein effizientes Verfahren bekannt, um diskrete Logarithmen zu berechnen [117].

Der diskrete Logarithmus lässt sich mit dem *Pohlig-Hellman-Algorithmus* berechnen. Der Rechenaufwand hängt vom größten Faktor der Gruppenordnung ab [138]. Ist die Gruppenordnung einem Angreifer bekannt, sollte aus diesem Grund die Gruppenordnung einen möglichst großen Primfaktor aufweisen. Eine Möglichkeit ist, den Modulus als sichere Primzahl oder als Produkt zweier sicherer Primzahlen zu wählen (vgl. Kapitel 2.1).

Das DLP lässt sich zum *Repräsentationsproblem* verallgemeinern [9].

Definition 2.2.2 (Repräsentationsproblem) *Es seien $g_0, \ldots, g_{N-1}$ die Generatoren einer endlichen zyklischen Gruppe $(G, \cdot)$, d. h. $G = \langle g_0, \ldots, g_{N-1} \rangle := \{g_0^{a_0} \cdot \ldots \cdot g_{N-1}^{a_{N-1}} | a_i \in \mathbb{Z}, i = 0, \ldots, N-1\}$. Die Repräsentation eines Elementes*

$h \in G$ ist ein N-Tupel $(a_0, \ldots, a_{N-1})$, so dass $\forall i : 0 \leq a_i < |G|$ gilt:

$$h = \prod_{i=0}^{N-1} g_i^{a_i}$$

Es sei eine Gruppe G mit den Generatoren $g_0, \ldots, g_{N-1}$ und ein Element $h \in G$ gegeben. Das Repräsentationsproblem ist die Suche nach einer Repräsentation für h in Abhängigkeit von $g_0, \ldots, g_{N-1}$.

Das DLP ist ein Sonderfall des Repräsentationsproblems für den Fall $N = 1$. Falls es einen effizienten Algorithmus gibt, um das DLP zu lösen, so gibt es auch einen effizienten Algorithmus, um das Repräsentationsproblem zu lösen [9].

2.2.2 Decisional-Diffie-Hellman Vermutung

Definition 2.2.3 (DDH Vermutung [20]) *Es sei G eine zyklische Gruppe der Ordnung q mit Generator g und $a_0, a_1, b \in_R \{0, \ldots, q-1\}$. Es gibt keinen effizienten Algorithmus, der zwischen den Verteilungen $\langle g^{a_0}, g^{a_1}, g^{a_0 a_1} \rangle$ und $\langle g^{a_0}, g^{a_1}, g^{b} \rangle$ unterscheiden kann.*

In einer Schnorr-Gruppe ist die *DDH (Decisional-Diffie-Hellman) Vermutung* erfüllt. Es seien p und q Primzahlen. Die Schnorr-Gruppe ist eine multiplikative Untergruppe von $\mathbb{Z}_p^*$ mit $p = q \cdot d + 1$ und $d \in \mathbb{Z}$ [150]. Es ist $g = h^d \bmod p$ mit $h \in_R \mathbb{Z}_p^*$ und $h^d \not\equiv 1 \pmod{p}$ ein Generator dieser Untergruppe. Nach dem Satz von Lagrange (Satz 2.1.1) hat sie die Ordnung q.

Die *SDDHI (Strong Decision-Diffie-Hellman Inversion) Vermutung* ist eine Variante der DDH Vermutung, bei der das Inverse des Exponenten betrachtet wird.

Definition 2.2.4 (SDDHI Vermutung [28]) *Es sei ℓ_ϕ ein Sicherheitsparameter, $\tilde{p}$ ein beliebiges Polynom und g ein Generator der Gruppe G mit der Ordnung $p \in \Theta(2^{\ell_\phi})$. Das Orakel $\mathcal{O}_a(\cdot)$ liefert für Eingaben $b \in \mathbb{Z}_p^*$ die Werte $g^{1/(a+b)}$ zurück. Es gilt für alle Angreifer $\mathcal{A}^{(\cdot)}$, die sich durch eine probabilistische Turing-Maschine beschreiben lassen können und ihr Ergebnis in polynomialer Zeit ausgeben:*

$$Pr\begin{bmatrix} a \leftarrow \mathbb{Z}_p^*; (b', state) \leftarrow \mathcal{A}^{\mathcal{O}_a}(g, g^a); h_0 = g^{1/(a+b')}; \\ h_1 \leftarrow G; d \leftarrow \{0,1\}; d' \leftarrow \mathcal{A}^{\mathcal{O}_a}(h_d, state) : d = d' \end{bmatrix} < 1/2 + 1/\tilde{p}(\ell_\phi)$$

Ein Angreifer kann bis auf b' beliebige Anfragen an das Orakel $\mathcal{O}_a(\cdot)$ stellen.

Die SDDHI Vermutung besagt, dass es ohne Kenntnis von a praktisch nicht möglich ist, zwischen einem zufälligen Gruppenelement und $g^{1/(a+b')}$ zu entscheiden.

2.2.3 Computational-Diffie-Hellman Vermutung

Eng verbunden mit der DDH Vermutung ist die *CDH (Computational-Diffie-Hellman) Vermutung*.

Definition 2.2.5 (CDH Vermutung [167]) *Es sei G eine zyklische Gruppe der Ordnung q mit Generator g und $a_0, a_1 \in_R \{0, \ldots, q-1\}$. Es gibt keinen effizienten Algorithmus, um $g^{a_0 \cdot a_1}$ aus $\langle g, g^{a_0}, g^{a_1} \rangle$ zu berechnen.*

Es wird vermutet, dass die CDH Vermutung eine schwächere Vermutung als die DDH Vermutung ist. Es gibt Gruppen, in denen das Lösen des CDH Problems schwierig ist, obwohl das Lösen des DDH Problems leicht ist (siehe Pairing-based Cryptography) [137].

2.2.4 RSA-Problem

Beim DLP wird die Frage beantwortet, wie häufig die Gruppenoperation auf einen Anfangswert angewendet werden muss, um einen Endwert zu erreichen. Beim *RSA-Problem* ist dieser Wert bekannt und es wird der unbekannte Anfangswert gesucht, d. h. die Wurzel eines Gruppenelementes [117].

Definition 2.2.6 (RSA-Problem) *Es sei G eine RSA-Gruppe mit unbekannter Ordnung. Es sind gegeben eine positive ganze Zahl a mit $ggT(a, (p-1)(q-1)) = 1$ und eine ganze Zahl h. Das RSA-Problem ist die Suche nach g, so dass $h = g^a$ gilt (g ist die a-te Wurzel von h).*

Das Lösen des RSA-Problems ist äquivalent zum Entschlüsseln eines beliebigen Ciphertextes mittels des RSA-Verfahrens, ohne den privaten Schlüssel zu kennen [145].

Das Lösen des RSA-Problems ist nicht schwerer als das Lösen des *Faktorisierungsproblems*, d. h. nicht aufwendiger als die Faktorisierung des RSA-Modulus n. Kann ein Angreifer $n = p \cdot q$ faktorisieren, kann er auch $\varphi(n) = (p-1)(q-1)$ berechnen. Daraus lässt sich mithilfe des euklidischen Algorithmus und dem öffentlichen RSA-Schlüssel der private RSA-Schlüssel effizient berechnen und damit jeder beliebiger Ciphertext entschlüsseln.
Es ist nicht bekannt, ob die Umkehrung gilt, d. h. ob das Lösen des Faktorisierungsproblems nicht schwerer als das RSA-Problem ist und damit beide Probleme vom Aufwand identisch sind [117]. Es wird vermutet, dass das RSA-Problem generell leichter zu lösen ist als das Faktorisierungsproblem[7] [21] [1].

[7] Bezogen auf das RSA-Verfahren können nach dem Lösen des Faktorisierungsproblems alle Ciphertexte entschlüsselt werden. Das Lösen des RSA-Problems erlaubt nur die Entschlüsselung eines Ciphertextes, so dass anzunehmen ist, dass es leichter zu lösen ist.

Die Einschränkungen gegenüber einem Angreifer können gelockert werden. Es ist zu vermuten, dass das RSA-Problem selbst dann nicht lösbar ist, wenn der Angreifer den Wert a frei wählen kann [30].

Definition 2.2.7 (Starke RSA-Vermutung) *Die starke RSA-Vermutung besagt, dass es praktisch nicht möglich ist, aus einem zufälligen RSA-Modulus n und einem Zufallselement $h \in \mathbb{Z}_n^*$ die Werte $a > 1$ und g zu berechnen, so dass $g^a \equiv h \pmod{n}$ gilt. Das Tupel (n, h) bezeichnet die* Instanz *des* flexiblen *RSA-Problems.*

Im weiteren Verlauf dieses Kapitels ist der nachfolgende Satz wichtig [36].

Satz 2.2.1 *Es sei $\mathbb{Z}_n^*$ eine RSA-Gruppe mit Modulus n und $g_0, g_1 \in_R (\mathbb{Z}_n^*)^2$, dann ist es unter der starken RSA-Vermutung schwierig, ein $h \in \mathbb{Z}_n^*$ und ganze Zahlen a_0, a_1, a_2 zu bestimmen, so dass gilt:*

$$h^{a_0} = g_0^{a_1} g_1^{a_2} \quad \text{und} \quad (a_0 \nmid a_1 \text{ oder } a_0 \nmid a_2)$$

Es ist leicht zu sehen, dass $a_0 \mid a_1$ und $a_0 \mid a_2$, wenn $h = g_0^{a_1'} g_1^{a_2'}$ gewählt wird und beide Seiten mit a_0 potenziert werden. Damit $a_0 \nmid a_1$ oder $a_0 \nmid a_2$ gilt, muss entweder $a_0 \cdot a_1'$ oder $a_0 \cdot a_2'$ um die Gruppenordnung reduziert werden. Da die Gruppenordnung allerdings unbekannt ist, ist das nicht möglich.

2.2.5 Schwierige zahlentheoretische Probleme und Homomorphismen

In der vorliegenden Arbeit werden schwierige zahlentheoretische Probleme in Form von Gruppenhomomorphismen, deren Urbilder nicht effizient berechenbar sind, beschrieben.
Es sei G eine Gruppe, in der diskrete Logarithmen nicht effizient berechenbar sind. Des Weiteren sei g ein Generator der Gruppe.

$$\phi_{\mathrm{DLP}} : \begin{cases} \mathbb{Z} & \to & G \\ i & \mapsto & g^i \end{cases} \qquad \phi_{\mathrm{DLP}'} : \begin{cases} \mathbb{Z}_{|G|} & \to & G \\ i & \mapsto & g^i \end{cases}$$

Beschrieben wird das DLP als Gruppenhomomorphismus bei unbekannter Gruppenordnung (links) oder bei bekannter Gruppenordnung (rechts). Mehrere Gruppenhomomorphismen können miteinander verknüpft werden. Es seien G_0, G_1 und $(H, \diamond)$ Gruppen und $\phi_{G_0} : G_0 \to H$ und $\phi_{G_1} : G_1 \to H$ zwei Homomorphismen, in denen Urbilder nicht effizient berechenbar sind, dann folgt, dass

$$\phi_{G_0 \times G_1} : \begin{cases} G_0 \times G_1 & \to & H \\ g_0, g_1 & \mapsto & \phi_{G_0}(g_0) \diamond \phi_{G_1}(g_1) \end{cases}$$

ebenfalls ein Homomorphismus ist.

2.3 Commitment-Verfahren

Ein Commitment-Verfahren ist ein Protokoll zwischen zwei Parteien und ein wichtiger Grundbaustein in Zero-Knowledge Protokollen [151]. Eine Partei speichert einen Wert (das Geheimnis) in dem sogenannten *Commitment* ab, veröffentlicht das Commitment und legt sich damit auf das Geheimnis fest, ohne es zu offenbaren. Commitments lassen sich durch Gruppenhomomorphismen mit dem Geheimnis als Urbild darstellen. Ein Commitment erfüllt zwei Eigenschaften [81].

1. *Binding*: Sobald ein Geheimnis in einem Commitment untergebracht ist, kann es praktisch nicht mehr unerkannt aus dem Commitment entfernt bzw. ausgetauscht werden. Die Binding-Eigenschaft lässt sich genauer spezifizieren.

    - *Perfect Binding*: Zu einem Commitment gibt es genau ein Urbild, so dass es unmöglich ist, für dasselbe Commitment ein weiteres Urbild zu finden. Das ist der Fall, wenn der Homomorphismus injektiv ist.

    - *Computational Binding*: Zu einem Commitment kann es zwar mehrere Urbilder geben, diese lassen sich aber nicht in der Praxis effizient berechnen.

2. *Hiding*: Aus dem Commitment kann nicht auf das Geheimnis geschlossen werden. Auch die Hiding Eigenschaft lässt sich genauer spezifizieren.

    - *Perfect Hiding*: Es ist unmöglich, aus einem Commitment eine Aussage über das Geheimnis zu machen. Ein Commitment besitzt diese Eigenschaft, falls jedes mögliche Geheimnis mit derselben Wahrscheinlichkeit auf jedes mögliche Commitment abgebildet wird.

    - *Statistical Hiding*: Dasselbe Geheimnis wird mit unterschiedlichen Wahrscheinlichkeiten auf verschiedene Commitments abgebildet. Ein Commitment ist Statistical Hiding, falls die Unterschiede in den Wahrscheinlichkeiten vernachlässigbar klein sind. Ist die Ordnung der Bildgruppe (Gruppe der Commitments) nicht bekannt und werden in der Urbildgruppe (Gruppe der Geheimnisse) Werte verwendet, die die Ordnung der Bildgruppe um Größenordnungen übersteigen, so ist das Commitment in der Regel Statistical Hiding.

    - *Computational Hiding*: Das Geheimnis kann aus dem Commitment zwar theoretisch, aber nicht praktisch berechnet werden. Das ist der Fall, wenn der Gruppenhomomorphismus injektiv ist, aber Urbilder nicht effizient berechenbar sind.

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

Tabelle 2.1: Mögliche Kombinationen an Eigenschaften von Commitments.

Ein Commitment kann nur die in Tabelle 2.1 dargestellten Kombinationen aus Binding- und Hiding Eigenschaften annehmen [81]. Perfect Binding und Perfect Hiding schließen sich gegenseitig aus, da bei einem injektiven Homomorphismus (Perfect Binding) nicht ein Geheimnis mit derselben Wahrscheinlichkeit auf jedes Commitment abgebildet werden kann. Dasselbe gilt für die Kombination aus Perfect Binding und Statistical Hiding.
Computational Binding und Computational Hiding schließen sich aus, weil nicht mehrere unterschiedliche Urbilder auf dasselbe Commitment abgebildet werden können (Computational Binding) und gleichzeitig ein eindeutiges Geheimnis aus dem Commitment berechnet werden kann (Computational Hiding).
Die drei folgenden Commitment-Verfahren erfüllen jeweils die drei möglichen Eigenschaften Perfect Binding / Computational Hiding, Computational Binding / Perfect Hiding und Computational Binding / Statistical Hiding.

2.3.1 Einfaches Commitment

Betrachtet werden zwei Gruppen W und G, zwischen denen ein Commitment als Gruppenmonomorphismus $\phi : W \to G$ definiert ist. Die Abbildung ist injektiv und damit das Commitment Perfect Binding und Computational Hiding. Ein Beispiel für ein einfaches Commitment ist $\phi(w) = g^w$ mit den Gruppen $W = \mathbb{Z}_q$ und $G = \mathrm{QR}\{\mathbb{Z}_q\}$, wobei p eine sichere Primzahl mit $p = 2q + 1$ ist und damit G die Ordnung q hat. Weiterhin gibt es einen Generator $\langle g \rangle = G$ und ein Geheimnis $w \in W$.

In der Praxis ergibt sich bei dem Verfahren ein Problem, falls das Geheimnis aus einer begrenzten Menge stammt.
Ein Angreifer kann das Commitment $\phi(w')$ für verschiedene Geheimnisse w' berechnen und überprüfen, ob $\phi(w') = \phi(w)$ gilt. Aufgrund der injektiven Abbildung gilt $w' = w$ falls $\phi(w') = \phi(w)$, so dass der Angreifer das Geheimnis berechnen kann.

2.3.2 Pedersen-Commitment

1991 stellte Torben Pedersen ein Commitment-Verfahren vor, das die Problematik bei einer begrenzten Geheimnismenge durch eine Zufallskomponente löst [134]. Ein *Pedersen-Commitment* ist Computational Binding und Perfect Hiding. Das heißt, es ist zwar theoretisch möglich, weitere Geheimnisse mit demselben Commitment zu berechnen, es lassen sich aber aus dem Commitment keine Rückschlüsse auf das Geheimnis ziehen.

Es sei $(W_0, \circ)$ die Gruppe der Geheimnisse, $(W_1, \diamond)$ die Gruppe der Zufallskomponenten und $(G, \star)$ die Gruppe der Commitments. Alle drei Gruppen haben die bekannte Ordnung q. Es werden zwei Gruppenisomorphismen $\phi_0 : W_0 \to G$ und $\phi_1 : W_1 \to G$, in denen Urbilder nicht effizient berechenbar sind und ein Gruppenepimorphismus $\phi : W_0 \times W_1 \to G, \phi(w, r) \mapsto \phi_0(w) \star \phi_1(r)$ definiert. Aus dem Geheimnis w und einer Zufallskomponente r berechnet sich damit das Commitment $C_t = \phi_0(w) \star \phi_1(r)$.

Das Pedersen-Commitment ist Perfect Hiding. Die Abbildungen ϕ_0 und ϕ_1 sind bijektiv und bilden jedes Geheimnis in W_0 auf jedes Commitment in G genau einmal und mit derselben Wahrscheinlichkeit ab.

Damit die Binding-Eigenschaft erfüllt ist, darf es nicht möglich sein, ein weiteres Urbildpaar (w', r') zu finden, so dass $\phi(w, r) = \phi(w', r')$ gilt.
Beispiel: Seien $(W_0, \circ) = (W_1, \diamond) = \mathbb{Z}_q$ und $(G, \star) = \mathrm{QR}_p$ Gruppen mit einer sicheren Primzahl $p = 2q + 1$, $\langle g \rangle, \langle h \rangle \in G$ und $\phi_0(w) = g^w$, $\phi_1(r) = h^r$, $\phi(w, r) = g^w \cdot h^r$. Ein Angreifer kann für das gleiche Commitment $\phi(w, r)$ ein neues Geheimnis w' und eine Zufallskomponente r' berechnen. Mit $w \in W_0$ folgt $r' = \log_h(g)(w - w') + r$. Die Berechnung ist allerdings nur theoretisch möglich, da für genügend große Werte der diskrete Logarithmus vom Generator g zur Basis h nicht effizient berechenbar ist. Das Pedersen-Commitment ist deswegen Computational Binding.

2.3.3 Damgård-Fujisaki-Commitment

Das Pedersen-Commitment hat den Nachteil, dass die Menge an Geheimnissen durch die Ordnung q der verwendeten Gruppen begrenzt wird. Damgård und Fujisaki beschreiben ein Commitment-Verfahren, bei dem als Geheimnis eine beliebige ganze Zahl $w \in \mathbb{Z}$ gewählt werden kann [49].

Der Modulus $n = p \cdot q$ wird als Produkt zweier natürlicher Zahlen $p, q \in \mathbb{N}$ mit den Eigenschaften $p \equiv q \equiv 3 \pmod 4$, $\mathrm{ggT}(p - 1, q - 1) = 2$ gewählt. Außerdem dürfen $p - 1$ und $q - 1$ nicht zu viele kleine Primfaktoren besitzen [49]. Die Anforderungen werden erfüllt, indem p und q als große sichere Primzahlen gewählt werden.

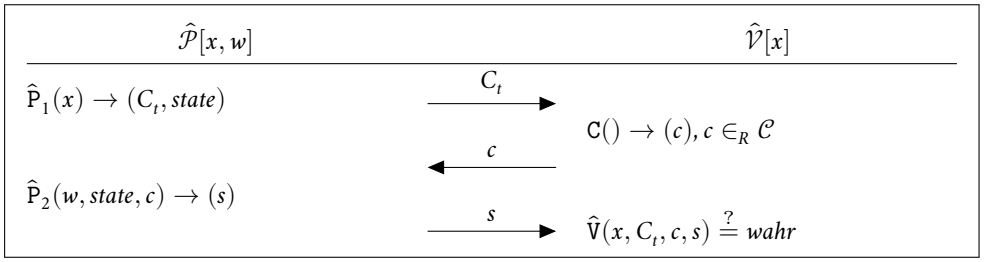


Abbildung 2.1: Interaktives Σ-Protokoll.

Es sei $(W_0, +) = \mathbb{Z}$ die Gruppe der Geheimnisse, $(W_1, +) = \mathbb{Z}$ die Gruppe der Zufallskomponenten und $(G, \cdot) = \langle g \rangle = \langle h \rangle \subseteq \mathbb{Z}_n^*$ die Gruppe der Commitments. Ein *Damgård-Fujisaki-Commitment* wird durch den Homomorphismus $\phi : W_0 \times W_1 \to G, (w, r) \mapsto g^w \cdot h^r$ beschrieben.
In der Praxis wird die Zufallskomponente $r \in_R \{0,1\}^{\ell_n + \ell_\phi}$ mit $\ell_n = \lceil \log_2(n) \rceil$ gewählt. Der Parameter ℓ_ϕ ist ein Sicherheitsparameter (in der Regel $\ell_\phi = 80$).

Das Damgård-Fujisaki-Commitment ist aufgrund der größeren Urbildgruppen W_0 und W_1 Statistical Hiding und aus demselben Grund wie das Pedersen-Commitment Computational Binding. Die Gruppengrößen müssen so gewählt werden, dass der diskrete Logarithmus $\log_g(h)$ oder $\log_h(g)$ praktisch nicht berechenbar ist.
Die Sicherheit des Damgård-Fujisaki-Verfahrens hängt zusätzlich von der Schwierigkeit des Faktorisierungsproblems ab. Denn ist die Gruppenordnung bekannt, kann die Gruppenordnung zum Geheimnis hinzuaddiert werden, ohne dass sich das Commitment ändert. Damit ist die Binding-Eigenschaft nicht mehr erfüllt.

2.4 Σ-Protokolle

Σ-Protokolle sind Protokolle zwischen einem *Prover* $\hat{\mathcal{P}}$ und einem *Verifier* $\hat{\mathcal{V}}$, durch die ein Prover einem Verifier beweisen kann, einen geheimen Wert w zu kennen. Es sei $R \subseteq \{0,1\}^* \times \{0,1\}^*$ eine binäre Relation. $\hat{\mathrm{P}}(x) \to (C_t)$ beschreibt einen Algorithmus $\hat{\mathrm{P}}$ mit dem Eingabeparameter x und Ausgabeparameter C_t. Ein Σ-Protokoll ist nach Bangerter et al. [9] wie folgt definiert.

Definition 2.4.1 (Interaktives Σ-Protokoll) *Es sei R eine binäre Relation mit $(x, w) \in R$ und den Algorithmen $\hat{\mathrm{P}}_1, \hat{\mathrm{P}}_2$ und $\hat{\mathrm{V}}$. Ein Protokoll zwischen einem Prover $\hat{\mathcal{P}} := \hat{\mathcal{P}}(x, w)$ und einem Verifier $\hat{\mathcal{V}} := \hat{\mathcal{V}}(x)$ heißt Σ-Protokoll mit der Challenge-Menge $\mathcal{C}$, falls folgende Bedingungen erfüllt sind:*

- *Es gibt ein 3-Wege-Protokoll wie in Abbildung 2.1 dargestellt.*

- Vollständigkeit*: Ist der Prover ehrlich, akzeptiert der Verifier die Protokollausführung und gibt am Ende* wahr *zurück.*

Der Wert x wird als *öffentlicher Wert* bezeichnet, der vom Geheimnis w abhängig ist. In der Regel ist x ein Commitment zu w.
Beim interaktiven Σ-Protokoll[8] (Abbildung 2.1) werden drei Nachrichten zwischen Prover und Verifier ausgetauscht. Der Prover überträgt als erste Nachricht das *Commitment*[9] C_t zum Verifier. Der Verifier antwortet mit der *Challenge* c. Als letzte Nachricht sendet der Prover die *Response* s an den Verifier. Das Tripel (C_t, c, s) wird *Transaktionstripel* genannt. Ein Tripel ist *korrekt*, falls der Algorithmus $\hat{\mathtt{V}}$ den Protokolldurchlauf akzeptiert und am Ende *wahr* zurückliefert.
Ein Σ-Protokoll definiert folgende Algorithmen:

- $\hat{\mathtt{P}}_1(x) \rightarrow (C_t, state)$: Der Algorithmus $\hat{\mathtt{P}}_1$ berechnet aus dem öffentlichen Wert x die Nachricht C_t. In der Regel ist die Nachricht C_t ein Commitment mit Zufallswerten. Die Zufallswerte werden im Status *state* gespeichert.
- $\mathtt{C}() \rightarrow (c)$: Der Algorithmus generiert die Challenge c aus der Challenge-Menge $\mathcal{C}$.
- $\hat{\mathtt{P}}_2(w, state, c) \rightarrow (s)$: Der Prover berechnet aus dem Geheimnis w, der Zustandsvariablen *state* und der Challenge c die Response s.
- $\hat{\mathtt{V}}(x, C_t, c, s)$: $\hat{\mathcal{V}}$ verwendet den Algorithmus, um ein Transaktionstripel auf Korrektheit zu prüfen. Im Falle eines Fehlers gibt $\hat{\mathtt{V}}$ *falsch* ansonsten *wahr* zurück.

2.4.1 Proof of Knowledge

Falls sich Prover and Verifier korrekt verhalten, kann der Prover mithilfe eines Σ-Protokolls den Verifier davon überzeugen, den geheimen Wert w zu kennen. Betrachtet wird der Fall, dass der Prover das Geheimnis nicht kennt und sich nicht korrekt verhält. Durch einen *PK (Proof of Knowledge)* kann der Prover nur mit einer Wahrscheinlichkeit „schummeln", die nicht höher als der sogenannte *Knowledge Error* $\varkappa(w)$ ist. Der Knowledge Error ist die Wahrscheinlichkeit, dass der Verifier am Ende der Protokollausführung *wahr* ausgibt, obwohl der Prover das Geheimnis w nicht kennt. Ein Proof of Knowledge ist wie folgt formal definiert [9] [13]:

[8] Initiiert der Verifier am Anfang die Kommunikation durch Senden einer zusätzlichen Nachricht an $\hat{\mathcal{P}}$, hat die Darstellung des Nachrichtenflusses die Form eines großen griechischen Sigmas.

[9] Dabei muss es sich nicht zwangsläufig um ein Commitment nach Kapitel 2.3 handeln. Mit der Bezeichnung „Commitment" wird lediglich zum Ausdruck gebracht, dass die Nachricht C_t an den Verifier $\hat{\mathcal{V}}$ gesendet wird, bevor $\hat{\mathcal{V}}$ eine Nachricht an den Prover schickt.

Definition 2.4.2 (Proof of Knowledge) *Ein Paar von Algorithmen* $(\hat{\mathrm{P}}, \hat{\mathrm{V}})$ *heißt Proof of Knowledge für eine Relation R mit Knowledge Error* $\varkappa$*, wenn gilt:*

- Vollständigkeit: *Gilt* $(x, w) \in R$ *und ist der Prover ehrlich, dann liefert der Verifier am Ende der Protokollausführung* wahr *zurück.*
- Soundness: *Es gibt eine probabilistische Turing-Maschine* $\mathcal{M}$ *(den* Knowledge Extractor*) mit rewindable Blackbox Zugriff auf den Prover. Es sei* $\eta(w)$ *die Wahrscheinlichkeit durch einen Prover-Algorithmus* $\hat{\mathrm{P}}^*$*, den Verifier von der Kenntnis des Geheimnisses w zu überzeugen. Des Weiteren sei* $\tilde{p}(\cdot)$ *ein Polynom. Falls* $\eta(w) > \varkappa(w)$ *gilt, beträgt die maximale Anzahl von Schritten*[a]*, das Geheimnis w zu extrahieren:*

$$\frac{\tilde{p}(|w|)}{\eta(w) - \varkappa(|w|)}$$

[a] wenn der Zugriff auf $\hat{\mathrm{P}}^*$ als ein Schritt zählt

Solange der Prover $\hat{\mathcal{P}}$ ehrlich ist, kann er den Verifier überzeugen (Vollständigkeit). Falls ein nicht-ehrlicher Prover einen Verifier dazu bringt, den Beweis mit hoher Wahrscheinlichkeit zu akzeptieren, kennt der Prover entweder das Geheimnis oder aber es gibt einen Knowledge Extractor $\mathcal{M}$, um das Geheimnis zu berechnen. Die Soundness-Eigenschaft stellt sicher, dass der Prover nur mit sehr geringer Wahrscheinlichkeit betrügen kann.

Der Knowledge Extractor $\mathcal{M}$ hat „Rewindable Blackbox Zugriff" auf $\hat{\mathcal{P}}$. So kann $\mathcal{M}$ beliebige Nachrichten an $\hat{\mathcal{P}}$ schicken und empfangen, allerdings nicht in $\hat{\mathcal{P}}$ „hinein schauen". $\mathcal{M}$ kann ein Protokoll mit $\hat{\mathcal{P}}$ ausführen und zu einem früheren Zustand im Protokoll zurückspringen („rewinding") und von dort aus die Ausführung mit neuen Nachrichten fortführen. Durch Angabe des Knowledge Extractors kann bewiesen werden, dass die Soundness-Eigenschaft eines Protokolls erfüllt ist.

Damgård hat gezeigt, dass es bei jedem Σ-Protokoll möglich ist, zwei Transaktionstripel (C_t, c, s) und (C_t, c', s') für $c \neq c'$ in der maximalen Laufzeit des Knowledge Extractors nach Definition 2.4.2 zu finden [48]. Ist es möglich, aus den beiden Transaktionstripeln das Geheimnis w zu extrahieren, ist das Σ-Protokoll ein Proof of Knowledge.

2.4.2 Zero-Knowledge Proof of Knowledge

Bei einem *ZPK (Zero-Knowledge Proof of Knowledge)* handelt es sich um einen Proof of Knowledge zwischen einem Prover $\hat{\mathcal{P}}$ und einem Verifier $\hat{\mathcal{V}}$. Der Prover überzeugt den Verifier, das Geheimnis zu kennen. Erhält der Verifier nach

dem Protokollablauf nur diese eine Information, handelt es sich um einen ZPK [9].

Definition 2.4.3 (Zero-Knowledge Proof of Knowledge) *Ein Paar* $(\hat{\mathrm{P}}, \hat{\mathrm{V}})$ *von Algorithmen heißt ZPK für eine Relation R, wenn gilt:*

- $(\hat{\mathrm{P}}, \hat{\mathrm{V}})$ *ist ein Proof of Knowledge nach Definition 2.4.2.*
- Zero-Knowledge: *Es gibt eine probabilistische Turing-Maschine* $\mathcal{T}$ *mit polynomialer Laufzeit, deren Transaktionen mit dem Verifier* $\hat{\mathcal{V}}$ *sich nicht von den Transaktionen zwischen Prover* $\hat{\mathcal{P}}$ *und Verifier* $\hat{\mathcal{V}}$ *unterscheiden.*

Die probabilistische Turing-Maschine $\mathcal{T}$ ist ein Simulator und simuliert die Funktionalität des Provers. Er kommuniziert mit dem Verifier und zeichnet die Transaktionen auf. Wenn sich die Transaktionen nicht von echten Transaktionen mit dem Prover unterscheiden, ist das Protokoll Zero-Knowledge. Im Gegensatz zum Prover kennt der Simulator das Geheimnis w nicht. Somit erhält der Verifier durch die Kommunikation mit dem Prover nicht mehr Informationen als er durch Analyse der öffentlichen Werte nicht ohnehin schon hatte.
Je nachdem, ob die simulierten und tatsächlichen Transaktionen nicht-unterscheidbar, statistisch nicht-unterscheidbar oder rechnerisch nicht-unterscheidbar sind, wird von *perfect*, *statistical* oder *computational Zero-Knowledge* gesprochen [9].

Bei ZPKs wird zwischen zwei Arten von Verifiern unterschieden [9]. Der *honest-but-curious Verifier* hält sich strikt an das Protokoll, kann aber versuchen, aus den übertragenen Informationen Erkenntnisse zu gewinnen. Der *malicious Verifier* dagegen weicht gezielt vom Protokollablauf ab, um das Geheimnis zu berechnen. Insbesondere muss der malicious Verifier die Challenge nicht zufällig wählen, was die Wahrscheinlichkeitsverteilung der Transaktionstripel beeinflusst. Aus diesem Grund wird in den folgenden Abschnitten vom ersten Verifier-Typ ausgegangen, was die Definition eines Simulators vereinfacht. Zero-Knowledge Protokolle mit solchen Verifiern werden *HVZK (Honest-Verifier Zero-Knowledge)* genannt [156]. Wenn ein Proof of Knowledge mit Knowledge Error $\varkappa$ für eine Relation R d-mal wiederholt wird, verringert sich der Knowledge Error auf $\varkappa^d$. Bei den Σ-Protokollen in dieser Arbeit hängt der Knowledge Error von der Challenge-Menge $\mathcal{C}$ ab. Um einen niedrigen Knowledge Error zu erreichen, gibt es zwei Möglichkeiten. Entweder wird die Challenge aus einer großen Menge $\mathcal{C}$ gewählt und das Protokoll einmal ausgeführt oder die Challenge-Menge ist klein (z. B. $\mathcal{C} = \{0, 1\}$) und das Protokoll wird mehrmals wiederholt [9].

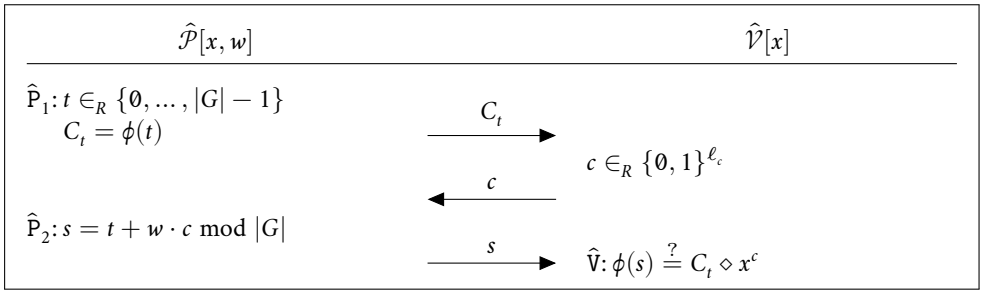


Abbildung 2.2: Σ^ϕ-Protokoll.

2.4.3 Σ^ϕ-Protokoll

Es seien $(W, +)$ und $(G, \diamond)$ zwei endliche zyklische Gruppen und $\phi : W \to G$ ein Gruppenmonomorphismus zwischen W und G mit $x = \phi(w)$, $x \in G$ und $w \in W = \{0, \ldots, |G|-1\}$. Ein Prover kann mit dem Σ^ϕ-Protokoll einem Verifier beweisen, dass er das Urbild zu ϕ bzw. das Geheimnis w kennt. Das Commitment x ist sowohl dem Prover als auch dem Verifier bekannt, das Geheimnis w kennt nur der Prover [9].

Abbildung 2.2 zeigt das Σ^ϕ-Protokoll zwischen einem Prover $\widehat{\mathcal{P}}$ und einem Verifier $\widehat{\mathcal{V}}$. Es handelt sich um einen ZPK, da die drei Eigenschaften Vollständigkeit, Soundness und Zero-Knowledge erfüllt sind.

- *Vollständigkeit*: Verhalten sich Prover und Verifier ehrlich, gilt $\phi(s) = C_t \diamond x^c$, da $\phi(s) = \phi(t + w \cdot c) = \phi(t) \diamond \phi(w \cdot c) = \phi(t) \diamond \phi(w)^c = C_t \diamond x^c$.

- *Soundness*: Die Soundness-Eigenschaft ist erfüllt, wenn es einen Knowledge Extractor mit rewindable Blackbox Zugriff auf $\widehat{\mathcal{P}}$ gibt, der das Geheimnis aus den Transaktionen berechnen kann.
  Der Extractor führt einen kompletten Protokolldurchlauf aus und speichert das Transaktionstupel (C_t, c, s). Danach spult der Extractor das Protokoll zurück bis zu der Stelle, an der er das Commitment C_t bekommen hat und führt das Protokoll mit neuer Challenge $c' \neq c$ erneut aus. Der Extractor erhält ein weiteres Transaktionstupel (C_t, c', s').

  Aus $\phi(s) = C_t \diamond x^c$, $\phi(s') = C_t \diamond x^{c'}$ und $x = \phi(w)$ folgt:

  $$\begin{aligned} & \phi(s) \diamond \phi(w)^{-c} = \phi(s') \diamond \phi(w)^{-c'} \\ \Leftrightarrow \quad & \phi(w \cdot (c' - c)) = \phi(s' - s) \end{aligned}$$

  Da die Ordnungen der Gruppen G und W bekannt sind, kann die Wurzel gezogen und das Geheimnis $w = \frac{s'-s}{c'-c}$ berechnet werden. Damit lässt sich

ein Knowledge Extractor angeben, der in polynomialer Zeit das Geheimnis aus den beiden Transaktionstupeln berechnen kann[10]. Die Soundness-Eigenschaft ist erfüllt.

- *Zero-Knowledge*: Das Σ^{ϕ}-Protokoll ist HVZK für beliebige Werte der Challenge-Menge $\{0,1\}^{\ell_c}$. Ein Simulator $\mathtt{T}$ lässt sich wie folgt angeben:

$$\mathtt{T}(x,c) = (\underbrace{\phi(s) \diamond x^{-c}}_{C_t}, c, s)$$

  Der Simulator wählt die Response s zufällig und berechnet aus Challenge c und Response s ein gültiges Commitment C_t. Die Wahrscheinlichkeitsverteilung der simulierten Transaktionstripel entspricht damit der Verteilung von Transaktionstripeln aus einer echten Kommunikation zwischen Prover und Verifier.

2.4.4 Σ^{GSP}-Protokoll

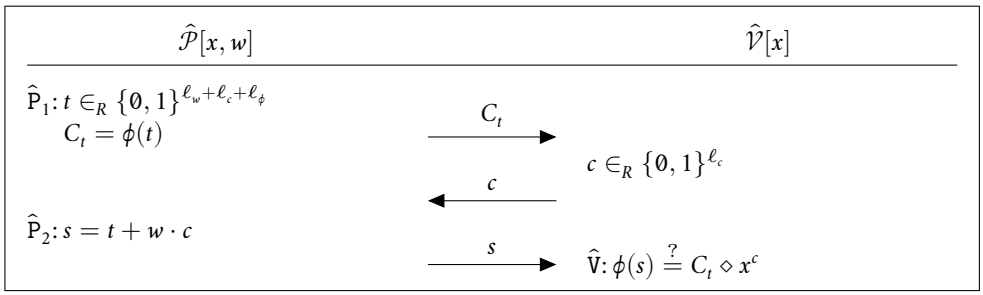


Abbildung 2.3: Σ^{GSP}-Protokoll.

Betrachtet werden wieder zwei endliche zyklische Gruppen $(W,+)$ und $(G,\diamond)$ mit dem Gruppenhomomorphismus $\phi : W \to G$. Abbildung 2.3 zeigt das Σ^{GSP}-*Protokoll (Generalized Schnorr Proof Protokoll)* [9]. Der Prover kennt beim Σ^{GSP}-Protokoll im Gegensatz zum Σ^{ϕ}-Protokoll nicht die Ordnung der Gruppe, aus denen die Urbilder des Homomorphismus ϕ gewählt werden. Das ist z. B. der Fall, wenn die Gruppe G eine RSA-Gruppe $\mathbb{Z}_n^*$ ist und der Prover die Faktorisierung des Modulus n nicht kennt.

Ist die Ordnung der Gruppe G nicht bekannt, muss der Prover t aus dem größeren Intervall $t \in_R \{0,1\}^{\ell_w+\ell_c+\ell_\phi}$ mit der Bitlänge des Geheimnisses ℓ_w, der Bitlänge der Challenge ℓ_c und einem Sicherheitsparameter ℓ_ϕ (in der Regel $\ell_\phi = 80$)

[10] Aus dem Beweis der Soundness-Eigenschaft folgt, dass der Prover t nicht doppelt wählen darf, da ansonsten der Verifier das Geheimnis aus den aufgezeichneten Transaktionen berechnen kann.

wählen. Damit liegt die statistische Zufallsverteilung der Response $s = t + w \cdot c$ nahe der Gleichverteilung $\{0,1\}^{\ell_w+\ell_c+\ell_\phi}$ und ein Angreifer kann aus C_t nicht auf das Urbild t des Homomorphismus $C_t = \phi(t)$ schließen [30].

Es sei G eine endliche zyklische Gruppe mit der modularen Multiplikation als Gruppenoperation. Das Σ^{GSP}-Protokoll ist ein ZPK und erfüllt die drei Eigenschaften Vollständigkeit, Soundness und Zero-Knowledge.

- *Vollständigkeit*: Die Richtigkeit der Eigenschaft folgt wie beim Σ^ϕ-Protokoll aus $\phi(s) = C_t \cdot x^c$.
- *Soundness*: Wie beim Σ^ϕ-Protokoll lässt sich ein Knowledge Extractor für zwei Transaktionstripel (C_t, c, s) und (C_t, c', s') o. B. d. A. mit $c' > c$ und $s \neq s'$ angeben. Aus $C_t = \phi(s) \cdot x^{-c} = \phi(s') \cdot x^{-c'} \pmod n$ folgt

  $$x^{c'-c} \equiv \phi(s' - s) \pmod n.$$

  Es wird zwischen zwei Fällen unterschieden:

  1. Falls $\ell_c = 1$ ist, gilt z. B. $c = 0$ und $c' = 1$. Damit lässt sich das Geheimnis w aus $x \equiv \phi(s' - s) \pmod n$ mit $w = s' - s$ extrahieren. Das Σ^{GSP}-Protokoll ist dann ein Proof of Knowledge.
  2. Gilt $\ell_c > 1$, muss die $(c' - c)$-te Wurzel von $\phi(s' - s)$ berechnet werden. Das ist bei einem hinreichend großen Modulus n praktisch nicht möglich. Unter der starken RSA-Annahme, $\phi(w) = g^w$ und Satz 2.2.1 gilt allerdings $(c' - c) \mid (s' - s)$ und es gibt ein Geheimnis $w \in \mathbb{Z}$ mit $(c' - c)w = (s' - s)$ [30].
     Trotzdem ist das Σ^ϕ-Protokoll für $\ell_c > 1$ kein Proof of Knowledge. Nach Definition 2.4.2 muss der Knowledge Extractor für jeden Prover $\hat{P}^*$ gültig sein. Kennt der Prover jedoch die Faktorisierung von n, kann er c, c', s und s' so wählen, dass $(c' - c) \nmid (s' - s)$ gilt [30].
     In der Regel spielt dieser Sachverhalt in der Praxis keine Rolle und das Σ^{GSP}-Protokoll wird trotzdem als Proof of Knowledge bezeichnet.
- *Zero-Knowledge*: Für das Σ^{GSP}-Protokoll lässt sich wie bereits für das Σ^ϕ-Protokoll ein Simulator erzeugen, so dass ein Verifier nicht zwischen echten und simulierten Transaktionen unterscheiden kann. Im Gegensatz zum Σ^ϕ-Protokoll ist das Σ^{GSP}-Protokoll allerdings statistical honest Verifier Zero-Knowledge, da es von der Wahl von ℓ_ϕ abhängig ist [30].

2.4.5 Nicht-interaktives Zero-Knowledge

Durch die Fiat-Shamir Heuristic lässt sich aus einem interaktiven Σ-Protokoll ein *nicht-interaktives Zero-Knowledge Protokoll* machen [63]. Das hat den Vorteil, dass nur noch eine Nachricht vom Prover an den Verifier gesendet werden

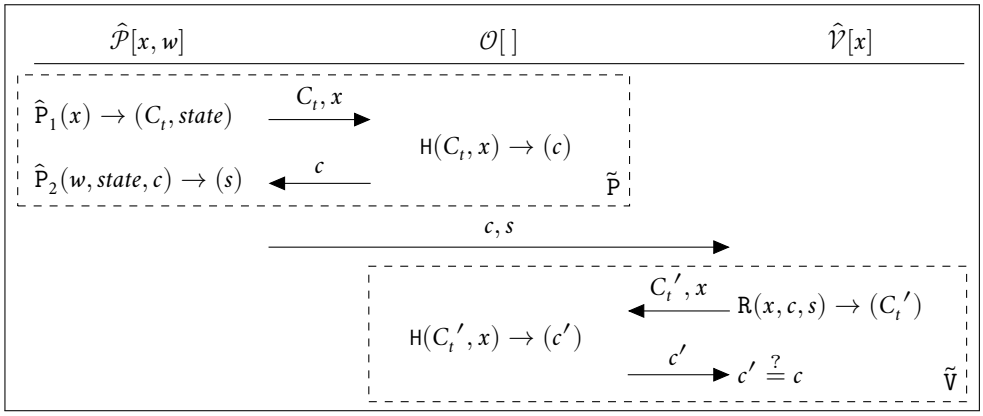


Abbildung 2.4: Umwandlung eines Σ-Protokolls in ein nicht-interaktives Zero-Knowledge Protokoll[11].

muss. Die Challenge wird nicht mehr durch den Verifier, sondern durch das *Zufallsorakel* (engl. *Random Oracle*) $\mathcal{O}$ bestimmt (siehe Abbildung 2.4). Dadurch kann die Zero-Knowledge Eigenschaft auch dann gezeigt werden, wenn es sich um einen malicious Verifier handelt [9].

Das Zufallsorakel lässt sich durch eine surjektive Funktion H modellieren, die Elemente einer unendlichen Definitionsmenge stochastisch gleichverteilt auf eine endliche Zielmenge abbildet. Die Abbildung gilt nur in eine Richtung, eine Umkehrabbildung existiert nicht. In der Praxis wird das Zufallsorakel durch eine kryptographische Hashfunktion implementiert, die lediglich eine Approximation eines Zufallsorakels darstellt. Der Sicherheitsbeweis eines Protokolls, dessen Sicherheit auf dem Random Oracle Modell basiert, ist dadurch streng genommen nicht mehr gültig [39]. Für praktische Anwendungen ist das allerdings häufig irrelevant.

Wird in Abbildung 2.4 der öffentliche Wert x durch eine beliebige Nachricht m ersetzt, wird aus dem Protokoll ein Signaturverfahren. Das Verfahren wird als *Signature Proof of Knowledge* (SPK) bezeichnet.

2.5 Komplexe Zero-Knowledge Proofs

In Kapitel 2.4 wurde dargestellt, wie ein Prover mithilfe des Σ^{ϕ}- und Σ^{GSP}-Protokolls beweisen kann, das Urbild eines Homomorphismus zu kennen. Im Folgenden wird gezeigt, wie die beiden Σ-Protokolle als Bausteine für komplexere Zero-

[11] Der Algorithmus $\mathrm{R}(x, c, s)$ berechnet das Commitment C_t'. Für das Σ^{ϕ}-Protokoll in Abbildung 2.2 würde z. B. $C_t' = \mathrm{R}(x, c, s) = \phi(s) \diamond x^{-c}$ gelten.

Knowledge Proofs verwendet werden können. Vorangestellt ist eine Einführung in die *CSN (Camenisch-Stadler Notation)*.

2.5.1 Camenisch-Stadler Notation

Die Notation von Camenisch und Stadler dient zur Beschreibung von Zero-Knowledge Proofs [29]. Das sei anhand eines Beispiels verdeutlicht:

$$\mathsf{PK}[\underbrace{(w, r)}_{\text{Variablenliste}} : \underbrace{x = g^w h^r}_{\text{Prädikat}} \ \wedge \ \underbrace{r \in [100, 200]}_{\text{Prädikat}}]$$

Die *Variablenliste* enthält eine Auflistung der Geheimnisse[12], die nur dem Prover bekannt sind. Alle weiteren Variablen sind öffentlich und sowohl dem Prover als auch dem Verifier bekannt.
Ein *Prädikat* ist ein mathematischer Ausdruck, der entweder *wahr* oder *falsch* sein kann. Mehrere Prädikate lassen sich durch boolesche logische Operatoren miteinander verknüpfen.

In dem Beispiel muss der Prover einen Verifier überzeugen, die Geheimnisse w und r zu kennen, so dass $x = g^w h^r$ mit $r \in [100, 200]$ erfüllt ist. Der Beweis kann entweder interaktiv oder nicht-interaktiv ausgeführt werden. Um eine Nachricht m zu signieren, schreibt man $\mathsf{SPK}\,[(\ldots) : \ldots]\,(m)$ und der Beweis wird nicht-interaktiv ausgeführt.
Bei der CSN handelt es sich um eine nicht-formale Darstellung. Es erfolgt z. B. keine Definition der verwendeten Gruppen.

2.5.2 „und“-Verknüpfung von Prädikaten

Prädikate lassen sich durch logische „und“-Operatoren miteinander verknüpfen. In Zero-Knowledge Protokollen beweist der Prover dann, dass er alle Geheimnisse kennt, um die Korrektheit der Prädikate zu bestätigen. Es wird folgender ZPK betrachtet.

$$\mathsf{PK}\,[(\mathcal{W}) : x_0 = \phi_0(\tilde{w}_0) \ \wedge \ \ldots \ \wedge \ x_{N-1} = \phi_{N-1}(\tilde{w}_{N-1})] \tag{2.1}$$

Es sei N die Anzahl der Prädikate, $\mathcal{W} = \{w_0, \ldots, w_{N_w - 1}\}$ eine Menge mit N_w Geheimnissen und $x_i = \phi_i(\tilde{w}_i)$ für $i = \{0, \ldots, N-1\}$ das Prädikat mit öffentlichem Wert x_i und Homomorphismus ϕ_i. Die Geheimnisse eines Prädikates lassen sich entweder in einem Tupel $\tilde{w}_i = (\tilde{w}_{i,0}, \ldots, \tilde{w}_{i,N_{\tilde{w},i}-1})$ mit $N_{\tilde{w},i} \leq N_w$ Komponenten

[12] In dieser Arbeit wird die Notation von Camenisch und Stadler in einer abgewandelten Variante verwendet, um die Lesbarkeit zu erhöhen. Die ursprüngliche Notation verwendet griechische Buchstaben zur Beschreibung der Geheimnisse.

und $\tilde{w}_{i,j} \in \mathcal{W}, j \in \{0, \dots, N_{\tilde{w},i} - 1\}$ oder als Menge $\tilde{\mathcal{W}}_i = \{\tilde{w}_{i,0}, \dots, \tilde{w}_{i,N_{\tilde{w},i}-1}\}$ beschreiben. Es muss $\tilde{\mathcal{W}}_i \subseteq \mathcal{W}$ gelten.

Es wird zwischen *unabhängigen* und *abhängigen Prädikaten* unterschieden. Für unabhängige Prädikate gilt $\tilde{\mathcal{W}}_i \cap \tilde{\mathcal{W}}_j = \emptyset$ ($i \neq j$ und $i, j = \{0, \dots, N-1\}$) und sie lassen sich getrennt voneinander betrachten. Bei abhängigen Prädikaten ist das nicht der Fall.

Beweise mit unabhängigen Prädikaten

Sind die Prädikate unabhängig, wird jeweils ein Σ-Protokoll pro Prädikat ausgeführt. Proof 2.1 lässt sich in N separate Beweise bzw. Unterprotokolle zerlegen.

$$\mathsf{PK}\left[\left(\tilde{w}_{0,0}, \dots, \tilde{w}_{0,N_{\tilde{w},0}-1}\right) : x_0 = \phi_0(\tilde{w}_0)\right]$$

$$\dots$$

$$\mathsf{PK}\left[\left(\tilde{w}_{N-1,0}, \dots, \tilde{w}_{N-1,N_{\tilde{w},N-1}-1}\right) : x_{N-1} = \phi_{N-1}(\tilde{w}_{N-1})\right]$$

Die Unterprotokolle werden zu einem Σ^{AND}-Protokoll zusammengefasst [9]. Die Ausführung der Algorithmen $\hat{\mathsf{P}}$ und $\hat{\mathsf{V}}$ bei interaktiven Beweisen bzw. $\tilde{\mathsf{P}}$ und $\tilde{\mathsf{V}}$ bei nicht-interaktiven Beweisen erfolgt in allen Unterprotokollen zur selben Zeit. Es wird bei allen Unterprotokollen dieselbe Challenge verwendet.

Beweise mit abhängigen Prädikaten

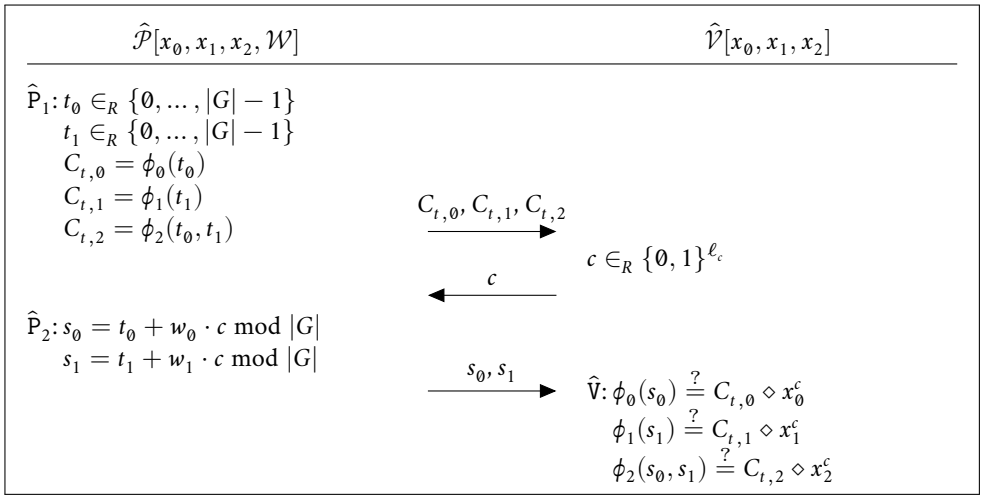


Abbildung 2.5: Σ^{ϕ}-Protokoll für Proof 2.2.

Soll dasselbe Geheimnis in verschiedenen Prädikaten verwendet werden, kann das Σ^{AND}-Protokoll nicht verwendet werden, da die Prädikate abhängig sind.
Ein Zero-Knowledge Proof mit abhängigen Prädikaten wird als ein Σ-Protokoll betrachtet. Für jedes Geheimnis w_i mit $i = \{0, \dots, N_w - 1\}$ wird ein Zufallswert t_i und eine Response s_i erzeugt. Jedes Prädikat erfordert jeweils ein Commitment sowie eine Verifizierungsgleichung. Die Challenge ist unabhängig von der Anzahl der Geheimnisse und Prädikate.
Das folgender Beispiel veranschaulicht die „und"-Verknüpfung von abhängigen Prädikaten.

$$\mathsf{PK}\left[(w_0, w_1) : x_0 = \phi_0(w_0) \ \wedge \ x_1 = \phi_1(w_1) \ \wedge \ x_2 = \phi_2(w_0, w_1)\right] \tag{2.2}$$

Abbildung 2.5 zeigt den Nachrichtenaustausch für den Proof.

2.5.3 „oder"-Verknüpfung von Prädikaten

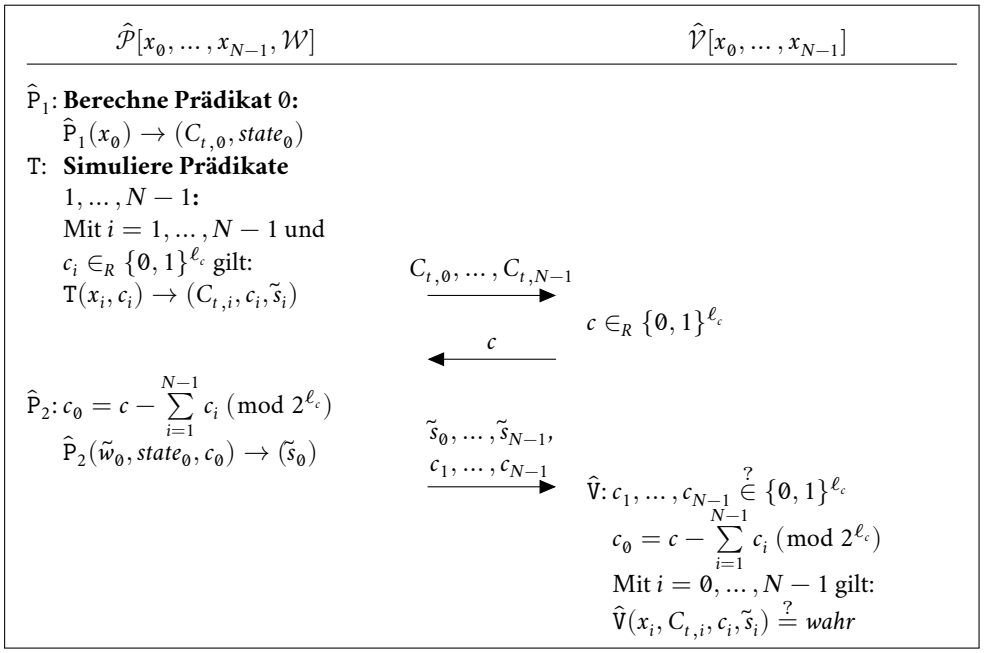


Abbildung 2.6: Σ^{OR}-Protokoll.

Der Beweis, dass ein Prover bei mindestens einem von N unabhängigen Prädikaten alle Geheimnisse kennt, erfolgt über das Σ^{OR}-Protokoll. Der Prover beweist für ein Prädikat, die Geheimnisse zu kennen, wobei die Kenntnis über die restlichen Geheimnisse der anderen Prädikate simuliert wird. Der Verifier weiß am Ende der Protokollausführung, dass der Provider die Geheimnisse mindestens eines Prädikates kennt, kann aber nicht sagen welche [9].

Es sei $\mathcal{W} = \{\tilde{w}_0, \dots, \tilde{w}_{N-1}\}$ die Menge mit Geheimnissen, wobei die Tupel $\tilde{w}_i$ für $i = \{0, \dots, N-1\}$ die Geheimnisse für das i-te Prädikat beschreiben.

$$\mathsf{PK}\left[(\tilde{w}_0, \dots, \tilde{w}_{N-1}) : x_0 = \phi_0(\tilde{w}_0) \ \vee \ \dots \ \vee \ x_{N-1} = \phi_{N-1}(\tilde{w}_{N-1})\right]$$

Es seien nur die Geheimnisse des ersten Prädikates $\tilde{w}_0$ dem Prover bekannt. Die Geheimnisse der anderen Prädikate werden mithilfe des Simulators aus Kapitel 2.4.3 simuliert. Abbildung 2.6 zeigt die Ausführung des Σ^{OR}-Protokolls.
Während der Protokollausführung muss für jedes Geheimnis eine Response berechnet werden. Entsprechend der Zusammenfassung mehrerer Geheimnisse eines Prädikates zu einem Tupel $\tilde{w}_i$, werden die Responses ebenfalls zu einem Tupel $\tilde{s}_i$ zusammengefasst.

„Und"- und „oder"-Proofs lassen sich kombinieren. Dazu werden die „und"-Verknüpfungen nach Kapitel 2.5.2 zu einem Σ-Protokoll zusammengefasst und die Algorithmen der Protokolle durch das Σ^{OR}-Protokoll aufgerufen. Cramer et al. haben gezeigt, wie solche Proofs weiter optimiert werden können [47].

2.5.4 Multiplikative Abhängigkeiten zwischen Geheimnissen

Es sei ϕ_g ein Homomorphismus, der ein Commitment-Verfahren repräsentiert und ein Geheimnis der Gruppe $(W_0, +)$ sowie einen Zufallswert der Gruppe $(W_1, +)$ auf eine multiplikative Gruppe $(G, \cdot)$ abbildet (siehe Kapitel 2.3.2 und 2.3.3). Die Gruppenelemente g und h sind Generatoren der Gruppe G.

$$\phi_g : \begin{cases} W_0 \times W_1 & \to & G \\ w, r & \mapsto & g^w \cdot h^r \end{cases}$$

Ein Prover kann in Zero-Knowledge zeigen, dass ein Geheimnis das Produkt zweier weiterer Geheimnisse ist. Es seien w_0, w_1 und w_2 Geheimnisse für die $w_2 = w_0 \cdot w_1$ gilt [9].

$$\mathsf{PK}\left[(w_i, r_i)_{i=0}^2 : \bigwedge_{i=0}^2 x_i = \phi_g(w_i, r_i) \ \wedge \ w_2 = w_0 \cdot w_1\right]$$

Es sei $x_2 = \phi_g(w_2, r_2) = \phi_g(w_0 \cdot w_1, r_2) = g^{w_0 \cdot w_1} h^{r_2}$. Um die Beziehung $w_2 = w_0 \cdot w_1$ in Zero-Knowledge zeigen zu können, muss das Prädikat umformuliert werden.

$$x_2 = g^{w_0 \cdot w_1} h^{r_2} = g^{w_0 \cdot w_1} h^{r_2} h^{r_0 \cdot w_1} h^{-r_0 \cdot w_1} = x_0^{w_1} h^{r_2 - r_0 \cdot w_1} = \phi_{x_0}(w_1, r_2 - r_0 \cdot w_1)$$

Daraus folgt:

$$\mathsf{PK}\left[\begin{array}{l}(w_i, r_i)_{i=0}^2 : x_0 = \phi_g(w_0, r_0) \wedge \\ x_1 = \phi_g(w_1, r_1) \ \wedge \ x_2 = \phi_{x_0}(w_1, r_2 - r_0 \cdot w_1)\end{array}\right]$$

2.6 Intervall-Proofs

In diesem Abschnitt werden mehrere Verfahren vorgestellt, mit deren Hilfe ein Prover beweisen kann, ein Geheimnis w aus dem Intervall $w \in [a, b]$ zu kennen.

2.6.1 Impliziter Beweis

Beim Σ^{GSP}-Protokoll wird die Response $s = t + w \cdot c$ in der Gruppe $\mathbb{Z}$ berechnet, so dass der Verifier Rückschlüsse auf das Intervall ziehen kann, aus dem das Geheimnis w gewählt wurde [30].
Nach Kapitel 2.4.4 kann der Knowledge Extractor unter der starken RSA-Vermutung[13] das Geheimnis $w' = (s - s')/(c' - c)$ aus dem Proof $\mathsf{PK}[(w') : x = g^{w'}]$ extrahieren. Wählt der Prover $t \in \{0,1\}^{\ell_w + \ell_c + \ell_\phi}$ und $w \in \pm\{0,1\}^{\ell_w}$ und der Verifier $c \in \{0,1\}^{\ell_c}$, dann folgt für die Response $s \in \pm\{0,1\}^{\ell_w + \ell_c + \ell_\phi + 1}$ und damit $w' \in \pm\{0,1\}^{\ell_w + \ell_c + \ell_\phi + 2}$.
Indem der Verifier die Bedingung $s \overset{?}{\in} \pm\{0,1\}^{\ell_w + \ell_c + \ell_\phi + 1}$ überprüft, ergibt sich ein einfacher Intervall-Proof.

$$\mathsf{PK}\left[(w) : x = \phi(w) \ \wedge \ w \in \pm\{0,1\}^{\ell_w + \ell_c + \ell_\phi + 2}\right]$$

Das Geheimnis muss aus einem wesentlich kleineren Intervall $\pm\{0,1\}^{\ell_w}$ gewählt werden, damit der Proof unabhängig von der Challenge c und dem Zufallswert t erfolgreich ausgeführt werden kann. Dadurch kann der Prover nicht zeigen, dass der Wertebereich des Geheimnisses w dem Intervallbereich entspricht. In der Praxis ist das allerdings häufig nicht notwendig.

Das Protokoll lässt sich erweitern, um zu beweisen, dass ein Geheimnis in einem festen Intervall $[a, b]$ liegt. Für die Intervallgrenzen gilt:

$$[a, b] = \left[\frac{a+b}{2} - \frac{b-a}{2}, \frac{a+b}{2} + \frac{b-a}{2}\right]$$

Damit kann der Proof $\mathsf{PK}[(w) : x = \phi(w) \ \wedge \ w \in [a, b]]$ umgeschrieben werden zu:

$$\mathsf{PK}\left[(w) : x = \phi(w + (a+b)/2) \ \wedge \ w \in \left[-\tfrac{b-a}{2}, \tfrac{b-a}{2}\right]\right]$$

Das Geheimnis w muss aus dem kleineren Intervall $\left[\ -\frac{b-a}{2 \cdot 2^{\ell_c + \ell_\phi + 2}}, \frac{a+b}{2 \cdot 2^{\ell_c + \ell_\phi + 2}}\ \right]$ gewählt werden.

[13] Der Prover kennt nicht die Faktorisierung des RSA-Modulus.

2.6.2 Binärzerlegung

Der Beweis $\mathsf{PK}[(w) : w \in [a,b]]$ entspricht dem „oder“-Beweis [38]:

$$\mathsf{PK}[(w) : w = a \vee w = a+1 \vee \ldots \vee w = b] \tag{2.3}$$

Es sei $w = w_{\ell_w-1}2^{\ell_w-1} + \ldots + w_1 2^1 + w_0$ die Binärzerlegung des Geheimnisses w und $C_i = g^{w_i}h^{r_i}$ die Commitments der einzelnen Bits mit Zufallskomponenten r_i für $i = 0, \ldots, \ell_w - 1$. Aus dem Commitment $C = g^w h^r$ und $C' = \prod_{i=0}^{\ell_w-1} C_i^{2^i}$ folgt $CC'^{-1} \equiv h^{r-r'}$. Während die Commitments C_i und C nur der Prover berechnen kann und vor Ausführung des Zero-Knowledge Proofs an den Verifier sendet, erfolgt die Berechnung von C' auch durch den Verifier. Dadurch kann der Verifier die Beziehung zwischen Bits und Geheimnis sicherstellen.

Mit der Binärzerlegung $a = a_{\ell_w-1}2^{\ell_w-1}+\ldots+a_1 2^1+a_0$ und $b = b_{\ell_w-1}2^{\ell_w-1}+\ldots+ b_1 2^1 + b_0$ kann ein Prover einen Verifier davon überzeugen, dass das Geheimnis w im Intervall $[a,b]$ liegt.

$$\mathsf{PK}\left[\begin{array}{l}\Big(w, r, w_0, \ldots, w_{\ell_w-1}, r_0, \ldots, r_{\ell_w-1}, (r-r')\Big) : \\ \Big(C_{\ell_w-1}/g^{a_{\ell_w-1}} = h^{r_{\ell_w-1}} \ \wedge \ \ldots \ \wedge \ C_0/g^{a_0} = h^{r_0}\Big) \ \vee \\ \vee \ \ldots \ \vee \\ \Big(C_{\ell_w-1}/g^{b_{\ell_w-1}} = h^{r_{\ell_w-1}} \ \wedge \ \ldots \ \wedge \ C_0/g^{b_0} = h^{r_0}\Big) \ \vee \\ C = g^w h^r \ \wedge \ CC'^{-1} = h^{(r-r')}\end{array}\right]$$

Unter der Bedingung, dass $a = 0$ und $b = 2^{\ell_w} - 1$ ist, vereinfacht sich der ZPK zu:

$$\mathsf{PK}\left[\begin{array}{l}\Big(w, r, w_0, \ldots, w_{\ell_w-1}, r_0, \ldots, r_{\ell_w-1}, (r-r')\Big) : \\ (C_0 = h^{r_0} \ \vee \ C_0/g = h^{r_0}) \ \wedge \ \ldots \ \wedge \\ \Big(C_{\ell_w-1} = h^{r_{\ell_w-1}} \ \vee \ C_{\ell_w-1}/g = h^{r_{\ell_w-1}}\Big) \ \wedge \\ C = g^w h^r \ \wedge \ CC'^{-1} = h^{(r-r')}\end{array}\right]$$

Das Verfahren ist nur für kleine Intervalle effizient, da die Anzahl der Prädikate mit größer werdenden Intervallgrenzen exponentiell zunimmt.

2.6.3 Lipmaa-Verfahren

Die Berechnungskomplexität beim *Lipmaa-Verfahren* ist unabhängig von der Intervallgröße und eignet sich deswegen für größere Intervalle. Das Verfahren basiert auf dem Beweis, dass eine natürliche Zahl nicht negativ ist [109].

Lagrange hat 1770 bewiesen, dass sich jede natürliche Zahl als Summe von vier Quadraten darstellen lässt [86]. Es sei $w = w_0^2 + w_1^2 + w_2^2 + w_3^2$ die Zerlegung des

Geheimnisses w und ϕ_g der Homomorphismus aus Kapitel 2.5.4. Die Ordnung der Bildgruppe von ϕ_g ist dem Prover unbekannt[14].

Zuerst wird der Beweis $\mathsf{PK}[(w) : w \geq a]$ betrachtet.

$$\mathsf{PK}\left[\begin{array}{l}(w, r, (w_i, r_i, w_i^2, r_i')_{i=0}^3) : C = \phi_g(w, r) \wedge \\ \bigwedge_{i=0}^3 C_i = \phi_g(w_i, r_i) \ \wedge \ \bigwedge_{i=0}^3 C_i' = \phi_g(w_i^2, r_i') \wedge \\ w = w_0^2 + w_1^2 + w_2^2 + w_3^2\end{array}\right]$$

Daraus folgt aufgrund der multiplikativen Abhängigkeiten zwischen den Geheimnissen (siehe Kapitel 2.5.4):

$$\mathsf{PK}\left[\begin{array}{l}(r, (w_i, r_i, w_i^2, r_i', (r_i \cdot w_i))_{i=0}^3) : \\ C = \phi_g(w_0^2 + w_1^2 + w_2^2 + w_3^2, r) \ \wedge \ \bigwedge_{i=0}^3 C_i = \phi_g(w_i, r_i) \wedge \\ \bigwedge_{i=0}^3 C_i' = \phi_g(w_i^2, r_i') \ \wedge \ \bigwedge_{i=0}^3 C_i' = \phi_{C_i}(w_i, r_i' - r_i \cdot w_i)\end{array}\right]$$

Ein Intervallbeweis $w \in [a, b]$ lässt sich in die beiden Beweise $w_a = w - a \geq 0$ und $w_b = b - w \geq 0$ unterteilen:
Mit $C_i' = \phi_g(w_i^2, r_i') = \phi_{C_i}(w_i, r_i' - r_i \cdot w_i) \Leftrightarrow 1 = (C_i)^{w_i} \cdot g^{-w_i^2} \cdot h^{-r_i \cdot w_i}$ folgt:

$$\mathsf{PK}\left[\begin{array}{l}\left(r_a, r_b, (w_{a,i}, r_{a,i}, w_{a,i}^2, r_{a,i}', (r_{a,i} \cdot w_{a,i}), w_{b,i}, r_{b,i}, w_{b,i}^2, r_{b,i}', (r_{b,i} \cdot w_{b,i})_{i=0}^3\right) : \\ C \cdot g^{-a} = \phi_g(w_{a,0}^2 + w_{a,1}^2 + w_{a,2}^2 + w_{a,3}^2, r_a) \wedge \\ \bigwedge_{i=0}^3 C_i = \phi_g(w_{a,i}, r_{a,i}) \ \wedge \ \bigwedge_{i=0}^3 1 = (C_i)^{w_{a,i}} g^{-w_{a,i}^2} h^{-r_{a,i} \cdot w_{a,i}} \wedge \\ C \cdot g^{-b} = \phi_g(-w_{b,0}^2 - w_{b,1}^2 - w_{b,2}^2 - w_{b,3}^2, r_b) \wedge \\ \bigwedge_{i=0}^3 C_i = \phi_g(w_{b,i}, r_{b,i}) \ \wedge \ \bigwedge_{i=0}^3 1 = (C_i)^{w_{b,i}} g^{-w_{b,i}^2} h^{-r_{b,i} \cdot w_{b,i}}\end{array}\right]$$

Gilt für das Geheimnis $w \neq 4^d(8 \cdot d' + 7), d, d' \in \mathbb{N}_0$, lässt sich w in die Summe aus drei (anstatt vier) Quadraten zerlegen [141]. Der Intervallbeweis ist dadurch effizienter durchführbar.

[14] Ansonsten könnte die Ordnung zu w addiert werden, ohne dass der Proof fehlschlägt.

Sicherheit und Datenschutz in Crowdsourcing-Szenarien

3

In diesem Kapitel werden das „Ticket-Chain“ und „Ticket-Spender“ Protokoll vorgestellt, die die Sicherheits- und Datenschutzanforderungen in Kapitel 1.5 erfüllen.

3.1 Lösungsansätze

Zur Sicherstellung der Authentizität in Crowdsourcing-Szenarien ist ein trivialer Lösungsansatz die Verwendung eines *Public-Key Verfahrens* wie z. B. RSA. Jede DCU generiert einen privaten und einen öffentlichen Schlüssel, wobei der öffentliche Schlüssel dem Provider bekannt ist. Die DCU signiert die Messdaten mit ihrem privaten Schlüssel und sendet Signatur, eine eindeutige Kennung[15] und Messdaten zum Provider. Der Provider verifiziert die Signatur mit dem öffentlichen Schlüssel.
Die DCU wird immer auch identifiziert, so dass ihre Anonymität nicht gewährleistet ist. Verwenden stattdessen alle DCUs das gleiche Schlüsselpaar und signieren ihre Messdaten mit demselben privaten Schlüssel, kann der Provider einzelne DCUs nicht mehr identifizieren. Das hat dann den Nachteil, dass einzelne DCUs nicht mehr von der weiteren Übertragung ausgeschlossen werden können.

Mithilfe einer *TTP (Trusted Third Party)* sind die oben genannten Nachteile vermeidbar. Eine DCU hat ein „festes“ und ein „temporäres“ Schlüsselpaar. Jedes Paar besteht aus einem privaten und einem öffentlichen Schlüssel. Die öffentlichen Schlüssel sind der TTP bekannt.

[15] Der Provider muss den passenden öffentlichen Schlüssel ermitteln.

Die DCU verwendet ihren privaten festen Schlüssel, um sich bei der TTP zu authentifizieren. Der zugehörige öffentliche Schlüssel identifiziert eine DCU eindeutig.
Das temporäre Schlüsselpaar wird vor jeder Übertragung von Messdaten zum Provider neu erzeugt. Dazu authentifiziert sich die DCU bei der TTP und übermittelt den öffentlichen temporären Schlüssel. Die TTP überprüft, ob die DCU berechtigt ist, Messdaten an den Provider zu senden. Falls das der Fall ist, erstellt die TTP ein Zertifikat für den öffentlichen temporären Schlüssel. Daraufhin kann die DCU Messdaten mit ihrem privaten temporären Schlüssel signieren und zusammen mit dem öffentlichen temporären Schlüssel und dem Zertifikat an den Provider senden. Der Provider überprüft anhand des Zertifikates die Integrität des öffentlichen Schlüssels und akzeptiert bei einem gültigen Zertifikat die Messdaten der DCU.

Der obige Ansatz trennt zwischen der *Rechtevergabe* und der *Rechteanwendung*. Eine DCU authentifiziert sich bei der TTP und bekommt nach erfolgreicher Authentifizierung „Rechte“ (z. B. das Recht, Messdaten an den Provider zu übermitteln) zugewiesen. Die Anwendung dieser Rechte erfolgt dann beim Provider. Dieses Konzept findet sich in der IT-Sicherheit in einer Vielzahl von Protokollen wieder.
Kerberos z. B. realisiert einen Authentifizierungsdienst, der eine Single Sign-on Funktionalität bereitstellt [157]. Die TTP ist ein Kerberos Server, bestehend aus dem Kerberos Authentication Server und dem Ticket Granting Service. Der Server führt die Authentifizierung eines Benutzers (oder Clients) durch und stellt einen Sitzungsschlüssel aus. Dieser Schlüssel kann danach (ohne sich ein weiteres Mal beim Kerberos Server authentifizieren zu müssen) vom Benutzer / Client verwendet werden, um Ressourcen (Server, Drucker etc.) nutzen zu können.
Ein ähnliches Konzept beschreibt die *WS-Trust Spezifikation* [163]. Die Spezifikation erweitert den WS-Security Standard, um Vertrauensbeziehungen zwischen Webdiensten zu realisieren. Auch hier wird eine TTP (der Security Token Service) verwendet, um die Rechtevergabe von Webdiensten zu steuern. Der „Transport dieser Rechte“ erfolgt dann in sogenannten *Security Tokens*, die vergleichbar mit Sitzungsschlüsseln in Kerberos sind.

Die Trennung zwischen Rechtevergabe und Rechteanwendung mithilfe einer TTP ist nachteilig. Erstens ist der Betrieb einer TTP mit zusätzlichen Kosten verbunden und zweitens werden hohe Anforderungen an die Verfügbarkeit, den Datenschutz und die Sicherheit gestellt. So lassen sich z. B. DCUs identifizieren, falls Provider und TTP kollaborieren.

3.2 Anonymous Credential System

Der Begriff *ACS (Anonymous Credential System)* wird 1985 von David Chaum eingeführt [41] und beschreibt ein kryptographisches Verfahren zur Authentifizierung eines Benutzers bei gleichzeitiger Sicherstellung seiner Anonymität [111].
Die Kernfunktionalität eines ACS besteht darin, dass eine Organisation signierte Berechtigungsnachweise für Benutzer ausstellt, wodurch diese - ohne ihre Identität preiszugeben - einer dritten Partei beweisen können Teil der Organisation zu sein. Moderne ACS sind nicht auf eine TTP angewiesen.

ACS ähneln Gruppensignaturverfahren, bieten allerdings keine Möglichkeit, die Identität des Benutzers im Ausnahmefall offenzulegen [42]. Dadurch gewährleisten ACS Benutzern einen besseren Schutz ihrer Privatsphäre.

3.3 CL-Signaturverfahren

Camenisch und Lysyanskaya (CL) haben ein effizientes Signaturverfahren vorgestellt, um in einem Commitment gespeicherte Geheimnisse zu signieren, ohne dass die signierende Partei die Geheimnisse kennen muss [33].
Im Gegensatz zu Signaturverfahren wie z. B. RSA, das sich um die Erstellung von *blinden* Signaturen erweitern lässt [73], hat das CL-Verfahren den Vorteil, dass die Gültigkeit der Signatur in Zero-Knowledge gezeigt werden kann. Die Geheimnisse und die Signatur müssen dazu nicht offengelegt werden. Das CL-Verfahren stellt somit ein einfaches ACS dar.

Bei den in dieser Arbeit vorgestellten Protokollen dient das CL-Verfahren als Grundlage. Die Beschreibung des Verfahrens weicht von der Veröffentlichung von Camenisch und Lysyanskaya ab, da hier zusätzliche Optimierungen von Brickell et al. [24] und Camenisch/Groth [31] berücksichtigt werden.

Ausgangspunkt sind die drei Parteien Benutzer $\mathcal{U}$, Signierer $\mathcal{S}$ und Verifier $\widehat{\mathcal{V}}$. Das CL-Verfahren lässt sich in die drei Abschnitte Setup, Signierung und Verifizierung unterteilen.

3.3.1 Setup

Die Anzahl der zu signierenden Nachrichten sei N. Der Signierer $\mathcal{S}$ wählt einen RSA-Modulus $n = p \cdot q$ der Länge ℓ_n mit zwei sicheren Primzahlen $p = 2p' + 1$ und $q = 2q' + 1$ sowie $\mathbf{R}_0, \ldots, \mathbf{R}_{N-1}, \mathbf{S}, \mathbf{Z} \in_R \mathrm{QR}_n$. Der öffentliche Schlüssel ist $(n, \mathbf{R}_0, \ldots, \mathbf{R}_{N-1}, \mathbf{S}, \mathbf{Z})$ und der geheime Schlüssel ist (p, q).

Wie später in Kapitel 3.3.4 gezeigt wird, ist es für die Sicherheit erforderlich, dass alle Gruppenelemente aus QR_n sind. Das ist allerdings ohne Kenntnis der Primfaktorzerlegung von n nicht möglich zu überprüfen[16].
Bei einem Zero-Knowledge Proof kann ein Prover einen Verifier mithilfe des Protokolls von Fiat und Shamir davon überzeugen, dass es sich bei den Gruppenelementen um quadratische Reste handelt [63]. Eine effizientere Möglichkeit hat Camenisch vorgestellt [30]. Der Prover führt anstatt des Beweises $\mathsf{PK}[(w) : x = \mathbf{g}^w]$, den Beweis $\mathsf{PK}[(w) : (x)^2 = (\mathbf{g}^2)^w]$ aus. Im Folgenden wird das durch die Notation $\mathsf{PK}[(w) : x = \pm\mathbf{g}^w]$ ausgedrückt.

3.3.2 Signierung

Das Protokoll zwischen einem Benutzer $\mathcal{U}$ und einem Signierer $\mathcal{S}$ zur Signierung des Nachrichtentupels $(m_0, \ldots, m_{N-1})$ für $m_i \in \pm\{0,1\}^{\ell_m}$ und $i = 0, \ldots, N-1$ zeigt Abbildung 3.1. Die Nachrichten werden in zwei Gruppen unterteilt. Die Nachrichten $(m_0, \ldots, m_{N'-1})$ für $N' \in \{0, \ldots, N-1\}$ sind dem Signierer $\mathcal{S}$ nicht bekannt, die Nachrichten $(m_{N'}, \ldots, m_{N-1})$ dagegen schon.

Das Protokoll umfasst sechs Schritte:

1. Benutzer $\mathcal{U}$ erstellt für die geheimen Nachrichten $(m_0, \ldots, m_{N'-1})$ ein Damgård-Fujisaki-Commitment **U**. Abweichend zur Veröffentlichung von Camenisch und Lysyanskaya gilt für den Wertebereich der Nachrichten $m_i \in \pm\{0,1\}^{\ell_m}$ anstatt $m_i \in \{0,1\}^{\ell_m}$ [31].

2. Commitment **U** wird zum Signierer übertragen.

3. Benutzer $\mathcal{U}$ zeigt in Zero-Knowledge und nicht-interaktiv, dass das Commitment **U** korrekt erzeugt wurde und $\mathcal{U}$ die Geheimnisse im Commitment kennt. Zur Überprüfung der Nachrichten, dass sie aus ihrem korrekten Intervall $m_i \in \pm\{0,1\}^{\ell_m}$ gewählt werden, zeigt $\mathcal{U}$, dass $m_i \in \pm\{0,1\}^{\tilde{\ell}_m}$ mit $\tilde{\ell}_m = \ell_m + \ell_\phi + \ell_c + 2$ für $i = 0, \ldots, N' - 1$ gilt (siehe Kapitel 2.6.1). Der Parameter ℓ_ϕ ist ein Sicherheitsparameter und Hashwerte haben die Bitlänge ℓ_c.

4. Signierer $\mathcal{S}$ signiert sowohl die geheimen Nachrichten $(m_0, \ldots, m_{N'-1})$ als auch die bekannten Nachrichten $(m_{N'}, \ldots, m_{N-1})$. Dazu wählt $\mathcal{S}$ eine Primzahl z, eine Zufallszahl v_2 der Länge $\ell_v = \ell_n + \tilde{\ell}_m + \ell_\phi$ und berechnet **A**.
   Damit die Sicherheit des CL-Verfahrens garantiert werden kann, muss

[16] Ein Ansatz ist die Berechnung des Jacobi-Symbols $\left(\frac{\mathbf{R}}{n}\right)$ in der Zeit $\mathcal{O}((\log(\mathbf{R}))(\log(n)))$ [45]. Falls das Jacobi-Symbol gleich -1 ist, gilt $\mathbf{R} \notin QR_n$. In allen anderen Fällen ist das Jacobi-Symbol 1 (mit $n \nmid \mathbf{R}$) und $\mathbf{R}$ *kann* ein quadratischer Rest sein. Für die Überprüfung $\mathbf{R} \in QR_n$ muss allerdings das *Quadratische-Reste-Problem* gelöst werden, was ohne Kenntnis der Primfaktorzerlegung von n nicht effizient machbar ist [17].

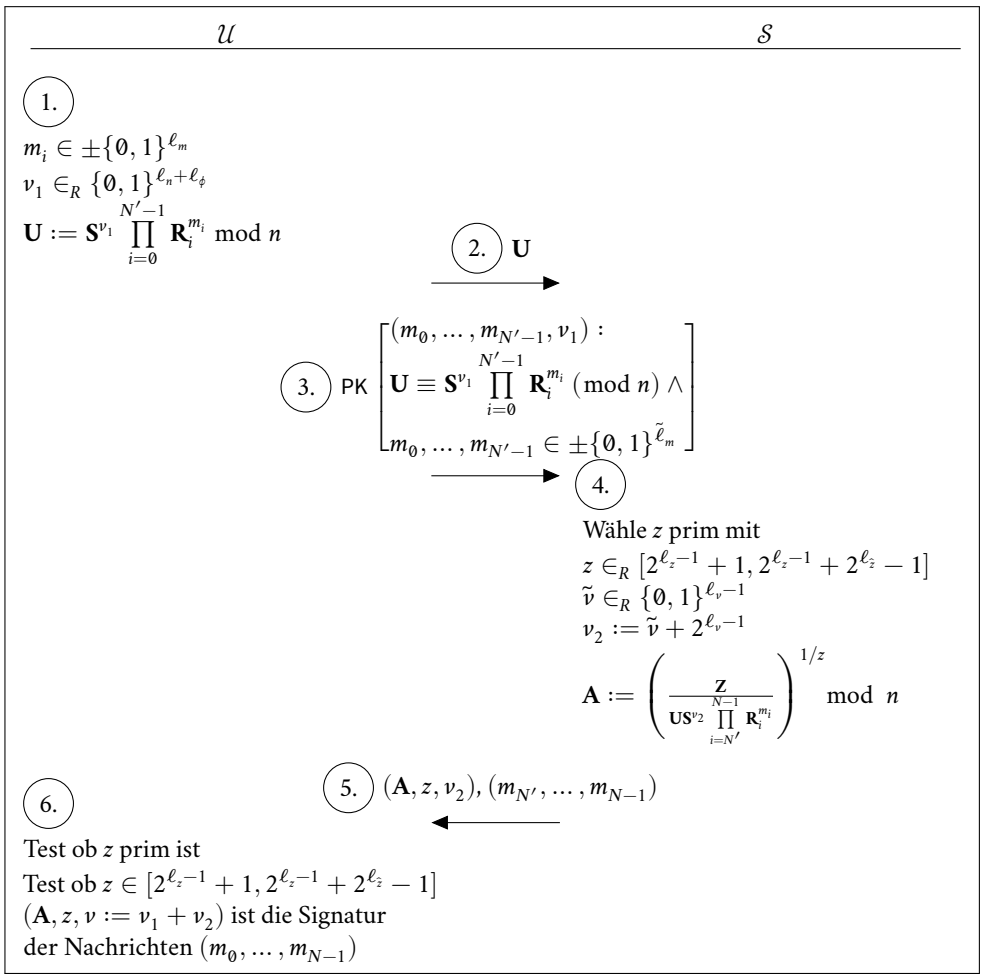


Abbildung 3.1: CL-Protokoll zur Signierung geheimer Nachrichten.

z eine Bitlänge von ℓ_z mit $\ell_z > \tilde{\ell}_m + 2$ haben[17]. In der Veröffentlichung von Camenisch und Lysyanskaya wird deswegen gefordert, dass $z \in_R [2^{\ell_z - 1} + 1, 2^{\ell_z} - 1]$ gilt. Soll später in Zero-Knowledge die Gültigkeit der Signatur gezeigt werden, sind komplexe Intervall-Proofs notwendig, um diese Bedingung zu zeigen. Wenn z stattdessen für $\ell_{\hat{z}} < \ell_z - \ell_\phi - \ell_c - 3$ aus dem Intervall $z \in_R [2^{\ell_z - 1} + 1, 2^{\ell_z - 1} + 2^{\ell_{\hat{z}}} - 1]$ gewählt wird, lässt sich die Intervall-Bedingung wesentlich effizienter zeigen (siehe Kapitel 3.3.3).

5. Benutzer $\mathcal{U}$ erhält die Teilsignatur $(\mathbf{A}, z, v_2)$ und die von $\mathcal{S}$ gewählten Nachrichten.

6. Im letzten Schritt berechnet Benutzer $\mathcal{U}$ die vollständige Signatur $(\mathbf{A}, z, v)$ der Nachrichten $(m_0, \ldots, m_{N-1})$ und überprüft ihre Korrektheit. Es muss gelten:

   a) $\mathbf{Z} \equiv \mathbf{A}^z \mathbf{R}_0^{m_0} \ldots \mathbf{R}_{N-1}^{m_{N-1}} \mathbf{S}^v \pmod{n}$

   b) $m_i \in \pm\{0,1\}^{\ell_m}$

   c) $z \in [2^{\ell_z - 1} + 1, 2^{\ell_z - 1} + 2^{\ell_{\hat{z}}} - 1]$

3.3.3 Verifizierung

Benutzer $\mathcal{U}$ kann den Besitz einer gültigen CL-Signatur für die Nachrichten $(m_0, \ldots, m_{N-1})$ in Zero-Knowledge zeigen, ohne diese offenlegen zu müssen.
Um die Bedingung $\mathbf{Z} \equiv \mathbf{A}^z \mathbf{R}_0^{m_0} \ldots \mathbf{R}_{N-1}^{m_{N-1}} \mathbf{S}^v \pmod{n}$ in Zero-Knowledge zu beweisen, können die in Kapitel 2 beschriebenen Techniken allerdings nicht direkt angewendet werden.
Die Basis $\mathbf{A}$ ist Teil der Signatur. Wird $\mathbf{A}$ veröffentlicht und beweist $\mathcal{U}$ mehrmals den Besitz einer gültigen Signatur, können die Beweise anhand von $\mathbf{A}$ verknüpft werden. Zur Verhinderung wird $\mathbf{A}$ verschleiert. Wenn $(\mathbf{A}, z, v)$ eine gültige Signatur ist, dann lässt sich zeigen, dass auch $(\hat{\mathbf{A}} := \mathbf{A}\mathbf{S}^{-\hat{r}} \mod n, z, \hat{v} := v + z\hat{r})$ für jedes beliebige $\hat{r} \in_R \{0,1\}^{\ell_n + \ell_\phi}$ eine gültige Signatur ist. Aus $\mathbf{A} \in \langle \mathbf{S} \rangle$ folgt, dass $\hat{\mathbf{A}}$ statistisch gleichverteilt über $\mathbb{Z}_n^*$ ist und daher offengelegt werden kann, ohne dass ein Verifier Rückschlüsse auf die Signatur ziehen kann [30].

Mit einem impliziten Intervall-Proof (siehe Kapitel 2.6.1) lässt sich in Zero-Knowledge zeigen, dass die Nachrichten $m_0, \ldots, m_{N-1}$ aus dem Intervall $\pm\{0,1\}^{\tilde{\ell}_m}$ gewählt werden.
Die Mindestlänge von z ist ähnlich beweisbar. Ohne Verletzung der Sicherheitseigenschaften wird der Wertebereich von z von $[2^{\ell_z - 1} + 1, 2^{\ell_z} - 1]$ geändert auf $z \in [2^{\ell_z - 1} - 2^{\ell_{\hat{z}}} + 1, 2^{\ell_z - 1} + 2^{\ell_{\hat{z}}} - 1]$. Der Term $2^{\ell_z - 1}$ lässt sich aus dem Intervall herausziehen, so dass $\hat{z} = z - 2^{\ell_z - 1} \in \pm\{0,1\}^{\ell_{\hat{z}}}$ gilt. Wie bei dem Intervall-

[17] Es gilt $\ell_z > \tilde{\ell}_m + 1$ mit $m \in \{0,1\}^{\tilde{\ell}_m}$. Aus $m \in \pm\{0,1\}^{\tilde{\ell}_m}$ folgt $\ell_z > \tilde{\ell}_m + 2$ [31].

Proof der Nachrichten, muss auch hier das Intervall für den Zero-Knowledge Proof auf $\tilde{\ell}_{\hat{z}} = \ell_{\hat{z}} + \ell_{\phi} + \ell_c + 2$ vergrößert werden.

Nach der Übertragung von $\hat{\mathbf{A}} := \mathbf{A}\mathbf{S}^{-\hat{r}} \bmod n$ an den Verifier kann mit dem folgenden PK der Besitz einer gültigen Signatur bewiesen werden.

$$\mathsf{PK}\left[\begin{array}{l}(\hat{z}, \hat{v}, m_0, \ldots, m_{N-1}) : \\ \mathbf{Z}\hat{\mathbf{A}}^{-2^{\ell_z-1}} \equiv \pm\mathbf{R}_0^{m_0} \cdot \ldots \cdot \mathbf{R}_{N-1}^{m_{N-1}} \hat{\mathbf{A}}^{\hat{z}} \mathbf{S}^{\hat{v}} \pmod n \wedge \\ m_i \in \pm\{0,1\}^{\tilde{\ell}_m} \text{ für } i \in \{0, \ldots, N\} \wedge \hat{z} \in \pm\{0,1\}^{\tilde{\ell}_{\hat{z}}}\end{array}\right]$$

Möchte der Prover die Nachrichten $m_{N'}, \ldots, m_{N-1}$ offenlegen, kann der Zero-Knowledge Proof vereinfacht werden.

$$\mathsf{PK}\left[\begin{array}{l}(\hat{z}, \hat{v}, m_0, \ldots, m_{N'-1}) : \\ \mathbf{Z}\hat{\mathbf{A}}^{-2^{\ell_z-1}} \mathbf{R}_{N'}^{-m_{N'}} \cdot \ldots \cdot \mathbf{R}_{N-1}^{-m_{N-1}} \equiv \\ \pm\mathbf{R}_0^{m_0} \cdot \ldots \cdot \mathbf{R}_{N'-1}^{m_{N'-1}} \hat{\mathbf{A}}^{\hat{z}} \mathbf{S}^{\hat{v}} \pmod n \wedge \\ m_i \in \pm\{0,1\}^{\tilde{\ell}_m} \text{ für } i \in \{0, \ldots, N'-1\} \wedge \hat{z} \in \pm\{0,1\}^{\tilde{\ell}_{\hat{z}}}\end{array}\right]$$

3.3.4 Sicherheitsanalyse

Camenisch und Lysyanskaya haben bewiesen, dass das Verfahren unter der starken RSA-Annahme sicher ist [33]. Für einen Angreifer mit Zugriff auf ein Signaturorakel ist es nicht praktikabel, eine gültige Signatur für eine beliebige Nachricht zu erzeugen [74]. Nur der Signierer kann mit Kenntnis der Primfaktorzerlegung von n eine gültige Signatur $(\mathbf{A}, z, v)$ erstellen.

Zeigt der Benutzer $\mathcal{U}$ mehrmals in Zero-Knowledge den Besitz einer Signatur, dann ist die Nicht-Verknüpfbarkeit zwischen den Beweisdurchläufen nur gewährleistet, wenn alle Operationen in derselben Untergruppe ausgeführt werden. Ein bösartiger Signierer könnte ansonsten in Protokollschritt 4 in Abbildung 3.1 $\mathbf{A} = g\left(\frac{\mathbf{Z}}{\mathbf{U}\ldots\mathbf{S}^{v_2}}\right)^{1/z}$ mit $g^z = 1$ und $g \notin \langle \mathbf{S} \rangle$ wählen. Bei der Verifizierung der Signatur erzeugt $\mathcal{U}$ das Commitment $\hat{\mathbf{A}} := \mathbf{A}\mathbf{S}^{-\hat{r}} \bmod n$, welches an den Verifier übertragen wird. Wenn nun $g \notin \langle \mathbf{S} \rangle$ gilt, dann gilt in der Regel auch $\hat{\mathbf{A}} \notin \langle \mathbf{S} \rangle$. Ist der Verifier gleichzeitig der Signierer $\mathcal{S}$, kann $\mathcal{S}$ prüfen, ob $g \notin \langle \mathbf{S} \rangle$ erfüllt ist und dadurch $\mathcal{U}$ identifizieren [24].

Camenisch und Lysyanskaya verhindern den Angriff, indem sie fordern, dass der Signierer zeigt, dass $n = p \cdot q$ das Produkt zweier sicherer Primzahlen $p = 2p' + 1$ und $q = 2q' + 1$ ist[18]. Dieser Zusammenhang lässt sich z. B. mit dem Verfahren

[18] Durch die Quadrierung der Basen in den Zero-Knowledge Proofs wird sichergestellt, dass alle Elemente aus QR_n sind (siehe Kapitel 3.3.1) und mit $n = p \cdot q$ die Gruppenordnung $p' \cdot q'$ ist. Aus Satz 2.1.1 sowie der Forderung $\ell_z < \log_2(\min(p', q'))$ folgt, dass ein Angreifer kein Element g mit $g^z = 1$ und $g \notin \langle \mathbf{S} \rangle$ finden kann.

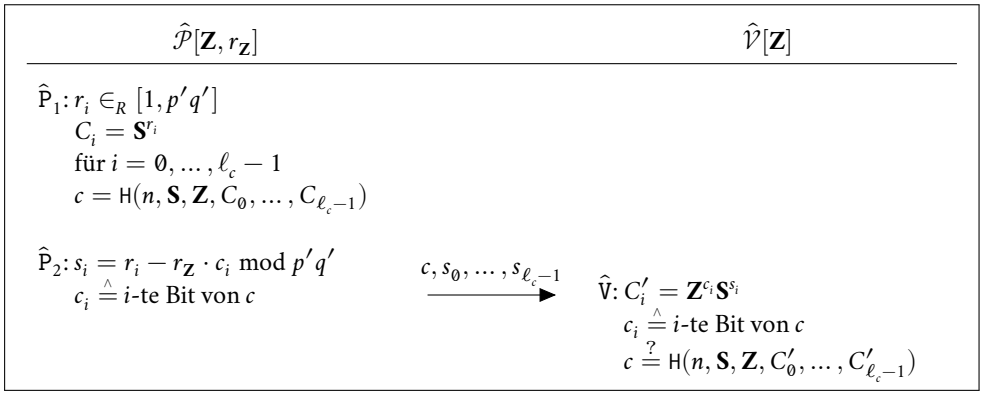


Abbildung 3.2: Nicht-interaktiver Zero-Knowledge Proof PK$\left[(r_{\mathbf{Z}}) : \mathbf{Z} \equiv \mathbf{S}^{r_{\mathbf{Z}}}\right]$ mit binären Challenges.

von Camenisch und Michels in Zero-Knowledge und ohne Offenlegung von p und q zeigen [35].
Das Verfahren hat den Nachteil, dass es ineffizient ist. Brickell et al. haben gezeigt, dass es ausreichend ist zu zeigen, dass sich alle Elemente in derselben Untergruppe befinden [24]. Dazu wählt der Signierer in der Setup-Phase $r_{\mathbf{Z}}, r_{\mathbf{R}_0}, \ldots, r_{\mathbf{R}_{N-1}}, \in [1, p'q']$ und berechnet mit $\mathbf{S} \in_R \mathrm{QR}_n$:

$$\mathbf{Z} = \mathbf{S}^{r_{\mathbf{Z}}} \bmod n \text{ und } \mathbf{R}_0 = \mathbf{S}^{r_{\mathbf{R}_0}} \bmod n, \ldots, \mathbf{R}_{N-1} = \mathbf{S}^{r_{\mathbf{R}_{N-1}}} \bmod n$$

Danach zeigt der Signierer allen Parteien, dass die Elemente richtig erzeugt wurden.

$$\mathrm{PK}\left[\begin{array}{l}(r_{\mathbf{Z}}, r_{\mathbf{R}_0}, \ldots, r_{\mathbf{R}_{N-1}}) : \mathbf{Z} \equiv \mathbf{S}^{r_{\mathbf{Z}}} \pmod n \wedge \\ \mathbf{R}_0 \equiv \mathbf{S}^{r_{\mathbf{R}_0}} \pmod n \wedge \ldots \wedge \mathbf{R}_{N-1} \equiv \mathbf{S}^{r_{\mathbf{R}_{N-1}}} \pmod n\end{array}\right]$$

Da der Signierer die Basis frei wählen kann, sind binäre Challenges notwendig, die den Proof ineffizient (aber effizienter als bei Camenisch/Michels) machen. Da der Proof allerdings nur einmal ausgeführt werden muss, kann das vernachlässigt werden. Abbildung 3.2 zeigt das Protokoll mit binären Challenges.
Zusätzlich muss der Signierer in Schritt 5 den folgenden Proof of Knowledge mit ausführen, um zu zeigen $1/z$ zu kennen [24].

$$\mathrm{PK}\left[(1/z) : \mathbf{A} \equiv \pm\left(\frac{\mathbf{Z}}{\mathbf{US}^{v_2}\prod_{i=N'}^{N-1}\mathbf{R}_i^{m_i}}\right)^{1/z} \pmod n\right]$$

3.4 „Ticket-Chain“ Protokoll

Das Protokoll basiert auf der Idee von Gruppensignaturverfahren, ist aber kein solches. Jede DCU ist ein Mitglied in einer Gruppe von DCUs. Der Provider

ist der Gruppenmanager. Damit kann er DCUs zur Gruppe hinzufügen bzw. entfernen. DCUs können Daten als Gruppenmitglieder signieren, so dass der Provider später anhand der Signatur überprüfen kann, ob die DCU zur Gruppe gehört, d. h. berechtigt ist, Daten an ihn zu senden [92] [90].
Einzelne DCUs bleiben anonym. Selbst wenn eine DCU zwei signierte Nachrichten an den Provider sendet, ist es für eine Partei nicht praktikabel herauszufinden, dass beide Signaturen von der gleichen DCU stammen (vgl. Kapitel 3.4.4).

Die Daten, die eine DCU an den Provider sendet, sind an ein sogenanntes *Ticket* gekoppelt. Ein gültiges Ticket ist der Nachweis, zur Gruppe valider DCUs zu gehören und kann nur einmal während eines festen Zeitintervalls verwendet werden. Parallel zur Datenübermittlung handelt die DCU mit dem Provider ein neues Ticket aus, dass für die nächste Transaktion verwendet wird. Aus diesem Grund wird das Protokoll hier als *„Ticket-Chain“ Protokoll* bezeichnet.

Das „Ticket-Chain“ Protokoll wird zwischen einer DCU $\mathcal{D}$ und einem Provider $\mathcal{P}$ ausgeführt. Es wird davon ausgegangen, dass die Kommunikation verschlüsselt (z. B. über TLS) erfolgt. Das Protokoll lässt sich in die drei Teile Setup, Registrierung und Reporting unterteilen.

3.4.1 Setup

Jede DCU $\mathcal{D}$ generiert zwei Schlüssel, die Teil eines Standard-Signaturverfahrens wie z. B. RSA oder ECDSA sind [117]. Der Verifizierungsschlüssel $\tilde{K}_{pub,\mathcal{D}}$ ist öffentlich, während der Signaturschlüssel $\tilde{K}_{sec,\mathcal{D}}$ geheim ist.
Die DCU sendet eine Kopie des öffentlichen Schlüssels an den Provider und authentifiziert und identifiziert sich mithilfe eines eindeutigen Authentifizierungsmerkmals. Im FCD-Szenario kann das z. B. die E-Mailadresse oder Telefonnummer des Benutzers[19] und im Smart Metering Szenario die Postanschrift des Kunden[20] sein.

Provider $\mathcal{P}$ generiert einen RSA-Modulus $n = p \cdot q$, der aus zwei sicheren Primzahlen $p = 2p' + 1$ und $q = 2q' + 1$ der Länge $\ell_n/2$ besteht. Des Weiteren erzeugt $\mathcal{P}$ $\mathbf{S} \in_R \mathrm{QR}_n$, wählt $r_{\mathbf{Z}}, r_{\mathbf{g}}, r_{\mathbf{h}}, r_{\mathbf{R}_0}, \ldots, r_{\mathbf{R}_3} \in_R [1, p'q']$ und berechnet für $i = 0, \ldots, 3$:

$$\mathbf{Z} = \mathbf{S}^{r_{\mathbf{Z}}} \bmod n, \quad \mathbf{g} = \mathbf{S}^{r_{\mathbf{g}}} \bmod n, \quad \mathbf{h} = \mathbf{S}^{r_{\mathbf{h}}} \bmod n, \quad \mathbf{R}_i = \mathbf{S}^{r_{\mathbf{R}_i}} \bmod n$$

[19] Der Benutzer registriert sich bei Google mit seiner E-Mailadresse bzw. Telefonnummer und Google sendet einen Aktivierungscode als „Challenge“ an die angegebene Adresse bzw. Nummer. Das Smartphone antwortet mit dem öffentlichen Schlüssel und dem Code und authentifiziert sich damit gegenüber Google.

[20] Wie im FCD-Szenario, nur sendet das EVU den Aktivierungscode per Post, und der Kunde trägt den Code im Smart Meter ein. Der Smart Meter überträgt dann öffentlichen Schlüssel und Code an das EVU.

Der Provider zeigt mit dem Protokoll in Abbildung 3.2 die korrekte Berechnung der Gruppenelemente.
Der öffentliche Schlüssel des Providers ist $(\tilde{K}_{pub,\mathcal{P}}, \mathbf{S}, \mathbf{Z}, \mathbf{g}, \mathbf{h}, \mathbf{R}_0, \mathbf{R}_1, \mathbf{R}_2, \mathbf{R}_3)$ und der geheime Schlüssel $(\tilde{K}_{sec,\mathcal{P}}, p, q)$.

3.4.2 Registrierung

Durch das *Registrierungsprotokoll* authentifiziert und identifiziert sich die DCU beim Provider. Damit hat der Provider die Möglichkeit, DCUs aus der Gruppe der DCUs zu entfernen. Ziel des Protokolls ist die Signierung eines geheimen Tickets w mithilfe des CL-Verfahrens (siehe Kapitel 3.3).
Bei der Ausführung des Registrierungsprotokolls erfolgt immer auch eine Identifizierung der DCU. Im Smart Metering Szenario bietet sich die Ausführung des Protokolls am Ende eines Abrechnungszeitraumes (z. B. am Ende eines Monats) an, da dann eine Identifizierung des Smart Meters notwendig ist.

Das Protokoll besteht aus fünf Schritten und ist in Abbildung 3.3 dargestellt.

1. Eine DCU generiert ein Ticket w bestehend aus einer Zufallszahl der Bitlänge ℓ_w. Das Ticket wird in einem Commitment $\mathbf{U}$ gespeichert. Die DCU signiert mit ihrem privaten Schlüssel $\tilde{K}_{sec,\mathcal{D}}$ das Commitment und die für die Registrierung notwendigen Daten $D_{\text{Registrierung}}$[21].

2. Commitment $\mathbf{U}$, für die Registrierung notwendige Daten $D_{\text{Registrierung}}$ und Signatur $\hat{S}$ werden an den Provider gesendet. Zusätzlich beweist die DCU die korrekte Erzeugung des Commitment in Zero-Knowledge und identifiziert sich gegenüber $\hat{\mathcal{P}}$ mithilfe des öffentlichen Schlüssels $\tilde{K}_{pub,\mathcal{D}}$.

3. Der Provider überprüft, ob die DCU bekannt ist und anonym Daten an den Provider senden darf. Ist das der Fall, erstellt der Provider eine Teilsignatur für das Ticket w und die beiden Zeitstempel $\overleftarrow{TS}$ und $\overrightarrow{TS}$. Das Ticket ist im Commitment versteckt, so dass der Provider den Wert nicht kennt. Der Zeitstempel $\overleftarrow{TS}$ gibt an, ab wann das Ticket w gültig ist. Der Zeitstempel $\overrightarrow{TS}$ legt die Ablaufzeit des Tickets fest.
   Der Zeitstempel $\overrightarrow{TS}$ wird zusammen mit dem öffentlichen Schlüssel $\tilde{K}_{pub,\mathcal{D}}$ in einer Datenbank gespeichert, denn die Teilsignatur wird vom Provider nur ausgestellt, wenn ein vorher ausgestelltes Ticket mit Ablaufzeit $\overrightarrow{TS}'$ seine Gültigkeit verloren hat. Es muss $\overrightarrow{TS}' < \overleftarrow{TS}$ gelten.

   Neben der Gültigkeit kann ein Ticket an einen Qualitätsparameter u gekoppelt werden, um ein (geringes) Maß an Verknüpfbarkeit zwischen Transaktionen zu realisieren (siehe Kapitel 3.4.3).

[21] Im Smart Metering Szenario entsprechen die Daten den akkumulierten Verbrauchswerten eines Abrechnungszeitraumes.

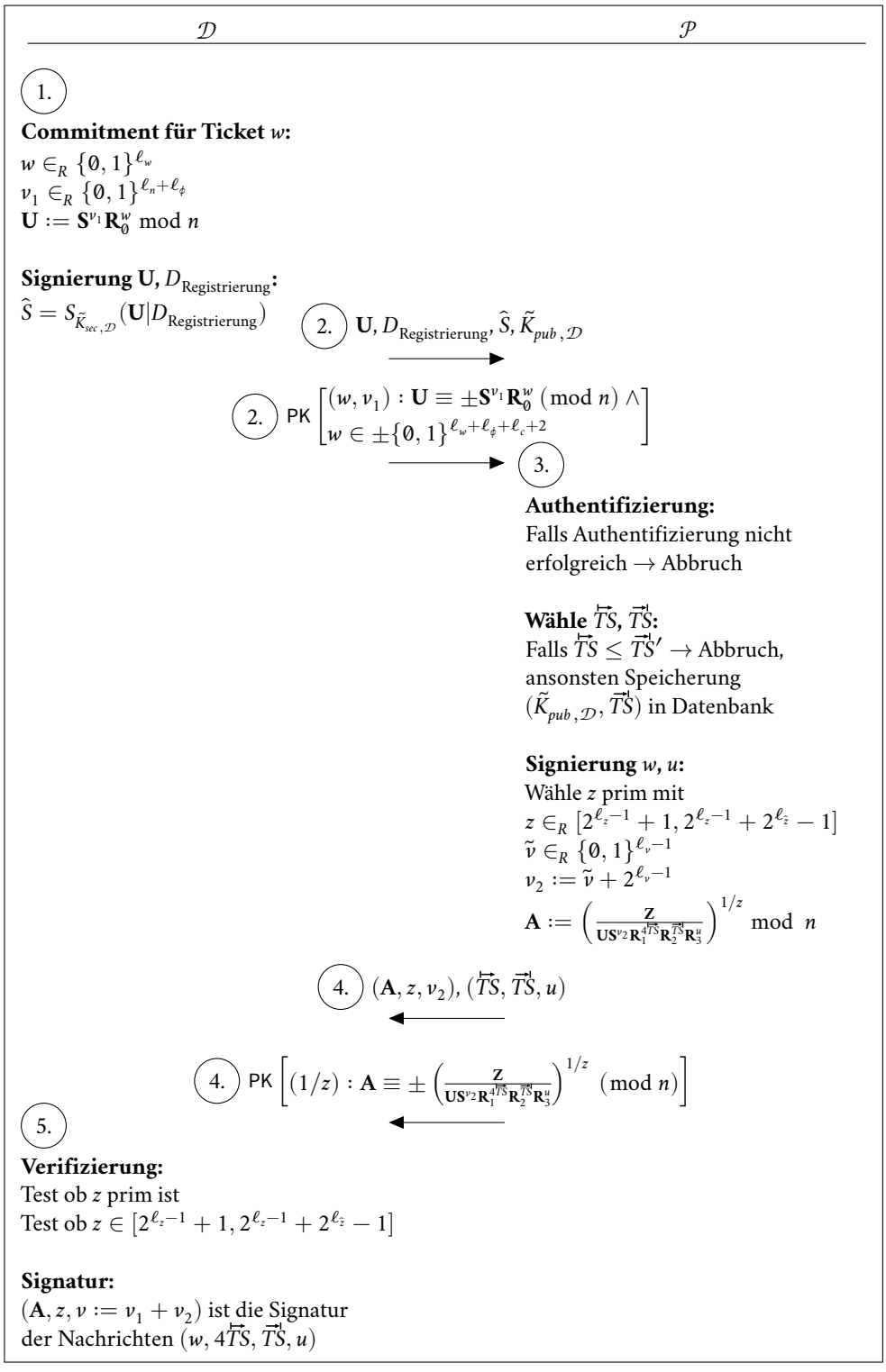


Abbildung 3.3: „Ticket-Chain“ Protokoll: Registrierung.

4. Die Teilsignatur und die beiden Zeitstempel werden zur DCU gesendet und der Provider zeigt in Zero-Knowledge und nicht-interaktiv, dass die Signierung korrekt durchgeführt wurde.

5. Die DCU erzeugt die vollständige CL-Signatur und überprüft ihre Richtigkeit. Die DCU besitzt nun ein Ticket, um sich gegenüber dem Provider zu authentifizieren und anonym Daten an ihn zu senden.

3.4.3 Reporting

Durch das *Reporting-Protokoll* sendet eine DCU $\mathcal{D}$ anonym Daten[22] $D_{\text{Reporting}}$ an den Provider. Trotz Anonymität kann der Provider überprüfen, ob $\mathcal{D}$ zur Gruppe der DCUs gehört. Es wird sichergestellt, dass nur DCUs erfolgreich Daten an den Provider senden können, die dazu autorisiert sind.

Technisch wird das durch einen Zero-Knowledge Proof erreicht. Die DCU beweist dem Provider, dass es ein vom Provider unterschriebenes Ticket besitzt. Sobald das Ticket eingelöst wird, kann es nicht noch einmal verwendet werden. Parallel dazu beantragt die DCU für die nächste Transaktion eine Signatur für ein neues Ticket w'.
Jedes Ticket ist in seiner Gültigkeit begrenzt und nur in einem vom Provider festgelegten Zeitintervall gültig. Durch Angabe des frühestmöglichen Zeitpunktes, zu dem das Ticket eingelöst werden darf, verhindert der Zeitstempel $\overrightarrow{TS}$, dass eine DCU zu schnell neue Daten an den Provider sendet. Zusätzlich ist die Ablaufzeit $\overrightarrow{TS}$ des Tickets begrenzt. Hat die DCU das Ticket nicht vor Ablauf der Ablaufzeit eingelöst, muss sie sich beim Provider durch das Registrierungsprotokolls erneut authentifizieren und identifizieren.

Abbildung 3.4 veranschaulicht die fünf Schritte des Reporting-Protokolls zwischen einer DCU $\mathcal{D}$ und einem Provider $\mathcal{P}$.

1. Eine DCU generiert ein neues Ticket w' und speichert es in dem Commitment $\mathbf{U}'$.

2. Das Commitment $\mathbf{U}'$ wird an den Provider gesendet und seine Korrektheit durch einen SPK bewiesen. Die DCU sendet zudem das Ticket w, die Ablaufzeit des Tickets $\overrightarrow{TS}$, den Qualitätsparameter u und die Daten $D_{\text{Reporting}}$ an den Provider.

   Der Provider berechnet aus dem Qualitätsparameter u und den Daten $D_{\text{Reporting}}$ mithilfe einer anwendungsabhängigen Funktion f einen neuen Parameter u'. Der Provider beurteilt dadurch die Qualität der Daten und koppelt dieses Ergebnis an das neue Ticket w'. Liefert eine DCU keine

[22] Bei den Daten handelt es sich im FCD-Szenario um Positions-, im Smart Metering Szenario um Verbrauchsdaten.

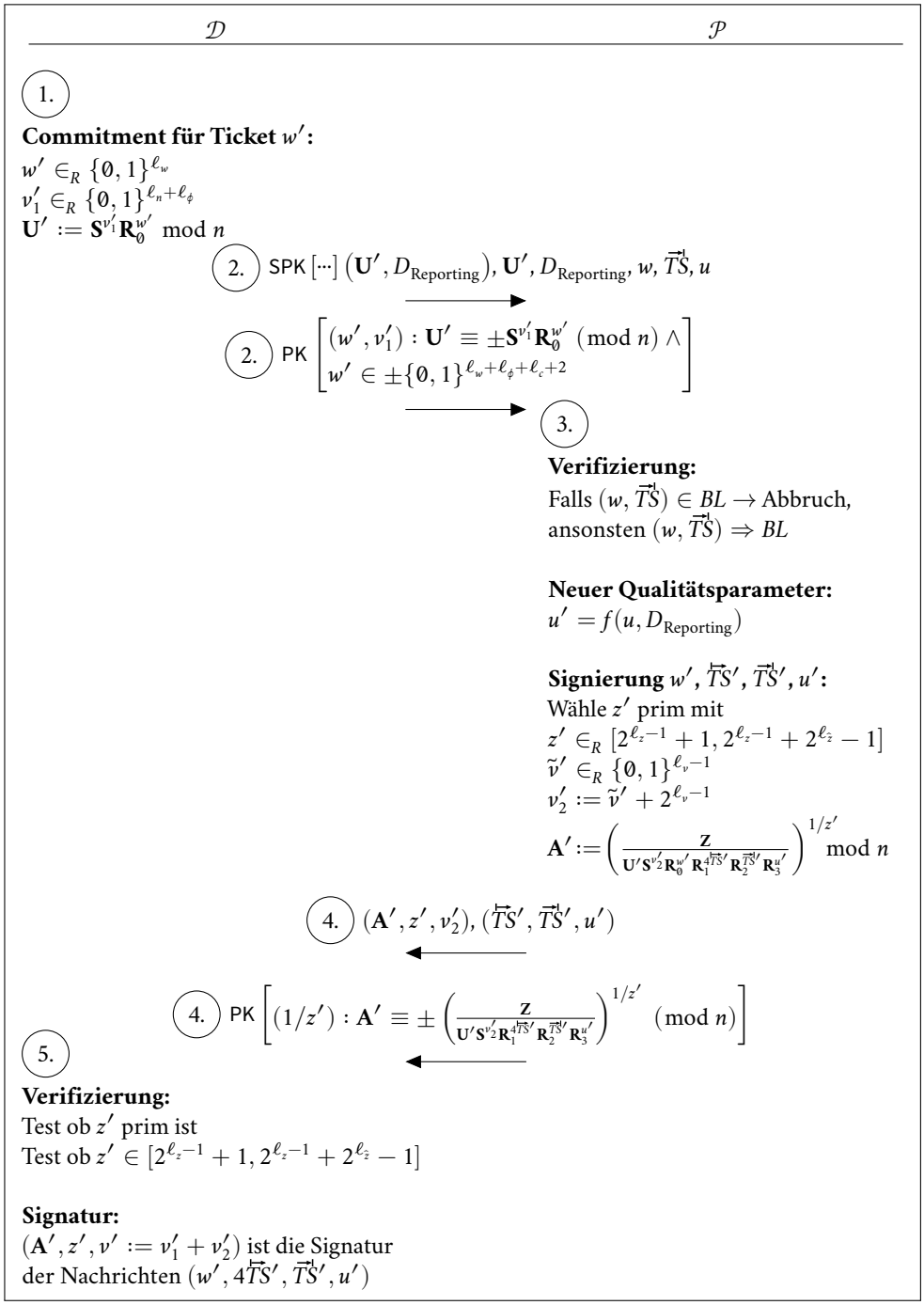


Abbildung 3.4: „Ticket-Chain“ Protokoll: Reporting.

plausiblen Daten, kann das bei zukünftigen Transaktionen berücksichtigt werden.

Die DCU authentifiziert sich gegenüber $\mathcal{P}$, in dem sie in Zero-Knowledge zeigt, dass sie eine gültige Signatur $(\mathbf{A}, z, v)$ für das Ticket w von $\mathcal{P}$ erhalten hat. Der Beweis wird als SPK ausgeführt, um Commitment $\mathbf{U}'$ und Daten $D_{\text{Reporting}}$ an die Signatur zu koppeln.

Zur Vermeidung, dass eine DCU den Provider mit Daten überschwemmt, wird die Gültigkeit jedes Tickets überprüft.
Es sei TS der Zeitstempel der zu übertragenden Daten, dann muss $TS - \overleftarrow{TS} \geq 0$ gelten. Dieser Zusammenhang wird durch einen Lipmaa-Intervall-Proof gezeigt. Nach Kapitel 2.6.3 kann der Proof effizienter ausgeführt werden, wenn $TS - \overleftarrow{TS} \neq 4^d(8 \cdot d' + 7)$ für $d, d' \in \mathbb{N}_0$ gilt. Die Bedingung ist erfüllt, falls $TS - \overleftarrow{TS}$ nicht durch vier teilbar ist, so dass anstatt $TS - \overleftarrow{TS} \geq 0$ die folgende Bedingung in Zero-Knowledge gezeigt wird:

$$\widehat{TS} = 4(TS - \overleftarrow{TS}) + 1 = \widehat{TS}_0^2 + \widehat{TS}_1^2 + \widehat{TS}_2^2 > 0$$

Der Zeitstempel $\overrightarrow{TS}$, der die Ablaufzeit des Tickets angibt, wird offen gelegt. Aus Kapitel 3.3.3 folgt mit $\widehat{\mathbf{A}} := \mathbf{A}\mathbf{S}^{-\widehat{r}} \pmod{n}$, $\widehat{v} = v + z\widehat{r}$ und $4\overleftarrow{TS} = 4TS + 1 - \widehat{TS}_0^2 - \widehat{TS}_1^2 - \widehat{TS}_2^2$:

$$\mathsf{SPK}\left[\begin{array}{l}(z, \widehat{v}, (\widehat{TS}_i, \widehat{TS}_i^2, r_{TS_i})_{i=0}^3) : \\ \mathbf{Z}\widehat{\mathbf{A}}^{-2^{\ell_z-1}}\mathbf{R}_0^{-w}\mathbf{R}_1^{-(4TS+1)}\mathbf{R}_2^{-\overrightarrow{TS}}\mathbf{R}_3^{-u} \equiv \\ \pm\mathbf{R}_1^{-\widehat{TS}_0^2-\widehat{TS}_1^2-\widehat{TS}_2^2}\widehat{\mathbf{A}}^z\mathbf{S}^{\widehat{v}} \pmod{n} \wedge \\ \bigwedge_{i=0}^{3} C_{\widehat{TS}_i} \equiv \mathbf{g}^{\widehat{TS}_i}\mathbf{h}^{r_{TS_i}} \pmod{n} \wedge \\ \bigwedge_{i=0}^{3} 1 \equiv \left(C_{\widehat{TS}_i}\right)^{\widehat{TS}_i}\mathbf{g}^{-\widehat{TS}_i^2}\mathbf{h}^{-(r_{TS_i}\cdot\widehat{TS}_i)} \pmod{n} \wedge \\ z \in \pm\{0,1\}^{\ell_z+\ell_\phi+\ell_c+2}\end{array}\right]\left(\mathbf{U}', D_{\text{Reporting}}\right)$$

3. Jedes eingelöste Ticket wird mit einer Blacklist BL abgeglichen. Die Blacklist enthält jedes eingelöste Ticket zusammen mit dem Zeitstempel $\overrightarrow{TS}$. Der Zeitstempel stellt sicher, dass die Größe der Blacklist nicht unbegrenzt wächst. Nicht mehr gültige Tickets können vom Provider entfernt werden. Befindet sich das Ticket w nicht auf der Blacklist und akzeptiert der Provider nach Überprüfung des SPK das Ticket, erzeugt der Provider eine Teilsignatur für ein neues Ticket w'.

4. Die Teilsignatur und die beiden Zeitstempel für das neue Ticket w' werden zur DCU gesendet. Das entspricht Schritt 4 im Registrierungsprotokoll.

5. Die DCU erzeugt die Signatur und überprüft ihre Korrektheit. Das entspricht Schritt 5 im Registrierungsprotokoll.

3.4.4 Sicherheitsanalyse

Das „Ticket-Chain“ Protokoll erfüllt die Sicherheitsanforderungen aus Kapitel 1.5.

1. *Authentifizierung und Integrität*
   Die Integrität der Messdaten wird im Registrierungsprotokoll durch die Signierung mit dem privaten Schlüssel $\tilde{K}_{sec,\mathcal{D}}$ und im Reporting-Protokoll durch den SPK sichergestellt. Durch die Signaturen authentifiziert sich eine DCU gleichzeitig gegenüber dem Provider.
   Damit sich ein Angreifer erfolgreich gegenüber dem Provider authentifizieren kann, muss er sich entweder mit einem Schlüsselpaar $(\tilde{K}_{pub,\mathcal{D}}, \tilde{K}_{sec,\mathcal{D}})$ beim Provider registrieren oder eine gültige CL-Signatur für ein Ticket w erstellen. Der Erfolg des ersten Angriffes hängt von dem verwendeten Authentifizierungsmerkmal[23] ab. Für den zweiten Angriff ist das Lösen des RSA-Problems nötig (siehe Kapitel 3.3.4).

2. *Widerrufbarkeit und Identifizierung*
   Bei jeder Ausführung des Registrierungsprotokolls wird die DCU identifiziert.
   Die Anforderung nach Widerrufbarkeit von DCUs wird durch den Zeitstempel $\vec{TS}$ erfüllt. Sobald ein Ticket abgelaufen ist, muss sich die DCU beim Provider identifizieren. Dadurch erhält der Provider die Möglichkeit, die DCU von der weiteren (erfolgreichen) Übermittlung von Daten auszuschließen.

3. *Anonymität und Nicht-Verknüpfbarkeit*
   Die Anonymität der DCU wird im Reporting-Protokoll durch die Zero-Knowledge Proofs sichergestellt. Eine Identifizierung ist nur im Registrierungsprotokoll möglich.
   Die Sicherheitseigenschaften des CL-Protokolls stellen sicher, dass ein Provider im Registrierungsprotokoll das Ticket w nicht lesen kann (siehe Kapitel 3.3.4). Die DCU kann Ticket und Zeitstempel $\vec{TS}$ im Reporting-Protokoll offenlegen, ohne dass der Provider eine Verknüpfung mit den aufgezeichneten Transaktionen im Registrierungsprotokoll und damit mit der Identität der DCU herstellen kann.
   Die Nicht-Verknüpfbarkeit mehrerer Ausführungen des Reporting-Protokolls wird durch die Zero-Knowledge Proofs sichergestellt. Damit eine Identifizierung durch den Provider verhindert wird, muss die Auflösung des Zeitstempels $\vec{TS}$ und des Qualitätsparameters u niedrig gewählt werden. In der Praxis ist die Angabe der Woche oder des Monats, an dem das Ticket abläuft, für $\vec{TS}$ ausreichend. Falls eine hohe Auflösung nötig ist, kann

[23] Im Smart Metering Szenario ist z. B. davon auszugehen, dass es für einen Angreifer nur unter sehr hohem Aufwand möglich ist, sich mit mehreren Postanschriften zu registrieren, da der Provider die Gültigkeit der Anschrift verifiziert.

z. B. die Gültigkeit des Zeitstempels $\overrightarrow{TS}$ durch einen Intervall-Proof gezeigt werden. Der Zero-Knowledge Proof in Schritt 2 des Reporting-Protokolls ließe sich zu diesem Zweck erweitern. Das hat allerdings einen negativen Einfluss auf die Effizienz des Protokolls, weshalb in dieser Arbeit darauf verzichtet wird.

4. *Robustheit*
   Eine DCU bleibt während der Ausführung des Reporting-Protokolls anonym. Damit eine kompromittierte DCU die Datenbasis des Providers nur begrenzt beeinflussen kann, gelten Messdaten nur für ein Zeitintervall *TS*. Neue Daten kann eine DCU erst nach Ablauf von $\overrightarrow{TS}$ an den Provider senden, so dass eine DCU den Provider nicht mit Daten überschwemmen kann (*Spamming-Angriff*). Die Blacklist unterbindet *Replay-Angriffe*, denn jedes Ticket kann nur einmal verwendet werden.
   Eine DCU könnte das Registrierungsprotokoll mehrmals ausführen und erhielte demzufolge mehrere Tickets. Damit wäre die DCU in der Lage, einen konzentrierten Angriff durchzuführen und alle Tickets auf einmal einzulösen. Das wird dadurch verhindert, dass neue Tickets im Registrierungsprotokoll erst gültig werden, wenn alle bisherigen Tickets ihre Gültigkeit verloren haben (siehe Schritt 3 im Registrierungsprotokoll).

Identifizierung einer DCU anhand ihrer Netzwerkadresse

Erfolgt die Ausführung des Registrierungs- und Reporting-Protokolls mit unterschiedlichen Absenderadressen, ist keine direkte Verknüpfung der Messdaten mit der Identität der DCU gegeben. Da das Reporting-Protokoll allerdings regelmäßig nach kurzen Zeitintervallen ausgeführt wird, ist eine Verknüpfung der Daten anhand gleicher Absenderadressen (z. B. IP-Adressen) im FCD- und Smart Metering Szenario möglich.

Im Idealfall kommunizieren DCUs über ein eigenes Netz mit dem Provider. Einige Smart Meter senden Daten über PLC (Powerline Communication) an das EVU bzw. den VNB [78]. Erfolgt die Kommunikation über Ethernet und TCP / IP, könnte der Smart Meter vor jeder Ausführung des Reporting-Protokolls die Hardwareadresse und IP-Adresse innerhalb eines Subnetzes zufällig wählen. Ein großer Adresspool würde sicherstellen, dass die Wahrscheinlichkeit vernachlässigbar ist, dass zwei Smart Meter dieselbe Adresse zur selben Zeit verwenden.

Bei der Kommunikation zwischen Smart Meter und VNB über GPRS [78] erfolgt die Kommunikation über einen Mobilfunkanbieter. Die zufällige Wahl der Absenderadresse ist in der Regel nicht möglich. Der Smart Meter kann jedoch die Mobilfunkverbindung vor jeder Übertragung neu aufbauen, so dass eine neue IP-Adresse zugewiesen wird.

Im FCD-Szenario ist ein Zurücksetzen der Verbindung nicht möglich, da die mobile Internetverbindung mit anderen Diensten geteilt wird. Das ergibt sich in der Regel auch bei Smart Metern, die über den DSL-Internetanschluss des Haushaltes mit dem EVU / VNB kommunizieren.
In Deutschland findet aber in der Regel alle 24 Stunden eine Zwangstrennung der Internetverbindung statt, so dass eine Verknüpfung der Daten nur eingeschränkt möglich ist [80].

Die Frage, nach wie vielen Ausführungen des Reporting-Protokolls mit derselben Absenderadresse Rückschlüsse auf den Benutzer bzw. Kunden gezogen werden können, wird in dieser Arbeit nicht diskutiert. Wenn ein hohes Maß an Datenschutz gefordert wird und für jede Übertragung eine neue Absenderadresse verwendet werden soll, bietet sich Tor [166] an. Dabei werden die Daten verschlüsselt über eine zufällige Route von Zwischenknoten gesendet.

3.5 „Ticket-Spender“ Protokoll

Das „Ticket-Chain“ Protokoll hat zwei Nachteile.

1. Der Intervall-Proof im Reporting-Protokoll, um die Gültigkeit eines Tickets zu zeigen, wirkt sich negativ auf die Ausführungszeit des Protokolls aus.

2. Die Ausführung des „Ticket-Chain“ Protokolls erfordert zusätzliche CPU Ressourcen. Im FCD-Szenario wirkt sich das z. B. negativ auf die Batterielaufzeit des Smartphones aus.

Das *„Ticket-Spender“ Protokoll* vermeidet diese Nachteile. Es ist eine modifizierte Variante des Protokolls von Camenisch et al. [28].
Beim „Ticket-Chain“ Protokoll besitzt eine DCU immer nur ein gültiges Ticket. Erst wenn die DCU das Ticket einlöst, bekommt sie die Signatur für ein weiteres Ticket.
Beim „Ticket-Spender“ Protokoll steht jeder DCU ein vom Provider signierter *Ticket-Spender* zur Verfügung. Durch den Ticket-Spender kann sich eine DCU selber gültige Tickets für bestimmte Zeitintervalle ausstellen. Die korrekte Erzeugung der Tickets beweist die DCU dem Provider in Zero-Knowledge. Der Beweis wird nicht-interaktiv als SPK ausgeführt, so dass die DCU gleichzeitig die Messdaten signieren kann.
Der Ticket-Spender hat ein Ablaufdatum, so dass eine DCU gezwungen wird, sich nach geraumer Zeit neu beim Provider zu authentifizieren. Durch die Offenlegung der Identität kann der Provider DCUs von der weiteren (erfolgreichen) Übermittlung von Daten ausschließen.

Ein Ticket-Spender wird formal durch eine *PRF (Pseudo-Random-Function Family)* beschrieben, die das Verhalten eines Zufallsorakels (siehe Kapitel 2.4.5) emuliert. Es sei $G = \langle g \rangle$ eine Gruppe mit primer Ordnung $\hat{q}$. Camenisch et al. verwenden in [28] eine effiziente PRF von Dodis und Yampolskiy, die Teil einer *VRF (Verifiable Random Function)*[24] ist [54]. Seien $\hat{m}$ und a die Eingaben der PRF, dann gilt:

$$f^{DY}_{g,\hat{m}}(a) := g^{1/(\hat{m}+a)}$$

Der Eingabeparameter $\hat{m}$ spezifiziert die Familie der PRF und wird hier als *Dispenser-Modul* bezeichnet. Camenisch et al. zeigen, dass unter der SDDHI Vermutung und für Eingaben $\hat{m}, a \in \mathbb{Z}^*_{\hat{q}}$ die Funktion $f^{DY}_{g,\hat{m}}(a)$ eine PRF im *generischen Gruppenmodell*[25] ist [32].

Mithilfe der PRF kann eine DCU gültige Tickets ohne Provider erzeugen und in Zero-Knowledge ihre Korrektheit beweisen.

Wie das „Ticket-Chain" Protokoll lässt sich das „Ticket-Spender" Protokoll in die drei Teile Setup, Registrierung und Reporting unterteilen. Die Kommunikation zwischen $\mathcal{D}$ und $\mathcal{P}$ erfolgt über TLS.

3.5.1 Setup

Wie beim „Ticket-Chain" Protokoll generiert jede DCU $\mathcal{D}$ einen geheimen Signaturschlüssel $\tilde{K}_{sec,\mathcal{D}}$ und einen öffentlichen Verifizierungsschlüssel $\tilde{K}_{pub,\mathcal{D}}$, dessen Kopie zum Provider gesendet wird. Der Schlüssel $\tilde{K}_{pub,\mathcal{D}}$ identifiziert eine DCU eindeutig (siehe Kapitel 3.4.1).

Ein RSA-Modulus n und Basen $\mathbf{S}, \mathbf{Z}, \mathbf{R}_0, \mathbf{R}_1 \in \mathrm{QR}_n$ werden vom Provider erzeugt und ihre korrekte Generierung in Zero-Knowledge bewiesen (siehe Kapitel 3.4.1). Neben der Gruppe $\mathbb{Z}^*_n$ erzeugt der Provider eine Schnorr-Gruppe $\mathbb{Z}^*_{\hat{p}}$ mit einer Primzahl $\hat{p}$ der Länge $\ell_{\hat{p}}$ als Modulus und die Generatoren $g, h \in \mathbb{Z}^*_{\hat{p}}$ (siehe Kapitel 2.2.2). Die Generatoren generieren eine Untergruppe der Ordnung $\hat{q}$, die prim ist.
Die korrekte Erzeugung der Generatoren validieren die DCUs, indem sie prüfen, ob $g, h = \{1, \ldots, \hat{q}-1\}$ gilt und $\hat{p}, \hat{q}$ prim sind. Außerdem muss $g^{\hat{q}} \equiv 1 \pmod{\hat{p}}$ und $h^{\hat{q}} \equiv 1 \pmod{\hat{p}}$ gelten.

Der öffentliche Schlüssel des Providers ist damit $(\tilde{K}_{pub,\mathcal{D}}, \mathbf{S}, \mathbf{R}_0, \mathbf{R}_1, \mathbf{Z}, g, h)$ und der geheime Schlüssel ist $(\tilde{K}_{sec,\mathcal{D}}, \hat{p}, \hat{q})$.

[24] Eine VRF ist eine PRF zusammen mit einem Beweis, dass die Ausgabe der PRF korrekt berechnet wurde. Statt eines Beweises zeigen Camenisch et al. die Korrektheit der PRF in Zero-Knowledge.

[25] Das generische Gruppenmodell ist ein idealisiertes Modell, bei dem ein Angreifer nur Zugriff auf eine zufällig gewählte Kodierung der Gruppenelemente hat. Ein generischer Algorithmus betrachtet nur die Gruppenoperationen und nicht die gruppenspezifische Kodierung der Gruppenelemente [155].

3.5.2 Registrierung

Das Registrierungsprotokoll ähnelt dem Registrierungsprotokoll im „Ticket-Chain“ Protokoll. Anstatt eines Tickets w signiert der Provider das Dispenser-Modul $\hat{m} \in_R \{0, \ldots, 2\hat{q}\}$ mithilfe des CL-Verfahrens. Das Dispenser-Modul wird vorher von der DCU und dem Provider gemeinsam erzeugt und spezifiziert eine Familie von PRFs. Es entspricht dem Initialisierungswert des „Ticket-Spenders“. Das Registrierungsprotokoll ist in Abbildung 3.5 dargestellt. Der Zeitstempel $\vec{TS}$ legt die Ablaufzeit des Dispenser-Moduls fest.

3.5.3 Reporting

Abbildung 3.4 veranschaulicht die drei Schritte des Reporting-Protokolls zwischen einer DCU $\mathcal{D}$ und einem Provider $\mathcal{P}$.

1. Der Zeitstempel TS repräsentiert ein Zeitintervall. Beträgt die zeitliche Auflösung z. B. 15 Minuten, wird jedem 15 Minuten-Intervall – beginnend von einem festen Starttermin – ein Zeitstempel zugewiesen. Ist z. B. der Startwert 01.01.13, bekommt das Datum 01.01.13 mit dem Zeitintervall 0:00 bis 0:15 Uhr den Wert $TS = 1$, das Intervall 0:15 bis 0:30 Uhr den Wert $TS = 2$ usw. zugewiesen.
   Eine DCU generiert aus Dispenser-Modul $\hat{m}$ und Zeitstempel TS ein Ticket w wie folgt:

$$w = f^{DY}_{g,\hat{m}}(TS) = g^{1/(\hat{m}+TS)} \bmod \hat{p} \tag{3.1}$$

   Das Dispenser-Modul $\hat{m}$ wird in einem Pedersen-Commitment gespeichert (siehe Kapitel 2.3.2).

2. Zum Beweis der Gültigkeit eines Tickets muss die DCU erstens zeigen, dass sie eine gültige CL-Signatur für das Dispenser-Modul $\hat{m}$ besitzt und zweitens, dass das Ticket nach Gleichung 3.1 korrekt erzeugt wurde.
   Es sei $\gamma = 1/(\hat{m} + TS)$, $r_2 = \gamma \cdot r_1$ und $\hat{\mathbf{A}} := \mathbf{A}\mathbf{S}^{-\hat{r}} \pmod{n}$, dann erfüllt der folgende Zero-Knowledge Proof diese Anforderungen [32].

$$\mathsf{SPK}\left[\begin{array}{l}(z, v, \hat{m}, \gamma, r_1, r_2) : \\ \mathbf{Z}\hat{\mathbf{A}}^{-2^{\ell_z-1}}\mathbf{R}_1^{-\vec{TS}} \equiv \pm\mathbf{R}_0^{\hat{m}}\hat{\mathbf{A}}^{z}\mathbf{S}^{v} \pmod{n} \wedge \\ w \equiv g^{\gamma} \pmod{\hat{p}} \ \wedge \ g \equiv (C \cdot g^{TS})^{\gamma} h^{-r_2} \pmod{\hat{p}} \wedge \\ C \equiv g^{\hat{m}} h^{r_1} \pmod{\hat{p}} \ \wedge \ z \in \pm\{0,1\}^{\ell_z+\ell_\phi+\ell_c+2}\end{array}\right]\left(C, D_{\text{Reporting}}\right)$$

   Die DCU führt den Beweis als SPK aus, um die Daten $D_{\text{Reporting}}$ zu signieren und an den Proof zu koppeln. Das Ticket w, Zeitstempel TS und die Ablaufzeit des Dispensers $\vec{TS}$ werden an den Provider gesendet.

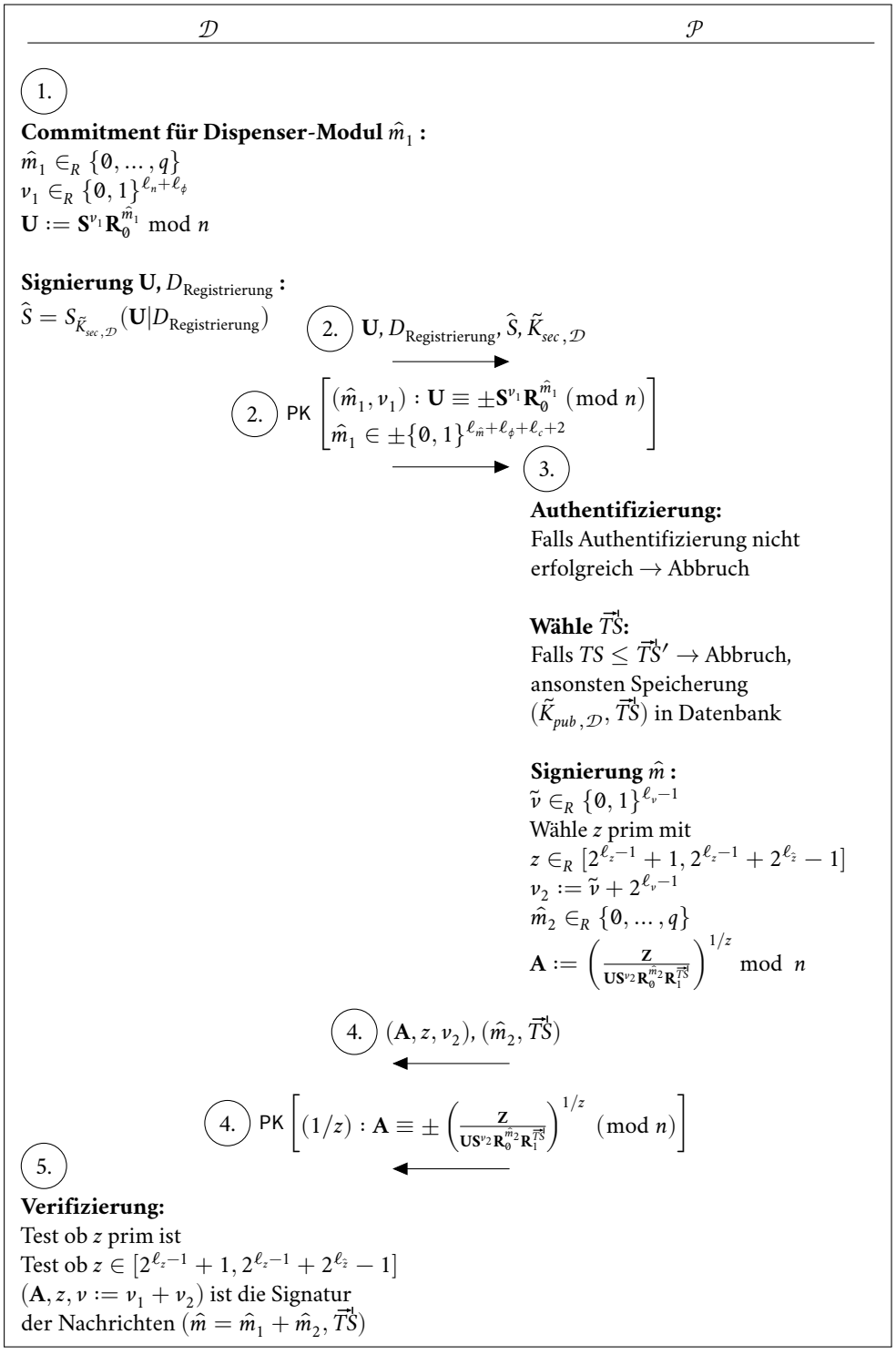


Abbildung 3.5: „Ticket-Spender“ Protokoll: Registrierung.

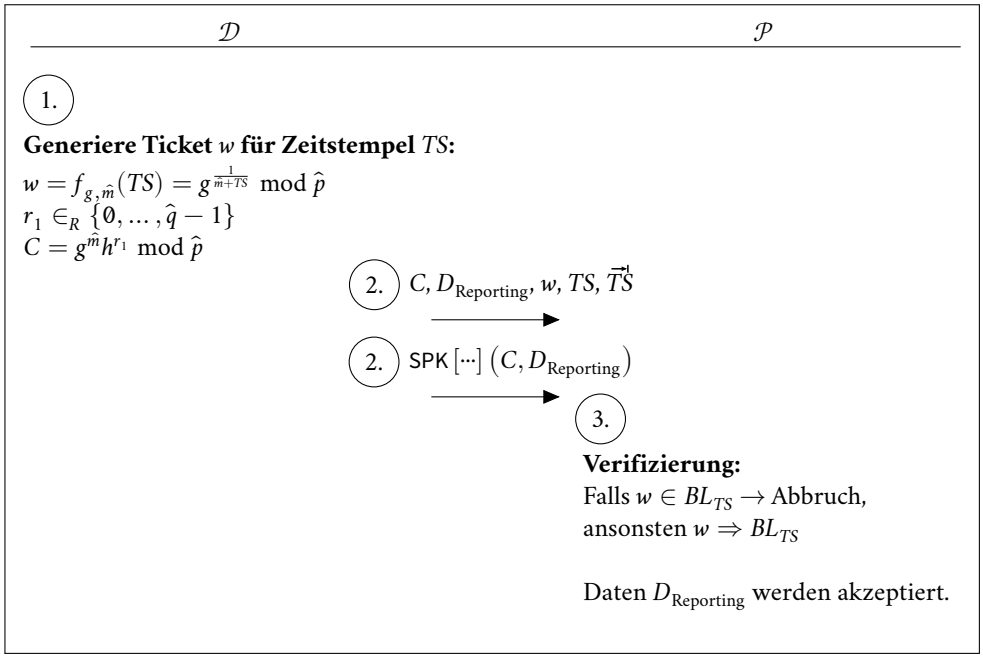


Abbildung 3.6: „Ticket-Spender“ Protokoll: Reporting.

3. Nur wenn der Beweis korrekt ausgeführt wurde, die Gültigkeit des Ticket-Dispensers nicht abgelaufen ist und das Ticket nicht bereits verwendet wurde, akzeptiert der Provider die Daten $D_{\text{Reporting}}$.
   Verbrauchte Tickets für das Zeitintervall TS speichert der Provider in einer Blacklist BL_{TS}. Ist das Zeitintervall TS verstrichen, kann der Provider die komplette Blacklist BL_{TS} löschen.

Das „Ticket-Spender“ Protokoll hat gegenüber dem „Ticket-Chain“ Protokoll den Vorteil, dass die Gültigkeit eines Tickets implizit an ein Zeitintervall TS gekoppelt ist. Ein rechenintensiver Intervall-Proof ist beim „Ticket-Spender“ Protokoll nicht notwendig.
Ein weiterer Vorteil ist die Vorberechnung einzelner Tickets. Das ist z. B. vorteilhaft im FCD-Szenario. Ein Smartphone kann im Hintergrund nicht nur die Tickets für zukünftige Zeitintervalle, sondern auch einen Teil der zugehörigen, rechenintensiven SPKs vorberechnen und speichern[26]. Sollen Daten durch das Reporting-Protokoll an den Provider gesendet werden, muss die DCU nur die Response des SPK berechnen. Nur sie ist von den Eingabedaten $D_{\text{Reporting}}$ abhängig. Damit beschränkt sich der Rechenaufwand neben Hash-Operationen auf wenige Langzahlmultiplikationen und -additionen. Die Berechnung rechenintensiverer modularer Exponentiationen entfällt.

[26] Das kann z. B. immer dann erfolgen, wenn das Smartphone aufgeladen wird, um Batterieressourcen zu sparen.

Das „Ticket-Chain“ Protokoll bietet dagegen den Vorteil, dass der Provider über den Qualitätsparameter Rückinformationen geben kann, die bei der nächsten Transaktionen wieder an den Provider gesendet werden. Diese Verknüpfbarkeit ist beim „Ticket-Spender“ Protokoll nicht möglich.

3.5.4 Sicherheitsanalyse

Das „Ticket-Spender“ Protokoll erfüllt die Sicherheitsanforderungen aus Kapitel 1.5.

1. *Authentifizierung und Integrität*
   Der Provider signiert anstatt dem Ticket w im „Ticket-Chain“ Protokoll das Dispenser-Modul $\hat{m}$ mithilfe des CL-Protokolls. Ansonsten entspricht die Authentifizierung und Sicherstellung der Integrität dem „Ticket-Chain“ Protokoll.

2. *Widerrufbarkeit und Identifizierung*
   Bei jeder Ausführung des Registrierungsprotokolls wird die DCU identifiziert.
   Die Signatur des Dispenser-Moduls ist nur bis zu dem Zeitpunkt $\vec{TS}$ gültig, so dass eine DCU gezwungen wird, sich danach erneut beim Provider zu registrieren. Der Provider hat die Möglichkeit, die DCU zu entfernen und von der weiteren (erfolgreichen) Übermittlung von Daten auszuschließen.

3. *Anonymität und Nicht-Verknüpfbarkeit*
   Das Dispenser-Modul bleibt dem Provider immer verborgen. Im Registrierungsprotokoll wird das durch die Eigenschaften des CL-Protokolls sichergestellt (siehe Kapitel 3.3.4). Des Weiteren haben Camenisch et al. gezeigt, dass es für eine Partei bei geeigneter Wahl der Sicherheitsparameter nicht möglich ist, aus dem Bild der Dispenser Funktion f, d. h. dem Ticket w, ein gültiges Dispenser-Modul $\hat{m}$ zu berechnen.
   Wie beim „Ticket-Chain“ Protokoll ist zu beachten, dass die Auflösung des Zeitstempels $\vec{TS}$ niedrig gewählt werden muss, um eine Identifizierung durch den Provider zu vermeiden.

4. *Robustheit*
   Ein Angreifer kann zwar beliebig viele Tickets erzeugen, ein Ticket gilt allerdings nur für ein Zeitintervall TS. Wird es verbraucht, speichert der Provider es in einer Blacklist. Spamming- und Replay-Angriffe werden dadurch verhindert.
   Damit ein Angreifer sich nicht mehrere Dispenser-Module und damit Ticket-Spender vom Provider signieren lassen kann, erstellt der Provider für eine DCU erst ein neues Dispenser-Modul, sobald das vorherige Modul seine Gültigkeit verloren haben.

3.6 Performance

	Init $\mathcal{D}$ [s]	Init $\mathcal{P}$ [s]	Registr. $\mathcal{D}$ [s]	Registr. $\mathcal{P}$ [s]	Report. $\mathcal{D}$ [s]	Report. $\mathcal{P}$ [s]
PC	1,40 0,60	1,41 0,60	<0,01 <0,01	0,01 0,01	0,02 <0,01	0,01 <0,01
Nexus S	31,09 12,01	30,09 12,11	0,14 0,16	0,27 0,21	0,33 0,15	0,29 0,13
MUC[27]	- -	- -	1,35 1,78	1,66 1,66	3,40 1,32	2,64 1,18

Abbildung 3.7: Performance-Vergleich des „Ticket-Chain“ (blau) und „Ticket-Spender“ (rot) Protokolls für $\ell_n = 1024$.

	Init $\mathcal{D}$ [s]	Init $\mathcal{P}$ [s]	Registr. $\mathcal{D}$ [s]	Registr. $\mathcal{P}$ [s]	Report. $\mathcal{D}$ [s]	Report. $\mathcal{P}$ [s]
PC	11,24 4,82	11,28 4,83	0,04 0,05	0,05 0,05	0,10 0,03	0,07 0,02
Nexus S	224,35 96,78	226,84 97,10	0,83 0,97	0,82 0,96	1,96 0,59	1,47 0,45
MUC[27]	- -	- -	7,09 8,37	7,27 9,58	17,58 4,96	15,18 3,81

Abbildung 3.8: Performance-Vergleich des „Ticket-Chain“ (blau) und „Ticket-Spender“ (rot) Protokolls für $\ell_n = 2048$.

Die Performance des „Ticket-Chain“ und „Ticket-Spender“ Protokolls wird auf verschiedenen Hardware-Plattformen ermittelt. Die Beschreibung der Protokolle erfolgt in der Eingabesprache des Crypto-Compilers (siehe Kapitel 4 und Anhang A.2, A.3). Der Compiler übersetzt die Programme in C-Code, die dann von einem GCC C-Compiler in ausführbaren plattformspezifischen Code kompiliert werden.

Für die Messung der Performance auf Provider-Seite wird ein Standard PC (Intel Core i7 CPU 920 @2,67 GHz x 8, 16 GB RAM, Ubuntu 12.04 64 Bit, GCC 4.6.3) verwendet[28]. In den FCD- und Smart Metering Szenarien verfügen die DCUs über eine deutlich geringere Leistung als ein PC, so dass die Protokolle zusätzlich auf einem Nexus S Smartphone (ARM Cortex-A8 @1 GHz, 512 MB RAM,

[27] Um mehr Ressourcen im Arbeitsspeicher zur Verfügung zu stellen, musste ein Watchdog deaktiviert werden. Dadurch führt das Gerät alle 30 Sekunden einen Neustart durch und die Ausführungszeit des Initialisierungsprotokolls kann nicht ermittelt werden.

[28] Bei allen Messungen wird nur ein Prozessorkern verwendet.

Android 4.4.4, GCC 4.6) und dem ZDUE-DSL MUC (Atmel AT91SAM9260 @210 MHz, 32 MB RAM, Embedded Linux, GCC 4.3.2) von Dr. Neuhaus ausgeführt werden. Bei dem ZDUE-DSL MUC handelt es sich um einen modifizierten MUC (Multi-Utility-Communication) Controller, der in industriellen Smart Metering Umgebungen zum Einsatz kommt.
Zur Berechnung von Langzahloperationen wird die MPIR (Multiple Precision Integers and Rationals) 2.6.0 Bibliothek und für Hash-Operationen die OpenSSL-Bibliothek 1.0.1e verwendet (siehe Kapitel 4.2.1).

ℓ_n	$\ell_{\hat{p}}$	$\ell_{\hat{q}}$	ℓ_c	ℓ_ϕ	ℓ_w	ℓ_z	$\ell_{\hat{z}}$	ℓ_{TS}
1024	1024	160	224 (SHA-224)	80	80	390	80	32
2048	2048	224	256 (SHA-256)	80	80	421	80	32

Abbildung 3.9: Wahl der Protokollparameter.

Die beiden Protokolle werden für RSA-Moduli der Länge $\ell_n = 1024$ und $\ell_n = 2048$ evaluiert. Die Wahl der Parameter zeigt Tabelle 3.9. Der Parameter $\ell_{\hat{p}}$ wird so gewählt, dass das DLP in der Gruppe $\mathbb{Z}^*_{\hat{p}}$ genauso schwierig zu lösen ist wie das Faktorisierungsproblem in der Gruppe $\mathbb{Z}^*_n$ [107].
Die Zeitstempel TS, $\overleftarrow{TS}$ und $\overrightarrow{TS}$ haben eine Länge von 32 Bit und werden zufällig unter der Bedingung $\overleftarrow{TS} < TS < \overrightarrow{TS}$ bestimmt.
In der Tabelle 3.7 und 3.8 sind die Benchmark-Ergebnisse für DCU und Provider dargestellt.

Die Ergebnisse zeigen die Praxistauglichkeit der Protokolle. Selbst auf dem Smartphone und dem MUC, dessen Prozessoren hinsichtlich geringem Energieverbrauch optimiert sind, werden die Protokolle schnell genug ausgeführt, um Daten mithilfe des Reporting-Protokolls in einem 15-Minuten-Fenster zum Provider zu senden.
Durch Optimierungen lassen sich die Ergebnisse weiter verbessern. In Kapitel 5 wird gezeigt, wie die Performance insbesondere auf der Seite des Providers durch den Einsatz von Grafikkarten weiter erhöht werden kann.

3.7 Stand der Forschung

Dieses Kapitel gibt einen Überblick über weitere Lösungen, um die Anforderungen in Kapitel 1.5 zu erfüllen.

3.7.1 FCD-Szenario

Die Übertragung von FCD unter Berücksichtigung strenger Datenschutz- und Authentifizierungsanforderungen ist bisher wenig erforscht. In der Regel konzen-

triert sich die Forschung auf den Datenschutz von Standortbezogenen Diensten (engl. *LBS (Location-Based Services)*) [43].
LBS bieten mobilen Benutzern Dienstleistungen auf Basis von Positionsdaten. Ein LBS wäre z. B. ein Dienst, der auf Grundlage der aktuellen Position einen Benutzer über die Standorte der umliegenden Tankstellen informiert. Ziel der Forschung ist die Entwicklung von Verfahren, die es Dienstanbietern auf Basis von mehreren Anfragen eines Benutzers unmöglich macht, Bewegungsprofile zu erstellen.
In Mokbel et al. werden zusammen mit der Position des Benutzers falsche Positionsangaben an den Dienstanbieter gesendet [119]. Hong et al. sowie Yiu et al. verschlechtern künstlich die Genauigkeit der zum Dienstanbieter gesendeten Positionsangaben [82] [176]. Die Forschungsergebnisse lassen sich nur schwer auf das FCD-Szenario übertragen. Eine hohe Genauigkeit der Positionsdaten ist erforderlich, um z. B. den Verkehrsfluss von Fahrzeugen berechnen zu können und damit Aussagen über mögliche Staus vorhersagen zu können[29].

Andere Ansätze verwenden den Einsatz einer TTP (auch *Anonymity Server* genannt), der die Standorte mobiler Benutzer unter Berücksichtigung individueller Datenschutzanforderungen verwischt [69]. Die Anforderungen lassen sich häufig im „k-Anonymity Model" beschreiben [160]. Der Anonymity Server stellt damit sicher, dass sich das Verhalten eines Benutzers nicht vom Verhalten k anderer Benutzer unterscheidet.
Lösungen mit einem Anonymity Server lassen sich zwar auf das FCD-Szenario übertragen, lösen aber nicht das Problem der Authentifizierung. Des Weiteren erschwert die Notwendigkeit einer TTP die praktische Umsetzung. Anstatt einer TTP schlagen manche Autoren die Kommunikation zwischen verschiedenen mobilen Benutzern vor, um die Position des einzelnen Benutzers zu verschleiern [70]. Neben der fehlenden Authentifizierung erfordern solche Lösungen einen deutlich höheren Kommunikationsaufwand als bei den in Kapitel 3 vorgestellten Verfahren.

Eichler berücksichtigt in seinem Verfahren die Übertragung von FCD unter Berücksichtigung von Datenschutz- und Authentizitätsanforderungen [60].
Grundlage sind drei Partien: *CC (Control Center)*, *SC (Service Center)* und *CS (Client System)*. Das CC ist für die Authentifizierung des Fahrzeuges zuständig und wird in der Regel vom Fahrzeughersteller betrieben. Das SC bietet Fahrzeugnutzern verschiedene Dienste an. Dazu gehört die Verarbeitung der vom Fahrzeug gesendeten FCD und die Berechnung von Verkehrsstaus. Es kann mehrere SCs geben. Das CS ist die Benutzerschnittstelle im Fahrzeug und verarbeitet die Daten vom SC.
Das Verfahren von Eichler trennt zwischen Rechtevergabe und Rechteanwendung (siehe Kapitel 3.1). Das CS generiert ein Pseudonym (den *Token*) und authentifiziert sich beim CC. Bei Erfolg sendet das CC eine Signatur für den Token

[29] Verkehrsstaus in Städten sind in der Regel stark räumlich begrenzt.

zurück. Daraufhin überträgt das CS den Token und die Signatur zusammen mit den FCD an das SC. Das SC akzeptiert die FCD, wenn die Signatur korrekt ist. Durch die Authentifizierung des CS bietet sich dem CC die Möglichkeit des Widerrufs.
Das Verfahren hat zwei wesentliche Nachteile, durch die eine Verknüpfung zwischen den FCD eines CS gegeben ist und die Anonymität der Benutzer eingeschränkt wird.

1. Der Token wird nicht bei jeder Transaktion zwischen CS und SC gewechselt[30].
2. Falls CC und SC kooperieren, lässt sich das CS bei der Übermittlung von Positionsdaten identifizieren. Vorausgesetzt es wird keine TTP verwendet, gilt das ebenfalls für eine Erweiterung des Verfahrens, das von Eichler diskutiert wird und auf blinden RSA-Signaturen[31] basiert.

Rass et al. haben ein Verfahren vorgestellt, bei dem einzelne Transaktionen nur in Ausnahmefällen miteinander verknüpft werden können [142].
Fahrzeuge sind bei dem Szenario mit einer *OBU (On-Board Unit)* ausgerüstet, die durch eine eindeutige *ID* identifiziert werden können. FCD werden an einen zentralen Server gesendet, ohne dass die OBU-ID offengelegt wird.
Positionsdaten werden bei dem Verfahren als *Samples* bezeichnet, wobei jedes Sample eindeutig einer *SID (Sample-ID)* zugeordnet ist. Ein Trip mit einer eindeutigen *TID (Trip-ID)* setzt sich aus einer Reihe von Samples zusammen. Grundlage des Verfahrens bildet die *Fusion Exponentiation*[32], die sicherstellt, dass IDs global eindeutig sind. Es ist weder möglich, aus einer SID oder TID auf die entsprechende OBU-ID zu schließen, noch kann eine Aussage getroffen werden, ob zwei SIDs bzw. TIDs von derselben OBU-ID stammen. Es gibt allerdings zwei Möglichkeiten, um Trips und Samples miteinander zu verknüpfen.

- Bei einer TTP können Informationen hinterlegt werden, so dass in bestimmten Fällen (z. B. zur Berechnung von Mautgebühren) eine Verknüpfung möglich ist.
- Durch die Verwendung von SIDs, TIDs und OBU IDs im Exponenten der Fusion Exponentiation können Beziehungen zwischen Samples in Zero-

[30] Das soll den Kommunikationsaufwand mit dem CC gering halten.

[31] Dabei multipliziert das CS den Token mit einem *„Blinding-Faktor"* bevor es ihn an das CC sendet. Das CC signiert den modifizierten Token mit seinem privaten Schlüssel und sendet ihn zurück an das CS. Das CS kann den „Blinding-Faktor" aus der Signatur entfernen, ohne dass die Signatur ungültig wird. Der „Blinding-Faktor" verschleiert den Token gegenüber dem CC.

[32] Es sei G eine Gruppe mit primer Ordnung q und es gilt $g_0, g_1 \in G$ und $a, b \in \mathbb{Z}_q$. Eine Fusion Exponentiation ist definiert als:

$$(g_0, g_1)^{(a,b)} := (g_0^a g_1^{-b}, g_0^b g_1^a)$$

Knowledge gezeigt werden. Die Autoren weisen in ihrer Veröffentlichung nur auf die generelle Machbarkeit hin. Sie erwähnen z. B. nicht, wie die Integrität der öffentlichen Werte ohne TTP sichergestellt werden kann.

Der wesentliche Nachteil des Verfahrens ist, dass sich die Autoren nur auf den Schutz der Privatsphäre konzentrieren. Eine Authentifizierung der OBU wird nicht vorgenommen, so dass sich OBUs nicht widerrufen lassen können. Die Integrität der Daten wird nicht sichergestellt und das Verfahren ist nicht robust gegenüber Spamming-Angriffe.

3.7.2 Smart Metering Szenario

Auf den folgenden Seiten wird eine Gruppe von N Smart Metern betrachtet, die Verbrauchsdaten sowohl zur Abrechnungs- als auch zur Lastprognose („Load Reporting“) an das EVU senden. Der Anschaulichkeit halber wird keine Unterscheidung zwischen EVU und VNB vorgenommen. Beide Parteien werden hier als EVU bezeichnet.

Abrechnung

In der Lösung von Bohli et al. senden die Smart Meter die Verbrauchswerte an eine TTP, welche die Werte unter Berücksichtigung eines Abrechnungsprofils akkumuliert und den Gesamtbetrag an das EVU weiterleitet [19].

Falls die Abrechnung mit dem Smart Meter erfolgt, ist eine TTP nicht erforderlich. Am Ende eines Abrechnungszeitraumes (z. B. eines Monats) wird die Gesamtsumme an das EVU gesendet. Dafür eignet sich das Registrierungsprotokoll im „Ticket-Chain“- bzw. „Ticket-Spender“-Protokoll, denn der Smart Meter identifiziert und authentifiziert sich beim EVU.

Rial und Danezis [144] sowie Jawurek et al. [89] haben Verfahren vorgestellt, um die Abrechnung außerhalb des Smart Meters vorzunehmen. Dabei muss dem Smart Meter die Abrechnungstabelle nicht zur Verfügung gestellt werden. Die Abrechnung erfolgt nicht auf dem Smart Meter, sondern auf dem Client[33] und soll die Berechnung des Gesamtverbrauches aus den Einzelverbrauchswerten für den Kunden besser nachvollziehbar machen.
Ein Smart Meter speichert dazu die einzelnen Verbrauchswerte in Commitments, signiert das Commitment und überträgt sie zum Client. Der Client summiert die einzelnen Verbrauchswerte (gewichtet mit den Werten einer Abrechnungstabelle) und sendet das Ergebnis zusammen mit den Commitments und Signaturen an das EVU. Der Client beweist dem EVU in Zero-Knowledge, dass die Aggregation der Verbrauchswerte auf Basis der Abrechnungstabelle korrekt erfolgt ist. Die

[33] Das kann z. B. der PC des Kunden sein.

Verbrauchswerte müssen nicht offen gelegt werden, wodurch die Privatsphäre des Kunden geschützt wird. Die signierten Commitments verhindern die Manipulation der Verbrauchswerte durch den Client.
Die Verfahren sind sehr rechenintensiv und können das Registrierungsprotokoll im „Ticket-Chain"- oder „Ticket-Spender"-Protokoll ergänzen.

Load Reporting

Kalogridis et al. [97], Varodayan / Khisti [170] und McLaughlin et al. [115] verwenden eine Batterie, um die Verbrauchskurve zu glätten und damit eine Identifizierung einzelner Haushaltsgeräte zu verhindern. Der größte Nachteil der Verfahren sind die relativ hohen zusätzlichen Kosten, die pro Haushalt für eine Batterie aufgewendet werden müssen.

Auf den folgenden Seiten sollen Verfahren, die ohne Batterie auskommen, hinsichtlich der Anforderungen aus Kapitel 1.5 evaluiert werden (siehe Tabelle 3.1). Die Verfahren lassen sich zwei Kategorien zuordnen, welche die Robustheit der Verfahren festlegen[34].

I *Das EVU kann nur die Summe der Verbrauchswerte ermitteln*:
Das EVU ist nicht in der Lage, einzelne Verbrauchswerte auf Plausibilität zu überprüfen. Das EVU kennt stattdessen nur den Gesamtverbrauch. Die DCUs senden ihre Daten verschlüsselt bzw. in einem Commitment versteckt zum Provider. Das hat den Nachteil, dass ein Angreifer mit Kontrolle über nur einen Smart Meter die Lastprognose maßgeblich negativ beeinflussen kann. Das EVU kann nicht zwischen dem (unrealistisch) hohen Verbrauch eines Smart Meters und dem (geringfügig) höheren individuellen Verbrauch von N Smart Metern unterscheiden.
Wenn die Smart Meter direkt mit dem EVU kommunizieren, hat die Verschleierung der Verbrauchswerte vor dem Provider den Vorteil, dass eine Authentifizierung (und Identifizierung) der Smart Meter beim EVU prinzipiell[35] möglich ist. Dadurch kann eine Obergrenze für die Anzahl der Übertragungen pro Smart Meter und Zeitintervall festgelegt werden und der Einfluss kompromittierter Smart Meter begrenzt werden.

II *Das EVU kennt die individuellen Verbrauchswerte*:
Das EVU kann zu hohe Messwerte einzelner Smart Meter erkennen und unrealistische Werte verwerfen. Auf der einen Seite hat das den Vorteil, dass ein Angreifer eine große Anzahl an Smart Metern unter seiner Kontrolle haben muss, um die Lastprognose nennenswert negativ beeinflussen

[34] Die Kategorie ist in Tabelle 3.1 in eckigen Klammern angegeben.
[35] Die hier beschriebenen Verfahren der Kategorie I führen keine Authentifizierung durch, lassen sich aber häufig dahingehend erweitern. Die Machbarkeit zeigt Tabelle 3.1.

zu können. Auf der anderen Seite ist eine Authentifizierung und Identifizierung des Smart Meters aufgrund der geforderten Anonymität nicht möglich, so dass Spamming-Angriffe anderweitig verhindert werden müssen.
Falls Spamming-Angriffe verhindert werden, sind damit Verfahren der Kategorie II robuster als Verfahren der Kategorie I.

Verfahren der Kategorie II sind flexibler einsetzbar. Anstatt Daten, die nur aufsummiert werden können, kann ein Smart Meter in der Regel beliebige Daten[36] an das EVU senden.

Bei der Lösung von Bohli et al. senden die Smart Meter die Verbrauchsdaten nicht nur zu Abrechnungszwecken, sondern auch für das Load Reporting an eine TTP [19]. Die Daten werden mit einem symmetrischen oder asymmetrischen Verfahren verschlüsselt[37] übertragen. Die TTP anonymisiert die Daten und leitet sie an das EVU weiter. Dadurch kann das EVU Transaktionen eines Smart Meters weder verknüpfen noch einen Smart Meter identifizieren.
Die Autoren geben nicht an, wie Smart Meter authentifiziert oder widerrufen werden und wie die Integrität der zu übertragenden Daten sichergestellt wird.
Das Verfahren gehört zur Kategorie II und ist damit flexibel einsetzbar. Aufgrund der fehlenden Authentifizierung kann ein Angreifer Messwerte beliebig häufig an die TTP senden. Spamming-Angriffe werden nicht verhindert und das Verfahren ist somit nicht robust.

Bei der Lösung von Petrlic sendet der Smart Meter verschlüsselte und signierte Verbrauchsdaten über einen *Kollektor* an das EVU [135]. Der Kollektor überprüft die Signatur und authentifiziert den Smart Meter. Ist die Integrität der Daten gewährleistet, d. h. die Signatur korrekt und der Smart Meter gültig, leitet der Kollektor die verschlüsselten Verbrauchsdaten anonymisiert an das EVU weiter. Die Verbrauchsdaten sind mit dem öffentlichen Schlüssel des EVU verschlüsselt, so dass der Kollektor die Verbrauchsdaten nicht lesen kann.
Der Kollektor ist eine TTP. Kooperieren EVU und Kollektor, können alle Verbrauchswerte einem Smart Meter zugeordnet werden.
Wie beim TTP Verfahren von Bohli et al. ist der Schutz der Privatsphäre hoch, da der Kollektor die Daten vor der Weitergabe an das EVU anonymisiert.
Das Verfahren gehört zur Kategorie II und bietet damit eine hohe Flexibilität. Es ist robust gegen Angriffe mit vollständig kompromittierten Smart Metern. Das EVU kann unrealistische Verbräuche identifizieren und Spamming-Angriffe können vom Kollektor durch den Authentifizierungsprozess erkannt werden.

Der Einsatz einer TTP bzw. eines Kollektors ist nicht zwingend notwendig. Bohli et al. [19], Wang et al. [174] und Lin et al. [108] stellen in ihren Veröffentlichungen

[36] Das kann z. B. die Akku-Ladekurve eines Elektrofahrzeuges sein, die dem EVU hilft, den zukünftigen Strombedarf besser abschätzen zu können.
[37] In der Veröffentlichung ist nur von Verschlüsselung die Rede.

	Bohli et al. (TTP-Variante) [19]	Petrlic [135]	Bohli et al. [19], Wang et al. [174], Lin et al. [108]	Garcia et al. [68]	Kursawe (low-overhead Prot.) [106]	Vetter et al. [171]	Mármol et al. [112]	Jeske
SM Kommunikation	über TTP	über Kollektor	direkt	über EVU zu SMs	direkt	direkt	über SMs	direkt
TTP	ja	ja	nein	ja (CA)	ja (CA)	ja (KA)	ja (SMs)	nein
Authent. / Widerrufbarkeit	nein	ja	möglich	möglich	möglich	möglich	nein	ja
Integritäts-überpüfung	nein	ja	möglich	möglich	möglich	möglich	nein	ja
Anonymität und Unverknüpfbar.	hoch	hoch	mittel	hoch	hoch	hoch	hoch	hoch / mittel
Robustheit	niedrig [II]	hoch [II]	niedrig [I]	niedrig [I]	niedrig [I]	niedrig [I]	niedrig [I]	hoch [II]
Flexibilität	hoch	hoch	niedrig	niedrig	niedrig	niedrig	niedrig	hoch
Rechenaufwand SM	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Rechenaufwand EVU	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N^2)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$
Kommunikationsaufwand (nach Initialisierung)	niedrig (SM kommuniziert direkt mit TTP)	niedrig (SM kommuniziert direkt mit Kollektor)	niedrig (SM kommuniziert direkt mit EVU)	hoch (jeder SM kommuniziert über EVU mit anderen SMs)	niedrig (SM kommuniziert direkt mit EVU)	niedrig (SM kommuniziert direkt mit EVU)	mittel (SM kommuniziert direkt mit EVU, SM kommuniziert mit Vor- und Nachfolger SM)	mittel (Tor) / niedrig (direkt)

Tabelle 3.1: Smart Metering Protokolle zur Lastprognose.

Lösungen ohne TTP vor, bei dem der Smart Meter die Verbrauchsdaten direkt an das EVU sendet. Zur Vermeidung einer Identifizierung von Haushaltsgeräten, erzeugt der Smart Meter einen Zufallswert nach einer bestimmten Verteilung[38], addiert diesen auf den Messwert und überträgt das Ergebnis zum EVU. Das EVU aggregiert für jedes Zeitintervall die „verrauschten" Verbrauchswerte von allen N Smart Metern. Da das EVU die Verteilung der Zufallswerte kennt, kann es die zusätzliche „Rauschkomponente" entfernen und erhält (näherungsweise) die Summe der Verbrauchswerte.
Die Smart Meter werden vom EVU nicht authentifiziert und die Integrität der Messdaten wird nicht sichergestellt. Im Vergleich zu anderen Verfahren ist weiterhin nachteilig, dass für ein hohes Maß an Datenschutz und Genauigkeit der akkumulierten Werte eine höhere Anzahl an Smart Metern notwendig ist [19].
Das Verfahren gehört zur Kategorie I (niedrige Flexibilität) und ist nicht robust gegen Angriffe durch vollständig kompromittierte Smart Meter. Wird das Verfahren erweitert und authentifiziert bzw. identifiziert sich ein Smart Meter beim EVU, kann das EVU Transaktionen verknüpfen. Das Verfahren erschwert zwar durch das Rauschen die Identifizierung genauer Verwendungszeiten von Haushaltsgeräten, die Autoren geben aber nicht an, wie generelle Aussagen, wie z. B. die Frage ob sich eine Person im Haushalt aufhält, verhindert werden können[39].

Beim Verfahren von Garcia und Jacobs werden die Messwerte mit dem *Paillier Kryptosystem* verschlüsselt [68]. Jeder Smart Meter zerlegt den Messwert für ein Zeitintervall in N Segmente, so dass die Summe der Segmente dem Messwert entspricht. Danach verschlüsselt jeder Smart Meter i mit $i = \{0, \dots, N-1\}$ bis auf das i-te Segment alle $N-1$ Segmente mit dem öffentlichen Schlüssel der anderen $N-1$ Smart Meter und sendet die Ciphertexte an das EVU. Das EVU multipliziert die $N-1$ Ciphertexte, die für einen Smart Meter bestimmt sind und leitet das Produkt an den entsprechenden Smart Meter weiter. Aufgrund der homomorphen Eigenschaft[40] des Paillier Kryptosystems entspricht das Produkt einer Verschlüsselung der aufsummierten Einzelwerte. Jeder Smart Meter entschlüsselt sein Produkt und addiert das fehlende Segment hinzu. Die Summe wird an das EVU gesendet. Die Addition der einzelnen Werte entspricht dem Gesamtverbrauch aller Smart Meter pro Zeitintervall.
Um die Integrität der öffentlichen Schlüssel sicherzustellen, erfordert das Verfahren eine *CA (Certification Authority)* als TTP [136]. Die Integrität der Messdaten wird nicht sichergestellt und das Paillier Kryptosystem hat den Nachteil, dass es

[38] z. B. Gaußverteilung

[39] Bei anderen Verfahren der Kategorie I werden die Verbrauchsdaten mit einem geheimen Schlüssel verschlüsselt. Hier werden die Verbrauchswerte mit einem Zufallswert addiert dessen Verteilung dem EVU bekannt ist.

[40] Es sei $\bullet$ eine algebraische Operation, die auf Klartexte angewendet wird und $\otimes$ eine auf Ciphertexte angewandte algebraische Operation. Die Verschlüsselungsoperation E ist homomorph, falls für zwei Nachrichten m_0 und m_1 gilt: $E(m_0 \bullet m_1) = E(m_0) \otimes E(m_0)$ [147].

verformbar[41] ist.
Das Verfahren lässt sich um ein einfaches Authentifizierungsverfahren (z. B. Challenge-Response-Authentifizierung) erweitern, da die individuellen Messdaten vom EVU nicht entschlüsselt werden können. Das EVU kann deshalb Smart Meter identifizieren, ohne die Datenschutzanforderungen zu verletzen.
Die Verschlüsselung individueller Messdaten macht das Verfahren allerdings nicht robust gegen Angriffe vollständig kompromittierter Smart Meter. Das Verfahren gehört zur Kategorie I und bietet deshalb eine geringe Flexibilität. Weitere Nachteile sind der hohe Rechen- und Kommunikationsaufwand, der von der Größe der Smart Meter Gruppe abhängig ist.

Kursawe et al. beschreiben in ihrer Veröffentlichung mehrere Verfahren, bei denen die Verbrauchswerte mit einem *Blindingwert* addiert und dadurch verschleiert werden [106]. Die Autoren unterscheiden zwei Anwendungsfälle. Im ersten Fall kennt das EVU den Gesamtverbrauch und vergleicht ihn mit der Summe einzelner Haushaltsmessungen, um Unstimmigkeiten im Verbrauch eines Versorgungsgebietes zu erkennen. Im zweiten Fall ermittelt das EVU den Verbrauch aller Haushalte in einem Versorgungsgebiet pro Zeitintervall. Das entspricht dem Szenario in dieser Arbeit.
An dieser Stelle soll das *Low-Overhead Protokoll* von Kursawe et al. betrachtet werden. Jeder Smart Meter a besitzt einen geheimen Schlüssel $\tilde{K}_{sec,a}$ und einen öffentlichen Schlüssel $\tilde{K}_{pub,a} = g^{\tilde{K}_{sec,a}}$ mit dem Generator g einer Gruppe, in der die CDH Vermutung in der Praxis nicht lösbar ist. Jeder Smart Meter kennt die öffentlichen Schlüssel der anderen Smart Meter. Wie beim Verfahren von Garcia et al. ist eine TTP in Form einer CA notwendig, um die Integrität der Schlüssel zu gewährleisten. Des Weiteren berechnet jeder Smart Meter a die Schlüssel $K_{a,b} = \mathrm{H}(\tilde{K}_{pub,b}^{\tilde{K}_{sec,a}})$ mit $a, b \in \{0, \ldots, N-1\}$ und $b \neq a$. Der Schlüssel $K_{a,b}$ ist der symmetrische Schlüssel zwischen Smart Meter a und Smart Meter b.
Um Messwerte für das Zeitintervall mit dem Zeitstempel TS an das EVU zu senden, berechnet jeder Smart Meter a zuerst einen Blindingwert $k_{a,TS}$:

$$k_{a,TS} = \sum_{i=0, i\neq a}^{N-1} (-1)^{i \overset{?}{<} a} \mathrm{H}(K_{a,i} || TS), \quad i \overset{?}{<} a := \begin{cases} 1, & i < a \\ 0, & i \geq a \end{cases}$$

Danach verschleiert der Smart Meter a die Messdaten $D_{a,TS}$ mit $D' = D_{a,TS} + k_{a,TS} \bmod 2^{32}$ und sendet das Ergebnis D' an das EVU. Das EVU akkumuliert die verschleierten Messdaten. Die Summe entspricht der Summe der Messdaten, d. h. $\sum_{i=0}^{N-1} D_{i,TS} = \sum_{i=0}^{N-1} D'_{i,TS}$.
Das EVU kann beim Verfahren von Kursawe et al. keine Rückschlüsse auf die Lebensgewohnheiten eines Haushaltes ziehen. Es ähnelt dem Verfahren von Garcia und Jacobs und lässt sich durch eine Smart Meter Authentifizierung und

[41] Ein Verschlüsselungsalgorithmus ist verformbar, wenn ein Angreifer einen Ciphertext in einen anderen Ciphertext überführen kann, so dass die Entschlüsselung einen ähnlichen Klartext ergibt [55].

eine Überprüfung der Datenintegrität ergänzen. Das Verfahren ist aus denselben Gründen wie das Verfahren von Garcia und Jacobs weder robust noch flexibel.

Bei dem Verfahren von Vetter et al. wird die Integrität der Daten durch einen *MAC (Message Authentication Code)* geschützt [171].
Es seien n_0 und n_1 zwei Moduli, die allen Smart Metern und dem EVU bekannt sind. Jeder Smart Meter $a \in \{0, \ldots, N-1\}$ besitzt einen „Root Encryption Key" K_a. Zusätzlich zum Smart Meter kennt den Schlüssel K_a nur der KA (Key Aggregator). Bei dem KA handelt es sich um eine TTP.
Bevor ein Smart Meter a die Messdaten $D_{a,TS}$ für ein Zeitintervall mit dem Zeitstempel TS zum EVU überträgt, werden sie mit dem Blindingwert $k_{a,TS} = \mathsf{H}(K_a||TS)$ verschleiert:

$$D'_{a,TS} = D_{a,TS} + k_{a,TS} = D_{a,TS} + \mathsf{H}(K_a||TS) \bmod n_0$$

Zusätzlich erzeugt Smart Meter a für die verschleierten Messdaten $D'_{a,TS}$ einen MAC mit dem Blindingwert $\tilde{k}_{a,TS} = M_0(K'_1||a||TS)$.
Es seien K'_0 und K'_1 zwei vom EVU gewählte Schlüssel und M_0 und M_1 zwei Pseudo-Zufallsfunktionen, dann gilt:

$$\begin{aligned} MAC(D'_{a,TS}) &= M_1(K'_0)D'_{a,TS} + \tilde{k}_{a,TS} \bmod n_1 \\ &= M_1(K'_0)D'_{a,TS} + M_0(K'_1||a||TS) \bmod n_1 \end{aligned}$$

Damit das EVU den MAC überprüfen und die Summe der Messwerte berechnen kann, sind die Summen der Blindingwerte notwendig. Der KA berechnet $k_{all,TS}$ und $\tilde{k}_{all,TS}$ und sendet sie zum EVU.

$$k_{all,TS} = \left(\mathsf{H}(K_0||TS) + \ldots + \mathsf{H}(K_{(N-1)}||TS)\right) \bmod n_0$$

$$\tilde{k}_{all,TS} = (M_0(K'_1||0||TS) + \ldots + M_0(K'_1||(N-1)||TS)) \bmod n_1$$

Das EVU aggregiert die verschleierten Messdaten aller N Smart Meter und subtrahiert vom Ergebnis $D'_{all,TS}$ den Blindingwert $k_{all,TS}$, um den Gesamtverbrauch zu erhalten.
Zur Überprüfung der Integrität der aggregierten Daten summiert das EVU die MACs der Smart Meter und vergleicht sie mit dem MAC der akkumulierten Messdaten:

$$\begin{aligned} \sum_{i=0}^{N-1} MAC(D'_{i,TS}) &\stackrel{?}{=} MAC(D'_{all,TS}) \\ &\stackrel{?}{=} M_1(K'_0)D'_{all,TS} + \tilde{k}_{all,TS} \bmod n_1 \end{aligned}$$

Der Nachteil im Vergleich zum Verfahren von Kursawe ist der höhere Kommunikations- und Rechenaufwand. Für jedes Zeitintervall muss der KA die Summen der Blindingwerte $k_{all,TS}$ und $\tilde{k}_{all,TS}$ neu berechnen und zum EVU senden.

Das Verfahren lässt sich um einen Authentifizierungsschritt erweitern und ist aus denselben Gründen wie das Verfahren von Garcia und Jacobs weder robust noch flexibel.

Marmol et al. stellen einen weiteren Ansatz unter Benutzung eines KA vor [112]. Im Unterschied zum Verfahren von Vetter et al. übernimmt die Funktionalität des KA ein Smart Meter. Durch ein Node Election Protokoll kann der KA für jedes Zeitintervall TS neu ermittelt werden.
Beim Verfahren von Marmol et al. wählt jeder Smart Meter a einen geheimen Schlüssel $K_{a,TS}$ und sendet diesen verschlüsselt an den KA. Der KA summiert alle Schlüssel auf und überträgt das Ergebnis K_{all} an das EVU.
Bevor ein Smart Meter die Messdaten $D_{a,TS}$ an das EVU sendet, verschleiert er sie mit dem Schlüssel $K_{a,TS}$.

$$D'_{a,TS} = D_{a,TS} + K_{a,TS} \bmod n$$

Das EVU akkumuliert die verschleierten Messdaten aller Smart Meter und subtrahiert vom Ergebnis den Schlüssel K_{all}, um den Gesamtverbrauch für ein Zeitintervall mit dem Zeitstempel TS zu erhalten.
Damit das EVU verschleierte Messdaten vom selben Smart Meter nicht verknüpfen kann, wechseln die Smart Meter nach jeder Übertragung ihren geheimen Schlüssel, ohne dass sich der Schlüssel K_{all} ändert. Alle Smart Meter sind in einer ringförmigen Struktur strukturiert und jedem Smart Meter ist ein Smart Meter als Vorgänger und ein Smart Meter als Nachfolger zugeteilt. Die Smart Meter wählen eine Zufallszahl $k_{a,TS+1}$, die vom aktuellen Schlüssel $K_{a,TS}$ abgezogen wird und an den Nachfolger gesendet wird. Die Zufallszahl vom Vorgänger $k_{a-1,TS+1}$ wird zu dem neuen Schlüssel addiert. Der neue Schlüssel $K_{a,TS+1}$ für das Zeitintervall mit dem Zeitstempel $TS + 1$ berechnet sich damit wie folgt:

$$K_{a,TS+1} = K_{a,TS} - k_{a,TS+1} + k_{a-1,TS+1}$$

Marmol et al. geben eine Lösung an, wie Smart Meter, die keine Daten senden, von der weiteren Verarbeitung ausgeschlossen werden. Allerdings kann ein Angreifer mit vollständiger Kontrolle über einen Smart Meter wie beim Protokoll von Vetter einen hohen Verbrauch simulieren, ohne erkannt zu werden. Im Gegensatz zum Protokoll von Vetter et al. wird die Integrität der Messdaten nicht sichergestellt.

4 Crypto-Compiler

Umfangreiche Zero-Knowledge Protokolle haben den Nachteil, dass deren Implementierung fehleranfällig und ein Debuggen des Codes nur unter hohem zeitlichen Aufwand durchführbar ist [37]. Software-Ingenieure verfügen in der Regel nicht über die nötige Expertise, um Zero-Knowledge Protokolle sicher und effizient zu implementieren [7].
In Crowdsourcing-Szenarien muss das gleiche Protokoll mehrmals auf unterschiedlichen Hardware-Plattformen und für verschiedene Betriebssysteme implementiert werden. Beispielsweise werden im FCD-Szenario Smartphones mit Betriebssystemen wie Android, iOS oder Windows Phone betrieben. Die aufwendige Implementierung der Protokolle für eine Vielzahl von Geräten macht die Entwicklung sehr zeitaufwendig.

Im Rahmen dieser Arbeit ist eine *domänenspezifische Eingabesprache* und ein Compiler entwickelt worden, um den Entwicklungsaufwand zu minimieren.
Eine domänenspezifische Eingabesprache ist für ein bestimmtes Problemfeld (in diesem Fall die Beschreibung von Zero-Knowledge Proofs) entworfen und hat gegenüber einer vorhandenen (höheren) Programmiersprache wie z. B. C++, Java, VHDL den Vorteil, dass sich Protokollspezifikationen mit geringerem Aufwand beschreiben lassen können [172].

Der Anwender beschreibt die Protokollspezifikation in einer Eingabesprache, die auf plattformübergreifenden *Zwischencode* abgebildet wird. Der Zwischencode ist unabhängig von einer konkreten Hardware/Software-Umgebung und wird von einem Compiler in die Zielsprache übersetzt. Aus einer Protokollbeschreibung in der Eingabesprache lässt sich Code erzeugen, um die Protokolle z. B. auf PC-Plattformen, Mobiltelefonen oder Grafikkarten zu implementieren oder sie als integrierte Schaltkreise zu realisieren.

An den Compiler und die Eingabesprache werden zusammenfassend die folgenden Anforderungen gestellt:

- *Einfache Eingabesprache bei hoher Ausdrucksmächtigkeit*: Die Eingabesprache sollte über eine möglichst geringe Anzahl von Sprachelementen verfügen, mit der sich Protokolle hoher Komplexität beschreiben lassen können.
- *Compiler*: In der Eingabesprache beschriebene Programme und Protokollspezifikationen werden in eine Zielsprache, die unabhängig von der Eingabesprache ist, übersetzt.
- *Interaktiver Modus*: Der Anwender soll in der Lage sein, den Compiler wie einen Interpreter interaktiv zu nutzen, um z. B. einen RAD-Ansatz zu verfolgen [113]. Eingaben in der Eingabesprache werden direkt ausgeführt und das Ergebnis ausgegeben.
- *Modularität*: Der Compiler soll eine einfache Austausch- und Erweiterbarkeit gewährleisten.
- *Benchmarking*: Protokolle werden häufig in Abhängigkeit von Parametern wie z. B. der Moduluslänge evaluiert. Aus diesem Grund müssen die kompilierten Programme über eine Schnittstelle verfügen, über die Protokollparameter geändert werden können. Dabei ist zu beachten, dass Änderungen eines Parameters weitere Änderungen nach sich ziehen können. Ändert sich z. B. der Faktor eines RSA-Modulus, muss der Modulus neu berechnet werden.

4.1 Formale Grammatiken und Sprachen

Jede Eingabe- bzw. Programmiersprache lässt sich durch eine *formale Grammatik* beschreiben, die den folgenden Definitionen und Gesetzen folgt [94].

Definition 4.1.1 (Alphabet) *Ein* Alphabet Γ *ist eine endliche, nicht-leere Menge von* Symbolen*. Symbole können nicht weiter unterteilt werden.*

Definition 4.1.2 (String) *Ein* String *aus einem Alphabet* Γ *ist eine endliche Folge von Symbolen aus der Menge* Γ*.*

Die Menge der möglichen Strings aus dem Alphabet Γ wird mit Γ^* bezeichnet. Die Anzahl der Symbole $|\mathtt{x}|$, aus denen ein String $\mathtt{x}$ besteht, ist die Länge des Strings. Ein String, der keine Symbole enthält, ist der leere String ϵ mit der Länge 0.

Definition 4.1.3 (Konkatenation von Strings) *Seien* $\mathtt{x}_0 = \mathtt{a}_0\mathtt{a}_1 \ldots \mathtt{a}_N$ *und* $\mathtt{x}_1 = \mathtt{b}_0\mathtt{b}_1 \ldots \mathtt{b}_N$ *zwei Strings. Die* Konkatenation *von* $\mathtt{x}_0$ *und* $\mathtt{x}_1$*, welche durch* $\mathtt{x}_0\mathtt{x}_1$ *dargestellt wird, ist der String* $\mathtt{a}_0\mathtt{a}_1 \ldots \mathtt{a}_N\mathtt{b}_0\mathtt{b}_1 \ldots \mathtt{b}_N$*.*

Es gilt für jeden String $\mathtt{x}$, $\epsilon\mathtt{x} = \mathtt{x}\epsilon = \mathtt{x}$. $\mathtt{x}^i$ bezeichnet die i-fache Konkatenation von $\mathtt{x}$.

Definition 4.1.4 (Sprache) *Es sei* Γ *ein Alphabet, dann ist eine Teilmenge von* Γ^* *die* Sprache $\mathtt{L}$*. Die Elemente einer Sprache werden als* Wörter *bezeichnet.*

Sprachen können wie Strings konkateniert werden.

Definition 4.1.5 (Konkatenation von Sprachen) *Seien* $\mathtt{L}_0$ *und* $\mathtt{L}_1$ *zwei Sprachen definiert über dem Alphabet* Γ*. Die Konkatenation von* $\mathtt{L}_0$ *und* $\mathtt{L}_1$*, welche durch* $\mathtt{L}_0\mathtt{L}_1$ *angegeben wird, ist die Sprache* $\{\mathtt{x}_0\mathtt{x}_1 | \mathtt{x}_0 \in \mathtt{L}_0, \mathtt{x}_1 \in \mathtt{L}_1\}$*.*

Sprachen sind Mengen, so dass sich Mengenoperationen wie Vereinigung, Durchschnitt und Komplementbildung auf Sprachen anwenden lassen können.

Definition 4.1.6 (Kleenesche Hülle) *Es sei* $\mathtt{L}$ *eine Sprache definiert über dem Alphabet* Γ*. Des Weiteren gilt* $\mathtt{L}^0 = \epsilon$ *und* $\mathtt{L}^i = \mathtt{L}\mathtt{L}^{i-1}$ *für* $i \geq 1$*. Die* Kleenesche Hülle $\mathtt{L}^*$ *ist die Sprache*

$$\mathtt{L}^* = \bigcup_{i \geq 0} \mathtt{L}^i.$$

In der Praxis ist die Beschreibung einer Sprache durch die Angabe aller gültigen Wörter häufig zu komplex. Aus diesem Grund werden Sprachen in der Regel durch reguläre Ausdrücke oder Grammatiken beschrieben.

Definition 4.1.7 (Reguläre Ausdrücke) *Reguläre Ausdrücke über einem Alphabet* Γ *und die durch reguläre Ausdrücke beschriebenen Sprachen sind wie folgt definiert.*

1. *Das Symbol* $\emptyset$ *ist ein regulärer Ausdruck und beschreibt die* leere Sprache*.*
2. *Das Symbol* ϵ *ist ein regulärer Ausdruck und bezeichnet eine Sprache, dessen einziges Mitglied der leere String* ϵ *ist.*
3. *Für jedes Symbol* $\mathtt{a} \in \Gamma$ *ist* $\mathtt{a}$ *ein regulärer Ausdruck und repräsentiert die Sprache* $\mathtt{a}$ *mit dem einzigen Mitglied* $\mathtt{a}$*.*
4. *Seien* $\mathtt{r}$ *und* $\mathtt{s}$ *reguläre Ausdrücke, die die Sprachen* $\mathtt{L}(\mathtt{r})$ *und* $\mathtt{L}(\mathtt{s})$ *repräsentieren. Die regulären Ausdrücke* $(\mathtt{r}|\mathtt{s})$*,* $(\mathtt{rs})$ *und* $(\mathtt{r}^*)$ *beschreiben die*

Sprachen $\mathtt{L}(\mathtt{r}) \cup \mathtt{L}(\mathtt{s})$, $\mathtt{L}(\mathtt{r})\mathtt{L}(\mathtt{s})$ *und* $\mathtt{L}(\mathtt{r})^*$.

Alle Operatoren sind *linksassoziativ*. Der * Operator hat die höchste Präzedenz, gefolgt von der Konkatenation und dem | Operator mit der niedrigsten Präzedenz. Durch Klammern kann die Präzedenz geändert werden.
Reguläre Ausdrücke werden in der Praxis um zusätzliche Operatoren wie z. B. den ? Operator erweitert, lassen sich aber immer auf die obige Definition zurückführen (z. B. $\mathtt{L}(\mathtt{r}?) \stackrel{\wedge}{=} \mathtt{L}(\mathtt{r}) \cup \mathtt{L}(\epsilon)$) [2].

Sprachen, die durch reguläre Ausdrücke beschrieben werden können, heißen *reguläre Sprachen*, wobei nicht jede Sprache regulär ist. Jede Sprache kann dagegen durch eine *Grammatik* beschrieben werden [94].

Definition 4.1.8 (Grammatik) *Ein Tupel* $(\Gamma, \mathtt{V}, \mathtt{S}, \mathtt{P})$ *heißt* Grammatik, *falls folgende Eigenschaften erfüllt sind.*

1. Γ *ist eine endliche nicht-leere Menge, die auch* Terminal-Alphabet *genannt wird. Die Elemente von* Γ *heißen* Terminalsymbole.
2. $\mathtt{V}$ *ist eine endliche nicht-leere Menge und es gilt* $\Gamma \cap \mathtt{V} = \emptyset$. *Die Elemente von* $\mathtt{V}$ *heißen* Nicht-Terminalsymbole *oder* Variablen.
3. $\mathtt{S} \in \mathtt{V}$ *ist ein spezielles Nicht-Terminalsymbol, das* Startsymbol.
4. $\mathtt{P}$ *ist eine endliche Menge von Regeln der Form*

   $$\alpha \rightarrow \beta$$

   mit $\alpha \in (\Gamma \cup \mathtt{V})^*\mathtt{V}(\Gamma \cup \mathtt{V})^*$ *und* $\beta \in (\Gamma \cup \mathtt{V})^*$, *d. h.* α *ist ein String von Terminal- und Nicht-Terminalsymbolen mit mindestens einem Nicht-Terminalsymbol und* β *ist ein String von Terminal- und Nicht-Terminalsymbolen.*

Formale Grammatiken lassen sich in vier Typen unterteilen, die sogenannte *Chomsky-Hierarchie* [94].

Definition 4.1.9 (Chomsky-Hierarchie) *Es sei* $\mathtt{G} = (\Gamma, \mathtt{V}, \mathtt{S}, \mathtt{P})$ *eine Grammatik mit* $\alpha \in (\Gamma \cup \mathtt{V})^*\mathtt{V}(\Gamma \cup \mathtt{V})^*$ *und* $\beta \in (\Gamma \cup \mathtt{V})^*$.

1. $\mathtt{G}$ *ist eine* Typ-$\mathtt{0}$ *oder eine* uneingeschränkte Grammatik.
2. $\mathtt{G}$ *ist eine* Typ-1 *oder* kontextsensitive Grammatik, *wenn für jede Regel* $\alpha \rightarrow \beta \in \mathtt{P}$ *gilt:* $|\alpha| \leq |\beta|$. *Falls das Startsymbol nicht in der rechten Seite der Regeln enthalten ist, kann zusätzlich die Regel* $\mathtt{S} \rightarrow \epsilon$ *gelten.*
3. $\mathtt{G}$ *ist eine* Typ-2 *oder* kontextfreie Grammatik, *wenn für jede Regel* $\alpha \rightarrow \beta \in \mathtt{P}$ *gilt:* $|\alpha| = 1$. *Daraus folgt nach der Definition, dass* α *ein Nicht-Terminalsymbol sein muss.*

4. $\mathtt{G}$ *ist eine* Typ-3 *oder* reguläre Grammatik*, wenn jede Regel für* $\mathtt{A}, \mathtt{B} \in \mathtt{V}$ *und* $\mathtt{a} \in \Gamma$ *eine der drei folgenden Formen hat:*

$$\mathtt{A} \rightarrow \mathtt{aB},\ \mathtt{A} \rightarrow \mathtt{a},\ \mathtt{A} \rightarrow \epsilon$$

Eine Sprache, die durch eine Typ-i Grammatik beschrieben wird, heißt *Typ-i Sprache* ($i \in \{0, \ldots, 3\}$). Es gelten die folgenden Zusammenhänge [94]:

Satz 4.1.1 *Die Klasse der Typ-3 Sprachen und die Klasse der regulären Sprachen sind gleich.*

Satz 4.1.2 *Für jedes* $i = 0, 1, 2$ *gilt, dass die Klasse der Typ-i Sprachen die Klasse der Typ-*$(i + 1)$ *Sprachen enthält.*

Eine Typ-1 Sprache wird auch als *kontextsensitive Sprache* (engl. CSL (Context Sensitive Language)) und eine Typ-2 Sprache als *kontextfreie Sprache* (engl. CFL (Context Free Language)) bezeichnet.

4.1.1 Kontextfreie Grammatiken

Es sei $\mathtt{G} = (\Gamma, \mathtt{V}, \mathtt{S}, \mathtt{P})$ eine kontextfreie Grammatik und $\mathtt{L}(\mathtt{G})$ eine Sprache, die durch $\mathtt{G}$ beschrieben wird. Werden Grammatiken in einem Compiler verwendet, sind zwei Probleme von besonderem Interesse [94].

Membership Problem

Definition 4.1.10 (Membership Problem) *Gegeben sei ein String* $\mathtt{x} \in \Gamma$*. Die Lösung des* Membership Problems *ist die Antwort auf die Frage, ob* $\mathtt{x} \in \mathtt{L}(\mathtt{G})$ *gilt.*

Das Lösen des *Membership Problems* beantwortet die Frage, ob ein Eingabestring zu einer Sprache gehört. Bei kontextfreien Grammatiken ist das Lösen des Membership Problems ein elementarer Bestandteil der lexikalischen Analyse [2]. Dabei werden durch den lexikalischen Scanner (*Lexer*) Zeichen im Quellcode zu *Lexemen* zusammengefasst. Zu jedem Lexem gibt der Lexer ein Terminalsymbol aus, dass in diesem Zusammenhang auch als *Token* bezeichnet wird[42]. Token und Lexeme repräsentieren die Bezeichner, Schlüsselwörter und Operatoren der Eingabesprache.

[42] Lexeme wären z. B. Eingabesymbole wie `-1`, `123`, `123.45`, die alle durch einen Token wie z. B. `NUMBER` repräsentiert werden.

Satz 4.1.3 *Das Membership Problem ist allgemein unentscheidbar für Typ-0 Grammatiken und entscheidbar für Grammatiken vom Typ-1. Dabei lässt sich für Typ-1 und Typ-2 Grammatiken eine Lösung in polynomialer und bei Typ-3-Grammatiken in linearer Laufzeit finden.*

Das Membership Problem ist bei regulären Grammatiken in linearer Laufzeit lösbar. Deshalb wird die Grammatik eines Lexers in der Regel als Typ-3 Grammatik angegeben [152].

Parsing Problem

Definition 4.1.11 (Parsing Problem) *Gegeben sei ein String* $\mathtt{x} \in \mathtt{L}(\mathtt{G})$. *Das* Parsing Problem *ist die Aufgabe, durch Anwendung der Regeln aus* $\mathtt{P}$ *vom Startsymbol* $\mathtt{S}$ *nach* $\mathtt{x}$ *abzuleiten.*

Ein *Parser* versucht das Parsing Problem durch Anwendung der Ableitungsregeln zu lösen (*syntaktische Analyse*). Dieser Vorgang ist äquivalent zur Erstellung eines *Parsebaumes*, der die grammatische Struktur der Token aus dem Eingabestroms wiedergibt[43]. Die inneren Knoten des Parsebaumes bestehen aus Nicht-Terminalsymbolen und die Blätter sind entweder der leere String ϵ oder Terminalsymbole. Die Wurzel des Baumes entspricht dem Startsymbol $\mathtt{S}$. Wird der Parser in einem Compiler verwendet, führt der Parser bei der Anwendung der Ableitungsregeln *semantische Aktionen* aus. Diese semantischen Aktionen erzeugen in einem Compiler den Zwischencode, der in die Zielsprache übersetzt wird [2].

Parser lassen sich als *Top-down-* oder *Bottom-up-Parser* klassifizieren. Bei einem Top-down Parser werden (virtuell) Ableitungsbäume von der Wurzel zu den Blättern erstellt. Bei einem Bottom-up Parser ist es umgekehrt.
Mit universellen Parsern wie dem CYK (Cocke-Younger-Kasami) Algorithmus (Bottom-up Parser) [77] oder dem Algorithmus von Earley (Top-down Parser) [57] lassen sich alle Eingaben parsen, die zu einer kontextfreien Sprache gehören. Die Parser haben eine Worst-Case Laufzeit von $\mathcal{O}(\mathtt{n}^3)$, wobei $\mathtt{n}$ der Länge des Strings $\mathtt{x}$ entspricht.
In der Praxis kommen üblicherweise LL- bzw. LR-Parser zum Einsatz, da mit ihnen in der Regel eine lineare Laufzeit erreicht werden kann. Allerdings eignen sie sich nur zum Parsen von Sprachen, die durch Untergruppen kontextfreier Grammatiken beschrieben werden [22].

Der *LL-Parser* ist ein Top-down-Parser, der aus einem Eingabestring, einem Stack und einer Parsing-Tabelle besteht. Die Elemente des Eingabestrings entsprechen Token, d. h. Terminalsymbolen, während die Elemente auf dem Stack Terminal-

[43] Ein Parser erzeugt nicht zwangsläufig einen Parsebaum.

und Nicht-Terminalsymbolen entsprechen. Die Parsing-Tabelle gibt an, welche Ableitungsregel aus P in Abhängigkeit des aktuellen Tokens im Eingabestring auf das oberste Stackelement angewendet werden soll.
Ein LL-Parser parst den Eingabestring von links nach rechts, was durch die Angabe des ersten „L“ in „LL“ zum Ausdruck gebracht wird. Das zweite „L“ bedeutet, dass der Parser eine *Linksreduktion* vornimmt, d. h. dass der Parser immer das am weitesten links stehende Nicht-Terminalsymbol ersetzt. Schaut ein LL-Parser beim Parsen k Tokens[44] voraus, wird dieser als *LL(k)-Parser* bezeichnet und verarbeitet Sprachen, die durch die sogenannten *LL(k)-Grammatiken* beschrieben werden. Ein LL-Parser hat eine lineare Laufzeit, wenn der Parser für eine kontextfreie LL(k)-Grammatik immer durch Untersuchung der nächsten k Token in der Eingabe (k ist begrenzt) entscheidet, welche Ableitungsregel zu wählen ist [2]. LL-Parser sind einfach im Aufbau und lassen sich mithilfe von Parser-Generatoren wie z. B. ANTLR automatisch erzeugen [133] [164].

LR-Parser gehören zur Gruppe der Bottom-up-Parser. Sie sind im Gegensatz zu LL-Parsern komplexer im Aufbau, haben aber den Vorteil, dass Syntaxfehler früher erkannt werden können [2]. Im Gegensatz zum LL-Parser nimmt der LR-Parser eine *Rechtsreduktionen* vor, d. h. der Parser versucht immer, das am weitesten rechts stehende Nicht-Terminalsymbol mithilfe einer Ableitungsregel zu ersetzen. Wie beim LL-Parser kann der LR(k)-Parser im Eingabestring k Token vorausschauen, um die richtige Ableitungsregel zu finden [40].
Der LR-Parser hat einen Eingabestring, eine Parsingtabelle mit Ableitungsregeln und einen Stack. Das oberste Element im Stack repräsentiert den aktuellen Zustand des Parsers. Die zwei wesentlichen Schritte sind der Zustandswechsel in Abhängigkeit des nächsten Tokens im Eingabestring (*shift*) und das Reduzieren der Elemente auf dem Stack in Abhängigkeit der Ableitungsregeln (*reduce*). Im Reduce-Schritt werden so viele Elemente vom Stack genommen, wie eine bestimmte Regel Symbole auf der rechten Seite hat. Danach wechselt der Parser seinen Zustand und legt diesen auf den Stack. LR-Parser können mehr Sprachen parsen als LL-Parser, da später entschieden wird, ob es zu einer Eingabe eine passende Produktionsregel gibt [100].
Die ursprüngliche Version eines LR-Parsers von Donald Knuth zeichnet sich durch einen hohen Speicherverbrauch aus. Optimierungen wie der *LALR-Parser (Lockahead LR-Parser)* oder *SLR-Parser (Simple LR-Parser)* sind spezielle LR-Parser, die effizienter sind [52].

Bei LR-Parsern können zwei Arten von Konflikten auftreten:

1. *Shift-Reduce-Konflikt*: Der Parser kann entweder das Symbol aus der Eingabe nehmen oder die obersten Elemente des Stacks entfernen.
2. *Reduce-Reduce-Konflikt*: Eine Reduktion kann durch mindestens zwei Produktionsregeln erfolgen.

[44] k wird als *Lockahead* bezeichnet.

Ein nicht-deterministischer Parser wie der *Tomita Parser* für kontextfreie Grammatiken löst Konflikte durch Verzweigungen bzw. Klonen des Parsers [169]. Jeder geklonte Parser endet entweder in einem Parser-Fehler und wird beendet oder vereint sich mit einem anderen Parser, weil beide Parser denselben Reduktionsschritt ausführen. Der Tomita Parser stellt eine Verallgemeinerung des LR(k)-Parsers dar und wird deswegen auch *GLR(k)-Parser* (Generalized LR(k)-Parsers) genannt [165]. Die Ausführungszeit von deterministischen LR-Parsern ist für kontextfreie Sprachen linear. Da GLR-Parser nicht-deterministisch sind, ist die Ausführungszeit des Parsers aufgrund der Verzweigungen nicht-linear. Können allerdings Konflikte vom Parser schnell gelöst werden, wirkt sich die Verwendung eines GLR-Parsers kaum auf die Ausführungszeit aus und die Vorteile wie z. B. die Verwendung einer einfacheren Grammatik überwiegen.

4.2 Konzept

Bevor die genaue Funktionsweise des Crypto-Compilers erläutert wird, soll in diesem Abschnitt der generelle Aufbau des Crypto-Compilers beschrieben werden (siehe Abbildung 4.1). Der Compiler lässt sich in die drei Komponenten *Frontend*, *Crypto-Framework* und *Backend* unterteilen.

Der Crypto-Compiler wurde unter anderem im Rahmen mehrerer Abschlussarbeiten an der Technischen Universität Harburg umgesetzt [81] [75] [79].

4.2.1 Crypto-Framework

Das Crypto-Framework ist unabhängig vom Rest des Compilers und kann ohne Front- und Backend verwendet werden. Es bildet den Kern des Compilers und stellt C++ Klassen bereit, um Protokolle, die auf zahlentheoretischen Problemen basieren, leichter implementieren zu können. Sobald die Klassen instanziiert sind, werden sie als *Crypto-Objekte* bezeichnet.

Crypto-Objekte

Crypto-Objekte können entweder vom Programmierer oder durch das Frontend erzeugt werden. Zurzeit sind zehn Typen von Crypto-Objekten implementiert, um die Protokolle in Kapitel 3 sowie weitaus komplexere Protokolle beschreiben zu können. Für die Schilderung der Funktionalität soll an dieser Stelle auf die Erläuterung der Eingabesprache in Kapitel 4.3 verwiesen werden, da jedes Sprachelement der Eingabesprache auf ein Crypto-Objekt abgebildet wird.

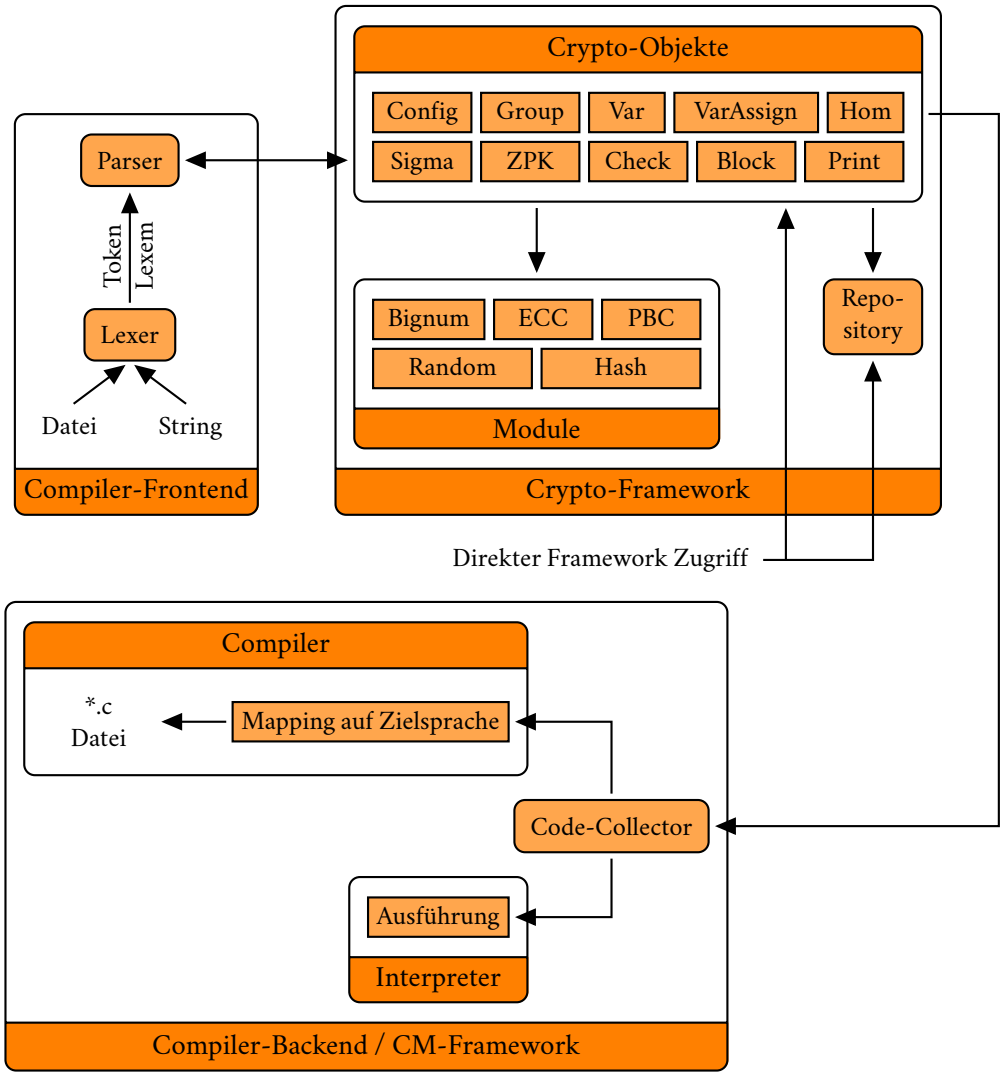


Abbildung 4.1: Aufbau des Crypto-Compilers.

Crypto-Objekte sind von anderen Crypto-Objekten abhängig. Beispielsweise enthält ein Crypto-Objekt, welches eine Variable abstrahiert, eine Referenz auf ein Crypto-Objekt, das eine Gruppe modelliert. Problematisch ist das Löschen von Crypto-Objekten. Es kommt zu einem Speicherleck, wenn z. B. das Gruppen-Objekt gelöscht wird und das Variablen-Objekt noch existiert.
Zur Vermeidung besitzt jedes Crypto-Objekt einen *Referenzzähler*. Referenziert ein Crypto-Objekt ein anderes Crypto-Objekt, wird der Referenzzähler des zweiten Objektes erhöht. Beim Löschen eines Crypto-Objektes wird der Referenzzähler aller referenzierter Objekte um eins dekrementiert. Ist der Referenzzähler Null, wird das Objekt automatisch entfernt. Durch dieses Konzept konnte ein einfacher Garbage Collector realisiert und die Gefahr von Speicherlecks reduziert werden [95].

Alle entweder durch den Programmierer oder den Parser erzeugten Crypto-Objekte werden im *Repository* gespeichert. Da jedes Crypto-Objekt über eine eindeutige ID verfügt, kann später direkt durch den Programmierer auf die Crypto-Objekte zugegriffen werden.

Module

Die Funktionalität der Crypto-Objekte basiert auf fünf austauschbaren Komponenten, den *Modulen*. Während die Schnittstelle der Module zum Rest des Crypto-Frameworks fest definiert ist, ist die Funktionalität der Module austauschbar. Dadurch wird eine maximale Flexibilität gewährleistet.

1. *Bignum*: In der Regel übersteigt bei kryptographischen Protokollen die Bitlänge der Zahlen die Wortbreite des Prozessors (typischerweise 32 oder 64 Bit). Aus diesem Grund müssen die arithmetischen Langzahloperationen wie z. B. Addition oder Multiplikation vor der Ausführung in mehrere Teiloperationen zerlegt werden. Das Bignum-Modul abstrahiert diese Langzahloperationen. Daneben stellt es weitere Funktionalitäten wie z. B. Primzahltests zur Verfügung.
   Das Bignum-Modul basiert auf der MPIR Bibliothek [71]. Die Bibliothek unterstützt eine Vielzahl von Hardware-Plattformen und enthält hochoptimierten Code für eine Vielzahl von Prozessorarchitekturen.

2. *ECC*: Basierend auf dem Bignum-Modul wurde im Rahmen einer Studienarbeit die Bibliothek „SimpleECC“ zur Verwendung elliptischer Kurven in C/C++ Projekten entwickelt [79]. Das ECC-Modul (Elliptic Curve Cryptography Modul) agiert als Schnittstelle zwischen dem Crypto-Framework und der SimpleECC Bibliothek.

3. *PBC*: Das PBC-Modul (Pairing Based Cryptography Modul) stellt PBC-Operationen für Pairing Crypto-Objekte zur Verfügung. Das Modul abstrahiert in erster Linie die Funktionalität der libPBC Bibliothek [14].

4. *Hash*: Nicht-interaktive Sigmaprotokolle erfordern die Implementierung von Hashfunktionen. Das Hash-Modul stellt eine einheitliche Schnittstelle zur Verwendung kryptographischer Hashfunktionen bereit. Im Rahmen dieser Arbeit sind die Hashfunktionen der OpenSSL-Bibliothek MD4, MD5, SHA-1, SHA-224, SHA-256, SHA-384 und SHA-512 implementiert [130].

5. *Random*: Das Random-Modul stellt eine einheitliche Schnittstelle zur Generierung von Zufallszahlen zur Verfügung. Neben Funktionen, um Zufallszahlen aus beliebigen Intervallen zu erzeugen, kann die Art der Zufallszahlen (Primzahlen, sichere Primzahlen etc.) gewählt werden. Das Random-Modul basiert auf dem Pseudozufallszahlengenerator der MPIR Bibliothek. Der Generator wird mit einem Wert (Seed) initialisiert und liefert daraufhin (Pseudo-) Zufallszahlen für das Crypto-Framework. Der Seed kann entweder durch das Betriebssystem oder durch eine spezielle Smartcard erzeugt werden. Die Erzeugung des Seeds durch die Smartcard, hat den Vorteil, dass „echte“ Zufallszahlen als Seed verwendet werden können [154].

4.2.2 Compiler-Frontend

Das Frontend des Compilers besteht aus Lexer und Parser und erweitert das Crypto-Framework dahingehend, dass Crypto-Objekte automatisch durch eine Eingabesprache erzeugt werden können.

Lexer

Der Lexer fasst Symbole aus einem Eingabebuffer zu Lexeme und Token zusammen und übergibt sie an den Parser. Die Daten werden dabei entweder aus einer ASCII-Datei oder einem String gelesen und in den Eingabebuffer geschrieben. Der Lexer entfernt Kommentare, sowie alle Zeichen, die keinerlei semantische Bedeutung haben, wie z. B. Leerzeichen, Tabulatoren und Zeilenumbrüche.

Der Lexer testet den Strom von Zeichen aus dem Eingabebuffer auf Bezeichner, Schlüsselwörter, Zahlen, Zeichenketten in Anführungszeichen und sonstige Zeichenketten.

- *Bezeichner*: Zur eindeutigen Identifikation von Objekten in der Eingabesprache (z. B. Variablen) dienen Bezeichner, die aus Groß-/Kleinbuchstaben (`A - Z` und `a - z`), Zahlen (`0 - 9`) und dem Unterstrich (_) bestehen dürfen. Ein Bezeichner darf nicht mit einer Zahl beginnen.

- *Schlüsselwörter*: Schlüsselwörter wie z. B. `ConfigParam` oder `export` haben in der Eingabesprache eine bestimmte Bedeutung und dürfen nicht als Bezeichner verwendet werden.
- *Zahlen*: Zahlen werden entweder in dezimaler oder hexadezimaler Notation angegeben, wobei hexadezimale Zahlen das Präfix `0x` haben müssen. Zur Darstellung negativer Zahlen kann das Minuszeichen (-) verwendet werden. Die Größe einer Zahl ist nur durch den zur Verfügung stehenden Arbeitsspeicher begrenzt.
- *Zeichenketten*: Der Lexer unterscheidet zwischen Zeichenketten in Anführungszeichen und normalen Zeichenketten. Der erste Fall beschreibt eine beliebige Folge von ASCII Zeichen, die durch " begrenzt wird. Normale Zeichenketten dürfen dagegen nur Groß-/Kleinbuchstaben, Ziffern und den Unterstrich enthalten.

In dieser Arbeit wird das Programm GNU Flex verwendet, um den Lexer zu erstellen [139]. Die Beschreibung des Lexers erfolgt in Form von regulären Ausdrücken (Typ-3 Grammatik). Nach Satz 4.1.3 ist die Laufzeit des Lexers damit linear.

Parser

Die manuelle Programmierung ist bei einer umfangreichen Eingabesprache, wie sie in dieser Arbeit verwendet wird, sehr komplex und fehleranfällig. Änderungen an der Grammatik führen zu aufwendigen Änderungen am Parser. Aus diesem Grund wird Bison, ein Parsergenerator eingesetzt, der aus einer kontextfreien Grammatik einen LR-Parser generiert [165]. Standardmäßig erzeugt Bison LALR(1)-Parser, aber aufgrund von Mehrdeutigkeiten in der Grammatik wird ein GLR-Parser generiert (siehe Kapitel 4.3).

Während des Parsens fordert der Parser vom Lexer Token an. Kann der Parser eine Grammatikregel anwenden, kommuniziert der Parser mit dem Crypto-Framework, um z. B. neue Crypto-Objekte anzulegen. Neben der Grammatik überprüft der Parser weitere Bedingungen[45] (siehe Kapitel 4.3).

4.2.3 Compiler-Backend

Soll die Funktionalität der Crypto-Objekte in eine Zielsprache übersetzt werden, ist das Compiler-Backend notwendig. Die Funktionalität der Crypto-Objekte wird durch das Backend als plattformunabhängiger Zwischencode in einer Datenstruktur beschrieben, der dann in einer beliebigen Zielsprache ausgegeben

[45] z. B. ob ein Bezeichner bereits verwendet wurde.

werden kann. Neben dem Compiler-Modus unterstützt der Crypto-Compiler einen *Interpreter-Modus*. In diesem Modus wird die Funktionalität der Crypto-Objekte direkt ausgeführt und nicht wie im Compiler-Modus nur aufgesammelt. Dadurch kann der Crypto-Compiler interaktiv verwendet werden.
Das Backend im Crypto-Compiler wird durch das *CMF (Code Management Framework)* realisiert, dass ein eigenständiges Projekt darstellt und in Kapitel 4.4 beschrieben wird.

4.3 Eingabesprache

Die Eingabesprache ist in zehn Sprachelemente untergliedert, wobei jedes Sprachelement durch ein Crypto-Objekt repräsentiert wird. Das besondere an der Eingabesprache ist, dass viele Sprachelemente sich durch *Tupel* gruppieren lassen können. Die Komponenten eines Tupels, die sogenannten *Childs* können wieder Tupel sein und aus weiteren Childs bestehen. Hat ein Sprachelement keine Childs, wird es als *atomar* bezeichnet. Die Verwendung von Tupeln hat zwei Vorteile:

1. *Vereinfachte Beschreibung komplexer Zero-Knowledge Proofs*: Die Objekte zur Beschreibung der Σ-Protokolle in Kapitel 2.5 sind entweder Tupel-Objekte (bei Σ^{AND}-, Σ^{OR}-Protokollen) oder atomare Objekte (bei Σ^{ϕ}-, Σ^{GSP}-Protokollen). Dadurch ist es möglich, eine beliebig komplexe Struktur von Prädikaten zu beschreiben, die jeweils mit „und"- und „oder"-Verknüpfungen kombiniert werden können.
   Die Commitments, Geheimnisse und Responses eines atomaren Σ^{ϕ}- oder Σ^{GSP}-Protokoll werden im Crypto-Framework durch Variablen-Objekte abstrahiert. Eine Variable ist entweder ein Tupel oder sie ist atomar. Das ermöglicht die Implementierung von Zero-Knowledge Proofs mit komplexen abhängigen Prädikaten, die durch ein logisches „und" miteinander verknüpft sind. Das soll anhand des Σ^{ϕ}-Protokolls in Abbildung 2.5 erläutert werden.
   Das Geheimnis w, Zufallskomponente t und Response s werden im Crypto-Framework jeweils als Tupel-Variable mit zwei Childs beschrieben. Die Homomorphismen ϕ_0, ϕ_1, ϕ_2 werden als Tupel-Homomorphismus mit drei Childs zusammengefasst. Durch die Verwendung der Tupel-Schreibweise reduziert sich damit das Protokoll auf das Σ^{ϕ}-Protokoll in Abbildung 2.2.

2. *Kurzschreibweise*: Werden Operationen auf einen Tupel angewendet, so werden diese auch auf die einzelnen Childs angewendet.

Kommentare in der Eingabesprache entsprechen Kommentaren in Programmiersprachen wie z. B. C++ und werden vom Lexer ignoriert. Durch `//` wird die Eingabe bis zum Ende der Zeile ignoriert. Die Zeichen `/*` und `*/` umschließen mehrzeilige Kommentare.

Die *EBNF (Erweiterte Backus-Naur-Form)* [85] ist ein Standard zur formalen Beschreibung kontextfreier Grammatiken und dient zur Darstellung der Grammatik des Crypto-Compilers in dieser Arbeit. In der EBNF werden die Produktionsregeln als Folge von Terminal- und Nicht-Terminalsymbolen angegeben, die jeweils mit einem Semikolon abgeschlossen werden. Terminalsymbole werden in Hochkommata (`'Terminalsymbol'`) gesetzt. Nicht-Terminalsymbole sind rekursiv dargestellt. Zur Beschreibung von Alternativen dient das Zeichen `|` und `[ .. ]` stellt eine optionale Folge von Symbolen dar.
Zum besseren Verständnis wird in diesem Kapitel eine vereinfachte Darstellung der Grammatik gewählt. Die vollständige Darstellung der Produktionsregeln ist in Anhang A.1 zu finden.

4.3.1 Konfigurationsparameter

Syntax:

```
stmt   = [ 'export' ] 'ConfigParam' IDENT [ ':' '=' number ]
       | [ 'export' ] 'ConfigParam' IDENT [ ':' '=' QUOTED_STR ] ;
number = POSNUMBER
       | NEGNUMBER;
```

Beispiel:

```
export ConfigParam p := 17;
export ConfigParam s := "Ein String";
G1 := Zadd(p, prime);
G2 := Zadd(17, prime);
G3 := ECprime(0x17, 0x1, 0x0, s, 0x17, 0x1);
```

Crypto-Objekte werden durch *Konfigurationsparameter* konfiguriert, die vor ihrer Verwendung deklariert werden müssen. Die Deklaration erfolgt in der Eingabesprache durch das Schlüsselwort `ConfigParam`, gefolgt von einem Bezeichner (`IDENT`) und einem optionalen Initialisierungswert, der entweder eine Zahl oder ein String in Anführungszeichen sein kann. Eine Zahl wird als Dezimal- oder Hexadezimalzahl angegeben und kann negativ sein.
Der Initialisierungswert legt den Typ eines Konfigurationsparameters (Zahl oder String) fest. Speichert der Konfigurationsparameter Strings, muss der Initialisierungswert in Anführungszeichen angegeben werden. Ist kein Initialisierungswert angegeben, wird der Konfigurationsparameter mit Null initialisiert und zur Speicherung von Zahlen konfiguriert.
Konfigurationsparameter haben im Gegensatz zu allen anderen Anweisungen einen ausschließlich globalen Gültigkeitsbereich und dürfen nur außerhalb von Blöcken definiert werden (siehe Kapitel 4.3.9). Das optionale Präfix `export` ist nur für das Compiler-Backend relevant und stellt eine Funktion im Zielcode

bereit, um den Konfigurationsparameter durch einen Funktionsaufruf ändern zu können.

Neben Konfigurationsparametern, die einen Bezeichner haben, gibt es anonyme Konfigurationsparameter ohne Bezeichner, die vom Parser direkt erzeugt werden. Anonyme Konfigurationsparameter sind konstant und können nicht geändert werden. Das obige Beispiel veranschaulicht die Verwendung von Konfigurationsparametern zur Definition von Gruppen (siehe Kapitel 4.3.2).
In beiden Gruppendefinitionen werden Konfigurationsparameter zur Beschreibung des Modulus verwendet. Im ersten Fall (`G1`) ist der Modulus ein Konfigurationsparameter. Bei der Deklaration von `G2`, ist der Modulus dagegen als Zahl angegeben und der Parser wandelt die Zahl automatisch in einen anonymen Konfigurationsparameter um.

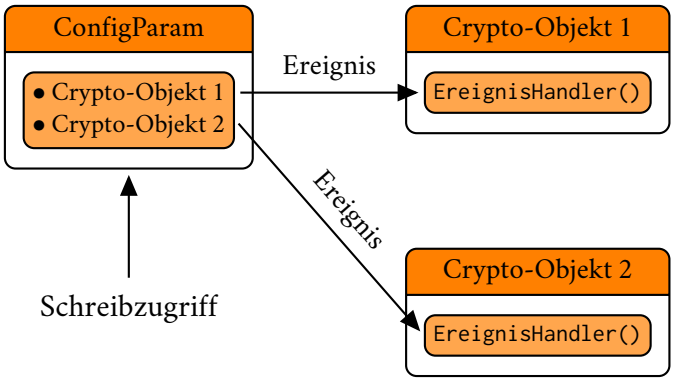


Abbildung 4.2: Wird ein Konfigurationsparameter geändert, werden alle Crypto-Objekte, die den Parameter verwenden, über die Änderung informiert.

Eine Besonderheit von Konfigurationsparametern ist ihre automatische Kopplung an Crypto-Objekte. Jeder Konfigurationsparameter enthält eine Liste mit Crypto-Objekten, die den Konfigurationsparameter verwenden. Wird ein Konfigurationsparameter geändert, wird die Änderung jedem Crypto-Objekt auf der Liste mitgeteilt (siehe Abbildung 4.2). Ein Ereignis-Handler in den jeweiligen Objekten aktualisiert das Crypto-Objekt auf Basis der Parameteränderung und überprüft, ob die Objekteigenschaften noch erfüllt sind. Wird die Funktionalität der Crypto-Objekte kompiliert, ist die Funktionalität des Ereignis-Handlers auch im kompilierten Code enthalten.
Im obigen Beispiel wird der Modulus einer additiven Gruppe als Konfigurationsparameter angegeben. Es wird gefordert, dass der Modulus prim ist. Wenn sich der Wert des Konfigurationsparameters ändert, überprüft der Ereignis-Handler des Crypto-Objektes, welches die Gruppe repräsentiert, ob der neue Wert des Konfigurationsparameters immer noch prim ist und damit die Gruppeneigenschaft weiterhin erfüllt ist. Ist das nicht der Fall, wird ein Fehler ausgegeben.

Neben der Initialisierung gibt es zwei weitere Möglichkeiten, einem Konfigurationsparameter einen neuen Wert zuzuweisen.

Zuweisungsoperationen

Syntax:

```
stmt         = confparambnz ':' '=' number
             | confparamstr ':' '=' QUOTED_STR ;
confparambnz = IDENT ;
confparamstr = IDENT ;
```

Beispiel:

```
ConfigParam l;
ConfigParam s := "";
l := 2048;
s := "Hallo Welt";
```

Durch eine Zuweisungsoperation wird einem Konfigurationsparameter ein neuer Wert zugewiesen. Der alte Wert wird überschrieben und von jedem Crypto-Objekt, das den Konfigurationsparameter verwendet, der Ereignis-Handler aufgerufen.

Zuweisung durch spezielle Funktionen

Syntax:

```
stmt = 'setConfigParam' '(' confparambnz ',' confparambnz ','
       'prime' ')'
     | 'setConfigParam' '(' confparambnz ',' confparambnz ','
       confparambnz ',' 'safe prime' ')'
     | 'setConfigParam' '(' confparambnz ',' confparambnz ','
       confparambnz ',' confparambnz ',' 'Schnorr' ')'
     | 'setConfigParam' '(' confparambnz ',' confparambnz ','
       confparambnz ',' confparambnz ',' 'RSA' ')'
     | 'setConfigParam' '(' confparambnz ',' confparambnz ','
       confparambnz ',' confparambnz ',' confparambnz ','
       confparambnz ',' 'safe prime' ',' 'RSA' ')' ;
```

Beispiel:

```
setConfigParam(p, l, prime);
setConfigParam(p, q, l, safe prime);
setConfigParam(p, q, lp, lq, Schnorr);
setConfigParam(n, p, q, l, RSA);
setConfigParam(n, p, q, p2, q2, l, safe prime, RSA);
```

Über spezielle Funktionen können Konfigurationsparameter mit bestimmten Eigenschaften konfiguriert werden. Im ersten Beispiel wird dem Konfigurati-

onsparameter p eine zufällig gewählte Primzahl der Bitlänge l zugewiesen. Der Parameter safe prime stellt sicher, dass es sich bei p um eine zufällige sichere Primzahl der Länge l handelt. Im dritten Beispiel werden zwei prime Konfigurationsparameter p und q mit einer Bitlänge von lp bzw. lq bestimmt, so dass p-1 ein Vielfaches von q ist.
In den beiden letzten Beispielen wird ein zufälliger RSA-Modulus der Bitlänge l erzeugt. Die Faktorisierung von n kann entweder aus zwei einfachen Primzahlen oder zwei sicheren Primzahlen bestehen. Alle Konfigurationsparameter bis auf l, lp und lq werden durch setConfigParam(..) überschrieben.

4.3.2 Gruppen

Im Gegensatz zu Programmiersprachen wie z. B. C entsprechen die Typen von Variablen in der Eingabesprache Gruppen. Bevor einer Variable eine Gruppe zugeordnet werden kann, muss die Gruppe definiert werden. Jede in der Eingabesprache definierte Gruppe hat einen Bezeichner, der die Gruppe eindeutig identifiziert. Daneben gibt es anonyme Gruppen, die vom Crypto-Framework im Zusammenhang mit anonymen Variablen automatisch erzeugt werden und keinen Bezeichner haben (siehe Kapitel 4.3.3). Als Gruppenparameter können Konfigurationsparameter oder Zahlen bzw. Strings verwendet werden. Im letzten Fall erzeugt der Crypto-Compiler automatisch anonyme Konfigurationsparameter (siehe Kapitel 4.3.1).

Im Crypto-Framework sind eine Reihe von Gruppen implementiert, die in Tabelle 4.4 zusammengefasst sind. Neben der Gruppenordnung ist angegeben ob ein Generatorelement der Gruppe bzw. einer großen Untergruppe automatisch generiert werden kann (siehe Kapitel 4.3.4).

Bitlänge von Gruppenelementen

Syntax:

```
bitlength      = /* empty */
               | '[' constnum ']' ;
constnum       = POSNUMBER
               | 'extract' '(' confparambnzin ')' ;
confparambnzin = IDENT
               | number;
```

Für einige Gruppen kann die maximale Bitlänge angegeben werden, die ein Gruppenelement im Speicher belegt. Das erlaubt verschiedene Optimierungen im Compiler-Backend (siehe Kapitel 4.4.11). Die Angabe der Bitlänge ist immer optional und kann entweder durch eine positive Zahl oder durch das Schlüssel-

Gruppe	Ordnung	Generator
`Z()`	-	-
`Z(min, max)`	-	✓
`Zadd(x)`	x	-
`Zadd(p, prime)`	p	✓
`ECprime(..)`	abhängig von Kurve	abhängig von Kurve
`Zmul(p, prime)`	$p - 1$	-
`Zmul(p, q, safe prime)`	$2 \cdot q$	-
`Zmul(p, q, safe prime, qr)`	q	✓
`Zmul(p, q, Schnorr)`	q	✓
`Zmul(n, p, q, prime)`	$(p-1)\cdot(q-1)$	-
`Zmul(n, p, q, p2, q2, safe prime)`	$4 \cdot p2 \cdot q2$	✓
`Zmul(n, p, q, p2, q2, safe prime, qr)`	$p2 \cdot q2$	✓
`ZmulPhi(G)`	abhängig von G	-
`[G1, G2]`	siehe G1, G2	siehe G1, G2

Tabelle 4.4: Eigenschaften der im Crypto-Framework implementierten Gruppen.

wort `extract` gefolgt von einem Konfigurationsparameter angegeben werden. Im letzteren Fall wird der aktuelle Wert des Konfigurationsparameters zur Kompilierzeit als Bitlänge verwendet.

Gruppe der ganzen Zahlen

Syntax:

```
stmt = IDENT ':' '=' 'Z' '(' ')' bitlength
     | IDENT ':' '=' 'Z' '(' confparambnzin [  ',' 'exact'  ] ')'
       bitlength
     | IDENT ':' '=' 'Z' '(' confparambnzin ',' confparambnzin ')'
       bitlength ;
```

Beispiel:

```
ConfigParam l := 2048;
G1 := Z()[extract(l)];
G2 := Z(1024);
G3 := Z(1024, exact);
G4 := Z(0, 100)[7];
```

Die Gruppe entspricht der mathematischen Gruppe $(\mathbb{Z}, +)$, d. h. der Gruppe der ganzen Zahlen mit Addition als Gruppenoperation. Elemente der Gruppen können theoretisch beliebig groß sein und sind nur durch die verwendete Langzahlbibliothek begrenzt.

Durch die Angabe von Parametern bei der Gruppendefinition kann das Intervall angegeben werden, aus welchem Zufallselemente gewählt werden. Das veranschaulicht das obige Beispiel. Im ersten Fall (`G1`) werden keine Intervallgrenzen angegeben. Das Ziehen eines Zufallselementes aus dieser Gruppe würde zu einem Fehler führen (siehe Kapitel 4.3.4). Bei der Gruppe im zweiten Fall (`G2`) werden Zufallselemente aus dem Intervall $[0, \ldots, 2^{1024} - 1]$ gezogen. Durch die Angabe des Parameters `exact` wird die Intervallgrenze als Bitlänge interpretiert, so dass Zufallswerte aus dem Intervall $[2^{1024-1}, \ldots, 2^{1024} - 1]$ gewählt werden (`G3`). Im letzten Beispiel (`G4`) sind die Intervallgrenzen direkt angegeben ($[0, 100]$).
Die Definition von `G1` zeigt, wie zur Angabe der maximalen Bitlänge der aktuelle Wert eines Konfigurationsparameters verwendet wird. Im letzten Beispiel wird die Bitlänge direkt angegeben.

Additive Restklassengruppe

Syntax:

```
stmt = IDENT ':' '=' 'Zadd' '(' confparambnzin [  ',' 'prime'  ] ')'
       bitlength ;
```

Beispiel:

```
G := Zadd(17, prime)[5];
```

Gruppen des Typs `Zadd` entsprechen der Klasse $\mathbb{Z}_n$, d. h. der additiven Restklassengruppen modulo einer natürlichen Zahl $n > 1$. Der erste Parameter gibt den Modulus an. Durch den optionalen Parameter `prime` muss der Modulus der Gruppe eine Primzahl sein.

Additive Restklassengruppe basierend auf elliptischen Kurven

Syntax:

```
stmt           = IDENT ':' '=' 'ECprime' '(' IDENT ')'
               | IDENT ':' '=' 'ECprime' '(' confparambnzin ','
                 confparambnzin ',' confparambnzin ','
                 confparamstrin ',' confparambnzin ','
                  confparambnzin ')' ;
confparamstrin = IDENT
               | QUOTED_STR ;
```

Beispiel:

```
G1 := ECprime(secp192r1);
G2 := ECprime(0x17, 0x1, 0x0, ”0415”, 0x17, 0x1);
```

Elliptische Kurven zur Beschreibung einer additiven Restklassengruppe können in der Eingabesprache entweder durch sechs Parameter oder durch Angabe des Kurvennamens spezifiziert werden. Die Definition elliptischer Kurven wird in [79] detailliert erläutert.

Multiplikative Restklassengruppe

Syntax:

```
stmt = IDENT ':' '=' 'Zmul' '(' confparambnzin ',' 'prime' ')'
       bitlength
     | IDENT ':' '=' 'Zmul' '(' confparambnzin ',' confparambnzin
       ',' 'safe prime' [  ',' 'qr'  ] ')' bitlength
     | IDENT ':' '=' 'Zmul' '(' confparambnzin ',' confparambnzin
       ',' 'Schnorr' ')' bitlength ;
```

Beispiel:

```
G := Zmul(17, prime)[5];
H := Zmul(7, 3, safe prime, qr)[3];
K := Zmul(43, 7, Schnorr)[6];
```

In der Eingabesprache lassen sich multiplikative Restklassengruppen mit primen Moduli beschreiben. Soll der Modulus p eine sichere Primzahl mit $p = 2q+1$ sein, muss die Primzahl q zusätzlich angegeben werden. Mit dem optionalen Parameter `qr` werden alle Gruppenoperationen in der Untergruppe der quadratischen Reste QR_n ausgeführt. Der Parameter `Schnorr` spezifiziert eine Schnorr-Gruppe.

RSA-Gruppe

Syntax:

```
stmt = IDENT ':' '=' 'Zmul' '(' confparambnzin ',' confparambnzin
       ',' confparambnzin ',' 'prime' ')' bitlength
     | IDENT ':' '=' 'Zmul' '(' confparambnzin ',' confparambnzin
       ',' confparambnzin ',' confparambnzin ',' confparambnzin
       ',' 'safe prime' [  ',' 'qr'  ] ')' bitlength ;
```

Beispiel:

```
G := Zmul(15, 3, 5, prime)[4];
H := Zmul(77, 7, 11, 3, 5, safe prime, qr)[7];
```

Anstelle einer Primzahl als Modulus kann auch das Produkt zweier Primzahlen p und q als Modulus n angegeben werden. Damit lässt sich eine RSA-Gruppe definieren. Durch den Parameter `prime` wird zum Ausdruck gebracht, dass es sich bei p und q um zwei Primzahlen handelt. Soll die Primfaktorzerlegung von n stattdessen aus zwei sicheren Primzahlen $p = 2p' + 1$ und $q = 2q' + 1$ bestehen, müssen zusätzlich die beiden Primzahlen p' und q' angegeben werden. Durch den Parameter `qr` werden alle Operationen in der Untergruppe der quadratischen Reste QR_n ausgeführt.

Multiplikative Restklassengruppe $\mathbb{Z}_\phi^*$

Syntax:

```
stmt  = IDENT ':' '=' 'ZmulPhi' '(' group ')' ;
group = IDENT ;
```

Beispiel:

```
G := Zmul(15, 3, 5, prime)[4];
H := ZmulPhi(G);
```

Die Gruppe $\mathbb{Z}_\phi^*$ repräsentiert eine multiplikative Restklassengruppe mit der Gruppenordnung einer anderen multiplikativen Restklassengruppe als Modulus. Die Gruppendefinition enthält neben einem Bezeichner den Namen der Gruppe, deren Ordnung als Modulus verwendet werden soll. Ist für die Gruppe die maximale Bitlänge der Gruppenelemente angegeben, so wird diese für die neue Gruppe übernommen.

Tupel-Gruppe

Syntax:

```
stmt      = IDENT ':' '=' '[' grouplist ']' ;
grouplist = group |grouplist ',' group ;
```

Beispiel:

```
G1 := Zadd(17, prime)[5];
G2 := Zmul(77, 7, 11, 3, 5, safe prime, qr)[7];
H := [G1, G2];
```

Durch Tupel-Gruppen lassen sich Gruppenhierarchien beschreiben. Jede Tupel-Gruppe hat einen Bezeichner und eine Liste mit Gruppen, die wiederum Tupel-Gruppen sein können. Die Gruppen müssen vorher definiert worden sein.

4.3.3 Variablen

Syntax:

```
stmt        = group ':' vardeflist ;
vardeflist  = [  vardeflist ','  ] vardef ;
vardef      = IDENT [  '=' varinit  ] ;
varinit     = number | 'O' | '<' numberlist '>' | '<' 'O' '>'
            | '[' varinitlist ']' ;
numberlist  = [  numberlist ','  ] number ;
varinitlist = [  varinitlist ','  ] varinit ;
```

Beispiel:

```
G := Zmul(15, 3, 5, prime)[4];
H := ECprime(secp192r1);
T1 := [G, H, H];
G : a;
T2 : b = [123, <1, 2>, O], c;
```

Eine *Variable* repräsentiert ein Gruppenelement und muss, bevor sie verwendet wird, deklariert werden. Eine Variablendeklaration besteht aus einem Bezeichner und dem Namen der Gruppe, dessen Elemente durch die Variable repräsentiert werden. Je nachdem, ob es sich um eine atomare Gruppe oder eine Tupel-Gruppe handelt, wird die erzeugte Variable entweder als *atomare Variable* oder als *Tupel-Variable* bezeichnet.

Variablen können definiert werden und dadurch gleich mit einem Wert initialisiert werden. Es gibt drei Arten der Variableninitialisierung.

1. Ist die Gruppe atomar und besteht ein Gruppenelement aus nur einer Langzahl (z. B. bei den Gruppen `Z`, `Zadd`, `Zmul`, `ZmulPhi`), kann der Wert direkt der Variablen zugewiesen werden.
2. Bei einer Gruppe, welche über eine elliptische Kurve definiert wird, sind zwei Werte zur Repräsentation eines Gruppenelementes notwendig[46]. In diesem Fall lässt sich ein Gruppenelement durch eine durch Kommata getrennte Liste von Komponenten darstellen. Die Zeichen '<' und '>' markieren Anfang und Ende des Komponentenvektors.
3. Auf ähnliche Art lassen sich Tupel-Variablen initialisieren. Anfang und Ende des Tupels sind durch spitze Klammern '[' und ']' festgelegt. Tupel-Elemente werden durch Kommata getrennt und können wiederum Zahlen, Komponentenvektoren oder Tupel sein, wodurch sich beliebige Hierarchien beschreiben lassen können.

[46] Zur Beschreibung des Punktes im Unendlichen dient das Symbol `O`.

Neben der Deklaration und Initialisierung einer Variablen erlaubt die Eingabesprache auch die Deklaration und Initialisierung mehrerer Variablen in einer Anweisung. Wird einer Variable kein Wert zugewiesen, wird sie mit der Gruppenidentität initialisiert.

Intern wird der Wert einer atomaren Variablen als Vektor von Langzahlen gespeichert, dessen Größe der Anzahl an Komponenten entspricht. Die maximale Größe eines Gruppenelementes ist nur von der verwendeten Langzahlbibliothek abhängig und in der Regel nur durch den zur Verfügung stehenden Speicher beschränkt.

4.3.4 Variablenzuweisungen

Syntax:

```
stmt      = variable '=' expr ;
expr      = expr ':' expr
          | addexpr ;
addexpr   = addexpr '+' addexpr
          | addexpr '-' addexpr
          | powexpr ;
powexpr   = powexpr '^' powexpr
          | unaryexpr ;
unaryexpr = '-' unaryexpr
          | dotexpr ;
dotexpr   = dotexpr '.' number
          | atomexpr ;
atomexpr  = '(' expr ')'
          | '$'
          | tmpvar
          | variable
          | IDENT '(' expr ')'
          | '?' group
          | '~' group
          | 'gen' '(' group ')'
          | number
          | 'O'
          | '<' numberlist '>'
          | '<' 'O' '>'
          | '[' exprlist ']' ;
tmpvar    = '#'
          | tmpvar '#' ;
exprlist  = [  exprlist ','  ] expr ;
variable  = IDENT ;
```

Eine Variablenzuweisung besteht aus einem Variablennamen, gefolgt von einem Gleichheitszeichen und einem Ausdruck. Zur Beschreibung komplexer gruppentheoretischer Zusammenhänge bestehen Ausdrücke aus Operatoren und weiteren Ausdrücken. Alle Operatoren sind bis auf den : Operator linksassoziativ. Das bedeutet, dass bei wiederholter Anwendung eines Operators (z. B. `exprA op exprB op exprC`) zuerst der linke Operator (`exprA op exprB`) ausgewertet wird[47].
Im Folgenden wird eine Liste aller Operatoren und Ausdrücke angegeben, die bei einer Variablenzuweisung Verwendung finden können.

- **Syntax:**
  ```
  atomexpr = number
           | '0'
           | '<' numberlist '>'
           | '<' '0' '>' ;
  ```
  Beispiel:
  ```
  a = 3;
  b = <1, 2>;
  c = 0;
  d = <0>;
  ```
  Die Angabe von Konstanten in einem Ausdruck erfolgt analog zur Variableninitialisierung und kann durch eine direkte Angabe einer Langzahl (Variable `a`) oder durch einen Komponentenvektor (Variable `b`) erfolgen. Die Variablen `c` und `d` sind identisch, denn ihnen wird der Punkt im Unendlichen zugewiesen.
  Konstanten werden vom Parser durch anonyme Variablen modelliert, die keinen Bezeichner besitzen.

- **Syntax:**
  ```
  atomexpr = '[' exprlist ']' ;
  ```
  Beispiel:
  ```
  a = [3, [<1, 2> + <3, 4>]];
  ```
  Ist die Variable eine Tupel-Variable, muss es sich bei dem Ausdruck um einen Tupel-Ausdruck handeln. Die Komponenten eines Tupel-Ausdruckes sind wieder Ausdrücke, so dass eine beliebig tiefe Verschachtelung von Ausdrücken möglich ist. Entspricht die Tupel-Struktur des Ausdruckes nicht der Tupel-Struktur der Variablen, bricht der Parser mit einer Fehlermeldung ab.

- **Syntax:**
  ```
  atomexpr = 'gen' '(' group ')' ;
  ```
  Beispiel:
  ```
  a = gen(G);
  ```

[47] Bei *rechtsassoziativen* Operatoren würde zuerst (`exprB op exprC`) ausgewertet.

Der Operator `gen(Gruppe)` erzeugt ein zufälliges Generatorelement aus der Gruppe `Gruppe`. Die Komplexität, ein Generatorelement zu finden, hängt von der verwendeten Gruppe ab und ist nicht immer trivial durchführbar. Tabelle 4.4 gibt eine Übersicht über Gruppen, auf die der Operator anwendbar ist. Ist die Funktionalität des Operators für eine Gruppe nicht implementiert, führt die Anwendung des Operators bei dieser Gruppe zu einem Fehler.

- **Syntax:**
  `atomexpr = '~' group ;`
  Beispiel:
  `a = ~G;`

  Der Operator liefert die Identität der angegebenen Gruppe zurück.

- **Syntax:**
  `atomexpr = '?' group ;`
  Beispiel:
  `a = ?G;`

  Das Ergebnis des Operators ist ein Zufallselement der angegebenen Gruppe oder eine Fehlermeldung, falls die Funktionalität des Operators für die Gruppe nicht implementiert ist. Das ist z. B. für die Gruppe `Z` der Fall, wenn das Intervall, aus dem Zufallselemente gezogen werden, nicht angegeben ist (siehe Kapitel 4.3.2).

- **Syntax:**
  `atomexpr = IDENT '(' expr ')' ;`
  Beispiel:
  `a = G(b);`

  Mithilfe des Casting-Operators wird der geklammerte Ausdruck zur Gruppe mit dem Bezeichner `IDENT` gecastet. Der Casting-Operator überprüft, ob die Anzahl der Komponenten der Gruppenelemente übereinstimmen. Eine Konvertierung z. B. von der Gruppe `ECprime` zur Gruppe `Zadd` und umgekehrt ist dadurch nicht möglich. Eine Überprüfung, ob das gecastete Gruppenelement in der neuen Gruppe liegt, wird aus Effizienzgründen nicht durchgeführt.

- **Syntax:**
  `atomexpr = variable ;`
  Beispiel:
  `a = b;`

  Falls Ausdrücke Variablen enthalten, müssen diese vorher deklariert worden sein und einen Wert besitzen.

- **Syntax:**
  ```
  expr     = expr ':' expr ;
  atomexpr = tmpvar ;
  ```
  Beispiel:
  ```
  a = ?G : [#1, #1];
  b = [?G, ?G];
  ```

  Durch den : Operator lassen sich Ausdrücke gruppieren, auf die mithilfe des # Operators zugegriffen werden kann. Werden Ausdrücke mehrfach verwendet, müssen diese nur einmal evaluiert werden.
  In dem Beispiel ist die Variable `a` eine Tupel-Variable, dessen Komponenten einem Zufallswert der Gruppe `G` entsprechen. Der Ausdruck `?G` wird nur einmal ausgewertet. Das bedeutet, dass beide Komponenten der Variable `a` gleich sind. Bei der Zuweisung zur Variablen `b` können die Komponenten dagegen unterschiedliche Werte annehmen.
  In einer Folge von Ausdrücken wird mit # auf das Ergebnis des vorherigen Ausdruckes, mit ## auf das Ergebnis des vor-vorherigen Ausdruckes usw. zugegriffen. Der entsprechende vorherige Ausdruck muss existieren.

- **Syntax:**
  ```
  atomexpr = '$' ;
  ```
  Der $ Operator erlaubt den Zugriff auf das Urbild eines Homomorphismus und wird nur der Vollständigkeit halber genannt. Er kann nur in Ausdrücken verwendet werden, die Homomorphismen beschreiben (siehe Kapitel 4.3.5).

- **Syntax:**
  ```
  atomexpr = '(' expr ')' ;
  ```
  Beispiel:
  ```
  a = (a + b)^c;
  ```

  Die Assoziativität der Operatoren lässt sich durch Klammern ändern.

- **Syntax:**
  ```
  dotexpr = dotexpr '.' number ;
  ```
  Beispiel:
  ```
  a = [[1, 2], 3].0.1;
  b = [1, 2] : #.1;
  ```

  Mit dem . Operator kann bei Tupel-Ausdrücken auf einzelne Komponenten eines Tupel-Ausdruckes zugegriffen werden. Durch mehrmalige Anwendung des '.' Operators (d. h. `expr . number . number` etc.) lässt sich durch eine komplette Hierarchie von Tupel-Ausdrücken navigieren. Der Variablen `a` wird im obigen Beispiel dadurch der Wert `2` zugewiesen.
  Der . Operator kann mit dem # Operator verknüpft werden, so dass sich

Tupel-Komponenten aus dem vorherigen Ausdruck auswählen lassen können. In dem Beispiel wird z. B. der Variablen `b` der Wert `2` zugewiesen.

- **Syntax:**
  ```
  unaryexpr = '-' unaryexpr ;
  ```
  Beispiel:
  ```
  a = -a;
  ```

  Der Operator liefert nach Auswerten des Ausdruckes und Bestimmung des Gruppenelementes das inverse Element zurück. Der Gruppentyp des inversen Ausdruckes entspricht dem Gruppentyp des Ausdruckes.

- **Syntax:**
  ```
  powexpr = powexpr '^' powexpr ;
  ```
  Beispiel:
  ```
  a = 2^4;
  ```

  Der `^` Operator potenziert den ersten mit dem zweiten Ausdruck. Der Exponent wird vor der Potenzierung nach `Z` gecastet.

- **Syntax:**
  ```
  addexpr = addexpr '+' addexpr
          | addexpr '-' addexpr ;
  ```
  Beispiel:
  ```
  a = 1 + 2;
  a = 1 - 2;
  ```

  Mit dem + Operator wird die Gruppenoperation auf die beiden Ausdrücke angewandt. Beide Ausdrücke müssen vom selben Gruppentyp sein. Mit dem - Operator wird vor der Anwendung der Gruppenoperation das Inverse des zweiten Operanden gebildet.

Syntaxbaum

Intern erzeugt das Crypto-Framework einen *abstrakten Syntaxbaum* mit der Variablenzuweisung als Wurzelelement. Jeder Knoten innerhalb des Syntaxbaumes ist entweder ein *Tupel-Knoten*, ein *Operator-Knoten* oder ein *Variablen-Knoten*.

- Tupel-Knoten kommen ausschließlich unterhalb des Wurzelelementes vor und repräsentieren einen Tupel-Ausdruck. Die Nachfolger eines Tupel-Knotens sind entweder weitere Tupel-Knoten, um verschachtelte Tupel-Ausdrücke darzustellen oder Operator- und Variablen-Knoten zur Darstellung von atomaren Ausdrücken. Die Anzahl der Nachfolger entspricht der Komponentenanzahl des Tupels. Eine atomare Variable besitzt keine Tupel-Knoten, da der Syntaxbaum den gesamten atomaren Ausdruck repräsentiert.

- Operator-Knoten bilden Operationen ab und haben abhängig von der Anzahl ihrer Operanden (Operanden sind Ausdrücke) keinen, einen oder zwei Unterbäume[48].
- Variablen-Knoten symbolisieren Variablen und Konstanten[49]. Da Variablen-Knoten keine Nachfolger haben, sind sie immer Blätter des Syntaxbaumes.

Die Beziehungen zwischen den Knoten-Typen soll anhand eines Beispiels veranschaulicht werden. Der Syntaxbaum der Anweisung `a = ~G + gen(G) : [(?G + #)^H(3), [b^4 + #]];` ist in Abbildung 4.3 vereinfacht[50] dargestellt. Die Variable a ist ein Element einer Tupel-Gruppe mit zwei Childs. Jeder Operator- und Variablen-Knoten (AtomicExpr) hat neben Zeigern auf die Operanden (a, b) eine Gruppe (Grp) und eine Variable (Dst). Die Gruppe der Variablen entspricht der Gruppe des Ausdruckes. Da jede Variable eine Gruppe hat, kann die Typsicherheit jederzeit gewährleistet werden. Eine Anwendung des + Operators auf Ausdrücke mit verschiedenen Gruppen lässt sich dadurch z. B. erkennen.

Der Syntaxbaum wird während des Parsens aus den Bäumen einzelner Teilausdrücke erzeugt. Sobald der Parser eine Regel anwenden kann, erweitert der Parser den Syntaxbaum auf seinem Stack mit dem geparsten Teilausdruck.

Die Darstellung als abstrakten Syntaxbaum hat den Vorteil, dass sich der Ausdruck nach dem Parsen leicht auswerten lässt. Dazu wird der Baum von den Blättern bis zur Wurzel durchlaufen. Das Zwischenergebnis einer Operation wird in einer anonymen Variablen (Dst) gespeichert. Zweige, die mehrmals durchlaufen werden, müssen dadurch nur einmal ausgewertet werden.
In Abbildung 4.3 wird das durch den Knoten für die + Operation mit `Refcount = 2` (Referenzzähler) veranschaulicht. Der Baum unterhalb dieses Operator-Knotens entspricht dem Ausdruck `~G + gen(G)` in der Eingabesprache. In der Variablenzuweisung wird der Ausdruck durch den '#' Operator zweimal referenziert. Aufgrund der Struktur des Syntaxbaumes kann beim Evaluieren der zweiten Referenz auf den Teilbaum das Ergebnis direkt aus der anonymen Variable `__Anon_20.0` gelesen werden.
Zur Erhöhung der Ausführungsgeschwindigkeit führt der Compiler eine automatische Optimierung des Syntaxbaumes durch. Das Crypto-Framework sucht im Syntaxbaum nach doppelten Teilbäumen und ersetzt sie durch Referenzen.

Konstanten werden beim Parsen auf Variablen abgebildet, wobei die Gruppe dieser Variablen unbekannt ist. Deshalb durchläuft der Parser nach Erstellung

[48] Ein Beispiel für eine Operation mit zwei Operanden ist der + Operator und mit einem Operand der Cast-Operator. Der Operator zur Erzeugung eines Generatorelementes einer Gruppe wäre dagegen ein Beispiel für einen Operator ohne Operanden.
[49] Konstanten werden durch Variablen repräsentiert.
[50] Jeder Knoten besitzt normalerweise eine Vielzahl weiterer Eigenschaften wie z. B. Referenzen auf übergeordnete Knoten, die das Navigieren innerhalb des Baumes erleichtern.

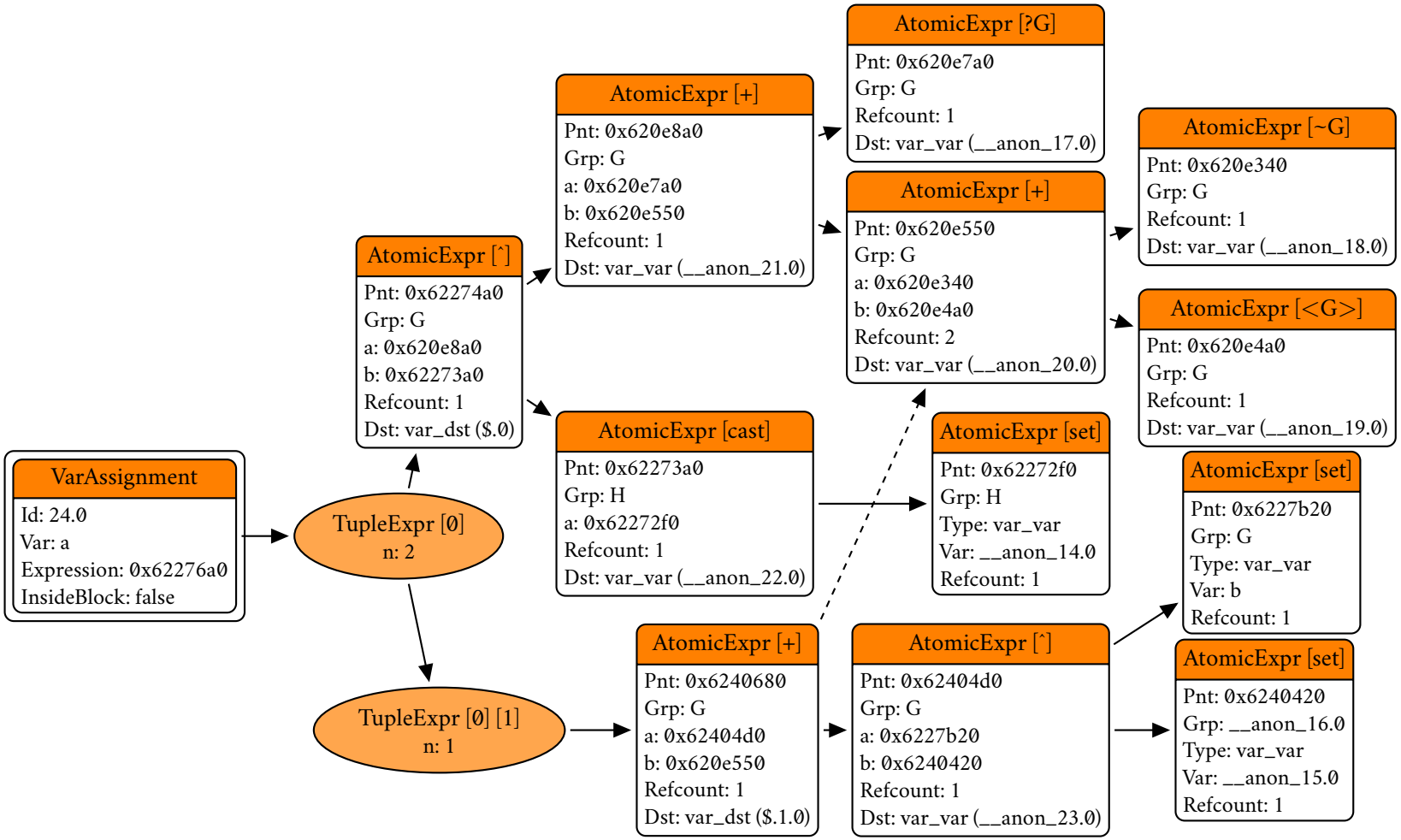


Abbildung 4.3: Syntaxbaum (ohne Variablen-Knoten) für die Variablenzuweisung `a = ~G + gen(G) : [(?G + #)^H(3), [b^4 + #]];`.

des Syntaxbaumes beginnend vom Wurzelelement den Baum und setzt die fehlenden Gruppen. Das ist möglich, da die Gruppe des Wurzelelementes, d. h. der Variablenzuweisung, bekannt ist.
Nicht immer ist dieses Verfahren möglich, wie z. B. die Anwendung des Potenzoperators `^` zeigt. Ist der Exponent eine Konstante und die Basis das Element einer Gruppe, welche über eine elliptische Kurve definiert wird, kann der Konstanten diese Gruppe nicht zugeordnet werden. Aus diesem Grund wird, falls die Gruppe des ersten Exponentenknotens nicht bekannt ist, immer eine neue Gruppe `Z` erzeugt und der Exponent auf diese Gruppe gecastet. Im Beispiel veranschaulicht das die Konstante `4`, die einer anonymen Variablen `__Anon_15.0` mit anonymer Gruppe `__Anon_16.0` vom Typ `Z` zugeordnet ist.

4.3.5 Homomorphismen

Syntax:
```
stmt = IDENT '[' group '-' '>' group ']' '=' expr ;
```

Beispiel:
```
phi1 [ W -> G ] = g^$;
phi2 [ WxW -> G ] = g^$.0 + h^$.1;
phi3 [ W -> GxWxG ] = ?G : [ [g^$, h^#] : #.0 + #.1, [$, #]];
```

Homomorphismen bilden Elemente einer Quellgruppe auf die Elemente einer Zielgruppe ab und dienen zur Beschreibung von Σ-Protokollen.
Im ersten Beispiel ist ein Homomorphismus mit dem Bezeichner `phi1` definiert, der Elemente der Gruppe `W` auf Elemente der Gruppe `G` abbildet. Für die Angabe des Ausdruckes gelten dieselben Regeln wie bei Variablenzuweisungen (siehe Kapitel 4.3.4). Zusätzlich kann noch der `$` Operator verwendet werden, um die Elemente der Urbildgruppe zu spezifizieren. Das erste Beispiel entspricht dem Homomorphismus:

$$\phi_0 : \begin{cases} W & \rightarrow & G \\ w & \mapsto & g^w \end{cases}$$

Der `$` Operator lässt sich mit dem `.` Operator kombinieren, falls die Quellgruppe eine Tupel-Gruppe ist (siehe zweites Beispiel). Der Homomorphismus `phi2` entspricht dem Homomorphismus:

$$\phi_1 : \begin{cases} W \times W & \rightarrow & G \\ (w_0, w_1) & \mapsto & g^{w_0} h^{w_1} \end{cases}$$

Durch die Beschreibung von Homomorphismen in der Eingabesprache muss das Bild der homomorphen Abbildung nicht mehr ausschließlich von der Eingabe

abhängen. Es lassen sich auch Homomorphismen beschreiben, die keine Homomorphismen im mathematischen Sinne darstellen. Eine Anwendung für einen solchen Homomorphismus zeigt das dritte Beispiel, dass ein Commitment mit einer Zufallskomponente beschreibt und mit $w' \in_R W$ der folgenden Abbildung entspricht.

$$\phi_3 : \begin{cases} W & \rightarrow & G \times (W \times G) \\ w & \mapsto & (g^w \cdot h^{w'}, (w, g^w \cdot h^{w'})) \end{cases}$$

4.3.6 Σ-Protokolle

Syntax:

```
stmt         = IDENT '=' 'SigmaPhi' '[' hom ',' variable ','
               variable ',' constnumhash ']'
             | IDENT '=' 'SigmaGsp' '[' hom ',' variable ','
               variable ',' constnum ',' constnumhash ']'
             | IDENT '=' 'SigmaAnd' '[' sigmalist ']'
             | IDENT '=' 'SigmaOr' '[' sigmalist ']' ;
constnumhash = POSNUMBER |hash
             | 'extract' '(' confparamin ')' ;
hash         = 'MD4' |'MD5' |'SHA1' |'SHA224' |'SHA256'
             | 'SHA384' |'SHA512' ;
sigmalist    = [  sigmalist ','  ] sigma ;
confparamin  = IDENT |number |QUOTED_STRING ;
hom          = IDENT ;
```

Beispiel:

```
sigmaphi1 = SigmaPhi [ phi, x, w, 80 ];
sigmaphi2 = SigmaPhi [ phi, x, w, extract(l) ];
sigmaphi3 = SigmaPhi [ phi, x, w, SHA256 ];
sigmagsp1 = SigmaGsp [ phi, x, w, 40, 80 ];
sigmagsp2 = SigmaGsp [ phi, x, w, 40, extract(l) ];
sigmaand  = SigmaAnd [ sigmaphi1, sigmaphi2 ];
sigmaor   = SigmaOr [ sigmagsp1, sigmagsp2 ];
```

Sigma Protokolle werden definiert durch einen Bezeichner, den Namen des Σ-Protokolls und Parameter, die das Σ-Protokoll spezifizieren. Das erste Beispiel zeigt die Definition eines Σ^{ϕ}-Protokolls. Als Homomorphismus wird `phi` mit dem Urbild `w` und dem Bild `x` verwendet. Der letzte Parameter legt die Obergrenze des Intervalls fest, aus dem die Challenges gewählt werden. Die Obergrenze wird als Bitlänge angegeben und kann z. B. auch durch einen Konfigurationsparameter festgelegt werden (siehe Kapitel 4.3.2). Eine weitere Möglichkeit, die

Obergrenze des Challenge-Intervalls anzugeben, ist die Angabe einer Hashfunktion, entweder als String oder als Konfigurationsparameter. Die Bitlänge entspricht der Länge eines Hashwertes der entsprechenden Hashfunktion. Die angegebene Hashfunktion wird zusätzlich verwendet, falls das Σ-Protokoll nicht-interaktiv implementiert wird. Ist wie in den ersten beiden Fällen keine Hashfunktion angegeben, wird SHA-224 verwendet.

Die Definition eines Σ^{GSP}-Protokolls entspricht bis auf einen zusätzlichen Parameter der Definition eines Σ^{ϕ}-Protokolls. Der zusätzliche Parameter gibt den Sicherheitsparameter an. Das veranschaulicht das zweite Beispiel mit $\ell_{\phi} = 40$ als Sicherheitsparameter.

Neben dem Σ^{ϕ}- und Σ^{GSP}-Protokoll lassen sich in der Eingabesprache Σ^{AND}- und Σ^{OR}-Protokolle definieren. Bei der Definition von Σ^{OR}-Protokollen müssen die einzelnen Prädikate unabhängig sein (die Definition von `sigmaor` im obigen Beispiel führt daher zu einem Fehler).

4.3.7 ZPK Anweisungen

Syntax:

```
stmt         = IDENT '=' 'PK' '[' '(' varidentlist ')' ':'
               preddisjop ',' 'SigmaPhi' ',' constnum ']'
             | IDENT '=' 'PK' '[' '(' varidentlist ')' ':'
               preddisjop [  ':' intchecklist  ] ','
               'SigmaGsp' ',' constnum ',' constnum ']'
             | IDENT '=' 'SPK' '[' '(' varidentlist ')' ':'
               preddisjop ',' 'SigmaPhi' ',' consthash
               [  ',' 'binary'  ] ']'
             | IDENT '=' 'SPK' '[' '(' varidentlist ')' ':'
               preddisjop ',' [  ':' intchecklist  ] ','
               'SigmaGsp' ',' constnum ',' consthash ']' ;
varidentlist = [  varidentlist ','  ] variable ;
preddisjop   = preddisjop '|' preddisjop
             | predkonjop ;
predkonjop   = predkonjop '&' predkonjop
             | predop;
predop       = '(' preddisjop ')'
             | expr '=' expr
             | expr;
intchecklist = [  intchecklist ','  ] intcheck ;
intcheck     = 'intcheck' '(' variable ',' constnum ')' ;
```

Beispiel:

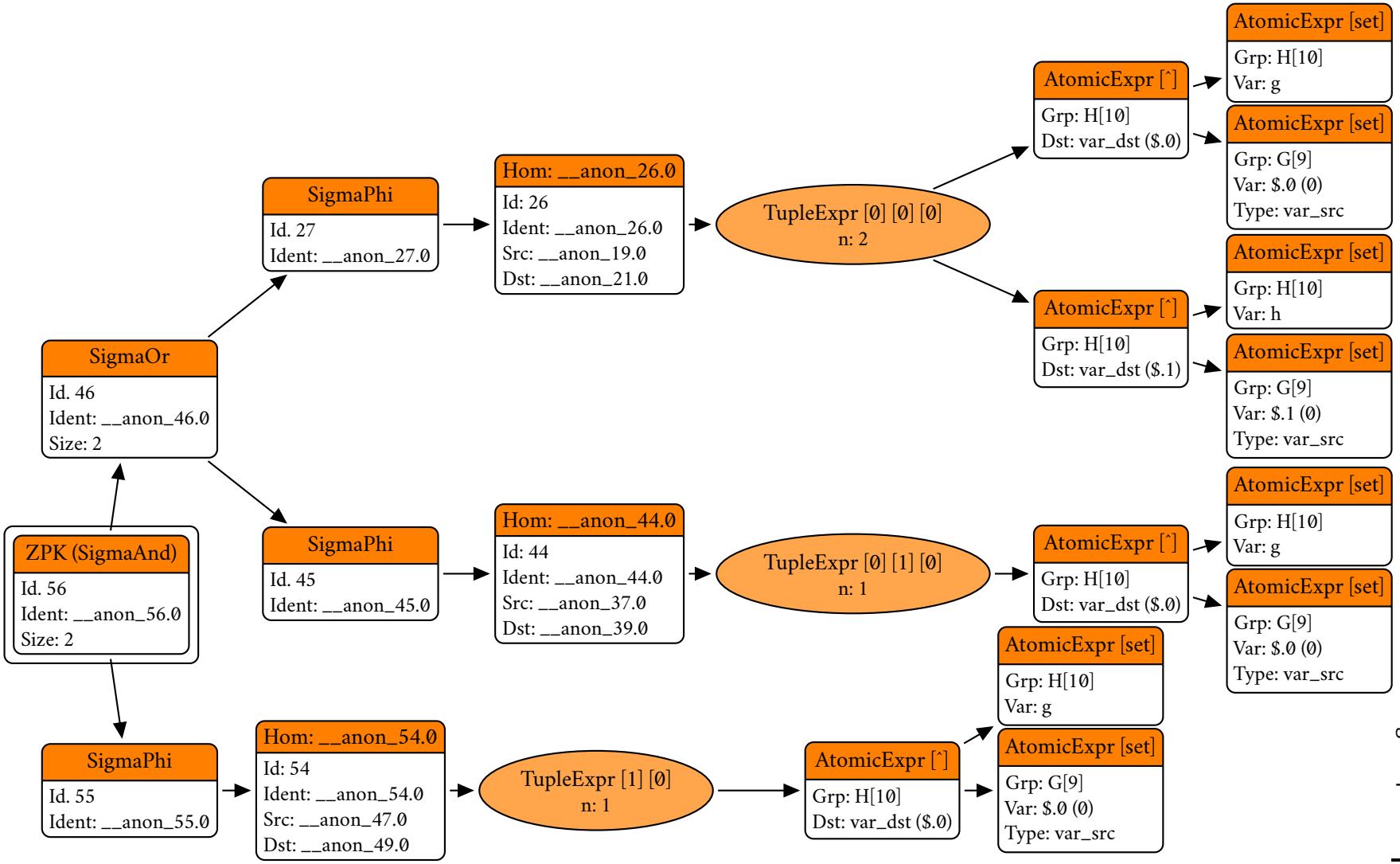


Abbildung 4.4: Vereinfachte Darstellung der Datenstruktur zur Beschreibung der Σ-Struktur von `PK [ (w) : (g^w.0 & h^w.1 | h = g^w.2) & g^w.3, SigmaPhi, 1];`.

```
zpk1 = PK [ (w) : (g^w.0 & h^w.1 | h = g^w.2) & g^w.3,
                  SigmaPhi, 1];
spk1 = SPK [ (w) : g^w.0, SigmaPhi, SHA256, binary];
spk2 = SPK [ (w) : g^w.0 : intcheck(w, 40), SigmaGsp, 80, SHA256];
```

Durch ZPK-Statements lassen sich Proofs in der CSN beschreiben. Die Geheimnisse werden am Anfang des Statements als Liste von Variablen angegeben[51]. Die Prädikate werden in der Notation der Eingabesprache beschrieben und lassen sich durch „und"- (`&`) bzw. „oder"- (`|`) Verknüpfungen kombinieren (siehe Kapitel 2.5).
„Und"-Verknüpfungen haben eine höhere Priorität als „oder"-Verknüpfungen, was durch Klammerung der Prädikate geändert werden kann. Prädikate, die durch „oder" verknüpft werden, müssen unabhängig sein (siehe Kapitel 2.5.3).
Die linke Seite eines Prädikates ist optional und wird automatisch berechnet, falls sie nicht angegeben ist. In einigen Protokollen (z. B. dem Protokoll in Kapitel 3) ist die linke Seite bei manchen Prädikaten konstant und muss nicht evaluiert werden. Das erhöht die Ausführungsgeschwindigkeit. In diesen Fällen kann in der Eingabesprache die linke Seite als Ausdruck angegeben werden.

Intern bildet der Parser die Prädikate auf eine Datenstruktur bestehend aus Σ-Protokollen ab. Abbildung 4.4 zeigt eine vereinfachte Darstellung dieser Datenstruktur für das erste Beispiel. Die Blätter der Σ-Struktur repräsentieren immer atomare Σ-Objekte, denen jeweils ein Homomorphismus zugeordnet ist. Neben den automatisch erzeugten Gruppen ist jedem Homomorphismus ein Tupel-Ausdruck zugeordnet.
Die inneren mit „und" verknüpften Prädikate (hier `g^w.0 & h^w.1`) werden immer durch ein atomares Σ-Objekt und einen Tupel-Ausdruck repräsentiert. Dadurch können diese Prädikate abhängig sein.

ZPK-Statements werden entweder durch ein Σ^{ϕ} (`SigmaPhi`) oder Σ^{GSP} (`SigmaGsp`) Protokoll interaktiv (`PK`) oder nicht-interaktiv (`SPK`) ausgeführt. Der erste Parameter eines ZPK-Statements gibt entweder die Obergrenze der Challenge-Menge (interaktiv) oder die Hashfunktion (nicht-interaktiv) an.
Soll ein Σ^{ϕ}-Protokoll nicht-interaktiv verwendet werden, kann zusätzlich mit dem Parameter `binary` festgelegt werden, binäre Challenges zu verwenden, um das Protokoll in Abbildung 3.2 zu implementieren.
Bei Σ^{GSP}-Protokollen spezifiziert ein weiterer Parameter den Sicherheitsparameter ℓ_{ϕ}. Implizite Intervallchecks (siehe Kapitel 2.6.1) werden bei Σ^{ϕ}-Protokollen mit `intcheck(..)` spezifiziert. So entspricht `intcheck(w, 40)` dem Beweis der Variablen[51] w, dass $w \in \pm\{0,1\}^{40+\ell_{\phi}+\ell_{c}+2}$ gilt.

[51] Bei einer Tupel-Variable werden alle Childs als Geheimnisse betrachtet.

4.3.8 Checks

Syntax:

```
stmt       = 'check' '(' variable comparator expr ')';
comparator = '=' '='
           | '!' '='
           | '>' '='
           | '<' '=';
```

Beispiel:

```
check(a == 1);
check(a != 2 + 3);
check(a >= 1);
check(a <= ?H);
```

Das check Statement vergleicht Variablen mit Ausdrücken. Es stehen die Vergleichsoperatoren „gleich“, „ungleich“, „größer gleich“ und „kleiner gleich“ zur Verfügung. Alle Ausdrücke der linken Seite einer Variablenzuweisung (siehe Kapitel 4.3.4) finden Verwendung.
Ist bei der Ausführung des kompilierten Programmes die Bedingung nicht erfüllt, bricht die Programmausführung mit einem Fehler ab. Im Interpreter-Modus wird eine Fehlermeldung ausgegeben.

4.3.9 Blöcke

Syntax:

```
block         = [  '__critical'  ] IDENT '(' blckparamlist ')'
                '{' blockcode '}';
blckparamlist = /* empty */
              | [  blckparamlist ','  ] blckparam;
blockcode     = /* empty */
              | stmtlist;
blckparam     = [  'in' |'out'  ] group ':' IDENT;
stmtlist      = [  stmtlist ','  ] stmt ;
```

Beispiel:

```
H := Zmul (1019, 509, safe prime, qr);
__critical func(out H: a, in H: b) {
  a = b + 1;
};
```

Durch *Blöcke* lassen sich Protokollbeschreibungen in der Eingabesprache strukturieren, die sich mit Funktionen vergleichen lassen können, wie sie z. B. in C

Verwendung finden.
Ein Block hat einen Bezeichner, Parameter und eine beliebige Anzahl von Anweisungen, die mit geschweiften Klammern umschlossen werden. Parameter und Anweisungen sind optional und können weggelassen werden. Über das optionale Präfix `__critical` lassen sich Blöcke kennzeichnen, deren Ausführung als besonders zeitkritisch gelten (siehe Kapitel 4.4.11).

Die Parameter eines Blockes sind Variablen, die lokal in den Blöcken verwendet werden können. Dabei unterscheidet das Crypto-Framework zwischen Eingabe- und Ausgabeparametern. Auf beide Parameter kann innerhalb des Blockes sowohl lesend als auch schreibend zugegriffen werden[52].

4.3.10 Variablenausgabe

Syntax:

```
stmt = 'print' '(' [  QUOTED_STR ','  ] IDENT [  ',' 'hex'  ] ')' ;
```

Beispiel:

```
ConfigParam a := 1;
b = 2;
print("Konfigurationsparameter a: ", a);
print("Variable b: ", b, hex);
```

Im kompilierten Code wird über die `print` Anweisung ein Konfigurationsparameter oder eine Variable ausgegeben. Das ist insbesondere beim Debuggen eines Protokolls vorteilhaft. Durch den optionalen Parameter `hex` wird der Wert nicht als Dezimal- (Standard), sondern als Hexadezimalzahl ausgegeben.

4.4 Code Management Framework

Das CMF bildet das Backend des Compilers und erstellt aus Crypto-Objekten Programmcode. Es werden folgende Anforderungen an das Backend gestellt:

- *Vollständige Ausdrucksmächtigkeit des Crypto-Frameworks*: Das Backend muss die komplette Funktionalität des Crypto-Frameworks in eine Zielsprache abbilden können.
- *Verwendung des Crypto-Framworks ohne Compiler-Backend*: Das gesamte Compiler-Backend kann deaktiviert werden, ohne die Funktionalität des

[52] Es ist zu beachten, dass ein Ausgabeparameter am Anfang des Blockes mit der Gruppenidentität initialisiert und ihr alter Wert überschrieben wird.

Crypto-Frameworks einzuschränken. Die Integration des Compiler-Backends wirkt sich nicht negativ auf die Performance des Crypto-Frameworks aus.

- *Unabhängigkeit der Zielsprache*: Das Backend ist unabhängig von der Zielsprache. Neben der Kompilierung in eine Maschinensprache lässt sich der Programmcode auch in höhere Programmiersprachen wie z. B. C, C++, Java oder Hardware Beschreibungssprachen wie z. B. VHDL, Verilog übersetzen.
- *Optimierung*: Vor der Umwandlung in die Zielsprache muss die Möglichkeit gegeben sein, den Code zu optimieren.
- *Erweiterbarkeit*: Das Crypto-Framework ist ohne Anpassung des Compiler-Backends um neue Funktionalität erweiterbar. Die zusätzliche Funktionalität wird automatisch in die Zielsprache übersetzt.

4.4.1 Konzept

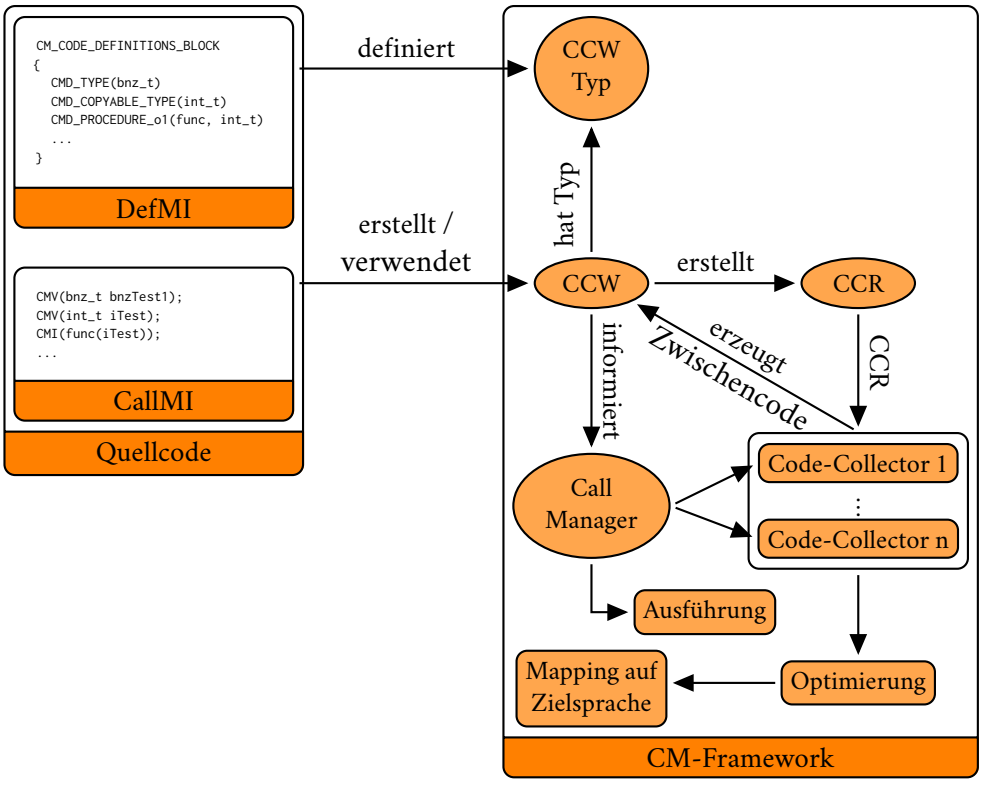


Abbildung 4.5: Konzept des CMF.

Beim Parsen einer Eingabesprache werden die semantischen Aktionen in einem plattformunabhängigen Zwischencode beschrieben, der dann in die Zielsprache

übersetzt wird [2]. Der C++ Code, der diese semantischen Aktionen beschreibt, ist in Form von Crypto-Objekten bereits implementiert. Er müsste nochmal als Zwischencode implementiert werden, um eine Kompilierung der Funktionalität der Crypto-Objekte zu ermöglichen.
In dieser Arbeit wird im Vergleich zum klassischen Compilerbau ein neuer Ansatz zur Beschreibung semantischer Aktionen eingeführt.

Die Idee beim CMF besteht darin, Variablendeklarationen und Anweisungen mithilfe spezieller Makros zur Laufzeit *aufzusammeln* und ihre Funktionalität auf eine Datenstruktur, den Zwischencode, abzubilden. Die aufgesammelten Anweisungen können wahlweise ausgeführt (Interpreter-Modus) oder am Ende des Aufsammelprozesses in die *Zielsprache* übersetzt werden (Compiler-Modus). Das CMF ermöglicht es, C++ Code in jede beliebige weitere Sprache zu übersetzen.

Abbildung 4.5 veranschaulicht das Konzept des CMF. Es ist ein eigenständiges Modul und unabhängig vom Crypto-Compiler.
Anweisungen werden durch Makros des *CallMI (Call Macro-Interface)* umschlossen (z. B. `CMV(...)`, `CMI(...)`) und *markieren* die Variablen und Anweisungen, die durch das CMF aufgesammelt werden. Der aufzusammelnde C++ Code wird im Folgenden als *Quellcode* bezeichnet. Das CallMI beschreibt die Quellcodeschnittstelle des CMF und bildet jede markierte Variable auf ein *CCW (Collected Code Wrapper)* im Namensraum des CMF ab. Funktionsaufrufe werden auf vordefinierte CCWs abgebildet. Damit repräsentieren CCWs die Funktionalität der aufgesammelten Anweisungen innerhalb des CMF. Die CCWs sind notwendig, um z. B. Destruktoraufrufe markierter Variablen zu erfassen, die ihren Gültigkeitsbereich verlieren (siehe Kapitel 4.4.3).

Variablentypen und Funktionsbezeichner, die aufgesammelt werden sollen, sind vorher dem CMF mithilfe des *DefMI (Definition Macro-Interface)* bekanntzugeben. Das *DefMI* definiert die Typen der CCWs im Namensraum des CMF.

Die CCWs informieren den *Code-Manager* über jeden Aufruf des CallMI und jede Zustandsänderung. Der Code-Manager gibt diese Information an die Code-Collectors weiter. Im Interpreter-Modus führt der Code-Manager den Code aus.

Die Code-Collectors initiieren die Erzeugung von Zwischencode, der zu den aufgesammelten Anweisungen äquivalent ist. Der Zwischencode setzt sich aus *CCRs (Collected Code Representatives)* zusammen, die jeder Code-Collector als Datenstruktur speichert.

Zur Überführung des aufgesammelten Codes in die Zielsprache wird die Datenstruktur mit CCRs durchlaufen und ihre Funktionalität auf die Zielsprache abgebildet. Vorher wird der Zwischencode optimiert und z. B. Redundanzen entfernt.

Beispiel

Anhand Listing 4.1 soll die Funktionsweise des CMF mit einem Beispiel erläutert werden.
Zeile 5 bis 7 zeigen die Definition aufsammelbarer Variablentypen und Funktionsaufrufe mithilfe des DefMI. Variablen vom Typ `bnz_t` und `int` sowie Funktionsaufrufe der Funktion `func` können vom CMF aufgesammelt werden. Das eigentliche Aufsammeln der Anweisungen erfolgt in der Funktion `my_Func`[53]. In Zeile 15 wird durch das CallMI die Deklaration der Variable `ccI` und in Zeile 17 ihre Zuweisung mit dem Wert `3` markiert und aufgesammelt. Zeile 21 zeigt die Markierung eines Funktionsaufrufes.

Voraussetzung für das Aufsammeln von Anweisungen ist die Präsenz eines aktiven Code-Collectors, wie er in Zeile 29 erzeugt wird. Ist kein Code-Collector aktiv, wird Funktion `my_Func` nur ausgeführt und keine Anweisungen werden aufgesammelt (Zeile 27).
Zeile 34 zeigt den Zugriff auf die Datenstruktur des Zwischencodes. Der Zwischencode kann mithilfe der Methoden `beginSubProgram` und `endSubProgram` hierarchisch strukturiert werden, um daraus z. B. Unterprogramme in der Zielsprache zu erstellen.

Rechts neben Listing 4.1 ist der Zwischencode des aufgesammelten Codes graphisch dargestellt. Neben den Anweisungen beinhaltet der Zwischencode auch die Abhängigkeiten zwischen den CCRs, wie z. B. Schreib-/Lesezugriffe auf Variablen[54].

Das Beispiel zeigt bereits ein wesentliches Merkmal des CMF. Zur Strukturierung des Programmcodes, der z. B. die Funktionalität der Crypto-Objekte beschreibt, können beliebig komplexe Sprachkonstrukte der Sprache C++ (z. B. Klassen, komplexe Entwurfsmuster etc.) verwendet werden. Durch das Markieren von Anweisungen werden die aufgesammelten Anweisungen mit Beachtung ihrer Ausführungsreihenfolge in eine einfache Darstellung überführt. Durch die `beginSubProgram` und `endSubProgram` Anweisungen lässt sich der Zwischencode neu strukturieren.

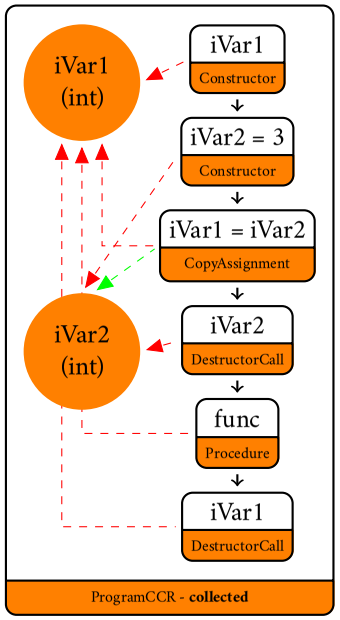
4.4.2 Macro-Interface

Das *MI (Macro-Interface)* besteht aus einer auf Makros basierten Schnittstelle des CMF. Das MI unterteilt sich in das *DefMI (Definition Macro-Interface)* und *CallMI (Call Macro-Interface)*.

[53] Die Funktion `my_Func` könnte z. B. die Methode eines Crypto-Objektes sein.
[54] Lesezugriffe auf eine Variable sind durch grüne, Schreibzugriffe durch rote Pfeile kenntlich gemacht.

```
1  extern void func(int&);
2
3  // DefMI
4  CM_CODE_DEFINITIONS_BLOCK_BEGIN
5      CMD_TYPE(bnz_t, bnz_t)
6      CMD_COPYABLE_TYPE(int, int)
7      CMD_PROCEDURE_o1(func, int)
8  CM_CODE_DEFINITIONS_BLOCK_END
9
10 // Funktion mit markierten Anweisungen
11 void my_Func()
12 {
13   int i;
14   // Markieren einer aufzusammelnden Variable
15   CMV(int ccI);
16   // Zuweisung
17   ccI = CMV_IN(3);
18   // Funktionsaufruf ausführen, nicht aufsammeln
19   func(i);
20   // Markieren eines aufzusammelnden Funktionsaufrufes
21   CMI(func(ccI));
22 }
23
24 void main()
25 {
26   // nur ausführen der Funktion
27   my_Func;
28   // ausführen und aufsammeln
29   CodeCollector cc("collector");
30   cc.beginSubProgram("collected");
31     my_Func();
32   cc.endSubProgram("collected");
33   // Zugriff auf den Zwischencode
34   ProgramCCR& rProgram = cc.getProgram();
35 }
```

Listing 4.1: Einführungsbeispiel



CallMI

```
1  // CMV(int ccI);
2  CMV::__cm_int ccI(CM_CREATE_DEBUG_INFO(..));
3  // CMV_POINTER(int, ccI2);
4  CMV::__cm_PointerBox_int ccI2(CM_CREATE_DEBUG_INFO(..));
5  // CMV_ARRAY(int, ccI3, 2);
6  CMV::__cm_int ccI3(new vector<CMV::__cm_int*>(), 2,
     ↳ CM_CREATE_DEBUG_INFO(..));
7  // CMI(func(ccI));
8  CMI::func(ccI).call(CM_CREATE_DEBUG_INFO(..));
```

Listing 4.2: Auflösung der wichtigsten CallMI-Makros.

Die Makros `CMV` und `CMI` werden als CallMI-Makros bezeichnet und dienen zur Markierung der aufzusammelnden Anweisungen im Quellcode. Listing 4.2 zeigt die Auflösung der Makros.
Das Makro `CMV` erzeugt für die eingeschlossene Variable ein CCW im CMF-Namensraum (Zeile 1 / 2, siehe Kapitel 4.4.3)[55]. Für jeden Funktionsaufruf existiert ein CCW im CMF.
Um das Verhalten von Zeigern zu abstrahieren, steht zusätzlich das CallMI-Makro `CMV_POINTER` zur Verfügung. Dadurch verhält sich die Variable `ccI2` wie eine kopierbare Variable, dessen Wert der Adresse von Daten auf dem Heap entspricht (Zeile 3 / 4).
Arrays werden über das `CMV_ARRAY`-Makro definiert (Zeile 5 / 6). Es werden jeweils ein CCW für die Array-Variable sowie alle Array-Elemente erstellt.
Das Makro `CMI(func)` ruft zur Registrierung des Funktionsaufrufes das passende CCW auf (Zeile 7 / 8).

Die CallMI-Makros übergeben über das Makro `CM_CREATE_DEBUG_INFO` zusätzlich Debugging-Informationen an die CCWs. Dazu gehören z. B. der Name der Quelldatei und die Zeile, in der die Anweisung markiert wurde. Im kompilierten Code lassen sich daraus Kommentare erzeugen, um das Debugging zu erleichtern.

Das CallMI verfügt über weitere Makros, die in den folgenden Kapiteln genauer beschrieben werden. Das Makro `CMV_IN` importiert Konstanten oder Werte nicht-markierter Variablen in das CMF, während das Makro `CMV_OUT` Werte von Variablen aus dem CMF exportiert (Kapitel 4.4.8). Weitere Makros stehen zur Beschreibung von Verzweigungen und Schleifen (Kapitel 4.4.10) zur Verfügung.

[55] Die CCWs befinden sich in einem eigenen Namensraum (`CMV::` für Variablen, `CMI::` für Instruktionen), um nicht mit den Bezeichnern im Quellcode zu kollidieren. Des Weiteren haben alle Typen der CCWs das Prefix `__cm_`, da CCWs C++ Objekte sind und Namen von Basistypen (z. B. `int`) als Klassennamen verboten sind.

DefMI

```
1  typedef const char* str;
2  extern str* str_Cstr(str rInitConst);
3  extern void str_Dest(str* pThis);
4
5  extern void bnz_init(bnz_t&);
6  extern void bnz_set(bnz_t&, const bnz_t&)
7  extern int bnz_set_Str(bnz_t&, str, int);
8  extern int hashUpdateBnz(POINTER(HashContext)&, const bnz_t&)
9
10 CM_CODE_DEFINITIONS_BLOCK_BEGIN
11     CMD_TYPE(bnz_t)
12     CMD_TYPE(HashContext)
13     CMD_COPYABLE_TYPE(int)
14     CMD_COPYABLE_TYPE(str)
15     CMD_CONST_INSTANCES_HANDLING(str, str_Cstr, str_Dest)
16
17     CMD_PROCEDURE_o1(bnz_init, false, true, bnz_t)
18     CMD_PROCEDURE_o1i1(bnz_set, false, false, bnz_t, bnz_t)
19     CMD_FUNCTION_o1i2(int, bnz_set_Str, false, false,
         ↳                bnz_t, str, int)
20     CMD_FUNCTION_o1i1(int, hashUpdateBnz, false, false,
         ↳                POINTER(HashContext), bnz_t)
21 CM_CODE_DEFINITIONS_BLOCK_END
```

Listing 4.3: Beispiel zur Nutzung des DefMI.

Aufzusammelnde Variablen und Funktionsaufrufe werden durch die `CMV`- und `CMI`-Makros auf CCWs im CMF abgebildet. Ein CCW ist ein C++ Objekt dessen Typ vom Typ der aufzusammelnden Variablen bzw. vom Prototyp der aufzusammelnden Funktion abhängig ist. Das DefMI definiert diese Typen. Nur durch das DefMI definierte Variablentypen und Funktionsaufrufe sind aufsammelbar.

DefMI-Definitionen werden durch *Definitionsblöcke* zusammengefasst. Ein C++ Projekt kann mehrerer solcher Definitionsblöcke enthalten, die üblicherweise in den Header-Dateien, in denen die Typen und Funktionen deklariert sind, stehen.

Das Listing 4.3 zeigt beispielhaft einen DefMI-Definitionsblock für die drei Typen `int`, `bnz_t`[56] und `str` sowie mehrere Funktionen. Durch das Makro `CM_CODE_DEFINITIONS_BLOCK_BEGIN`[57] in Zeile 10 wird der DefMI-Definitionsblock eingeleitet und durch das Makro in Zeile 21 beendet.

[56] Der Typ `bnz_t` repräsentiert im Crypto-Compiler eine Langzahl. Bei Funktionen kennzeichnet das Präfix `bnz_` Operationen, die auf Variablen vom Typ `bnz_t` angewendet werden [71].

[57] Zusätzlich gibt es das Makro `CM_CODE_DEFINITIONS_BLOCK_EX_BEGIN(namespace)`, das innerhalb des Definitionsblockes Zugriff auf den Namensraum `namespace` gestattet.

Das CMF unterscheidet zwischen *einfachen* (Zeile 11) und *kopierbaren* Variablen (Zeile 13 / 14). Kopierbare Variablen-CCWs[58] kann über den Ist-gleich-Operator ein neuer Wert zugewiesen werden. Bei nicht-kopierbaren bzw. einfachen CCW-Variablen ist das nicht möglich, sondern nur über ihren Kopierkonstruktor (siehe Kapitel 4.4.8)[59].
Einen Sonderfall stellt der Typ `str` zur Repräsentation von C-Zeichenketten im CMF dar. Variablen des Typs speichern nur die Anfangsadresse eines Arrays von `char` Werten. Soll z. B. eine konstante Zeichenkette über das `CMV_IN`-Makro importiert werden (siehe Kapitel 4.4.8), würde nur die Speicheradresse kopiert. Eine *flache Kopie* wird erstellt [159]. Das Makro `CMD_CONST_INSTANCES_HANDLING` in Zeile 15 definiert deshalb spezielle Konstruktor- und Destruktoraufrufe, die aufgerufen werden, sobald Variablen des angegebenen Typs erzeugt bzw. zerstört werden. Für den Typ `str` legt der Konstruktor `str_Cstr` eine *tiefe Kopie* der Variablen an [159].

Zeile 17 bis 20 zeigt die Definition von Funktionsaufrufen im DefMI. Es wird zwischen *Prozeduren* und *Funktionen* unterschieden. Funktionen, deklariert durch das Makro `CMD_FUNCTION`, haben einen Rückgabewert und entsprechen klassischen C++ Funktionen mit return-Wert. Prozeduren dagegen, deklariert durch das `CMD_PROCEDURE`-Makro, entsprechen C++ Funktionen ohne Rückgabewert bzw. mit Rückgabewert `void`.

Im CMF werden Funktionen und Prozeduren als *Instruktionen* bezeichnet. Aufrufe von Instruktionen können Parameter besitzen, wobei zwischen Eingabe- und Ausgabeparametern unterschieden wird. Ausgabeparameter können während der Ausführung der Instruktion verändert werden, bei Eingabeparametern ist das nicht möglich[60]. Die differenzierte Betrachtung ist insbesondere bei der Optimierung des Zwischencodes von Bedeutung (Kapitel 4.4.11).

Das Suffix im Bezeichner eines DefMI-Makros gibt die Anzahl der Eingabe- und Ausgabeparameter der Funktion bzw. Prozedur an. In Zeile 17 weist das Suffix `o1` darauf hin, dass die Prozedur genau einen Ausgabeparameter (`o`, engl. output) hat. Die Funktion in Zeile 19 hat dagegen zwei Eingabe- und einen Ausgabeparameter (`i`, engl. input).

Der erste Parameter im `CMD_PROCEDURE`-Makro gibt den Namen der C++ Funktion an, dessen Aufrufe aufgesammelt werden sollen. Der nächste Parameter spezifiziert, ob die Anweisung *Seiteneffekte* hat, d. h. ob sich durch die Ausführung der Anweisung der Zustand (bis auf die Änderung der Ausgabeparameter) des Programmes ändert. Der dritte Parameter legt fest, ob es sich bei der Funktion

[58] Kopierbare Variablen-CCWs beschreiben Basistypen wie z. B. `int`, `long`, `bool`.
[59] Dazu gehören z. B. Variablen vom Typ `bnz_t`, die nicht direkt kopiert werden können.
[60] Streng genommen kann eine Änderung der Variable immer noch explizit durch den Programmierer mithilfe von `const_cast` verändert werden [159].

um einen speziellen Konstruktor-/Destruktoraufruf handelt[61]. Beide Parameter sind für die Optimierung des Zwischencodes relevant (Kapitel 4.4.11). Die folgenden Parameter geben die Variablentypen der Eingabe- und Ausgabeparameter an, wobei die Typen der Eingabeparameter als letztes deklariert werden. Die Ausgabeparameter der aufzusammelnden Funktion werden immer als Referenzparameter übergeben[62]. Eingabeparameter müssen entweder als Referenz- (bei einfachen Variablen) oder als Wertparameter (bei kopierbaren Variablen) definiert werden[63].
Die Beschreibung der aufzusammelnden Instruktionen über das DefMI gewährleistet die Typsicherheit. Fehlerhafte Aufrufe werden bereits zur Kompilierzeit erkannt.

Das `CMD_FUNCTION`-Makro wird für das Aufsammeln der Funktionsaufrufe mit Rückgabewert verwendet. Es entspricht dem Aufbau des `CMD_PROCEDURE`-Makros. Lediglich ein zusätzlicher Parameter am Anfang der Parameterliste definiert den Rückgabetyp (Zeile 19). Nur kopierbare Typen sind als Rückgabewert erlaubt.

Falls der Zeiger auf eine Variable als Parameter einer Instruktion übergeben werden soll, ist der Typ der Variablen mit dem Makro `POINTER(..)` zu kennzeichnen (siehe Zeile 20).

Deaktivierung des CMF

```
1  int ccI;        // CMV(int ccI);
2  int* ccI2;      // CMV_POINTER(int, ccI2);
3  int ccI2[2];    // CMV_ARRAY(int, ccI2, 2);
4  func(ccI);      // CMI(func(ccI));
```

Listing 4.4: Auflösung der in Listing 4.1 verwendeten CallMI-Makros bei Deaktivierung des CMF.

Während der Quellcode kompiliert wird, muss das Makro `CM_ENABLE_CODE_COLLECTING` global definiert sein, damit Anweisungen vom CMF aufgesammelt werden können. Soll das Crypto-Framework ohne Compiler-Funktionalität verwendet werden, wird das Makro nicht definiert und die Markierungen durch die CallMI-Makros durch den Präprozessor entfernt (siehe Listing 4.4). Bei der Ausführung des Crypto-Frameworks ohne Backend kommt es zu keinerlei Performanceeinbußen.

[61] Die Funktionen `bnz_init` und `bnz_clear` zur Erzeugung bzw. Zerstörung einer Langzahl sind spezielle Konstruktor- bzw. Destruktoraufrufe.
[62] Die Funktion `bnz_init()` hat nach Zeile 17 z. B. einen Ausgabeparameter. Der Prototyp der Funktion in Zeile 5 definiert den Parameter entsprechend als Referenzparameter.
[63] Variablen vom Typ `int` werden z. B. als Wertparameter während Variablen vom Typ `bnz_t` als Referenzparameter übergeben werden (Zeile 14).

```
1  extern void func(int);
2
3  void cc_ToCollect()
4  {
5    int i1;
6    int i2;
7    CMV(int ccI1);
8    CMV(int ccI2);
9    func(i1);          // Ok
10   CMI(func(ccI1));   // Ok
11   i2 = i1;           // Ok
12   ccI2 = ccI1;       // Ok
13   i2 = ccI1;         // Verboten!
14   ccI2 = i1;         // Verboten!
15   func(ccI2);        // Verboten!
16   CMI(func(i2));     // Verboten!
17 }
```

Listing 4.5: Einige typische Anwendungsfehler beim Markieren des aufzusammelnden Quellcodes.

4.4.3 Collected Code Wrappers

Das CallMI bildet markierte Quellcodeabschnitte auf CCWs ab. Es gehört zu jeder markierten Variable im Quellcode ein CCW (auch *Variablen-CCW* genannt). Dadurch ist es möglich, den Destruktoraufruf markierter Variablen zu erfassen, sobald ihr Gültigkeitsbereich verlassen wird[64].

Häufig wird aufsammelbarer und nicht-aufsammelbarer Code im selben Projekt verwendet. Dadurch dass CCWs und nicht-markierte Anweisungen in unterschiedlichen Namensräumen ausgeführt werden, können viele Markierungsfehler bereits zur Kompilierzeit erkannt werden.
Listing 4.5 veranschaulicht das anhand eines Beispiels. Am Anfang werden vier Variablen des Typs `int` deklariert, von denen die letzten beiden als aufsammelbar markiert sind und die ersten beiden nicht. Zeile 12 zeigt eine Zuweisung von zwei CCWs. Dabei handelt es sich um eine *implizite Markierung*, denn die Anweisung wird vom CMF aufgesammelt, obwohl sie nicht markiert ist. Der Grund ist, dass beide Variablen `ccI1` und `ccI2` bereits markiert sind und die zugehörigen CCWs im CM-Namensraum existieren. Der Zuweisungsoperator ist überladen, so dass eine Markierung nicht notwendig ist.

Die Verwendung von markiertem und nicht-markiertem Code in derselben Zuweisung bzw. Anweisung führt zu einem Fehler (siehe Zeile 13 bis 16). Aufgrund

[64] Im Einführungsbeispiel endet der Gültigkeitsbereich der Variablen `ccI`, nachdem die Funktion `my_Func()` ausgeführt wurde. Die rechte Seite mit der CCR-Struktur zeigt, dass der Destruktoraufruf für `ccI` aufgesammelt wurde.

der unterschiedlichen Namensräume des Quellcodes und der CCWs können Fehler dieser Art bereits zur Kompilierzeit entdeckt werden. Benutzt der Programmierer eine moderne Entwicklungsumgebung wie z. B. Eclipse [58], werden Markierungsfehler bereits während des Programmierens angezeigt.

Ein CCW führt die Funktionalität der aufgesammelten Anweisung nicht aus, sondern gibt die für die Ausführung notwendigen Informationen an den Code-Manager weiter. Das ist erforderlich, weil z. B. das Aufsammeln von Verzweigungen im Programmfluss (If-Then-Else-Blöcke) eine sofortige Ausführung nicht ermöglichen (siehe Kapitel 4.4.10).

Typumwandlungen

Werden Operationen auf Variablen mit Basistypen ausgeführt und sind die Typen unterschiedlich, führt der C / C++ Compiler automatisch eine Typumwandlung der Operanden durch. Da der aufgesammelte Code durch CCWs, d. h. durch C++ Objekte repräsentiert wird, kann dieses *implizite Casting* vom CMF nicht immer vorgenommen werden (siehe Listing 4.6).

```
1  CMV(short ccS);
2  CMV(long ccL);
3  CMV(int ccI);
4
5  ccL = ccS;                                          // Ok
6  ccI << ccS;                                         // Ok
7  ccS = ccL;                                          // Warnung
8
9  ccL = ccS + ccS;                                    // Warnung
10 ccI = ccI + ccL;                                    // Fehler!
11
12 ccL = CMV_COPYCONSTR(int, ccS) +
     ↳  CMV_COPYCONSTR(int, ccS);                      // Ok
13 ccI = ccI + CMV_COPYCONSTR(int, ccL);               // Warnung
14
15 ccL = CMV_CAST(long, ccS) + CMV_CAST(long, ccS);    // Ok
16 ccI = ccI + CMV_CAST(int, ccL);                     // Ok
17
18 ccL = CMV_CAST(long, ccS + ccS);                    // Warnung
```

Listing 4.6: Typumwandlungen markierter Variablen.

In Zeile 5 bis 7 werden Variablen unterschiedlichen Typs zugewiesen. Ähnlich wie bei nicht-markierten Variablen gibt der Compiler[65] für Zeile 7 eine Warnung aus, da der Wertebereich einer `short` Variablen kleiner als der Wertebereich einer `long` Variablen ist.

[65] g++ Version 4.6.3 mit Parameter -Wall [72]

In Zeile 9 und 10 werden Operationen auf markierten Variablen unterschiedlichen Typs ausgeführt. Im ersten Fall gibt der Compiler eine Warnung aus. Anders als bei nicht-markiertem Code bricht der Compiler im zweiten Fall die Kompilierung mit einem Fehler ab.
Zeile 12 und 13 zeigen die Lösung des Problems. Es wird der Kopierkonstruktor [159] der Variablen-CCW aufgerufen.
Für Zeile 13 gibt der Compiler immer noch eine Warnung aus, da eine Variable vom Typ `long` auf einen kleineren Wertebereich abgebildet wird. Die Lösung ist ein *explizites Casting* durch das CallMI-Makro `CMV_CAST`. Die Anwendung des Operators ist in Zeile 15 und 16 dargestellt.

Zeile 18 zeigt einen Sonderfall. Bei der Anweisung kommt es trotz explizitem Castings zu einer Compilerwarnung. Die Ursache ist eine Typaufweitung (engl. *Promotion*), bei der ein C / C++ Compiler aus Performancegründen die Operandentypen in Typen mit größerem Wertebereich umwandelt [87]. Das temporäre Ergebnis der Addition ist vom Typ `short`. Das CallMI-Makro `CMV_CAST` konvertiert das Ergebnis in `long`. Da aber der Compiler die Operanden vom Typ `short` durch die Typaufweitung intern vor der Addition nach `int` casted, kommt es zu einer Compiler Warnung.

4.4.4 Code-Manager

Der Code-Manager ist im CMF die zentrale Instanz zur Verwaltung des markierten Quellcodes und ist nach dem Singleton-Entwurfsmusters der „Gang of Four“ entworfen [67]. Intern besteht der Code-Manager aus einer öffentlichen, globalen Instanz einer Klasse, auf die alle CCWs und das CallMI Zugriff haben. Der Code-Manager wird durch Verwendung des Makros `CM_INIT_CODE_MANAGER` initialisiert und hat zwei Aufgaben.

- Der Code-Manager gibt die von den CCWs erhaltenen Informationen über den aufzusammelnden Quellcode an die aktiven Code-Collectors weiter (siehe Kapitel 4.4.5). Code-Manager und Code-Collectors stellen nach der „Gang of Four“ einzelne *Observer* (dt. Beobachter) dar, die über Aufrufe des eingewickelten Quellcodes benachrichtigt werden [67].
- Der Code-Manager sorgt für die korrekte Ausführung der markierten Quellcodeabschnitte. Das realisiert ein eigener Code-Collector, der während der gesamten Ausführungszeit alle markierten Anweisungen aufsammelt. Der Code-Manager entscheidet nach jedem Aufruf, ob die Anweisung ausgeführt werden soll[66].

[66] Bei Verzweigungen wird der Code z. B. erst später ausgeführt (siehe Kapitel 4.4.10)

```
1  CodeCollector cc("collector");
2  cc.beginSubProgram("addition");
3    CMV(int ccIout);
4    CMV_ARRAY(int, ccIin, 2);
5
6    ccIout = ccIin[0] + ccIin[1];
7
8    cc.addOutputVariable(ccIout, "iOut");
9    cc.addInputVariable(ccIin, "iIn");
10 cc.endSubProgram("addition");
```

Listing 4.7: Setzen von Eingabe- und Ausgabeparametern eines aufgesammelten Unterprogrammes.

4.4.5 Code-Collector

Der Code-Collector erzeugt aus CCWs die entsprechenden CCRs, den Zwischencode des CMF (siehe Kapitel 4.4.6). Der Programmierer hat über den Code-Collector die Möglichkeit, markierte Quellcodeabschnitte zu gruppieren und mit Eingabe- und Ausgabeparametern zu versehen. Gruppierte Quellcodeabschnitte werden in dieser Arbeit als *Unterprogramme* bezeichnet. Sie sind vergleichbar mit C-Funktionen, können im Gegensatz dazu aber weitere Unterprogramme enthalten. Die oberste Hierarchieebene wird als *globaler Code* bezeichnet.

Listing 4.1 zeigt die Anwendung eines Code-Collectors. Zeile 29 instanziiert den Code-Collector `cc`. Der Code-Collector registriert sich beim Code-Manager als *Observer*, so dass er über jede aufgesammelte Anweisung informiert wird [67]. Die Methode `beginSubProgram()` in Zeile 30 und `endSubProgram()` in Zeile 32 gruppieren Quellcodeabschnitte. Alle markierten Anweisungen zwischen den beiden Methoden, d. h. der markierte Quellcode der Funktion `cc_ToCollect()` wird zu einem Unterprogramm mit dem Bezeichner `"collected"` zusammengefasst. Das „globale Programm“ hat den Bezeichner `"collector"`.
Auf diese Weise lassen sich beliebig viele (verschachtelte) Unterprogramme erstellen. Im weiteren Verlauf dieser Arbeit wird darauf verzichtet, da die Zielsprache ANSI-C ist und verschachtelte Funktionen (sogenannte *nested functions* [159]) im Standard nicht unterstützt werden.

Unterprogramme lassen sich durch Eingabe- und Ausgabevariablen parametrisieren. In Listing 4.7 addiert das Unterprogramm zwei Variablen `ccIin[0]` und `ccIin[1]` und speichert das Ergebnis in `ccIout`. Die Variablen werden durch die Funktionen `addInputVariable` und `addOutputVariable` als Eingabe- bzw. Ausgabeparameter definiert und erwarten als Parameter ein CCW und einen Bezeichner, der im kompilierten Code den Namen des Parameters angibt.
Das Beispiel zeigt, dass auch Array-Variablen als Parameter verwendet werden können.

4.4.6 Collected Code Representatives

Der Zwischencode ist eine Datenstruktur, die durch die Code-Collectors erstellt wird und aus miteinander verknüpften CCR-Objekten besteht. Die Ausdrucksmächtigkeit einzelner CCR-Objekte ist gering, so dass in der Regel mehrere CCR-Objekte notwendig sind, um komplexe Anweisungen im Zwischencode zu beschreiben. Das erhöht zwar die absolute Anzahl an CCRs, eine niedrigere Abstraktionsebene führt aber zu einer besseren Optimierung des Zwischencodes und einer einfacheren Übertragung des Zwischencodes auf die Zielsprache.

CCRs lassen sich klassifizieren als *Variablen*, *Variablen-Instruktionen*, *Instruktionen* und *Sichtbarkeitsblöcke*. Daneben gibt es CCRs vom Typ `CommentCCR` zum Einfügen von zusätzlichen Kommentaren in den Zielcode.
Abbildung 4.6 zeigt eine vereinfachte Darstellung des Klassendiagrammes aller CCRs. Die Basisklasse ist die abstrakte Klasse `CCR`, von der die beiden Klassen `VariableCCR` und `ExecutableCCR` abgeleitet werden. Die erste Klasse repräsentiert Variablen, während die zweite Klasse zur Beschreibung von Anweisungen (z. B. Funktionsaufrufen) und Gültigkeitsbereichen dient. `ExecutableCCR`-Objekte werden auch als *Code-Objekte* bezeichnet. Ein Gültigkeitsbereich beschreibt einen Codeabschnitt mit Anweisungen und Variablen, die nur in diesem Abschnitt gültig sind.

4.4.7 CCR-Variablen

CCR-Variablen entsprechen Variablen-CCWs im Zwischencode und lassen sich wie Variablen-CCWs in einfache und kopierbare Variablen unterteilen (siehe Kapitel 4.4.2). Da einfache Variablen nur eine Einschränkung von kopierbaren Variablen sind, beschränken sich die folgenden Erläuterungen auf kopierbare CCR-Variablen.
Im Zwischencode ist allen Variablen ein Konstruktor und Destruktor zugewiesen. Variablen werden am Anfang eines Gültigkeitsblockes durch den Konstruktor erzeugt und am Ende durch den Destruktor zerstört.

Der Zugriff auf Variablen kann entweder lesend oder schreibend erfolgen. Aus diesem Grund hat jede Variable zwei Listen, in denen die Schreib-/Lesezugriffe festgehalten werden. Das ist insbesondere zur Optimierung des Zwischencodes hilfreich (siehe Kapitel 4.4.11).

4.4.8 CCR-Variablen-Instruktionen

Die Klasse `VarSpecialCallCRR` ist die Basisklasse aller Variablen-spezifischen Anweisungen. Neben den beiden CCR-Klassen `ConstructorCallCCR` und `DestructorCallCCR`, welche die Konstruktor- und Destruktoraufrufe der Varia-

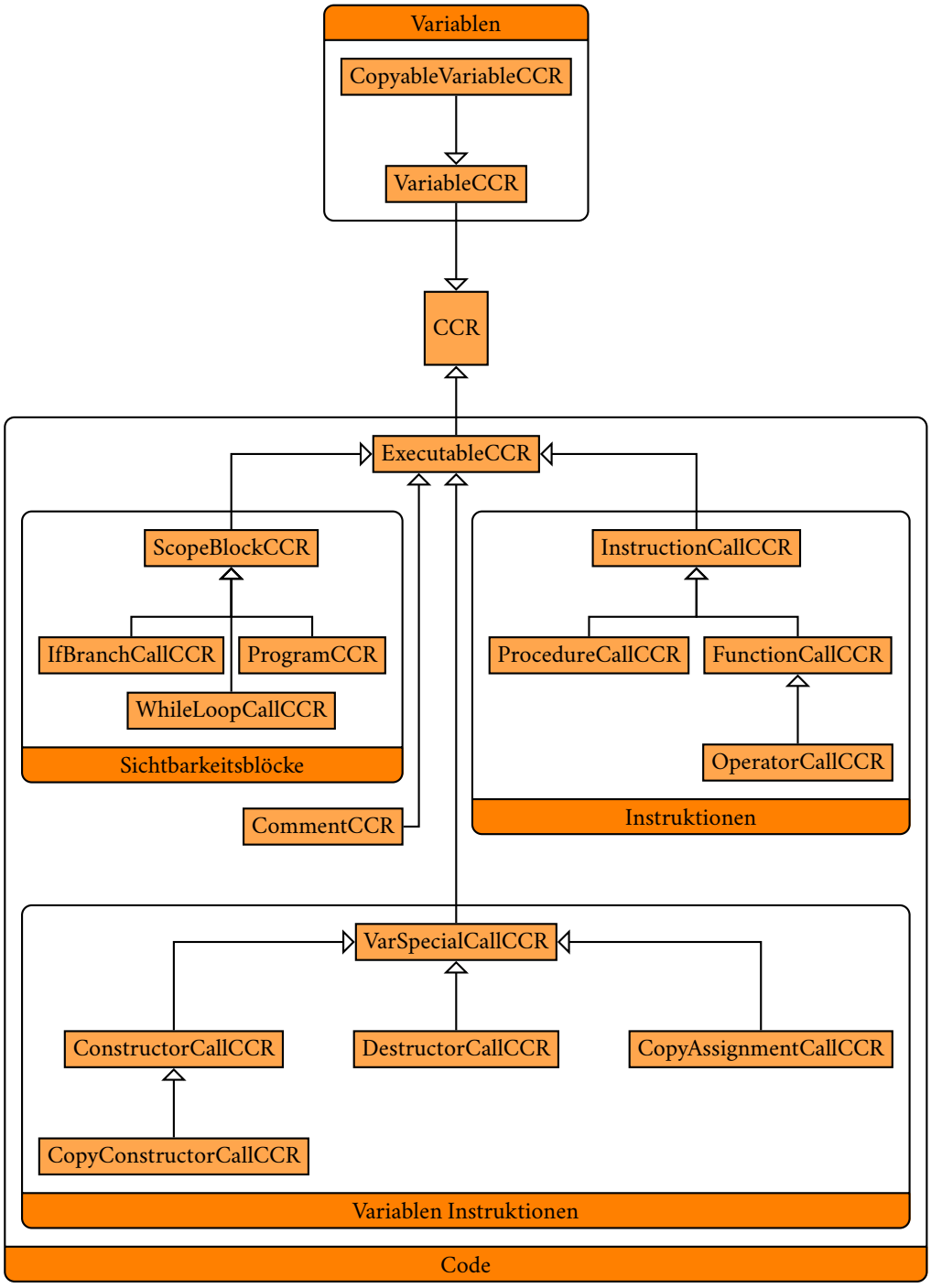


Abbildung 4.6: CCR-Klassenhierarchie.

blen abstrahieren, gibt es zwei weitere Klassen, die Variablenzuweisungen in der Zwischensprache repräsentieren. Die Klasse `CopyAssignmentCallCCR` repräsentiert eine Variablenzuweisung durch den Ist-Gleich-Operator während die Klasse `CopyConstructorCallCCR` die Zuweisung durch den Kopierkonstruktor beschreibt (siehe Kapitel 4.4.3).

Quellcode	CCR-Pseudocode
1 `int i1;`	1
2 `int i2;`	2
3 `CMV(int ccI1);`	3 `int& ccI1 = *new int();`
4 `CMV(int ccI2);`	4 `int& ccI2 = *new int();`
5 `CMV(str ccStr);`	5 `str& ccStr = *new str_Cstr();`
6 `i1 = 23;`	6
7 `ccI1 = CMV_IN(42);`	7 `const int& tmp1=*new int(42);`
8	8 `ccI1 = tmp1;`
9	9 `delete &tmp1;`
10 `ccI2 = CMV_IN(i1);`	10 `const int& tmp2=*new int(23);`
11	11 `ccI2 = tmp2;`
12	12 `delete &tmp2;`
13 `ccStr = CMV_IN((str)"Foo");`	13 `const str& tmp3 = *new str_Cstr("Foo");`
14	14 `ccStr = tmp3;`
15	15 `delete &tmp3;`
16 `ccI1 = ccI2;`	16 `ccI1 = ccI2;`
17 `CCW_TYPE(int) ccI3 = ccI2;`	17 `int& ccI3 = *new int(ccI2);`
18 `CCW_TYPE(int)* pCCI = new CMV(int);`	18 `int* pCCI1 = new int();`
19 `delete pCCI;`	19 `delete pCCI1;`
20 `pCCI = new CMV(int);`	20 `pCCI2 = new int();`
21 `delete pCCI;`	21 `delete pCCI2;`
22 `i2 = CMV_OUT(ccI1);`	22
23	23 `delete &ccI3;`
24	24 `delete &ccStr;`
25	25 `delete &ccI2;`
26	26 `delete &ccI1;`

Listing 4.8: Verwendung von speziellen Variablen-Instruktionen.

Listing 4.8 veranschaulicht die Verwendung der Variablen-Instruktionen. Die linke Seite zeigt den markierten Quellcode. Auf der rechten Seite steht der entsprechende Zwischencode in Form von CCR-Pseudocode. Jede nicht-leere Zeile auf der linken Seite wird durch ein CCR repräsentiert.
Zeile 3 und 4 zeigen, wie durch das CallMI-Makro `CMV(..)` Konstruktoraufrufe als CCRs erzeugt werden. Destruktoraufrufe in Form von `DestructorCallCCR`-Objekten werden am Ende des Gültigkeitsbereiches automatisch eingefügt (z. B.

Zeile 23 bis 26).[67]

Eine markierte Variable lässt sich einer anderen markierten Variablen direkt zuweisen (siehe Zeile 16), vorausgesetzt beides sind kopierbare Variablen nach Kapitel 4.4.7. In diesem Fall wird ein CCR vom Typ `CopyAssignmentCallCCR` angelegt.

Einen Sonderfall für einen Konstruktoraufruf zeigt Zeile 17. Die Zuweisung erfolgt durch den Aufruf des Kopierkonstruktors der Variablen `ccI3`. Im Gegensatz zu den vorherigen Definitionen wird der Typ des CCWs explizit durch das Makro `CCW_TYPE` verwendet und nicht das CallMI-Makro `CMV`. Die Repräsentation der Anweisung im Zwischencode entspricht einem `CopyConstructorCallCCR`-Objekt.
In Zeile 18 wird ein CCW dynamisch auf dem Heap erzeugt und in der nächsten Zeile wieder zerstört. Entsprechende CCWs und CCRs werden erzeugt. Der Zeiger auf das CCW wird in Zeile 20 wiederverwendet, um eine weitere Variable dynamisch zu erzeugen. Aufgrund des Destruktoraufrufes können die CCWs und CCRs nicht wiederverwendet werden. Zur Lösung dieses Problems wird bei jedem Aufruf eines CCW-Konstruktors ein neues CCW- und CCR-Objekt erzeugt. Die Objekte haben eine eindeutige Identifikationsnummer. Ursprüngliche Variablen-Bezeichner aus dem Quellcode – falls diese vorhanden sind – werden ausschließlich zu Debugging-Zwecke mit aufgesammelt.

Variablen Im- und Export

Einen besonderen Konstruktoraufruf zeigt Zeile 7 (Listing 4.8). Das CallMI-Makro `CMV_IN` sammelt externe Werte auf und importiert sie in das CMF. Dabei werden drei CCRs erstellt. Zuerst wird eine temporäre Variable erzeugt und mit dem Wert `42` initialisiert. Danach wird in Zeile 8 die temporäre Variable der Variablen `ccI1` zugewiesen. Die temporäre Variable wird nicht mehr gebraucht, so dass in Zeile 9 ihr Destruktor aufgerufen wird.
Zeile 13 zeigt das Importieren einer Zeichenkette durch Aufruf des speziellen Konstruktors `str_Cstr` (vgl. Kapitel 4.4.2).

Nicht nur Konstanten können importiert werden, sondern auch nicht-markierte Variablen (siehe z. B. Zeile 10). Der aktuelle Wert der Variablen `i1` wird ausgelesen und `ccI2` zugewiesen.

Um Konstanten und Variablen aus dem CMF zu exportieren, wird das CallMI-Makro `CMV_OUT` verwendet (siehe z. B. Zeile 22). Der aktuelle Wert der Variablen `ccI1` wird der Variablen `i2` zugewiesen. Kein Quellcode wird aufgesammelt.

[67] Wenn CCWs ihren Gültigkeitsbereich verlieren, wird ihr Destruktor aufgerufen und der Code-Manager informiert. Der Code-Manager gibt die Information über den Aufruf an die Code-Collectors weiter, die für das CCW `DestructorCallCCR`-Objekte erstellen.

4.4.9 CCR-Instruktionen

Die Klasse `InstructionCallCCR` repräsentiert den Aufruf einer Instruktion (Prozeduren, Funktionen und Operatoren). Es wird zwischen *Funktionsaufrufen* (`FunctionCallCCR`) und *Prozeduraufrufen* (`ProcedureCallCCR`) unterschieden. Funktionsaufrufe haben mindestens einen Rückgabeparameter und entsprechen vom Konzept C-Funktionen mit Rückgabewert. Operatoren werden durch die Klasse `OperatorCallCCR` beschrieben, die eine Spezialisierung der Klasse `FunctionCallCCR` darstellt.

Operatoren

Quellcode	CCR-Pseudocode
`CMV(int ccI1);`	`int& ccI1 = *new int();`
`CMV(int ccI2);`	`int& ccI2 = *new int();`
`CMV(int ccI3);`	`int& ccI3 = *new int();`
`ccI2 = -(ccI1 = CMV_IN(42));`	`const int& tmp1 = *new int(42);`
	`ccI1 = tmp1;`
	`int& tmp2 = *new int();`
	`tmp2 = -ccI1;`
	`ccI2 = tmp2;`
	`delete &tmp2;`
	`delete &tmp1;`
`ccI3 = ccI2 + CMV_IN(2) * ccI1;`	`const int& tmp3 = *new int(2);`
	`int& tmp4 = *new int();`
	`tmp4 = tmp3 * ccI1;`
	`int& tmp5 = *new int();`
	`tmp5 = ccI2 + tmp4;`
	`ccI3 = tmp5;`
	`delete &tmp5;`
	`delete &tmp4;`
	`delete &tmp3;`
	`delete &ccI3;`
	`delete &ccI2;`
	`delete &ccI1;`

Listing 4.9: Aufsammeln von Operatoren.

Dem Programmierer stehen die folgenden Operatoren zur Verfügung:

- *Arithmetische Operatoren:*
  +, -, *, /, %, ++, --
- *Logische Operatoren:*
  ==, !=, >, <, >=, <=, !

- *Bitweise Operatoren:*
  ~, &, |, ^, <<, >>
- *Kombinierte Zuweisungsoperatoren:*
  +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=

Listing 4.9 zeigt, wie Ausdrücke mit Operatoren im Zwischencode abgebildet werden und komplexe Ausdrücke in einzelne atomare Operationen zerlegt werden. Jedes CCR vom Typ `OperatorCallCCR` repräsentiert eine Operator-Operation. Jede Operation liefert einen Rückgabewert zurück, der Operand einer weiteren Operation sein kann.

Die beiden logischen Operatoren `&&` (und) und `||` (oder) sind Sonderfälle, da sie den Kontrollfluss des ausgeführten Programmes verändern. Liefert z. B. die Auswertung des linken Operanden eines `&&`-Ausdruckes den Wert `false` zurück, wird der rechte Operand nicht mehr ausgeführt[159]. Da die Syntax eines Ausdruckes nicht mit aufgesammelt wird, werden die beiden logischen Operatoren durch das CMF verboten. Stattdessen lässt sich dieselbe Logik durch die Operatoren `&` und `|` (beide Seiten werden ausgewertet) sowie If-Then-Else-Konstrukte (siehe Kapitel 4.4.10) beschreiben.

4.4.10 CCR-Sichtbarkeitsblöcke

Die Klasse `ScopeBlockCCR` beschreibt einen Quellcode-Block von Code-Objekten und Variablen mit demselben Gültigkeitsbereich. Ein `ScopeBlockCCR`-Objekt enthält neben einer Liste von Variablen eine Referenz auf das erste und letzte `ExecutableCCR`-Objekt. `ExecutableCCR`-Objekte sind untereinander verlinkt, um die Ausführungsreihenfolge zu beschreiben. Da ein `ScopeBlockCCR`-Objekt auch ein `ExecutableCCR`-Objekt ist, lässt sich ein Quellcode-Block vom CMF wie eine einzelne Anweisung behandeln und kann z. B. weitere `ExecutableCCR`-Objekte als Vorgänger bzw. Nachfolger haben.
Jeder Variablen ist ein Sichtbarkeitsbereich in Form einer Referenz auf ein `ScopeBlockCCR`-Objekt zugewiesen.

Die Klassen `ProgramCCR`, `IfBranchCallCCR` und `WhileLoopCallCCR` beschreiben drei spezielle Sichtbarkeitsblöcke. Die erste Klasse repräsentiert Unterprogramme mit Eingabe- und Ausgabevariablen, die zweite Klasse dient zur Beschreibung von Verzweigungen und die letzte Klasse beschreibt Schleifen im Programmfluss.

Unterprogramme

Unterprogramme werden durch die Code-Collectors verwaltet und im Zwischencode durch `ProgramCCR`-Objekte abstrahiert. Ein `ProgramCCR`-Objekt enthält

VariableCCR- und ExecutableCCR-Objekte. Da ExecutableCCR-Objekte auch ProgramCCR-Objekte sein können, lassen sich Hierarchien von Unterprogrammen beschreiben. Auch wenn keine verschachtelten Unterprogramme beabsichtigt sind, repräsentiert die oberste Ebene immer den globalen Code.
Im Einführungsbeispiel (Listing 4.1) werden zwei ProgramCCR-Objekte mit Bezeichner "collector" und "collected" erstellt. Das erste ProgramCCR-Objekt abstrahiert den globalen Code und bindet das Unterprogramm "collected" als weiteres ProgramCCR-Objekt ein.

Quellcode	CCR-Pseudocode

```
1  CodeCollector cc("global");
2  CCW_TYPE(int)* pCCI;
3
4  cc.beginSubProgram("constr");
5    pCCI = new CMV(int);
6  cc.endSubProgram("constr");
7
8  cc.beginSubProgram("init");
9    *pCCI = CMV_IN(42);
10
11
12 cc.endSubProgram("init");
13
14 cc.beginSubProgram("destr");
15   delete pCCI;
16 cc.endSubProgram("destr");
```

```
1
2  int* pCCI;
3
4  void constr() {
5    pCCI = new int();
6  }
7
8  void init() {
9    const int& tmp1 = *new
       ↳ int(42);
10   *pCCI = tmp1;
11   delete &tmp1;
12 }
13
14 void destr() {
15   delete pCCI;
16 }
```

Listing 4.10: Aufsammeln einer Variablen in unterschiedlichen Unterprogrammen.

Der Code-Collector sorgt für das richtige Setzen der Sichtbarkeitsbereiche im aufgesammelten Programm (siehe Abschnitt 4.4.5).

Die Verwendung von ein und derselben Variable in unterschiedlichen, vom Code-Collector aufgesammelten Unterprogrammen (z. B. durch die Benutzung von Zeigern) ist problematisch. In Listing 4.10, Zeile 2, wird die Variable pCCI deklariert, dessen Äquivalent im Zielcode in den drei Unterprogrammen "init", "constr" und "destr" erzeugt, initialisiert und zerstört wird.
Während der Quellcode aufgesammelt wird, erzeugt der Code-Collector in Zeile 5 das VariableCCR-Objekt pCCI und fügt es in das ProgramCCR-Objekt "constr" ein. Der erste Zugriff auf die Variable pCCI erfolgt in Zeile 9 und damit nicht nur außerhalb des Unterprogrammes "constr", sondern innerhalb eines neuen Unterprogrammes "init". Zur Lösung dieses Problems verschiebt der Code-Collector das VariableCCR-Objekt solange in der Hierarchie der Unterprogramme nach oben, bis die Variable im Unterprogramm "init" sichtbar ist. Die Variable im Zielcode wird global. Die temporäre CCR-Variable, die für die Zuweisung

nötig ist, behält dagegen ihren lokalen Sichtbarkeitsbereich im Unterprogramm "`init`" bei (Zeile 9). Beim letzten Unterprogramm "`destr`" ist der Sichtbarkeitsbereich der Variablen `pCCI` korrekt (global) und es wird ein `DestructorCallCCR`-Objekt erstellt.

Verzweigungen

Das CMF unterstützt If-Then-Else-Verzweigungen, um den Programmfluss in Abhängigkeit vom Wert einer Variablen zu beeinflussen. Die Herausforderung bei If-Then-Else-Anweisungen besteht darin, dass der Code beider Zweige aufgesammelt werden muss. Nach dem Kompilieren des aufgesammelten Codes kann erst zur Laufzeit entschieden werden, welcher Zweig ausgeführt werden soll.

If-Then-Else-Konstrukte lassen sich mithilfe der CallMI-Makros `CMM_IF`, `CMM_ELSE` und `CMM_ENDIF` aufsammeln. Zusätzlich steht das Makro `CMM_ELIF` als Kurzschreibweise für `CMM_ELSE CMM_IF` zur Verfügung. Die Makros übermitteln die notwendigen Informationen an die Code-Collectors, um die Zweige ohne Ausführung aufzusammeln. Zur Vermeidung einer fehlerhaften Verwendung der Makros (z. B. durch ein fehlendes `CMM_ENDIF`-Makro) kapseln die Makros die If-Zweige mit geschweiften Klammern.

Die linke Seite von Listing 4.11 zeigt die Verwendung der CallMI-Makros zum Aufsammeln einer If-Then-Else-Verzweigung. Die zugehörigen CCRs sind auf der rechten Seite in Form von CCR-Pseudocode und in Abbildung 4.7 dargestellt. Ein If-Then-Else-Block besteht neben einem *Then-* und einem optionalen *Else-Block* aus zwei weiteren Sichtbarkeitsblöcken, dem *Pre-* und *Post-Block*. Der Pre-Block berechnet die Verzweigungsbedingung und erzeugt bei komplexen Bedingungen temporäre Variablen, deren Destruktoren im Post-Block zusammengefasst sind.
Befindet sich das CMF im Interpreter-Modus, merkt sich der Code-Manager den aktuellen Wert der If-Bedingung und führt den richtigen Zweig aus, nachdem die komplette If-Anweisung aufgesammelt wurde.

Bei der Verwendung von markierten If-Then-Else-Konstrukten sind zwei Sonderfälle zu beachten.

- Ist das Makro `CM_ENABLE_CODE_COLLECTING` im Quellcode definiert, d. h. das CMF aktiv, werden immer beide Zweige einer markierten If-Then-Else-Anweisung aufgesammelt und der nicht-markierte Code beider Zweige ausgeführt.
  Das zeigt Listing 4.12. Der nicht-markierten Variablen `i` wird sowohl im Then- als auch im Else-Zweig ein Wert zugewiesen. Obwohl nach der Bedingung in Zeile 9 `i = 1` sein muss, ist stattdessen `i = 2`, so dass die Zuweisung in Zeile 16 nicht aufgesammelt wird.
  Wenn das CMF deaktiviert ist, wandeln die Makros die If-Anweisung

Quellcode	CCR-Pseudocode
`CMV(int ccI);`	`int& ccI = *new int();`
`ccI = CMV_IN(42);`	`const int& tmp1 = *new int(42);`
	`ccI = tmp1;`
	`delete &tmp1;`
	`const int& tmp2 = *new int(23);`
	`bool& tmp3 = *new bool();`
	`tmp3 = (tmp2 != ccI);`
	`bool& tmp4 = *new bool();`
	`tmp4 = tmp3;`
`CMM_IF(ccI != CMV_IN(23))`	`if (tmp4) {`
`  ccI = CMV_IN(0);`	`  const int& tmp5 = *new int(0);`
	`  ccI = tmp5;`
	`  delete &tmp5;`
`CMM_ELSE`	`} else {`
`  ccI = CMV_IN(1);`	`  const int& tmp6 = *new int(1);`
	`  ccI = tmp6;`
	`  delete &tmp6;`
`CMM_ENDIF`	`}`
	`delete &tmp4;`
	`delete &tmp3;`
	`delete &tmp2;`
	`delete &ccI;`

Listing 4.11: Aufsammeln einer If-Then-Else-Verzweigung.

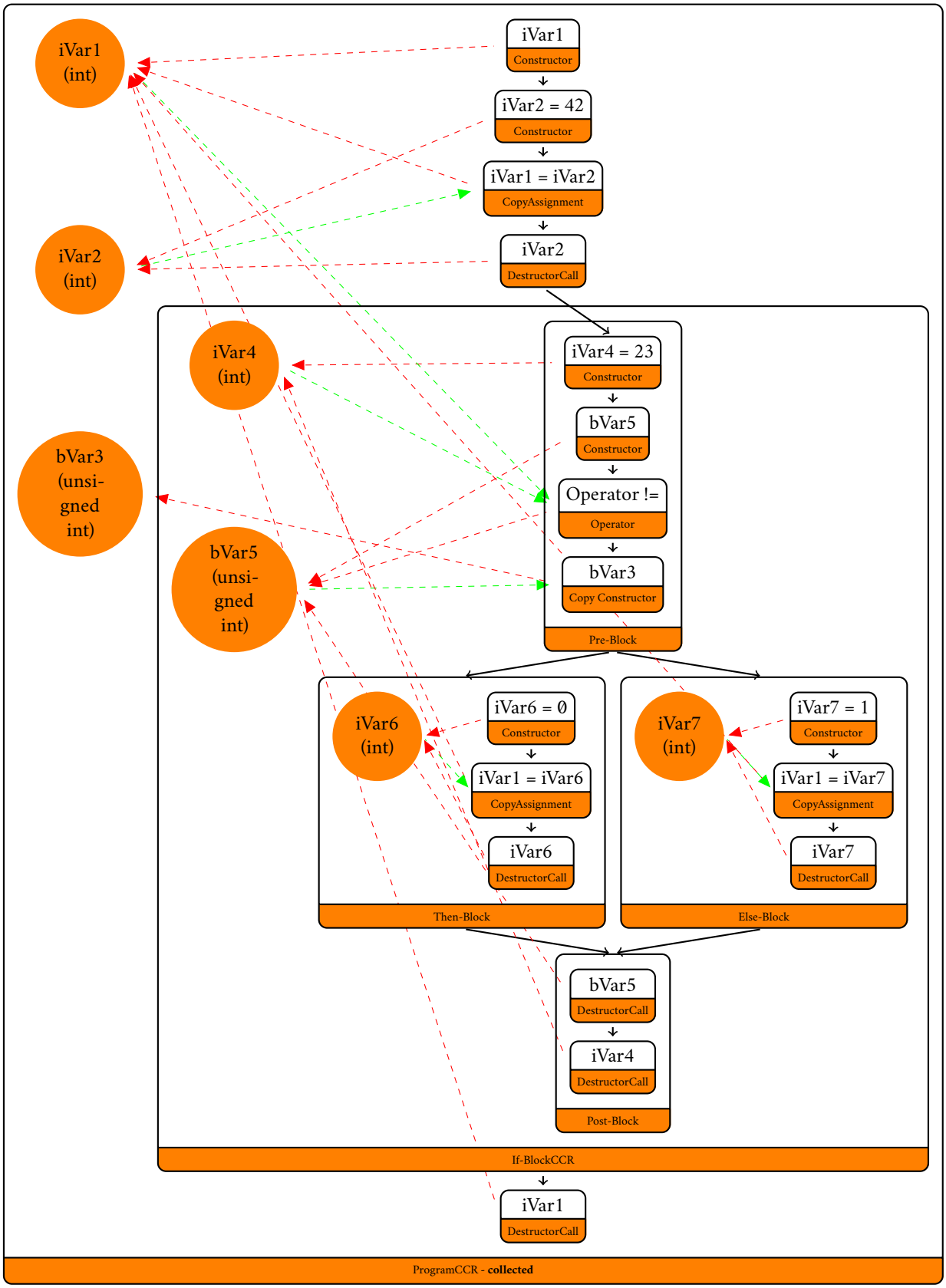


Abbildung 4.7: Visualisierung der CCR-Datenstruktur von Listing 4.11. Lesezugriffe auf eine Variable sind durch grüne, Schreibzugriffe durch rote Pfeile kenntlich gemacht.

```
1  int i = 0;
2  CMV(int ccI1);
3  CMV(int ccI2);
4  CMV(int ccI3);
5
6  ccI1 = CMV_IN(2);
7  ccI2 = CMV_IN(2);
8
9  CMM_IF(ccI1 == ccI2)
10     i = 1;
11 CMM_ELSE
12     i = 2;
13 CMM_ENDIF
14
15 if(i == 1) {
16     ccI3 = CMV_IN(3);
17 }
```

Listing 4.12: Vermischung aufsammelbare / nicht aufsammelbare Quellcodeabschnitte.

in eine konventionelle If-Anweisung um, so dass `iVar = 1` und Zeile 16 ausgeführt wird.

- Anweisungen, die den Programmfluss unterbrechen, dürfen nicht in aufgesammelten If-Then-Else-Konstrukten verwendet werden. Dazu gehören z. B. Anweisungen wie `return`, `break`, `continue`, `goto` oder `throw`. Funktionsaufrufe innerhalb der If-Then-Else-Konstrukte sind erlaubt, wenn der Kontrollfluss wieder an die Stelle des Aufrufes zurückkehrt.

Schleifen

Das CMF erlaubt die Beschreibung von while-Schleifen. Der Schleifenkörper wird durch einen Sichtbarkeitsblock und die Abbruchbedingung durch eine Verzweigung modelliert.

While-Schleifen werden mithilfe der CallMI-Makros `CMM_WHILE` und `CMM_ENDWHILE` aufgesammelt und wie bei If-Then-Else-Verzweigungen übermitteln die Makros die notwendigen Informationen an die Code-Collectors. Eine fehlerhafte Verwendung wird durch eine automatische Kapselung mit geschweiften Klammern verhindert.

Abbildung 4.13 zeigt die Verwendung der CallMI-Makros zum Aufsammeln einer While-Schleife. Die Abbruchbedingung wird wie eine If-Verzweigung durch einen Pre-, Post- und Then-Block modelliert. Der Then-Block entspricht dem Schleifenkörper.

Quellcode	CCR-Pseudocode
`CMV(int ccI1);`	`int& ccI1 = *new int();`
`CMV(int ccI2);`	`int& ccI2 = *new int();`
`ccI1 = CMV_IN(42);`	`const int& tmp1 = *new int(42);`
	`ccI1 = tmp1;`
	`delete &tmp1;`
	`bool& tmp2 = *new bool();`
`CMM_WHILE(ccI1 != ccI2)`	`bool& tmp3 = *new bool();`
	`while (tmp2 = (ccI1 != ccI2),`
	`tmp3 = tmp2,`
	`tmp3) {`
`ccI2 = ccI2 + CMV_IN(1);`	`const int& tmp4 = *new int(1);`
	`int& tmp5 = *new int();`
	`tmp5 = ccI2 + tmp4;`
	`ccI2 = tmp5;`
	`delete &tmp5;`
	`delete &tmp4;`
`CMM_ENDWHILE`	`}`
	`delete &tmp3;`
	`delete &tmp2;`
	`delete &ccI2;`
	`delete &ccI1;`

Listing 4.13: Aufsammeln einer While-Schleife.

Anweisungen zur Unterbrechung des Programmflusses dürfen im Schleifenkörper aus demselben Grund wie bei If-Then-Else-Anweisungen nicht verwendet werden.

4.4.11 Optimierungen

Bevor die CCR-Struktur in die Zielsprache übersetzt wird, bietet das CMF verschiedene Möglichkeiten, die Geschwindigkeit des kompilierten Codes zu erhöhen. Alle Variablen und Funktionsaufrufe sind als CCRs dargestellt. Dadurch ermöglicht das Konzept des CMF Optimierungen, über die z. B. ein C++ Compiler nicht verfügt.

Optionale If-Anweisungen

```
1  CMM_IF_OPT(inGroup(var, grp))
2    CMI(printf(CMV_IN((cchar_t) "Kein Gruppenelement!\n")));
3  CMM_ENDIF
```

Listing 4.14: Optionale If-Anweisungen.

Das Crypto-Framework enthält Codeblöcke, um z. B. Gruppenparameter zu überprüfen. Solcher „Validierungscode“ vereinfacht zwar das Debuggen von Anwendungen, wirkt sich aber negativ auf die Ausführungsgeschwindigkeit des kompilierten Codes aus.

Im CMF lassen sich If-Anweisungen durch das CallMI-Makro `CMM_IF_OPT` als optional kennzeichnen. Dadurch kann die gesamte If-Anweisung vom Kompilierungsprozess ausgeschlossen werden. Listing 4.14 zeigt die Anwendung des `CMM_IF_OPT`-Makros. Die Funktion `inGroup` liefert `true` oder `false` zurück, je nachdem, ob die Variable `var` in der Gruppe `grp` liegt. Der „Gruppentest“ kompromittiert nicht die Sicherheit des Protokolls und kann deshalb wahlweise weggelassen werden.

Entfernen redundanter Anweisungen

Das CMF kann redundante Instruktionen in der CCR-Struktur finden und entfernen. Zwei CCRs müssen sich im selben `ScopeBlockCCR` befinden, denselben Prozedur- bzw. Funktionsaufruf abstrahieren[68] und auf dieselben Variablen lesend zugreifen. Zur Löschung des am spätesten ausgeführten Objektes dürfen die Variablen, auf die beide Objekte lesend zugreifen, nicht zwischen den beiden

[68] unter der Voraussetzung, dass der Aufruf keine Seiteneffekte hat

```
1  CMV(int ccIin);
2  CMV(int ccI1);
3  CMV(int ccI2);
4  CMV(int ccI3);
5  ccI1 = CMI(func(ccIin));
6  ccIin = CMV_IN(3);
7  ccI2 = CMI(func(ccIin));
8  ccI3 = ccI2;
9  ...
```

Listing 4.15: Entfernen redundanter Anweisungen.

Objekten beschrieben werden.
Listing 4.15 veranschaulicht diesen Zusammenhang. Der zweite Funktionsaufruf in Zeile 7 kann z. B. nicht gelöscht werden, da die Variable `ccIin` in Zeile 6 überschrieben wird. Existiert die Zuweisung in Zeile 6 nicht, wird der zweite Funktionsaufruf aus der CCR-Struktur entfernt. Alle Variablen, auf die der zweite Funktionsaufruf schreibend zugreift, werden durch Schreibzugriffe auf das erste Objekt ersetzt. Zeile 8 würde sich z. B. zu `ccI3 = ccI1` ändern.

Der Algorithmus hat für jedes `ScopeBlockCCR` eine quadratische Ausführungszeit.

Löschen nicht-verwendeter CCR-Objekte

```
1  CodeCollector cc("collector");
2  cc.beginSubProgram("DeleteCCRs");
3      CMV(bnz_t w);
4      CMV(bnz_t w0);
5      CMV(bnz_t w1);
6      CMV(bnz_t w2);
7      CMV(bnz_t w3);
8
9      CMI(bnz_foursquares(w0, w1, w2, w3, w));
10
11     cc.addOutputVariable(w1, "outW1");
12     cc.addOutputVariable(w2, "outW2");
13     cc.addOutputVariable(w3, "outW3");
14     cc.addInputVariable(w, "inW");
15 cc.endSubProgram("DeleteCCRs");
```

Listing 4.16: Löschen nicht-verwendeter CCRs.

Variablen lassen sich aus der CCR-Struktur entfernen, wenn sie zur Ausführung des aufgesammelten Codes nicht relevant sind. Dazu gehören alle Variablen auf

```
1  CodeCollector cc("collector");
2  cc.beginSubProgram("DeleteCCRs2");
3      CMV(bnz_t bnzTest);
4      CMV(bnz_t outputVar);
5
6      CMI(bnz_init(bnzTest));
7      CMI(bnz_set_ui(bnzTest, CMV_IN((unsigned)123)));
8
9      // initialisiere SHA-256 Hashfunktion
10     CMV_POINTER(HashContext, hashContext);
11     hashContext = CMI(hashInit(CMV_IN((hashFunctions_t)
         ↳                              HASH_SHA256)));
12
13     // hash bnzTest
14     CMI(hashUpdateBnz(hashContext, bnzTest));
15
16     // speichere aktuellen Hashwert in outputVar
17     CMI(hashFinalBnz(hashContext, outputVar,
         ↳             CMV_IN((hashFunctions_t)HASH_SHA1)));
18
19     CMI(bnz_clear(bnzTest));
20
21     // aufräumen
22     CMI(hashCleanup(hashContext));
23
24     cc.addInOutVariable(output, outputVar, "outVar");
25 cc.endSubProgram("DeleteCCRs2");
```

Listing 4.17: Probleme beim Löschen nicht-verwendeter CCRs.

die schreibend zugegriffen wird[69]. Zusätzlich können Ausgabeparameter mit ausschließlich Lesezugriffen entfernt werden.

Beim Löschen einer Variablen sind auch alle `ExecutableCCR`-Objekte zu entfernen, die auf die zu löschende Variable schreibend zugreifen und die keine Seiteneffekte haben. Das ist nicht immer möglich, wie Listing 4.16 zeigt. Die Funktion `bnz_foursquares(w0, w1, w2, w3, w)` in Zeile 9 zerlegt die Variable `w` in die Quadratwurzeln `w0`, `w1`, `w2`, `w3` (siehe Kapitel 2.6.3). Alle Variablen bis auf `w0` sind als Ausgabeparameter definiert, so dass nur `w0` alle Kriterien zum Löschen erfüllt. Zum Entfernen von `w0` muss die Funktion `bnz_foursquares(w0, w1, w2, w3, w)` aus der CCR-Struktur gelöscht werden. In diesem Fall ist das allerdings nicht möglich, weil die Funktion noch schreibenden Zugriff auf die Ausgabeparameter `w1`, `w2` und `w3` hat.

Eine Variable kann nur gelöscht werden, wenn (spezielle[70]) Konstruktor- bzw. Destruktoraufrufe die einzigen Schreibzugriffe auf die Variable sind. Gibt es Schreibzugriffe von anderen `ExecutableCCR`s, wird die Variable nicht gelöscht, sondern nur zum Löschen markiert. Alle `ExecutableCCR`-Objekte, die auf die Variable schreibend zugreifen, werden hinsichtlich des Löschens überprüft. Das kann nur dann geschehen, wenn alle `VariableCCR`-Objekte zum Löschen markiert sind, auf die das `ExecutableCCR`-Objekt schreibend Zugriff hat.
Diese Bedingung ist nicht ausreichend, wie Listing 4.17 zeigt. Die Variable `hashContext` ist eine Status-Variable, die den aktuellen Status der Hash-Operation festhält. Sie wird wie die Variable `bnzTest` nur beschrieben. Damit sind beide Variablen zum Löschen markiert und `hashUpdateBnz(..)` könnte entfernt werden. Das ist unzulässig, da sich die Semantik des aufgesammelten Codes ändert. Die Instruktion `hashFinalBnz(..)` kann aufgrund der Variable `outputVar` nicht entfernt werden.
Die Folge ist, dass ein `ExecutableCCR`-Objekt nur aus der CCR-Struktur entfernt wird, wenn erstens alle Variablen, auf die das `ExecutableCCR`-Objekt schreibend zugreift, zum Löschen markiert sind und zweitens das `ExecutableCCR`-Objekt den letzten Schreibzugriff (ausgenommen (spezielle) Konstruktor- / Destruktoraufrufe) auf die Variable ausführt.

Wird ein `ExecutableCCR`-Objekt gelöscht, können auch die Lesezugriffe des `ExecutableCCR`-Objektes auf die entsprechenden Variablen entfernt werden. Das hat zur Folge, dass beim Löschen eines `ExecutableCCR`-Objektes weitere Variablen zum Entfernen in Frage kommen.
Der Algorithmus hat dadurch bezogen auf die Anzahl der Variablen im schlechtesten Fall eine kubische Laufzeit. In der Praxis wird allerdings eine fast lineare

[69] `ExecutableCCR`-Objekte haben eine Liste mit Variablen (`VariableCCR`-Objekte), auf die sie schreibend bzw. lesend zugreifen. Umgekehrt enthält ein `VariableCCR`-Objekt Informationen über Schreib-/Lesezugriffe von `ExecutableCCR`-Objekten.

[70] Spezielle Konstruktor-/Destruktoraufrufe sind z. B. `bnz_init` und `bnz_clear`.

Laufzeit erreicht, da die Abhängigkeiten zwischen den `ExecutableCCR`-Objekten in der Regel gering sind.

Lookup-Tabelle

Durch eine *LUT (Lookup Tabelle)* werden komplexe Rechenoperation durch einen einfachen Speicherzugriff ersetzt. Daten werden vorausberechnet und ihre Speicherung erfolgt in einer tabellenförmigen Datenstruktur.

Das CMF kann für eine Variable eine LUT berechnen, wenn die folgenden Anforderungen erfüllt sind.

- Die Variable muss vom Typ `bnz_t` sein und als Basis in einer modularen Exponentiation verwendet werden.
- Die Variable darf nicht in einem mit `__critical` gekennzeichneten Block geändert werden (siehe Kapitel 4.3.9) und kein Eingabe- bzw. Ausgabeparameter sein.
  Die Erstellung einer LUT ist sehr rechenintensiv. Ein Geschwindigkeitsvorteil ergibt sich nur dann, wenn die LUT während einer Initialisierungsphase erzeugt und während der (zeitkritischen) Protokollausführung verwendet wird.
- Die Bitlänge der Variable muss bekannt sein.
- Die Bitlängen der Exponenten müssen bekannt sein.

Die letzten beiden Anforderungen gewährleisten, dass die Größe der LUT bekannt ist. Die Größe einer LUT berechnet sich aus dem Produkt der Bitlänge der Variablen und der Anzahl an Einträgen. Der größte Exponent legt die Anzahl der Einträge fest. Da Exponenten auch negativ sein können, wird eine LUT sowohl für positive als auch negative Exponenten erzeugt.

Das CMF überprüft alle Variablen in der CCR-Struktur, ob die Voraussetzungen für eine LUT erfüllt sind. Ist das der Fall, werden die Exponentiationen durch eine Anweisung zur Abfrage der LUT ersetzt. CCRs werden eingefügt, die die LUT neu berechnen, sobald sich die Basis oder der Modulus einer modularen Exponentiation ändert.

4.4.12 Kompilierung des Zwischencodes

Die Kompilierung des CCR-Zwischencodes erfolgt durch Code-Generatoren, die definieren wie CCR-Objekte in der Zielsprache beschrieben wird. Das CMF kann über eine beliebige Anzahl an Code-Generatoren verfügen. Im Rahmen dieser Arbeit wurde ein Code-Generator für ANSI-C (C89 Standard) [5] implementiert.

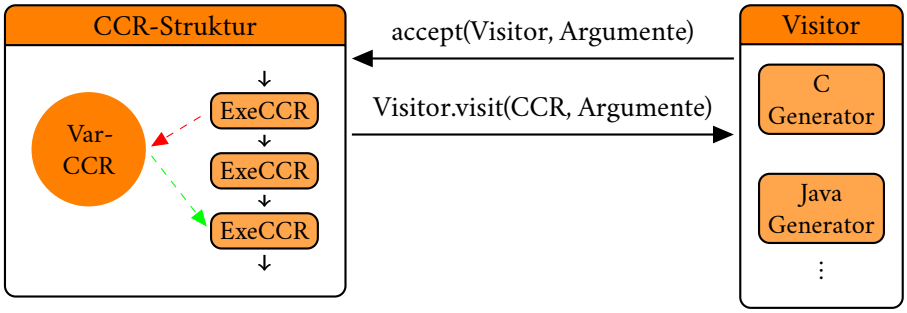


Abbildung 4.8: Visitor-Entwurfsmuster zur Kompilierung des Zwischencodes.

Des Weiteren stehen Generatoren zur Visualisierung der CCR-Struktur in DOT [143] und Tikz [162] zur Verfügung.

Die CCR-Klassen und Code-Generatoren sind nach dem Visitor-Entwurfsmuster aufgebaut (siehe Abbildung 4.8) [67]. Ein Code-Generator, d. h. ein *Visitor*, navigiert durch die CCR-Struktur, indem die `accept` Funktion der CCR-Objekte aufgerufen wird. Die `accept` Funktion ist *polymorph* [159], so dass automatisch die zum CCR-Objekt gehörige Funktion aufgerufen wird. Ihre einzige Aufgabe besteht im Aufruf der `visit` Funktion, die eine Beschreibung des CCR-Objektes in der Zielsprache enthält.

Der Vorteil bei der Verwendung des Visitor-Entwurfsmusters liegt darin, dass für jede Zielsprache nur eine Klasse mit `visit` Funktionen für jede CCR-Klasse definiert werden braucht. Dadurch ist eine Trennung zwischen der Beschreibung eines CCR-Objektes in der Zielsprache und dem Rest des CMF gewährleistet. Ein aufwendiges Casting der CCR-Objekte in der Code-Generator-Klasse ist nicht erforderlich[71].

4.5 Evaluierung

Tabelle 4.21 vergleicht den Crypto-Compilers mit anderen Compilern und Frameworks.

Das CACE (Computer Aided Cryptography Engineering) Projekt ist ein europäisches Projekt zur Entwicklung eines Frameworks für die Erstellung kryptographischer Software [27]. ZPKs werden durch eine PSL (Protocol Specification Language) beschrieben und ein Compiler überführt die Spezifikation in eine

[71] Ein `ProgramCCR`-Objekt enthält z. B. eine Liste von `ExecutableCCR`-Objekten. Zur richtigen Beschreibung des Objektes in der Zielsprache müsste beim Kompilieren des `ProgramCCR`-Objektes jedes `ExecutableCCR`-Objekt überprüft werden, um was für ein `ExecutableCCR`-Objekt es sich handelt (`IfBranchCallCCR`, `WhileLoopCallCCR` etc.). Das erfolgt durch Casting auf den entsprechenden Typ.

		CACE	cPLC	Charm	ZKPDL	Crypto-Compiler
Framework		-	-	✓	✓	✓
Compiler		✓	✓	-	-	✓
Abstraktionsgrad Eingabesprache		niedrig	hoch	niedrig	hoch	hoch
ZPK	Proofs	beliebige Gruppenhomomorphismen	beliebige Gruppenhomomorphismen in CSN	einfache Proofs	komplexe Proofs, keine „oder“-Proofs	beliebige Gruppenhomomorphismen in CSN
	Sigma	Phi, Gsp, Exp	siehe CACE	allg. Sigma Klasse	Phi, Gsp (nur nicht-interaktiv)	Phi, Gsp
	Verknüpfung	und, oder	siehe CACE	und	und	und, oder
Beschreibung komplexer Protokolle		-	✓	✓	-	✓
Optimierungen		Optimierung von ZPK-Statements	siehe CACE	-	Optimierung von ZPK-Statements, LUT	Optimierung von ZPK-Statements, globale Optimierung von Langzahloperationen, LUT
Zielsprache Compiler		C, Java, LaTeX	Java	-	-	C, Tikz, DOT
Implementierung		Python	Python	Python	C++	C++
Besonderheiten		Verification Toolbox	siehe CACE	-	-	optionale Anweisungen

Tabelle 4.21: Vergleich des Crypto-Compilers mit anderen Compilern und Frameworks.

Turing-vollständige Zwischensprache, die PIL (Protocol Implementation Language). Der Compiler wandelt den PIL Code dann in C oder Java Code um. Zusätzlich kann der CACE Compiler komplexe logische Verknüpfungen zwischen Prädikaten optimieren [4].
Der PIL Code wird mithilfe der PVT (Protokoll Verification Toolbox) verifiziert. Die Toolbox erzeugt aus der Spezifikation in der PSL und der Implementierung in der PIL einen Soundness Beweis [7] [8].
Die Eingabesprache des CACE Compilers hat einen niedrigeren Abstraktionsgrad als die Eingabesprache des Crypto-Compilers. Dadurch sind zur Beschreibung eines Proofs mehr Anweisungen in der Eingabesprache notwendig. Der CACE Compiler unterstützt neben dem Σ^{ϕ}- und Σ^{GSP}-Protokoll auch das Σ^{exp}-Protokoll, welches in der Praxis allerdings eine geringe Bedeutung hat[72].

Bangerter et al. haben eine Erweiterung des CACE Compilers vorgestellt, dessen Eingabesprache auch die Beschreibung komplexer Crypto-Protokolle erlaubt [11]. Für die Kompilierung von ZPKs wird der CACE Compiler verwendet. Im Gegensatz zur ursprünglichen Eingabesprache des CACE Compilers bietet ihre Eingabesprache einen höheren Abstraktionsgrad. So lassen sich z. B. ZPKs ähnlich der CSN beschreiben. Der Compiler unterstützt neben allgemeinen Gruppen auch Gruppen, welche über elliptische Kurven definiert sind, und Standardalgorithmen wie z. B. AES, SHA-2. Am Ende des Kompilierungsvorganges gibt der Compiler Java Code aus.

Charm ist ein auf Python basiertes Framework für das Rapid Prototyping von komplexen Kryptosystemen [3]. Es werden die kryptographischen Protokolle in einer Eingabesprache beschrieben, die sehr an die Syntax von Python angelehnt ist. Mathematischen Operationen sind als C-Module implementiert und werden während der Interpretierung der Eingabesprache ausgeführt. Akinyele et al. haben gezeigt, dass trotz Implementierung rechenintensiver Operationen in C, die Performance bei einigen Protokollen deutlich unter der Performance einer reinen C-Implementierung liegt [3].
Das Charm-Framework stellt eine Sigma-Basisklasse bereit, dass den Programmierer bei der Implementierung von Σ-Protokollen unterstützt und bietet nur eine einfache Unterstützung für ZPKs. So lassen sich z. B. keine oder-Verknüpfungen zwischen Prädikaten beschreiben.

Meiklejohn et al. haben die Eingabesprache ZKPDL zur Beschreibung von ZPKs und einen Interpreter für die Sprache vorgestellt [116]. Die Eingabesprache ist insbesondere für die Beschreibung von Proofs, die in eCash-Protokollen verwendet werden, ausgelegt. Der Interpreter führt wie der Crypto-Compiler verschiedene Optimierungen durch. Dazu gehören z. B. die Entfernung von Redundanzen in Zero-Knowledge Proofs oder die Anwendung von LUTs zur Beschleunigung von

[72] Das Σ^{exp}-Protokoll wurde im Rahmen des CACE Projektes entwickelt und zeichnet sich durch einen geringen Knowledge Error bei nicht-interaktiver Ausführung ohne Zufallsorakel aus [10].

Exponentiationen. Eine Kompilierung des Codes ist nicht möglich. Die Eingabesprache erlaubt zwar die Beschreibung komplexer Zero-Knowledge Proofs, „oder"-Verknüpfungen zwischen den Prädikaten können allerdings nicht verwendet werden. Die Ausführung der ZPKs erfolgt ausschließlich nicht-interaktiv.

Modulare Exponentiationen auf Grafikkarten

5

Die Überprüfung von SPKs, welche auf multiplikativen Gruppen basieren, lässt sich beschleunigen, indem die Berechnung der modularen Exponentiationen auf Grafikkarten durchgeführt wird. Es werden N Prover betrachtet, die einen nicht-interaktiven Zero-Knowledge Proof mit dem Provider durchführen. Der Anschaulichkeit halber besteht jeder Proof aus nur einem Prädikat und jedes Prädikat aus $N' + 1$ Exponentiationen. Alle Exponenten seien positiv. Der Provider führt die folgenden Berechnungen aus[73]:

$$\begin{array}{rrcccccc} P_0: & C'_{t,1} & = & g_0^{s_{0,0}} & \cdot \ldots \cdot & g_{N'-1}^{s_{0,N'-1}} \cdot x_0^{-c_0} \\ \vdots & \vdots & & \vdots & & \vdots \\ P_{N-1}: & C'_{t,N-1} & = & g_0^{s_{N-1,0}} & \cdot \ldots \cdot & g_{N'-1}^{s_{N-1,N'-1}} \cdot x_{N-1}^{-c_{N-1}} \end{array} \tag{5.1}$$

Die Challenge c, Response s, Commitment C_t' und öffentlicher Wert x sind entsprechend Kapitel 2.4.3 gewählt. Alle Berechnungen werden in einer RSA-Gruppe mit $g_i, x_j \in \mathbb{Z}_n^*$ für $i = 0, \ldots, N' - 1$ bzw. $j = 0, \ldots, N - 1$ durchgeführt.
Der Anschaulichkeit halber wird auf den folgenden Seiten eine modulare Exponentiation nach Gleichungssystem 5.1 als $\mathsf{x} = g^{\mathsf{e}} \bmod n$ beschrieben.

5.1 Binäre modulare Exponentiation

Bei der *binären modularen Exponentiation* wird das Ergebnis durch abwechselndes Quadrieren und Multiplizieren berechnet (siehe Algorithmus 5.1). Zu Anfang wird der Wert x mit Eins initialisiert und der Exponent e in eine Binärdarstellung

[73] Die Berechnungen entsprechen Algorithmus $\mathtt{R}(x, c, s)$ (siehe Kapitel 2.4.5).

Algorithmus 5.1 : Binäre modulare Exponentiation.

```
1 g ← Basis
2 e ← Exponent
3 x := 1
4 for e_i ← höchstwertige Bit to niedrigstwertige Bit in e do
5 |   x := x^2 mod n
6 |   if e_i = 1 then
7 |   |   x := x · g mod n
8 return x
```

überführt. Beim bitweisen Durchlaufen des Exponenten vom höchstwertigen zum niedrigstwertigen Bit wird das Zwischenergebnis in Abhängigkeit der Bits entweder quadriert (Bit ist 0) oder multipliziert und quadriert (Bit ist 1). Nach jeder Operation wird der Rest (mod n) gebildet [101]. Im Gegensatz zur einfachen Potenzierung von g^e, bei der $(e-1)$ Multiplikationen notwendig sind, sind bei der binären modularen Exponentiation nur $log_2(e) + 1/2 \cdot log_2(e)$ Multiplikationen[74] notwendig. Dadurch ergibt sich im Gegensatz zu einer linearen Laufzeit von $\mathcal{O}(e)$ bei der einfachen Potenzierung eine Laufzeit von $\mathcal{O}(log_2(e))$.

5.1.1 Chinesischer Restsatz

Die Berechnung einer modularen Exponentiation kann durch Anwendung des *Chinesischen Restsatzes* beschleunigt werden [117].

Satz 5.1.1 (Chinesischer Restsatz) *Es seien $a_0, a_1, \ldots, a_{d-1}$ für $d \in \mathbb{N}$ paarweise teilerfremde ganze Zahlen und $x_0, \ldots, x_{d-1} \in [0, a_i - 1]$ für $i = 0, \ldots, d$. Das System von linearen Kongruenzen*

$$\begin{aligned} x &\equiv x_0 \pmod{a_0} \\ x &\equiv x_1 \pmod{a_1} \\ &\vdots \\ x &\equiv x_{d-1} \pmod{a_{d-1}} \end{aligned} \tag{5.2}$$

hat die eindeutige Lösung x modulo $n = a_0 \cdot a_1 \cdot \cdots \cdot a_{d-1}$.

Quisquater and Couvreur haben gezeigt, wie mit dem chinesischen Restsatz die Verschlüsselungs- bzw. Signaturoperation beim RSA-Verfahren beschleunigt werden kann [140]. Ihre Ergebnisse lassen sich auch auf die Berechnung der modularen Exponentiation $g^e \bmod n$ anwenden, da der Provider die Faktorisierung

[74] Die Wahrscheinlichkeit ist $1/2$, dass ein Bit Eins ist.

von n in zwei sichere Primzahlen p und q kennt. Durch den chinesischen Restsatz kann die Berechnung von

$$x = g^e \bmod p \cdot q \tag{5.3}$$

in zwei Teile aufgespaltet werden:

$$x_0 \equiv g^e \pmod{p} \tag{5.4}$$
$$x_1 \equiv g^e \pmod{q} \tag{5.5}$$

Durch den *kleinen Satz von Fermat* wird das Kongruenzsystem weiter vereinfacht [117].

Satz 5.1.2 (Kleiner Satz von Fermat) *Es seien a und p zwei teilerfremde ganze Zahlen und p eine Primzahl, d. h. $ggT(a,p) = 1$, dann gilt $a^{p-1} \equiv 1 \pmod{p}$.*

Es seien a und b zwei beliebige ganze Zahlen und p eine Primzahl. Der Exponent e in Gleichung 5.3 kann umgeschrieben werden zu $e = (p-1)a + b$. Daraus folgt:

$$b \equiv e \pmod{p-1}$$

Angewandt auf die Kongruenzrelation 5.4 ergibt sich zusammen mit dem kleinen Satz von Fermat:

$$x_0 = g^e = g^{(p-1)a+b} \pmod{p} \equiv g^b \pmod{p}$$

Daraus folgt für die Kongruenzrelationen 5.4 und 5.5.

$$\begin{aligned} x_0 &\equiv g^{e \bmod p-1} \pmod{p} \\ x_1 &\equiv g^{e \bmod q-1} \pmod{q} \end{aligned}$$

Die Bitlängen der Exponenten entsprechen damit in etwa der Hälfte der ursprünglichen Bitlänge von e.

Der Anschaulichkeit halber werden die folgenden Variablen definiert:

$$\begin{aligned} x_0 &\equiv g_0^{e_0} \pmod{n_0} \quad \text{mit} \quad g_0 = g \bmod p, \quad e_0 = e \bmod p-1, \quad n_0 = p \\ x_1 &\equiv g_1^{e_1} \pmod{n_1} \quad \text{mit} \quad g_1 = g \bmod q, \quad e_1 = e \bmod q-1, \quad n_1 = q \end{aligned}$$

Rücktransformation

Um aus den Teilergebnissen $x_0, x_1, \ldots, x_{d-1}$ im Kongruenzsystem 5.2 das Endergebnis $x \bmod n$ zu berechnen, gibt es den *SRC (Single Radix Conversion)* und *MRC*

(Mixed Radix Conversion) Algorithmus [102]. Der SRC Algorithmus berechnet $x \bmod n$ durch

$$x = \sum_{i=0}^{d-1} x_i c_i Q_i \pmod{n}$$

mit $Q_i = p_0 \cdot p_1 \cdots p_{i-1} \cdot p_{i+1} \cdots p_{d-1} = \frac{n}{p_i}$ und dem multiplikativen Inversen c_i von $Q_i \bmod p_i$, d. h. $c_i Q_i = 1 \pmod{p_i}$. Für $d = 2$ mit $c_0 = q^{-1} \bmod p$ und $c_1 = p^{-1} \bmod q$ folgt:

$$x = x_0 c_0 \frac{pq}{p} + x_1 c_1 \frac{pq}{q} \bmod n \tag{5.6}$$

$$\Leftrightarrow x = (x_0 (q^{-1} \bmod p) q + x_1 (p^{-1} \bmod q) p) \bmod n \tag{5.7}$$

Um beim MRC Algorithmus das Ergebnis x zu ermitteln, wird zuerst eine Tabelle mit den folgenden Werten berechnet.

$$\begin{array}{ccccc} x_{11} & & & & \\ x_{21} & x_{22} & & & \\ x_{31} & x_{32} & x_{33} & & \\ \vdots & \vdots & \vdots & \ddots & \\ x_{d1} & x_{d2} & \cdots & \cdots & x_{dd} \end{array}$$

Die Werte werden sequentiell mit der Formel $x_{i,j+1} = (x_{ij} - x_{jj}) c_{ji} \bmod p_i$ und dem multiplikativen Inversen c_{ji} von $p_j \bmod p_i$ berechnet. Die Werte der ersten Spalte x_{i1} entsprechen den gegebenen Werten $x_{i1} = x_{i-1}$. Das Ergebnis berechnet sich aus den Elementen ganz rechts in der Tabelle mit

$$x = x_{11} + x_{22} p_1 + x_{33} p_1 p_2 + \ldots + x_{dd} p_0 p_1 \cdots p_{d-1}$$

Bei zwei Kongruenzen besteht die Tabelle nur aus den ersten beiden Zeilen. Das Ergebnis der Exponentiation lautet:

$$x = x_0 + [(x_1 - x_0) \cdot (p^{-1} \bmod q) \bmod q] \cdot p$$

5.2 Montgomery-Verfahren

1985 stellte Peter L. Montgomery einen Algorithmus vor, mit dem sich modulare Multiplikationen effizient und ohne die Notwendigkeit von Probedivisionen berechnen lassen können [120]. Es sei ℓ_n die Bitlänge des Modulus n, r eine ganze Zahl mit $r \geq 2^{\ell_n}$ und $\mathrm{ggT}(r, n) = 1$ (r wird als *Montgomery-Faktor* bezeichnet). *Montgomerys Algorithmus* berechnet für $a, b \in \mathbb{Z}_n$ mit dem multiplikativen Inversen $r^{-1} \bmod n$ von r:

$$\mathtt{MonPro}(a, b) = a \cdot b \cdot r^{-1} \bmod n$$

Üblicherweise ist r ein Vielfaches der Potenz des zugrundeliegenden Zahlensystems. Typische Werte für r sind z. B. bei Computersystemen mit einer Wortbreite von 32 Bit ein Vielfaches von 2^{32}.
Das Inverse r^{-1} von r mod n sowie der Cofaktor n' von n lassen sich mithilfe des erweiterten euklidischen Algorithmus berechnen, so dass n und r teilerfremd sind [117].

$$\mathsf{r} \cdot \mathsf{r}^{-1} - n \cdot n' = 1 \tag{5.8}$$

5.2.1 Montgomery-Multiplikation

Der Idee der *Montgomery-Multiplikation* liegt zugrunde, dass bevor die modulare Multiplikation $a \cdot b \bmod n$ ausgeführt wird, die ganzen Zahlen a und b in ihre Montgomery-Darstellung $\bar{a}$ und $\bar{b}$ überführt werden. Falls r ein Vielfaches der Potenz des zugrundeliegenden Zahlensystems entspricht, wird in der Montgomery-Darstellung aus einer rechenintensiven Modulo-Operation mit Probedivisionen eine Shift-Operation um den Faktor r. Durch Multiplikation mit r lassen sich die Operanden in die Montgomery-Darstellung überführen:

$$\bar{a} \equiv a \cdot \mathsf{r} \pmod{n}$$
$$\bar{b} \equiv b \cdot \mathsf{r} \pmod{n}$$

Die Rücktransformation erfolgt durch Multiplikation mit dem multiplikativen Inversen r^{-1}:

$$a = \mathtt{MonPro}(\bar{a}, 1) = \bar{a} \cdot 1 \cdot \mathsf{r}^{-1} = a \cdot \mathsf{r} \cdot \mathsf{r}^{-1} \bmod n$$

Zwei ganze Zahlen in der Montgomery-Darstellung können direkt miteinander addiert oder subtrahiert werden. Das Ergebnis ist wieder in der Montgomery-Darstellung. So gilt z. B. :

$$\begin{aligned} \bar{a} + \bar{b} \bmod n &= a \cdot \mathsf{r} + b \cdot \mathsf{r} \bmod n \\ &= (a + b) \cdot \mathsf{r} \bmod n \end{aligned}$$

Damit bei einer Multiplikation zweier ganzer Zahlen das Ergebnis in der Montgomery-Darstellung bleibt, muss es mit r^{-1} multipliziert werden. Dieser Schritt wird als *Montgomery-Reduktion* bezeichnet [120].

$$\begin{aligned} \mathtt{MonPro}(\bar{a}, \bar{b}) &= \bar{a} \cdot \bar{b} \cdot \mathsf{r}^{-1} \bmod n \\ &= a \cdot \mathsf{r} \cdot b \cdot \mathsf{r} \cdot \mathsf{r}^{-1} \bmod n \end{aligned} \tag{5.9}$$

Mit Gleichung 5.8 lässt sich das Montgomery-Produkt umschreiben:

$$\begin{aligned} \mathtt{MonPro}(\bar{a}, \bar{b}) &= \bar{a} \cdot \bar{b} \cdot \mathsf{r}^{-1} \bmod n \\ &= \bar{a} \cdot \bar{b} \cdot \mathsf{r}^{-1} \cdot \mathsf{r} \cdot 1/\mathsf{r} \bmod n \\ &= \bar{a} \cdot \bar{b} \cdot (1 + n \cdot n') \cdot 1/\mathsf{r} \bmod n \\ &= \left(\bar{a} \cdot \bar{b} + \bar{a} \cdot \bar{b} \cdot n' \cdot n\right) \cdot 1/\mathsf{r} \bmod n \end{aligned}$$

Algorithmus 5.2 zeigt die Berechnung des Montgomery-Produktes nach Montgomery [120] und dient als Basis für die effiziente Implementierung des Montgomery-Verfahrens (siehe Kapitel 5.3). Der Algorithmus hat den Vorteil, dass die Reduktion mit modulo n effizienter als in Gleichung 5.9 durchgeführt werden kann.

Algorithmus 5.2 : Montgomery-Multiplikations-Algorithmus $\texttt{MonPro}(a, b)$.

```
1 t := ā · b̄
2 m := t · n′ mod r
3 t := (t + m · n)/r
4 if t ≥ n then
5 |   return t − n
6 else
7 |   return t
```

Die Division mit r in Zeile 3 lässt sich als günstige Shift-Operation realisieren. Es gilt:

$$(\mathsf{t} + \mathsf{m} \cdot n) \equiv \mathsf{t} \pmod{n}$$

Damit $(\mathsf{t} + \mathsf{m} \cdot n)$ durch r teilbar ist, muss gelten:

$$\mathsf{t} + \mathsf{m} \cdot n \equiv 0 \pmod{\mathsf{r}} \tag{5.10}$$

Wird $\mathsf{m} = \mathsf{t} \cdot n' \bmod \mathsf{r}$ in die Kongruenzrelation 5.10 eingesetzt, gilt:

$$\begin{aligned} \mathsf{t} + \mathsf{m} \cdot n &= \mathsf{t} + (\mathsf{t} \cdot n' \bmod \mathsf{r}) \cdot n \\ &\equiv \mathsf{t} + \mathsf{t} \cdot n' \cdot n \pmod{\mathsf{r}} \end{aligned}$$

Mit Gleichung 5.8 ist die Kongruenzrelation 5.10 erfüllt.

$$\begin{aligned} \mathsf{t} + \mathsf{t} \cdot n' \cdot n &\equiv \mathsf{t} + \mathsf{t} \cdot (\mathsf{r}^{-1} \cdot \mathsf{r} - 1) \pmod{\mathsf{r}} \\ &\equiv 0 \pmod{\mathsf{r}} \end{aligned}$$

5.2.2 Modulare Montgomery-Exponentiation

Die Berechnung einer einzelnen modularen Multiplikation mithilfe des Montgomery-Verfahrens lohnt sich aufgrund des höheren Rechenaufwandes durch die Konvertierung und die Berechnung von n' nicht.
Bei der Berechnung von $\mathsf{x} = g^{\mathsf{e}} \bmod n$ werden allerdings mehrere Multiplikationen auf dieselbe Variable angewendet. Eine Konvertierung der Basis und des Initialisierungswertes muss nur einmal vorgenommen werden (siehe Algorithmus 5.3).

Algorithmus 5.3 : Montgomery-Exponentiation $\mathtt{MonExp}(g, \mathsf{e})$.

```
1 ḡ = g · r mod n; x̄ = 1 · r mod n
2 for e_i ← höchstwertige to niedrigstwertige Bit in e do
3 |   x̄ := MonPro(x̄, x̄)
4 |   if e_i = 1 then
5 |   |   x̄ := MonPro(x̄, ḡ)
6 return MonPro(x̄, 1)
```

5.3 Varianten der Montgomery-Multiplikation

Koç et al. vergleichen in ihrer Veröffentlichung verschiedene Montgomery-Algorithmen hinsichtlich Speicherbedarf und Performance [103]. Die Autoren unterteilen die Berechnung des Montgomery-Produktes in einen Multiplikations- und einen Reduktionsschritt und klassifizieren ihre Algorithmen auf Grundlage dieser Unterteilung. Multiplikation und Reduktion können entweder separat (*separated*) oder ineinander integriert (*integrated*) ausgeführt werden. Je nachdem, wie häufig zwischen Multiplikation und Reduktion gewechselt wird, unterscheiden die Autoren zwischen einer feinkörnigen (*fine-grained*) oder grobkörnigen (*coarse-grained*) Integration.

Es sei NO_WORDS die Anzahl der Wörter, aus denen ein Operand besteht und WORDSIZE die Wortbreite des Prozessors. Der Montgomery-Faktor r sei in dieser Arbeit definiert als:

$$\mathsf{r} = (2^{\mathtt{WORDSIZE}})^{\mathtt{NO\_WORDS}}$$

(C, S) ist das Ergebnis einer Addition mit Summe S und Carry C. Die Operation $ADD(..)$ addiert zwei Wörter unter Berücksichtigung des Carry-Flags. So addiert z. B. die Anweisung $ADD(\mathtt{t}[0], \mathsf{C})$ das Carry C zum Wort $\mathtt{t}[0]$. Entsteht dabei ein Übertrag, wird das entstandene Carry-Flag zu $\mathtt{t}[1]$ hinzuaddiert und die Addition neuer Carry-Flags solange fortgesetzt, bis kein Übertrag mehr erzeugt wird.
In den folgenden Abschnitten werden drei Algorithmen beschrieben, die als Grundlage für einen GPU-Algorithmus dienen. Vor der Ausführung der Algorithmen ist das temporäre Array t immer mit Null initialisiert.

5.3.1 SOS Variante

Bei der *SOS (Separated Operand Scanning) Variante* (siehe Algorithmus 5.4) ist die Multiplikation (Zeile 1 - 6) von der Reduktion (Zeile 7 - 13) getrennt.

Stephen Dussé und Burton Kaliski haben gezeigt, dass im Reduktionsschritt n' durch $n'[0] = n' \bmod 2^{\mathtt{WORDSIZE}}$ ersetzt werden kann [56]. Dadurch ändert sich

Algorithmus 5.4 : SOS Variante $\texttt{MonPro}(a, b)$.

```
1  for i ← 0 to NO_WORDS − 1 do
2  |  C := 0
3  |  for j ← 0 to NO_WORDS − 1 do
4  |  |  (C, S) := t[i + j] + a[j] ∗ b[i] + C
5  |  |  t[i + j] := S
6  |  t[i + NO_WORDS] := C
7  for i ← 0 to NO_WORDS − 1 do
8  |  C := 0
9  |  m := t[i] · n′[0] mod 2^WORDSIZE
10 |  for j ← 0 to NO_WORDS − 1 do
11 |  |  (C, S) := t[i + j] + m · n[j] + C
12 |  |  t[i + j] := S
13 |  ADD(t[i + NO_WORDS], C)
14 C := 0
15 for i ← 0 to NO_WORDS − 1 do
16 |  (C, S) := t[i + NO_WORDS] − n[i] − C
17 |  t[i] := S
18 (C, S) := t[2 · NO_WORDS] − C
19 if C = 0 then
20 |  return t[0], t[1], ... , t[NO_WORDS − 1]
21 else
22 |  return t[NO_WORDS], t[NO_WORDS + 1], ... , t[2 · NO_WORDS − 1]
```

zwar das temporäre Ergebnis t, allerdings ist die Kongruenzrelation 5.10 weiterhin erfüllt. Diese Erkenntnis wird bei allen nachfolgenden Algorithmen ebenfalls angewandt.
Die SOS Variante hat aufgrund des möglichen Überlaufs in Zeile 13 einen Speicherbedarf von $2 * \texttt{NO\_WORDS} + 1$ Wörtern.

Algorithmus 5.5 : Quadrierung, SOS Variante.

```
1 for i ← 0 to NO_WORDS − 1 do
2 |   (C, S) := t[i + i] + a[i] ∗ a[i]
3 |   for j ← i + 1 to NO_WORDS − 1 do
4 |   |   (C, S) := t[i + j] + 2 ∗ a[j] ∗ a[i] + C
5 |   |   t[i + j] := S
6 |   t[i + NO_WORDS] := C
```

Sind die Operanden zur Berechnung des Montgomery-Produktes gleich (Zeile 3, Algorithmus 5.3), kann die Berechnung des Produktes $a \cdot b$ optimiert werden, da $a_i \cdot a_j = a_j \cdot a_i$ gilt. Bei $a = b$ kann Zeile 1 bis 6 durch Algorithmus 5.5 ersetzt werden. Das hat den Vorteil, dass etwa die Hälfte der Multiplikationen durch eine Shift-Operation ersetzt wird. Der Nachteil ist allerdings, dass aufgrund des Shifts das Carry C mehr als zwei Bits aufnehmen muss.

5.3.2 CIOS Variante

Bei der *CIOS (Coarsely Integrated Operand Scanning) Variante* (siehe Algorithmus 5.6) wird in der äußeren Schleife zwischen Multiplikation und Reduktion gewechselt, da der i-te Reduktionsschritt nur von $t[i]$ abhängig ist und $t[i]$ im vorhergehenden Multiplikationsschritt vollständig berechnet wird. Die CIOS Variante hat gegenüber der SOS Variante den Vorteil, dass für t statt $2 \cdot \texttt{NO\_WORDS} + 1$ nur $\texttt{NO\_WORDS} + 2$ Wörter an Speicherplatz erforderlich sind.

5.3.3 FIOS Variante

Im Gegensatz zur CIOS Variante verfügt die *FIOS (Finely Integrated Operand Scanning) Variante* (siehe Algorithmus 5.7) nur über eine innere Schleife. Der Vorteil der FIOS Variante besteht im Gegensatz zur CIOS Variante in der höheren Code-Lokalität, da der Multiplikations- und Reduktionsschritt im j-ten Schritt zusammen ausgeführt wird. Nachteilig sind zwei Carrys, die durch die beiden Multiplikationen $a[j] \cdot b[i]$ und $m \cdot n[j]$ erzeugt werden können (Zeile 8) und einen zusätzlichen Aufruf der $ADD()$-Funktion benötigen.

Algorithmus 5.6 : CIOS Variante $\texttt{MonPro}(a, b)$.

```
1  for i ← 0 to NO_WORDS − 1 do
2      C := 0
3      for j ← 0 to NO_WORDS − 1 do
4          (C, S) := t[j] + a[j] · b[i] + C
5          t[j] := S
6      (C, S) := t[NO_WORDS] + C
7      t[NO_WORDS] := S
8      t[NO_WORDS + 1] := C
9      C := 0
10     m := t[0] · n′[0] mod 2^WORDSIZE
11     (C, S) := t[0] + m · n[0]
12     for j ← 1 to NO_WORDS − 1 do
13         (C, S) := t[j] + m · n[j] + C
14         t[j − 1] := S
15     (C, S) := t[NO_WORDS] + C
16     t[NO_WORDS − 1] := S
17     t[NO_WORDS] := t[NO_WORDS + 1] + C
```

Algorithmus 5.7 : FIOS Variante $\texttt{MonPro}(a, b)$.

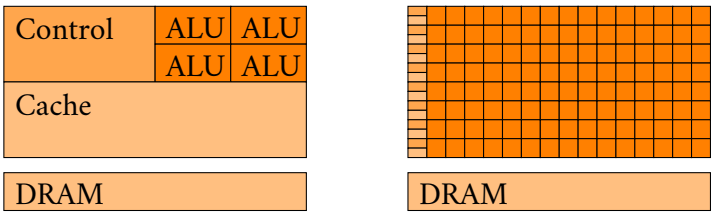
```
1  for i ← 0 to NO_WORDS − 1 do
2      (C, S) := t[0] + a[0] · b[i]
3      ADD(t[1], C)
4      m := S · n′[0] mod 2^WORDSIZE
5      (C, S) := S + m · n[0]
6      for j ← 1 to NO_WORDS − 1 do
7          (S, C) := t[j] + a[j] · b[i] + C
8          ADD(t[j + 1], C)
9          (C, S) := S + m · n[j]
10         t[j − 1] := S
11     (C, S) := t[NO_WORDS] + C
12     t[NO_WORDS − 1] := S
13     t[NO_WORDS] := t[NO_WORDS + 1] + C
14     t[NO_WORDS + 1] := 0
```

Wie bei der CIOS Variante ist ein Speicherplatz von `NO_WORDS` + 2 Wörtern für das temporäre Array `t` erforderlich.

5.4 GPU-Computing

Für die Entwicklung effizienter GPU-Algorithmen sind Kenntnisse über Aufbau und Funktionsweise eines modernen Grafikprozessors notwendig. In diesem Abschnitt wird eine kurze Einführung gegeben.

5.4.1 Motivation



(a) CPU **(b)** GPU

Abbildung 5.1: Bei GPUs wird mehr Transistorfläche für arithmetische Operationen verwendet.

Abbildung 5.1 stellt den Die[75] einer CPU dem Die einer GPU (Graphics Processing Unit) gegenüber. Bei einer GPU wird das gleiche Programm parallel auf unterschiedliche Daten angewendet. Die Datenflusssteuerung ist weniger komplex und Techniken wie Branch Prediction oder Speculative Execution kommen nicht zum Einsatz [99]. Dadurch stehen bei einer GPU wesentlich mehr Transistoren zur Datenverarbeitung zur Verfügung als bei einer CPU. Anstatt Speicherzugriffe durch Cache-Speicher zu beschleunigen, wird stattdessen die Ausführung des aktuellen Programmes unterbrochen und die Wartezeit mit Berechnungen anderer Daten überbrückt. GPUs sind im Gegensatz zu CPUs für rechenintensive, hoch-parallele Berechnungen optimiert und erfüllen dadurch die Anforderungen der Computerspielindustrie nach schneller Grafikberechnung in Echtzeit [99]. Da Computerspiele komplexer werden und neben dem reinen 3D-Rendering auch physikalische Prozesse in Echtzeit simuliert werden müssen, hat sich der Aufgabenbereich der Grafikprozessoren in den letzten Jahren gewandelt. Vertex- und Fragment-Shader, die zur Geometrie- bzw. Pixelberechnung dienen, wurden zusammengelegt und machen den Grafikprozessor

[75] ein ungehäuster Halbleiterchip

flexibler hinsichtlich beliebiger arithmetischer Berechnungen. Aus einer GPU wird eine *GPGPU (General Purpose Computing on Graphics Processing Unit)*.

Aufgrund der hohen Rechenleistung bei gleichzeitig niedrigen Kosten sind Grafikkarten für die Berechnung von modularen Exponentiationen besonders geeignet.

5.4.2 SIMT-Architektur

Mit *SIMT (Single Instruction, Multiple Threads)* bezeichnet Nvidia die Architektur ihrer aktuellen Grafikkarten. Auch wenn die Bezeichnung ursprünglich nur für Nvidia Grafikkarten galt, haben anderer Hersteller inzwischen die Bezeichnung übernommen [149].

Ein Grafikprozessor besteht aus einem Array von *SMPs (Streaming Multiprocessors)* [46]. Jeder SMP ist für die parallele Ausführung von mehreren hundert Threads optimiert und besteht aus einer Vielzahl von *Cores*, die einzelne Threads ausführen. Befehle werden zusätzlich in einer Pipeline abgearbeitet, um eine zusätzliche Parallelisierung innerhalb eines Threads zu ermöglichen.

Ein SMP organisiert Threads in sogenannten *Warps*, wobei ein Warp einer Gruppe von 32 Threads entspricht [149]. Threads innerhalb eines Warp werden mit derselben Programmadresse gestartet und führen einen Befehl mit unterschiedlichen Daten aus.
Die SIMT-Architektur ähnelt der *SIMD (Single Instruction, Multiple Data)* Architektur, ist dazu aber nicht äquivalent [46]. Bei der SIMD Architektur wird auf Datenvektoren derselbe Befehl angewendet. Es gibt einen Ausführungspfad, der sich bei Vektoroperationen auf mehrere Pfade aufteilt. Die Ausführung erfolgt parallel.
Bei der SIMT-Architektur erfolgt die Ausführung immer parallel (Parallelisierung auf Thread- anstatt auf Daten-Ebene). Das hat z. B. zur Folge, dass sich der Gesamtbedarf an Registern aus der Anzahl der Register eines einzelnen Programmes multipliziert mit der Anzahl an Threads ergibt. Dieser Nachteil kann durch Verwendung eines gemeinsamen Speicherbereichs (Shared Memory) vermindert werden.

Kommt es innerhalb eines Warp zu einer Verzweigung, von der nicht alle Threads betroffen sind (*Warp-Divergence*), muss trotzdem der gesamte Warp beide Ausführungspfade ausführen [61]. Von der Verzweigung nicht betroffene Threads sind während der Ausführung deaktiviert. Da beide Pfade seriell abgearbeitet werden, verlängert sich die Gesamtausführungszeit des Warp entsprechend. Die Warp-Divergence kann nur innerhalb eines Warp auftreten. Verschiedene Warps werden unabhängig voneinander ausgeführt und können somit unterschiedliche Codepfade nehmen.

Der *Warp-Scheduler* ist für die Ausführung der Warps verantwortlich und wählt Warps aus, die bereit sind, die nächste Instruktion auszuführen. Warps werden vom Scheduler unterbrochen, falls für den nächsten Befehl z. B. Operanden aus dem Speicher geladen werden müssen. Das Umschalten zwischen zwei Warps erfolgt dabei ohne Unterbrechung, da der gesamte Ausführungskontext (Register, Programmzähler etc.) über die gesamte Lebensdauer des Warp fest den SMP Ressourcen zugeordnet ist. Eine vergleichsweise teure Swapping-Operation von Registern, wie sie beim Kontextwechsel in der CPU verwendet wird, ist damit überflüssig. Da Warps unabhängig voneinander ausgeführt werden können, braucht der Scheduler nicht auf Abhängigkeiten zwischen Threads unterschiedlicher Warps zu achten.
Die Wartezeit in Taktzyklen, in der ein Warp nicht ausgeführt werden kann, wird hier als *Latency* bezeichnet. Um eine möglichst vollständige Auslastung des SMP zu erreichen, wird die Latency durch die Ausführung mehrere Warps überbrückt. Die Gesamtanzahl an Threads, die gleichzeitig einem SMP zugeordnet ist, ist neben einem Hardwarelimit vom Ressourcenbedarf der einzelnen Threads abhängig.

5.4.3 Speicherhierarchie Fermi-Architektur

Speicher	on-chip	Device	Host	Größe
Global	nein	r/w	r/w	max. 6 GB pro Device
Constant	nein	r	r/w	64 KB pro Device
Texture	nein	r	r/w	max. 6 GB pro Device
Shared	ja	r/w	-	16 KB oder 48 KB pro SMP
Register	ja	r/w	-	128 KB pro SMP
Local	nein	r/w	-	512 KB pro Thread
L1-Cache	ja	r/w	-	16 KB oder 48 KB pro SMP
L2-Cache	ja	r/w	-	768 KB pro Device
Constant-Cache	ja	r	-	8 KB
Texture-Cache	ja	r	-	6-8 KB pro SMP

Tabelle 5.1: Übersicht der Speicher und Caches in der Fermi-Architektur (r: Lesezugriff, w: Schreibzugriff).

An dieser Stelle soll eine Einführung in die unterschiedlichen Speicher und Caches einer GPU gegeben werden (siehe Tabelle 5.1) [125]. Als Grundlage dient die Fermi-Architektur von Nvidia, da sie am weitesten verbreitet ist und sich mittlerweile Architekturen anderer Grafikkartenhersteller an dieser Architektur

orientieren. In den folgenden Kapiteln wird zwischen dem *Device* und dem *Host* unterschieden. Ein Device bezeichnet eine[76] GPU, die für den Host, auf dem das Hauptprogramm ausgeführt wird, als Coprozessor agiert.

Globaler Speicher

Der *globale Speicher* befindet sich im DRAM. Daten werden in Blöcken zu je 32 bzw. 128 Byte mit einer Zugriffszeit von 400 bis 800 Taktzyklen aus dem Speicher geladen.

Werden Datenelemente im L1- und L2-Cache zwischengespeichert, hat jede Speichertransaktion eine Länge von 128 Byte. Der L1-Cache kann vom Programmierer deaktiviert werden, so dass Speicher-Transaktionen eine Breite von 32 Byte haben. Das bietet eine höhere Performance, falls Daten verstreut im DRAM liegen.

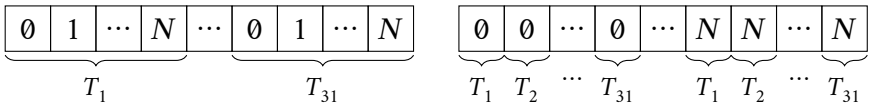


Abbildung 5.2: Threads $T_0, \ldots, T_{31}$ laden $N - 1$ Wörter aus dem DRAM, Speicherzugriffe sind *uncoalesced* (links) und *coalesced* (rechts).

Erzeugt ein Warp einen Speicherzugriff, um z. B. Operanden zu laden, werden die Speicherzugriffe aller Threads innerhalb des Warp zusammengelegt und entsprechende Speicheranfragen generiert. Die Anzahl dieser Anfragen hängt zum einen von der Größe des angeforderten Datenelementes eines Threads sowie zum anderen von der Position der Daten im Speicher ab. Bei einem *uncoalesced* Speicherlayout wird z. B. für jede 4 Byte Anfrage eines Threads eine 128 Byte Speichertransaktion erzeugt. Der Durchsatz sinkt, während bei einem *coalesced* Speicherlayout die Daten mehrerer Threads eines Warp im Speicher beieinander liegen. Dadurch wird bei Speicherzugriffen die gesamte Bandbreite genutzt. Abbildung 5.2 veranschaulicht diesen Zusammenhang.

Constant-Memory

Jedes Device verfügt für konstante Werte über einen Speicherbereich von 64 KB, der sich im DRAM befindet. Damit gelten dieselben Zugriffszeiten wie für den globalen Speicher. Der *Constant-Memory* hat den Vorteil, dass Speicherzugriffe durch einen 8 KB Cache-Speicher geleitet werden, der sich in jedem SMP befindet und den L1-Cache entlastet.

[76] Grafikkarten können auch zusammengeschlossen werden, fungieren hier aber immer als ein Device.

Texture Memory

Teile des globalen Speichers lassen sich als *Texture Memory* deklarieren. Dadurch werden Speicherzugriffe in einem Cache zwischengespeichert, der für zweidimensionale Zugriffsmuster optimiert ist. Zugriffe auf Daten im Cache verringern zwar nicht die Zugriffszeit, entlasten aber das DRAM-Speicherinterface. Sind Daten nicht coalesced, bietet der Texture Memory gegenüber Global und Constant-Memory eine bessere Performance.

Shared Memory

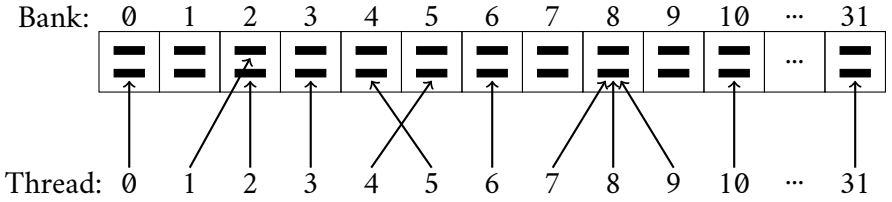


Abbildung 5.3: Der Zugriff auf Speicherbank 8 durch Thread 7, 8 und 9 veranschaulicht Broadcasting. Thread 1 und 2 verursachen einen Bankkonflikt. Alle anderen Threads verursachen keine Bankkonflikte.

Der Programmierer kann 64 KB pro SMP in 48 KB gemeinsam genutzten Speicher (*Shared Memory*) und 16 KB L1-Cache oder umgekehrt aufteilen.
Der Shared Memory erlaubt einen Datenaustausch von Threads untereinander und dadurch eine Kooperation der Threads. Daten können ohne das Zurückschreiben in den DRAM wiederverwendet werden. Der Shared Memory erlaubt die Implementierung eines anwendungsabhängigen Caching-Schemas, wodurch eine bessere Gesamtperformance erreicht werden kann als durch das automatische Caching über den L1-Cache.
Da sich der Shared Memory auf dem Chip befindet, sind die Zugriffszeiten mit 2 Taktzyklen wesentlich schneller als Zugriffe auf den globalen Speicher. Der Shared Memory besteht aus 32 Speicherbänken, die fortlaufend mit jeweils 32 Bit pro Bank beschrieben werden.

Falls keine Bankkonflikte auftreten, können 32 Threads gleichzeitig auf den Shared Memory zugreifen. Greifen in einem Warp Threads auf unterschiedliche Datenelemente zu, die sich in derselben Speicherbank befinden, kommt es zu einem *Bankkonflikt*. Die Folge ist die Sequenzierung der Speicherzugriffe und damit eine Verschlechterung der Performance.
Falls mehrere Threads eines Warp auf dasselbe Datenelement in einer Speicherbank zugreifen, kommt es bei Lesezugriffen (Schreibzugriffe auf dieselbe Speicherstelle sind nicht-deterministisch) nicht zu einem Bankkonflikt.
Beim *Broadcasting* wird der Speicherinhalt derselben Speicherzelle mehreren

Threads ohne Performanceeinbußen zur Verfügung gestellt. Abbildung 5.3 veranschaulicht Bankkonflikte und Broadcasting.

Register

Register stellen die schnellsten Speicherelemente dar. Jeder SMP verfügt über 32768 Register (128 KB), die Threads zugewiesen werden. Im Gegensatz zum globalen Speicher und Shared Memory können sich Threads Register nicht teilen.

Local Memory

Da die Anzahl an Registern pro Thread begrenzt ist, kann der SMP Variablen in den *Local Memory* auslagern (*Register Spilling*)[77]. Der lokale Speicher befindet sich im DRAM der Grafikkarte und es gelten für Speicherzugriffe dieselben Eigenschaften wie für Zugriffe auf den globalen Speicher.
Falls genügend ausführbare Warps einem SMP zur Verfügung stehen, führt Register Spilling zu keinen Performanceeinbußen, denn die Wartezeit wird durch Berechnungen anderer Warps überbrückt.

5.4.4 CUDA

CUDA (Compute Unified Device Architecture) ist eine von Nvidia 2006 eingeführte logische Abstraktionsschicht der Grafikkarte, um Programme auf Grafikkarten auszuführen (siehe Abbildung 5.4). Ein CUDA Programm wird als *Kernel* bezeichnet, wobei ein Kernel parallel durch mehrere Threads ausgeführt wird [125]. Mehrere Threads werden in einem *Thread-Block* zusammengefasst und einem SMP zugewiesen. Jeder Thread führt eine Instanz des Kernels auf einem Core aus und besitzt eine im Thread-Block eindeutige *Thread-ID*. Diese ID ist mehrdimensional. Ihre Struktur ist von der Modellierung des Problems abhängig. Threads eines Thread-Blockes haben Zugriff auf die Ressourcen des SMP. Die Threads können miteinander kooperieren, sich synchronisieren und über den Shared Memory Daten austauschen.
Die maximale Anzahl an Threads pro Block wird durch ihren Ressourcenbedarf (z. B. Anzahl an Registern, Bedarf an Shared Memory) sowie durch die Architektur[78] beschränkt. Alle Threads eines Blockes teilen sich die Ressourcen desselben SMP. Mehrere durch eine Block-ID identifizierbaren Thread-Blöcke sind in *Grids* organisiert. Die Block-ID kann wie die Thread-ID mehrdimensional sein.

[77] Dieses trifft z. B. auch auf Arrays zu, bei denen die Zugriffsindizes nicht konstant sind.
[78] Bei der Fermi-Architektur liegt das Maximum bei 1024 Threads.

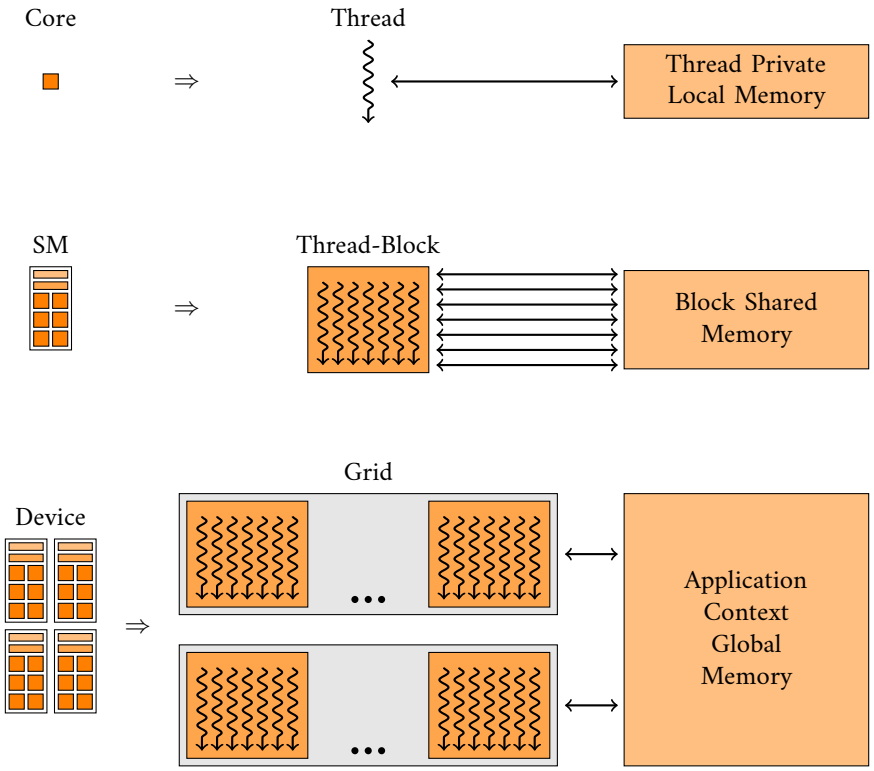


Abbildung 5.4: Cuda Hierarchie.

Alle Threads eines Grids haben Zugriff auf denselben globalen Speicher (Global-Memory, Constant-Memory, Texture Memory).

Die Anzahl der Cores pro SMP und die Anzahl der SMPs pro Grafikkarte ist nicht konstant und abhängig vom konkreten Grafikkartenmodell. Da ein Thread-Block in beliebiger Reihenfolge, gleichzeitig oder nacheinander auf einem der SMPs ausgeführt werden kann, hat die Strukturierung der Threads in Thread-Blöcke und Grids den Vorteil, dass eine automatische Skalierung auf die zugrundeliegende Hardware stattfindet. Kann z. B. eine GPU mit vier SMPs nur vier Thread-Blöcke parallel ausführen, sind es bei einer GPU mit acht SMPs doppelt so viele.

5.5 Montgomery auf GPUs

In den folgenden Abschnitten wird die Berechnung des Gleichungssystems 5.1 auf Grafikkarten gezeigt. Nur die modularen Exponentiationen mit fester Basis[79] werden auf der GPU berechnet. Im Anschluss können die Ergebnisse entweder in einem zweiten Kernel auf der GPU oder auf der CPU multipliziert werden. Auf die

[79] Dazu gehören alle Basen, die für alle Prover konstant sind und z. B. im „Ticket-Chain“ und „Ticket-Spender“ Protokoll während der Setup-Phase generiert werden.

Berechnung der Exponentiation mit variabler Basis auf der GPU wird verzichtet, da sie RSA-Operationen entsprechen und es bereits mehrere Veröffentlichungen zu diesem Thema gibt (siehe Kapitel 5.6).

Die Performance der Algorithmen wird für Operanden der Bitlänge $\texttt{NO\_BITS} \in \{512, 768, 1024, \ldots, 4096\}$ nachgewiesen, die nach Kapitel 5.1.1 vor der Berechnung in zwei Teile aufgespaltet werden, so dass sich die effektive Bitlänge der Operanden halbiert. Bei allen Implementierungen werden die Moduli p und q im Constant-Memory der Grafikkarte abgelegt. Die Anzahl der Basen N' soll beschränkt werden auf $N' = 5$, was einer typischen Anzahl von Basen entspricht. Theoretisch kann eine höhere Anzahl von Basen verwendet werden, ohne dass sich das negativ auf die Performance auswirkt. Im pessimistischen Fall ($\texttt{NO\_BITS} = 4096$) können $N' = 14$ Basen (14 Basen + p + q = 16 Werte $\hat{=}$ 8 KB) im Constant-Memory Cache abgelegt werden. Darauf wird verzichtet, um einen einheitlichen Versuchsaufbau zu gewährleisten und Messungen zu beschleunigen.

Die Algorithmen werden in CUDA C implementiert, mit dem Nvidia Cuda Compiler 4.2 bzw. 5.5[80] kompiliert und auf einer GeForce GTX 580 ausgeführt [124]. Als Vergleich dient ein Intel Core i7 CPU 920 Prozessor mit 2,67 GHz und vier Kernen (acht virtuelle Kerne durch Hyper-Threading). Die Anschaffungskosten entsprechen in etwa denen der GTX 580. Die maximale Verlustleistung liegt bei 130 W (CPU) bzw. 244 W (GTX 580). Zur Implementierung der arithmetischen Operationen wird die MPIR 2.6.0 Bibliothek verwendet [71]. Zur vollständigen Nutzung des Potentials der CPU wird die 64 Bit Unterstützung des Prozessors verwendet und die Implementierung auf Basis der OpenMP Bibliothek parallelisiert. Dadurch sind alle acht virtuellen Kerne für Berechnungen nutzbar [129]. In Cuda C ist kein direkter Zugriff auf das Carry-Flag möglich. Aus diesem Grund wird der überwiegende Teil der Algorithmen in PTX (Parallel Thread Execution) Assembler Code implementiert [127].

Bei allen Zeitmessungen der GPU-Implementierungen ist die Zeit für die Übertragung der Daten zum bzw. vom Device enthalten[81]. Bei allen Implementierungen wird die MRC Konvertierung verwendet, die auf der CPU ausgeführt wird. Kapitel 5.5.5 zeigt den Einfluss der SRC Konvertierung auf die Gesamtperformance.

5.5.1 CIOS Variante

[80] Version 5.5 des Cuda Compilers basiert im Gegensatz zu Version 4.2 auf dem LLVM Compiler Framework. Dadurch kann sich die Performance des kompilierten Codes je nach Kernel gegenüber Version 4.2 sowohl verbessern als auch verschlechtern. In dieser Arbeit werden beide Compiler verwendet und das beste Ergebnis ermittelt.

[81] Durch die Verwendung von Streams lässt sich diese Zeit theoretisch auch für weitere Berechnungen nutzen [125].

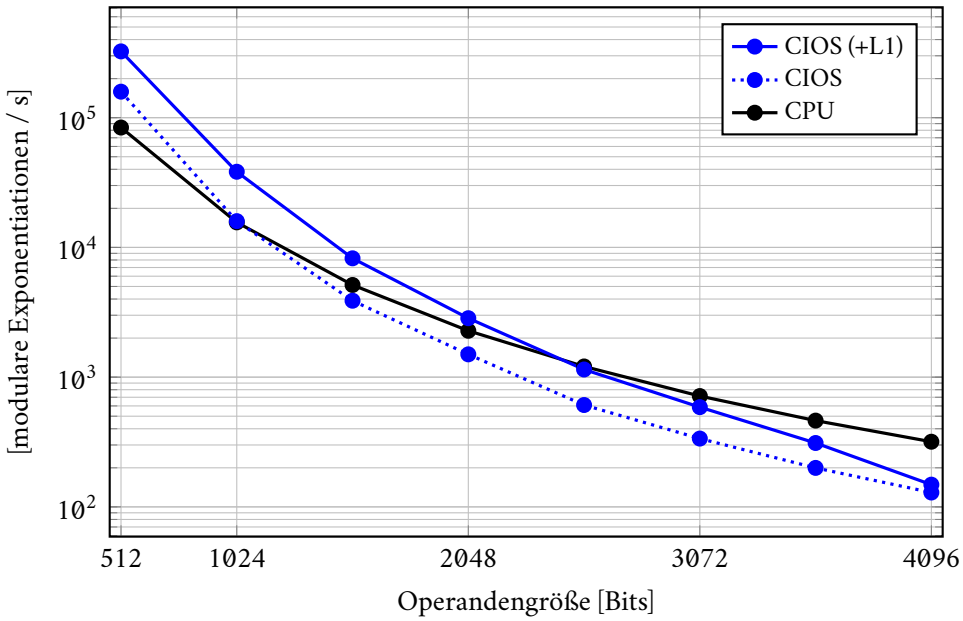


Abbildung 5.5: Performance CIOS Variante.

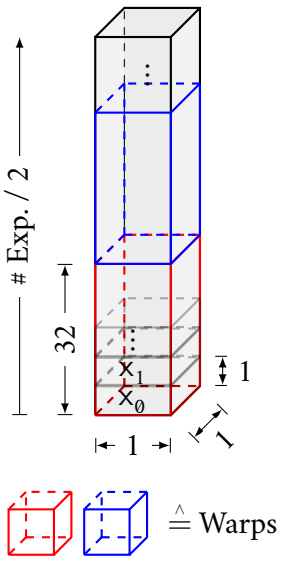


Abbildung 5.6: Modellierung eines Thread-Blockes (CIOS Variante).

Çetin Kaya Koç et al. haben gezeigt, dass die CIOS Variante auf einer Pentium-90 CPU der SOS und FIOS Variante überlegen ist [103]. In einem ersten Ansatz wird deshalb Algorithmus 5.6 auf einer GPU implementiert.

Zwei Threads berechnen eine Exponentiation. Ein Thread berechnet die Teilexponentiation $x_0 = g^{e_0} \bmod p$, der andere $x_1 = g^{e_1} \bmod q$, so dass ein Warp 16 Exponentiationen gleichzeitig berechnet. Abbildung 5.6 zeigt die Modellierung des Thread-Blockes. Die optimale Anzahl und Größe der Thread-Blöcke wird dynamisch ermittelt und die Konfiguration gewählt, die die höchste Performance bietet. Die optimale Konfiguration lässt sich zwar theoretisch berechnen [124], liefert aber in der Praxis häufig nicht die höchste Performance.

Abbildung 5.5, Kurve ···●··· zeigt, dass für Operanden ab 1024 Bit die CPU-Implementierung eine bessere Performance bietet als die GPU-Implementierung. Eine Auswertung mithilfe des Nvidia Profilers [126] lässt erkennen, dass alle SMPs die meiste Zeit auf Daten aus dem langsamen Global-Memory warten müssen.

Die Wartezeit kann selbst durch eine hohe Anzahl von gestarteten Threads nicht überbrückt werden. Die Daten werden zwar im L1- und L2-Cache zwischengespeichert, in Algorithmus 5.6 ist die Lokalität der Daten aber zu gering und die Caches zu klein, um einen positiven Effekt zu erzielen. Die Zeit zwischen denen eine Variable wiederverwendet wird ist so hoch, dass sie durch einen neuen Wert im Cache verdrängt wird.

Größerer L1-Cache

Um Speicherzugriffe auf den Global-Memory zu beschleunigen, wird die Größe des L1-Cache von 16 auf 48 KB erhöht. Abbildung 5.5, Kurve zeigt, dass bei allen Operandengrößen eine Verbesserung der Performance erreicht werden kann. Bei größeren Operanden macht sich allerdings der größere L1-Cache immer weniger bemerkbar. Die Anzahl der Cache-Misses nimmt zu, so dass bei ca. 4096 Bit kaum noch eine Verbesserung zu erreichen ist.

5.5.2 Vergleich SOS / CIOS / FIOS Variante

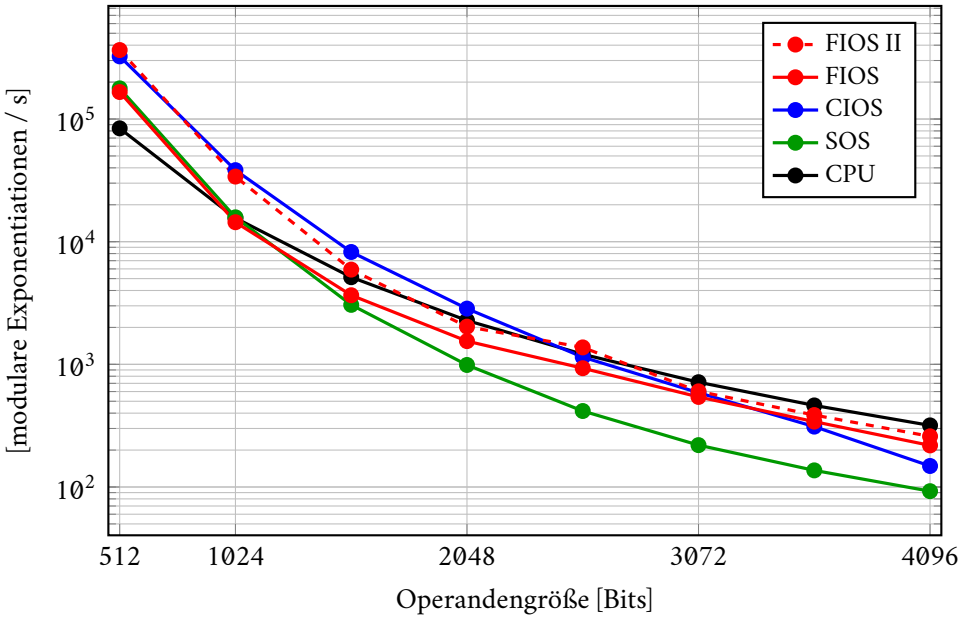


Abbildung 5.7: Performancevergleich SOS / CIOS / FIOS Variante.

Für Operanden mit einer Größe kleiner 2560 Bit übertrifft die Performance der GPU CIOS Variante die Performance der CPU-Implementierung (vgl. Kapitel 5.5.1). Die geringe Lokalität der Daten und die hohe Anzahl an Speicherzugriffen wirken sich negativ aus.
Als Nächstes soll die SOS und FIOS Variante auf der Grafikkarte getestet und

mit der CIOS Variante verglichen werden. Bei allen Implementierungen wird ein größerer L1-Cache verwendet. Die Ergebnisse zeigt Abbildung 5.7.

Die SOS Variante (Kurve ) hat aufgrund des größeren Speicherplatzbedarfes die schlechteste Performance. Die FIOS Variante (Kurve ) ist bei Operanden mit einer Größe ab $\approx$ 3584 Bit der CIOS Variante überlegen.

Optimierte FIOS Variante (FIOS II)

Durch das Zusammenlegen der inneren Schleifen erhöht die FIOS Variante die Lokalität der Daten. Allerdings kann das Carry-Handling in Zeile 8 (Algorithmus 5.7) auf einer Grafikkarte nur sehr ineffizient implementiert werden. Es kann der Fall eintreten, dass die Addition des Übertrages einen neuen Übertrag generiert, der bei der Addition mit einem Element aus dem temporären Ergebnis t einen weiteren Übertrag generiert usw. Die $ADD(..)$ Anweisungen müssen durch eine Schleife implementiert werden, ohne dass der Compiler die Anzahl der Schleifendurchläufe ermitteln kann. Das schränkt die Optimierungsmöglichkeiten des Compilers ein, so dass z. B. ein Abrollen der Schleife nicht möglich ist [125]. Außerdem ist die Ausführungszeit eines Warp vom langsamsten Thread innerhalb des Warp abhängig. In dem Beispiel wäre das der Thread, der am meisten Schleifendurchläufe braucht. Die Ausführung wird ineffizient.

Das Problem lässt sich durch die Verwendung von zwei Carry-Flags C1 und C2 lösen. Bei der FIOS II Variante wird die innere For-Schleife (Zeile 6 bis 10) in Algorithmus 5.7 durch die folgenden Anweisungen ersetzt [93].

Algorithmus 5.8 : FIOS II Variante (innere Schleife).

```
1 for j ← 1 to NO_WORDS − 1 do
2 |  (C1, x̄[j]) := x̄[j] + ā[j] · b̄[i] + C1
3 |  (C2, x̄[j − 1]) := x̄[j] + m · n[j] + C2
```

Die Ergebnisse in Abbildung 5.7 (Kurve ) zeigen, dass trotz zweier zusätzlicher Additionen und Variablen die Performance der FIOS Variante verbessert wird. Die FIOS II Variante ist für Operanden ab einer Größe von 2560 Bit sowie bei kleinen Operanden von 512 Bit der CIOS Variante überlegen.

5.5.3 Beschleunigung durch Lookup Tabellen

Neben der Optimierung des Montgomery-Produktes lässt sich die Montgomery-Exponentiation (Algorithmus 5.3) für Grafikkarten optimieren. Der Quadrierungsschritt in Zeile 3 kann eliminiert werden, indem $\bar{t} = \texttt{MonPro}(\bar{t}, \bar{g})$ in Zeile 5 durch $\bar{t} = \texttt{MonPro}(\bar{t}, \bar{g}^{(2^i)})$ für $i = 0, \ldots, \texttt{NO\_BITS}/2 - 1$ ersetzt wird. Dadurch

kann $\bar{g}^{(2^i)}$ in einer LUT abgelegt werden (siehe Abbildung 5.8). Der Quadrierungsschritt wird in eine Leseoperation überführt. Die Performance erhöht sich.

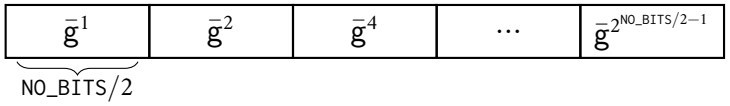


Abbildung 5.8: Lookup Tabelle für die Basis $\bar{g}$ (LUT_DEPTH $= 1$).

Die Anzahl der Schleifendurchläufe in Zeile 2 (Algorithmus 5.3) kann verringert werden, indem die Größe der LUT erhöht wird. Dazu wird der Exponent in Segmente der Größe LUT_DEPTH unterteilt. Die *Segmentgröße* LUT_DEPTH ist definiert als die Anzahl von Exponenten-Bits, die pro Schleifendurchgang verarbeitet werden. Damit entspricht die Anzahl der Segmente $u = (\text{NO\_BITS}/2)\ /\text{LUT\_DEPTH}$. Mit

$$e_i \in \{0, \ldots, 2^{\text{LUT\_DEPTH}} - 1\}$$

lässt sich die Berechnung von $\bar{g}^e$ zerlegen in

$$\bar{g}^e = \prod_{i=0}^{u-1} \bar{g}^{e_i \cdot (2^{\text{LUT\_DEPTH}})^i}$$

Aufgrund der Zerlegung in Kapitel 5.1.1 sind pro Basis zwei LUTs notwendig. Der Speicherbedarf wächst mit zunehmender Segmentgröße LUT_DEPTH exponentiell an und beträgt pro Basis:

$$\text{LUT\_SIZE} = 2 \cdot \frac{\frac{(\text{NO\_BITS}/2)^2}{\text{LUT\_DEPTH}} \cdot (2^{\text{LUT\_DEPTH}} - 1)}{8 \cdot 1024^2}\ \text{MB}$$

In dieser Arbeit wird LUT_DEPTH $= \{1, 2, 4, 8\}$ gewählt. Im pessimistischen Fall LUT_DEPTH $= 8$, NO_BITS $= 4096$ ergibt sich ein Speicherbedarf von LUT_SIZE $=$ 31,875 MB. Wenn kein Padding verwendet werden soll[82], sind größere Werte für LUT_DEPTH nicht praktikabel. Für LUT_DEPTH $= 16$ übersteigt der erforderliche Speicherbedarf von 4 GB die Speichergröße der für die Messungen verwendete Grafikkarte.

In Algorithmus 5.9 ist die Erweiterung von Algorithmus 5.3 um LUTs dargestellt. Die LUTs werden für alle $N' = 5$ Basen als Textur auf der Grafikkarte gespeichert. Daraus ergeben sich zwei Vorteile. Erstens ist die Texture-Fetching-Unit optimiert, Daten aus nicht zusammenhängenden Speicherblöcken aus dem Global-Memory zu laden. Zweitens werden die Daten nicht im L1-Cache, sondern in einem separaten Texture-Cache zwischengespeichert und damit der L1-Cache entlastet (siehe Kapitel 5.4.3).

[82] Padding ist erforderlich, wenn LUT_DEPTH keiner 2-er Potenz entspricht.

Algorithmus 5.9 : Montgomery-Exponentiation $\texttt{MonExp}(g, \texttt{e})$ mit Lookup Tabelle.

1 $\bar{\texttt{g}}\,(i,j) \leftarrow \bar{\texttt{g}}^{j\cdot(2^{\texttt{LUT\_DEPTH}})^i} \bmod n, j \in \{0,\ldots,2^{\texttt{LUT\_DEPTH}}-1\}$ (LUT)
2 $\bar{\texttt{t}} = 1 \cdot \texttt{r} \bmod n$
3 **for** $i \leftarrow (\texttt{NO\_BITS}/2)/\texttt{LUT\_DEPTH} - 1$ **to** 0 **do**
4 $\quad \texttt{e}_i =$ die nächsten LUT_DEPTH Bits aus dem Exponenten
5 $\quad$ **if** $\texttt{e}_i \neq 0$ **then**
6 $\quad\quad \bar{\texttt{t}} := \texttt{MonPro}(\bar{\texttt{t}}, \bar{\texttt{g}}(i, \texttt{e}_i))$
7 **return** $\texttt{t} = \texttt{MonPro}(\bar{\texttt{t}}, 1)$

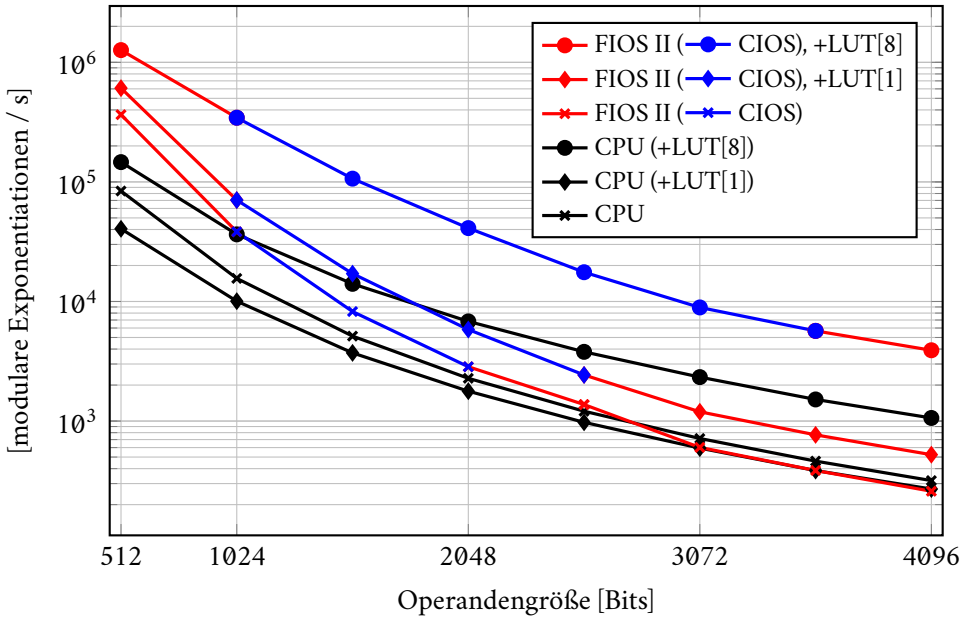


Abbildung 5.9: Performance beste Variante (CIOS / FIOS II) mit LUTs (LUT_DEPTH $= 1$ und LUT_DEPTH $= 8$).

Auf Basis der CIOS / FIOS II Variante wird Algorithmus 5.9 mit einer LUT basierten CPU-Implementierung verglichen. Abbildung 5.9 zeigt für die unterschiedlichen Operandengrößen die Variante mit der größten Performance. Mit steigender Segmentgröße nimmt sowohl die Performance der GPU- als auch der CPU-Implementierung zu. Bei einer Segmentgröße von `LUT_DEPTH` $= 8$ werden auf der GPU erstmals für alle Operandengrößen mehr modulare Exponentiationen pro Sekunde berechnet als auf der CPU. Die GPU- und CPU-Implementierungen profitieren unterschiedlich stark von der Verwendung von LUTs. Das Laden der Operanden ist auf der GPU deutlich effizienter als auf der CPU.

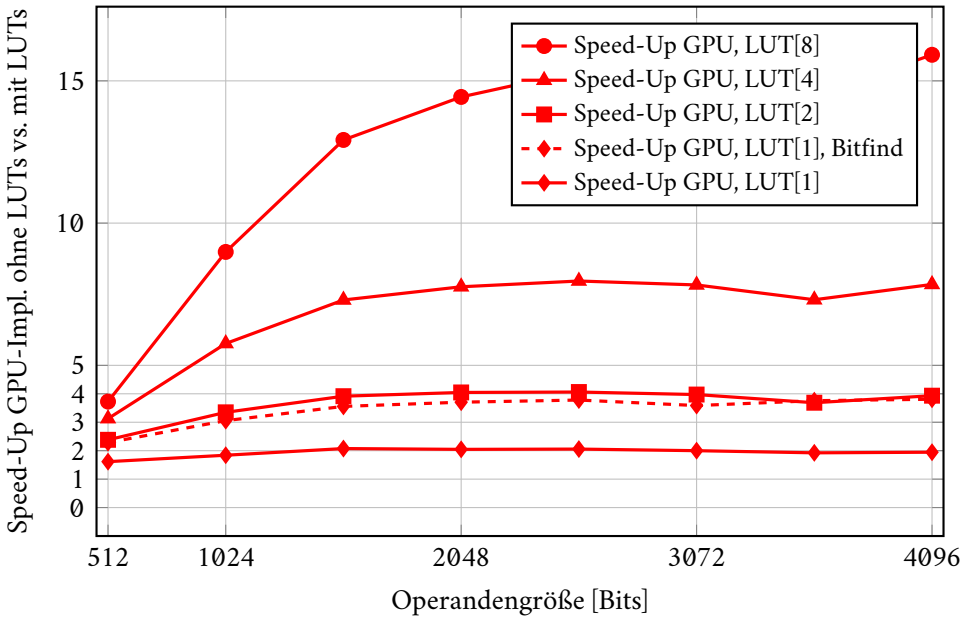


Abbildung 5.10: Speed-Up durch die Verwendung von LUTs.

Bei einer Segmentgröße von 1 verschlechtert sich die CPU-Performance im Vergleich zur CPU-Implementierung ohne LUTs. Das ist bei der GPU-Implementierung nicht der Fall. Die GPU profitiert insbesondere bei Operanden mit großen Bitlängen von der Verwendung von LUTs. Abbildung 5.10 veranschaulicht diesen Zusammenhang anhand des Speed-Ups (Faktor um den die GPU-Implementierung mit LUTs schneller als die GPU-Implementierung ohne LUTs ist).

5.5.4 „Bitfind“-Methode

Wenn die If-Anweisung in Zeile 5 (Algorithmus 5.9) nicht für alle Threads eines Warp ausgeführt wird, kommt es in der If-Anweisung zu einer Warp-Divergence. Zur Bestimmung des Einflusses soll die Laufzeit der For-Schleife für verschiedene

Segmentgrößen als Vielfaches der Ausführungszeit einer einzelnen $\texttt{MonPro}(\cdot,\cdot)$ Operation berechnet werden.
Die Ausführungszeit eines Schleifendurchlaufes ist nur dann Null, wenn für alle Threads innerhalb eines Warp $\mathsf{e}_i = 0$ für $i = 0, \ldots, 31$ gilt. Es sei $1/2$ die Wahrscheinlichkeit, dass ein Bit im Exponenten e 1 ist. Die Wahrscheinlichkeit, dass der komplette Warp die If-Anweisung ausführt, ergibt sich damit zu:

$$1 - \frac{1}{\left(2^{\texttt{LUT\_DEPTH}}\right)^{32}}$$

Daraus lässt sich die durchschnittliche Laufzeit $\mathrm{LZ}_{\mathrm{GPU}}$ für die Ausführung der For-Schleife berechnen.

$$\mathrm{LZ}_{\mathrm{GPU}} = \left(1 - \frac{1}{\left(2^{\texttt{LUT\_DEPTH}}\right)^{32}}\right) \cdot \frac{\texttt{NO\_BITS}/2}{\texttt{LUT\_DEPTH}}$$

Bei der CPU-Implementierung kommt es nicht zu einer Warp-Divergence. Einzelne Threads können unabhängig voneinander den If-Zweig ausführen. In diesem Idealfall berechnet sich die durchschnittliche Laufzeit eines Threads zu:

$$\mathrm{LZ}_{\mathrm{ideal}} = \left(1 - \frac{1}{2^{\texttt{LUT\_DEPTH}}}\right) \cdot \frac{\texttt{NO\_BITS}/2}{\texttt{LUT\_DEPTH}}$$

Der Einfluss der Warp-Divergence auf die Performance wird an dieser Stelle als *Effizienz* bezeichnet und ergibt sich aus dem Verhältnis zwischen idealer Laufzeit $\mathrm{LZ}_{\mathrm{ideal}}$ und durchschnittlicher Laufzeit $\mathrm{LZ}_{\mathrm{GPU}}$. Bei einer Effizienz von 1 tritt keine Warp-Divergence auf.

LUT_DEPTH	$\mathrm{LZ}_{\mathrm{ideal}}$ [# Operationen]	$\mathrm{LZ}_{\mathrm{GPU}}$ [# Operationen]	Effizienz $[\mathrm{LZ}_{\mathrm{ideal}}/\mathrm{LZ}_{\mathrm{GPU}}]$
1	512	1024	0,5
2	384	512	0,75
4	240	256	0,9375
8	127,5	128	0,9961

Tabelle 5.2: Einfluss der Warp-Divergence auf die Laufzeit ($\texttt{NO\_BITS} = 2048$).

Für den Fall $\texttt{NO\_BITS} = 2048$ ist in Tabelle 5.2 exemplarisch die Laufzeit unter Berücksichtigung unterschiedlicher Segmentgrößen dargestellt. Bei steigender Segmentgröße sinkt der Einfluss der Warp-Divergence. Ab einer Segmentgröße von $\texttt{LUT\_DEPTH} = 4$ sind die Performanceeinbußen durch die Warp-Divergence vernachlässigbar.

Die Effizienz kann für den Fall $\texttt{LUT\_DEPTH} = 1$ erhöht werden, indem aus einer Kontrollflussabhängigkeit eine Datenflussabhängigkeit gemacht wird. Damit

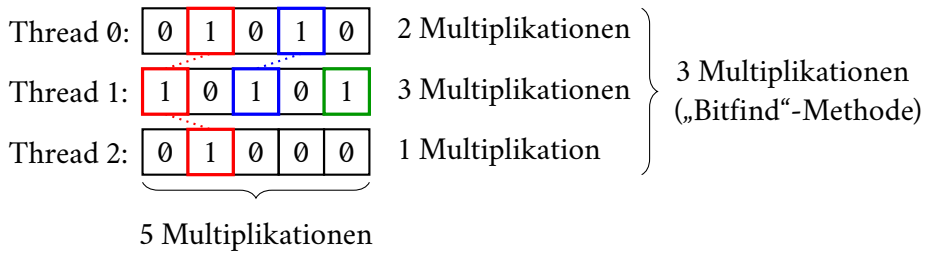


Abbildung 5.11: Die Ausführungszeit einzelner Threads eines Warp entspricht der Ausführungszeit des Threads mit der höchsten Anzahl an Einsen.

nicht alle Threads eines Warp ihre Exponenten in der For-Schleife bitweise scannen, sucht jeder Thread nach dem nächsten gesetzten Bit und führt für dieses Exponentenbit die Montgomery-Multiplikation aus. Abbildung 5.11 veranschaulicht die *„Bitfind“-Methode*. Exponentenbits derselben Farbe werden gleichzeitig verarbeitet. In dem Beispiel sind deshalb nur drei anstatt fünf Schritte notwendig, um das Ergebnis von allen drei Threads zu berechnen. Die gesamte Ausführungszeit entspricht dem Thread mit den meisten Einsen im Exponenten.

Der PTX Befehlssatz stellt zum Suchen des ersten gesetzten Bit innerhalb eines 32 Bit Wortes einen Assemblerbefehl bereit. Dadurch lässt sich das Konzept effizient auf der GPU implementieren. Obwohl mehr Rechenaufwand erforderlich ist (ein Exponent besteht aus mehreren Wörtern), zeigen die Ergebnisse in Abbildung 5.10 (Kurve -♦-) eine nahezu Verdoppelung der Performance im Vergleich zur einfachen Variante ohne „Bitfind“-Methode.

5.5.5 SOS-parallel Variante

Messungen mit dem Nvidia Visual Profiler ergeben, dass die einzelnen GPU-Cores bei der LUT Implementierung mit `LUT_DEPTH` $= 8$ (Abbildung 5.9, Kurve —●— und —●—) insbesondere bei größeren Operanden nicht vollständig ausgelastet sind (33 % Auslastung bei `NO_BITS` $= 2048$, 23 % Auslastung bei `NO_BITS` $= 4096$). Die Wartezeit bei Zugriffen auf Daten aus dem Global-Memory kann nur unzureichend durch Berechnungen überbrückt werden. Ein Grund hierfür ist die hohe Anzahl an Registern, die zur Ausführung eines Threads notwendig sind. Messungen ergeben, dass 38 % des Datenverkehrs mit dem DRAM bei `NO_BITS` $= 2048$ für das Register Spilling aufzuwenden sind. Für größere Operanden steigt das Verhältnis auf 96 % bei `NO_BITS` $= 4096$ an. Der hohe Bedarf an Registern kommt hauptsächlich durch das temporäre Array `t` zustande. Zwei Maßnahmen sollen die Anzahl an Registern pro Thread verringern.

1. Die Berechnung einer Exponentiation wird auf mehrere Threads verteilt. Es können dadurch zwar weniger Exponentiationen auf einem SMP par-

Algorithmus 5.10 : SOS-parallel Variante MonPro(a, b), *threadIdx*: aktuelle Thread-ID (*threadIdx* $\in \{0, \dots, 15\}$ für NO_BITS $= 1024/3072$, *threadIdx* $\in \{0, \dots, 31\}$ für NO_BITS $= 2048/4096$), KoggeStoneAdd(dst, opA, opB, P, G): paralleler Kogge-Stone Addierer, KoggeStoneSub(dst, opA, opB, P, G): paralleler Kogge-Stone Subtrahierer.

```
1  C := 0
2  t[threadIdx] := 0
3  t[NO_WORDS + threadIdx] := 0
4  for j ← 0 to NO_WORDS − 1 do
5  |  (C, t[threadIdx + j]) := t[threadIdx + j] + a[j] ∗ b[threadIdx] + C
6  KoggeStoneAdd(t[NO_WORDS], t[NO_WORDS], C, t[0], t[NO_WORDS])
7  t[threadIdx] := 0
8  t[NO_WORDS + threadIdx] := 0
9  C := 0
10 for j ← 0 to NO_WORDS − 1 do
11 |  (C, t[threadIdx + j]) := t[threadIdx + j] + t[j] · n′[threadIdx] + C
12 C := 0
13 for j ← 0 to NO_WORDS − 1 do
14 |  (C, t[threadIdx + j]) := t[threadIdx + j] + t[j] · n[threadIdx] + C
15 KoggeStoneAdd(t[NO_WORDS], t[NO_WORDS], C, t[0], t[NO_WORDS])
16 if t[2 ∗ NO_WORDS − 1] != 0   or
   t[2 ∗ NO_WORDS − 1] >= n[NO_WORDS − 1] then
17 |  KoggeStoneSub(t[0], t[NO_WORDS], n[0], t[0], t[NO_WORDS])
18 |  return t[0], t[1], ... , t[NO_WORDS − 1]
19 else
20 |  return t[NO_WORDS], t[NO_WORDS + 1], ... , t[2 ∗ NO_WORDS − 1]
```

allel berechnet werden, dafür benötigt ein einzelner Thread aber auch weniger Ressourcen.

2. Durch die Reduzierung der gleichzeitig ausgeführten Exponentiationen lässt sich das temporäre Array im Shared Memory speichern. Der L1-Cache wird auf 16 KB verringert und der Shared Memory auf 48 KB erhöht.

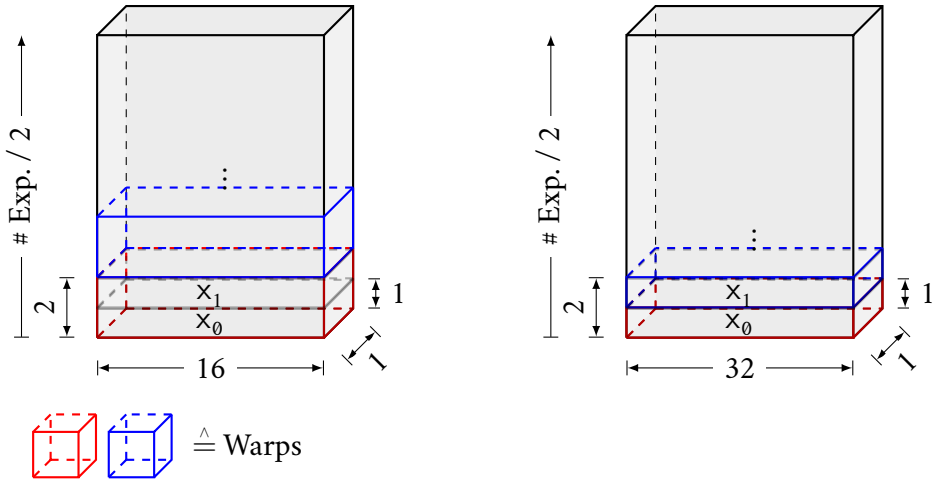


Abbildung 5.12: Modellierung eines Thread-Blockes (SOS-parallel Variante), links für `NO_BITS` = 1024 und `NO_BITS` = 3072, rechts für `NO_BITS` = 2048 und `NO_BITS` = 4096.

Die Anzahl der Threads, die eine Exponentiation ausführen, ist von der Bitlänge der Operanden abhängig. Vorerst sollen nur Operanden der Länge `NO_BITS` = 1024 und `NO_BITS` = 2048 Verwendung finden.

Jeder Thread bearbeitet ein Wort. Operanden der Bitlänge `NO_BITS` = 1024 werden in zwei Teile aufgespaltet, so dass jeder Operand aus 16 Wörtern besteht (siehe Kapitel 5.1.1). Dadurch führt ein Warp (32 Threads) eine Exponentiation durch, wobei die eine Hälfte des Warp x_0 und die andere Hälfte x_1 berechnet. Bei `NO_BITS` = 2048 haben die Operanden eine Größe von 1024 Bit (≙ 32 Wörtern), so dass ein Warp eine Teilexponentiation x_0 bzw. x_1 berechnet. Abbildung 5.12 zeigt das Thread-Modell für verschiedene Operandengrößen. Es ergeben sich folgende Vorteile.

1. Eine Synchronisierung der Threads ist nicht notwendig, da Threads innerhalb eines Warp automatisch synchronisiert sind [149].

2. Bankkonflikte im Shared Memory werden vermieden, da zu jeder Zeit zwei gleichzeitig ausgeführte Threads immer auf unterschiedliche Speicherbänke zugreifen.

3. Ein Coalescing der Operanden, welches bei allen vorherigen Implementierungen vorher auf der CPU ausgeführt werden muss, ist nicht erforderlich.

Die Fälle NO_BITS = 3072 und NO_BITS = 4096 stellen Spezialfälle der obigen Thread-Modelle dar. Im ersten Fall wird das Thread-Modell für NO_BITS = 1024 verwendet. Operanden haben nach der Zerlegung eine Größe von 1536 Bit. Diese werden weiter in drei Teile zerlegt, um jeweils 16 Wörter des Operanden parallel zu bearbeiten. Bei NO_BITS = 4096 haben die Operanden eine Bitlänge von 2048 Bit und lassen sich in zwei Teile zerlegen und mit dem Thread-Modell für NO_BITS = 2048 berechnen.
Andere Bitlängen lassen sich mit der SOS-parallel Variante nur ineffizient implementieren, da entweder nicht alle Threads eines Warp verwendet werden können oder ein Coalescing der Operanden durchgeführt werden muss. An dieser Stelle soll auf diese Implementierungen verzichtet werden.

Unter allen Varianten eignet sich die SOS Variante am besten zur Parallelisierung, da der Schleifenkörper aller äußeren Schleifen (Algorithmus 5.4) auf 16 bzw. 32 Threads aufgeteilt werden kann. Algorithmus 5.10 veranschaulicht die SOS-parallel Variante. Die Verarbeitung des Carrys in Zeile 13 stellt allerdings ein Problem dar, da bei der Addition mit dem nächsthöheren Wort ein weiteres Carry entstehen kann. Im ungünstigsten Fall sind 15 bzw. 31 weitere Additionen notwendig, die sequentiell auszuführen sind.

Aus diesem Grund werden für alle Additions- und Subtraktionsoperationen[83] Kogge-Stone Addierer verwendet [104]. Ein *Kogge-Stone Addierer* ist ein Paralleladdierer mit Übertragsvorausberechnung. Die Funktionsweise veranschaulicht Abbildung 5.13.
Jede vertikale Stufe *j* erzeugt ein „Propagate"-Bit P und ein „Generate"-Bit G in Abhängigkeit von der vorherigen Stufe. Die Summe der beiden Operanden A und B wird aus den „Generate"-Bits der letzten Stufe und den „Propagate"-Bits der ersten Stufe berechnet. Da die GPU über 32 Bit Addierer verfügt, operiert der Kogge-Stone Addierer nicht wie dargestellt bitweise, sondern auf Basis von 32 Bit Wörtern.

Der Kogge-Stone Addierer hat im ungünstigsten Fall eine logarithmische Laufzeit. Sind alle „Propagate"-Bits einer Stufe Null, ist die Berechnung weiterer Stufen nicht erforderlich, da die „Propagate"-Bits der nachfolgenden Stufen ebenfalls Null sind und die „Generate"-Bits den „Generate"-Bits vorheriger Stufen entsprechen. Ob ein Wert bei allen aktiven Threads eines Warp ungleich Null ist, lässt sich auf der GPU effizient ausführen[84].

Der SOS-parallel Algorithmus benötigt in jedem Schritt zur Berechnung einer modularen Exponentiation nicht mehr als 4 ∗ NO_WORDS Wörter im Shared Memory. Der Speicherplatz wird abwechselnd zur Speicherung von Zwischenergebnissen, Operanden und der „Propagate"- und „Generate"-Bits während der

[83] Aus einem digitalen Addierer wird ein Subtrahierer durch Bildung des Zweierkomplementes des Subtrahenden und Addition mit dem Minuend [18].
[84] z. B. mit der Warp Vote Funktion __any in CUDA [125].

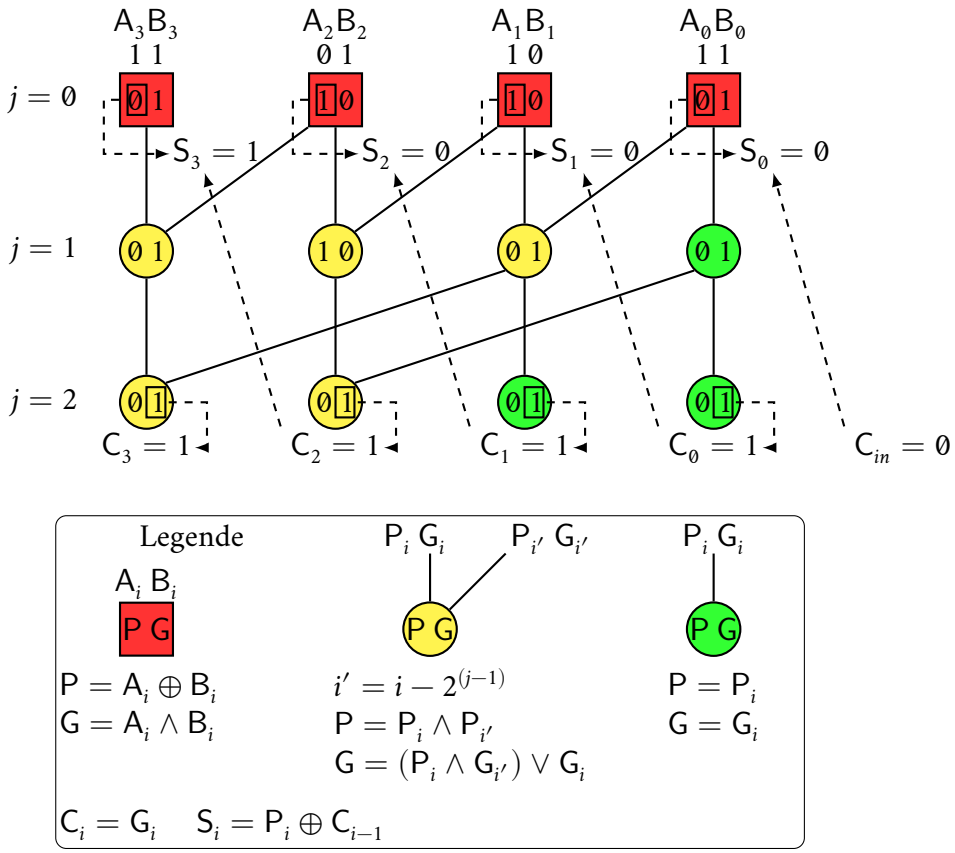


Abbildung 5.13: Funktionsweise des Kogge-Stone Addierers bei der bitweisen Addition von $1011 + 1101 = 11000$.

Ausführung des Kogge-Stone Addierers genutzt.
Für NO_BITS $= 3072$ führt ein Warp eine Exponentiation aus. Theoretisch lassen sich maximal 1024 Threads bzw. 32 Exponentiationen auf einem SMP ausführen. Damit ergibt sich ein Speicherbedarf von $32 \cdot (4 \cdot \texttt{NO\_BITS}) \mathrel{\hat{=}} 48$ KB, was dem gesamten Shared Memory entspricht. Bei größeren Operanden wie z. B. NO_BITS $= 4096$ halbiert sich die Anzahl der gleichzeitig auf einem SMP ausgeführten Exponentiationen, so dass nur ein Speicherbedarf von 32 KB erforderlich ist.

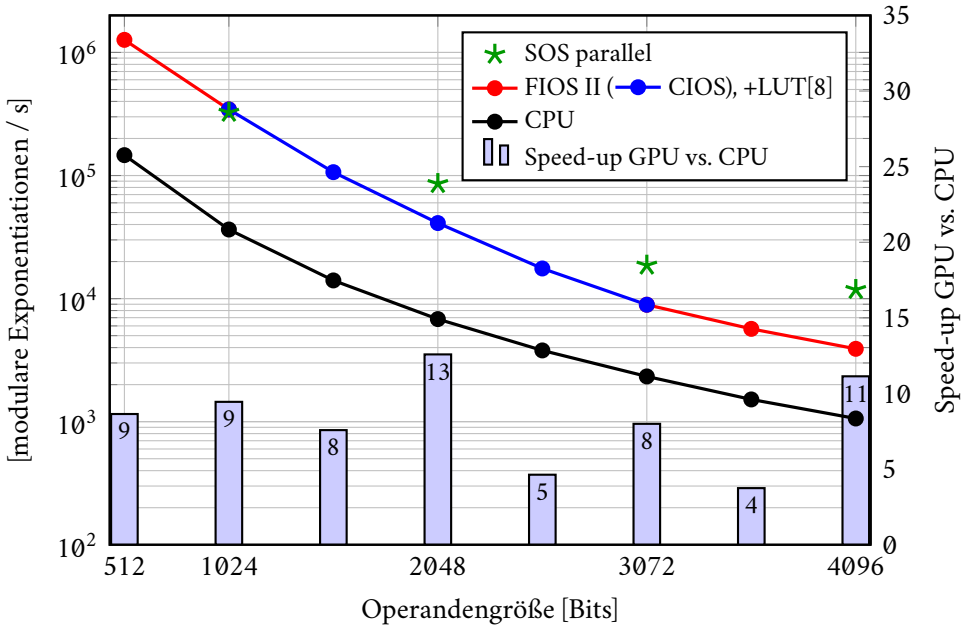


Abbildung 5.14: Performance CIOS / FIOS II vs. SOS-parallel (LUT, LUT_DEPTH $= 8$) und maximaler Speed-Up durch Berechnung von modularen Exponentiationen auf der Grafikkarte unter Berücksichtigung aller Algorithmen.

Abbildung 5.14 vergleicht die SOS-parallel Variante mit der CIOS / FIOS II Variante bei Verwendung von LUTs mit LUT_DEPTH $= 8$. Bei Operanden größer gleich 2048 Bit ist die SOS-parallel Variante den anderen Algorithmen überlegen. Das Balkendiagramm in Abbildung 5.14 zeigt den maximalen Geschwindigkeitsgewinn, der sich durch Auslagerung von modularen Exponentiationen auf die Grafikkarte ergibt. Neben der Performance bekräftigen der Preis und das Performance-pro-Watt-Verhältnis von Grafikkarten die Praxistauglichkeit der in dieser Arbeit vorgestellten Algorithmen.

Einfluss des Konvertierungsalgorithmuses

Bisher wird bei allen Implementierungen die MRC Konvertierung auf der CPU ausgeführt, um das Endergebnis aus den beiden Teilergebnissen x_0 und x_1 zu

	1024 bit [Exp./s]	2048 bit [Exp./s]	3072 bit [Exp./s]	4096 bit [Exp./s]
MRC	325337	85790	18637	11822
SRC	291660	77552	17898	11327
SRC GPU	251544	75078	17643	11410

Tabelle 5.3: Einfluss des Konvertierungsalgorithmuses auf die Performance bei der SOS-parallel Variante mit LUT_DEPTH $= 8$.

berechnen. Nach Gleichung 5.7 lassen sich die beiden Summanden getrennt voneinander parallel berechnen. Deshalb wird der SRC Algorithmus sowohl auf der CPU als auch auf der Grafikkarte implementiert. Das Ergebnis zeigt Tabelle 5.3 (SOS-parallel Variante mit LUT_DEPTH $= 8$). Deutliche Unterschiede in der Performance lassen sich bei Operanden niedriger Bitlänge erkennen. Bei allen Bitlängen ist beim MRC Algorithmus eine größere Performance als beim SRC Algorithmus zu erkennen.

5.6 Stand der Technik und Vergleich

Brickell und Gordon haben ein Verfahren zur schnellen modularen Exponentiation mit Vorberechnungen vorgestellt [23]. Es sind $(1 + o(1))(\log_2(\mathtt{NO\_BITS})/\log_2(\log_2(\mathtt{NO\_BITS})))$ Multiplikationen notwendig, um das Endergebnis bei einer LUT mit $\mathcal{O}(\log_2(\mathtt{NO\_BITS})/\log_2(\log_2(\mathtt{NO\_BITS})))$ Elementen zu berechnen. Kern des Verfahrens ist, den Exponenten e als Vielfaches einer Basis b zu berechnen:

$$e = \sum_{i=0}^{u-1} (p_i k_i b^i)$$

Die Koeffizienten $p_i \in M$ werden als *Multiplizierer* bezeichnet und beeinflussen die Größe der LUT. Multiplizierer können mehrfach verwendet werden, wobei ihr Einfluss auf die Zerlegung durch den Koeffizienten k_i gesteuert wird.

Das in dieser Arbeit verwendete Verfahren basiert auf LUTs mit $b = 2^{\mathtt{LUT\_DEPTH}}$ und $p_i \in \{1, \ldots, 2^{\mathtt{LUT\_DEPTH}}\}$ und $k_i = \{0, 1\}$ und lässt sich als ein Sonderfall des Verfahrens von Brickell und Gordon betrachten.
Brickell und Gordon haben gezeigt, dass bei Inkaufnahme einer geringen Verschlechterung der Laufzeit, die Größe der LUT durch Zerlegungen verringert werden kann. Ihre Ergebnisse lassen sich allerdings nur schwer auf Grafikkarten übertragen. Grafikkarten verfügen in der Regel über einen Speicher von mehreren Gigabyte, so dass auch größere Tabellen gespeichert werden können. Die Zerlegung des Exponenten erfordert dagegen einen nicht zu vernachlässigenden Rechenaufwand.

RSA-Operationen auf Grafikkarten

Das in dieser Arbeit vorgestellte Verfahren ist nach aktuellem Kenntnisstand des Autors das erste Verfahren, um modulare Exponentiationen mit festen Basen auf Grafikkarten zu berechnen. Es gibt allerdings eine Reihe von Veröffentlichungen, die zeigen, wie RSA-Operationen, d. h. modulare Exponentiationen mit variablen Basen auf Grafikkarten berechnet werden können. An dieser Stelle soll gezeigt werden, dass die Ergebnisse dieser Arbeit auch zur Beschleunigung von RSA-Operationen auf GPUs verwendet werden können.

Erstmals hat Fleissner 2007 gezeigt, wie Teile der Montgomery-Exponentiation auf Grafikkarten ausgelagert werden können [65]. Er verwendet eine traditionelle Grafikkarte (GeForce 8800 GTX) und beschränkt sich beim Benchmarking auf Operanden geringer Größe (192 Bit). Sein Algorithmus berechnet die Montgomery-Multiplikation auf der Grafikkarte, der Exponentiationsalgorithmus wird auf der CPU ausgeführt.
Szerwinski und Güneysu führen dagegen die komplette Montgomery-Exponentiation auf der Grafikkarte aus [161]. Die Autoren implementieren für Operanden der Länge 1024 und 2048 Bit die CIOS Variante auf der Grafikkarte in CUDA.

2011 hat Kaihara 2048 Bit RSA auf Grafikkarten implementiert [96]. Zwei Warps berechnen eine RSA-Operation. Da die Operanden aus 64 Wörtern bestehen, werden bei der Addition auftretende Überträge in einem Carry-Array gespeichert und später zum Ergebnis hinzuaddiert. In logarithmischer Zeit wird überprüft, ob weitere Überläufe generiert wurden. Ist das der Fall, werden sie zum Endergebnis addiert.
Im selben Jahr haben Jang et al. gezeigt, dass GPUs als Beschleuniger für SSL verwendet werden können [88]. Die Autoren konnten Montgomery-Exponentiationen für Operanden der Größe 1024 und 2048 Bit auf der Grafikkarte implementieren. Wie bei Kaihara werden Carrys in einem Array gespeichert und später zum Ergebnis hinzuaddiert. Falls bei der Addition weitere Überträge hinzukommen, werden diese in linearer Laufzeit zum Ergebnis addiert.
Die vorliegende Arbeit hat gezeigt, wie sich das Carry-Handling so verbessern lässt, dass im ungünstigsten Fall nur eine logarithmische Laufzeit erforderlich ist, um alle Carrys zu verarbeiten. Die Wahrscheinlichkeit ist in der Praxis gering, dass ein Carry ein zweites Carry generiert, wodurch sich die Laufzeit des Verfahren von Jang et al. im Durchschnitt kaum verbessern ließe. Die Autoren betrachten in ihrer Veröffentlichung allerdings nicht die Gefahr von Seitenkanalangriffen in SSL [25]. Um solche Angriffe zu verhindern, muss jede RSA-Operation die gleiche Ausführungszeit haben, so dass sich die logarithmische Laufzeit beim Carry-Handling positiv auf die Performance auswirkt.

Karatsuba Multiplikation

Die Langzahlmultiplikation bei der SOS Variante (Algorithmus 5.4, Zeile 1 bis 6) kann durch den *Karatsuba Algorithmus* ersetzt werden, der eine Laufzeit von $\mathcal{O}(\texttt{NO\_WORDS}^{\log_2(3)}) = \mathcal{O}(\texttt{NO\_WORDS}^{1,585})$ anstatt $\mathcal{O}(\texttt{NO\_WORDS}^2)$ hat [98]. Zhao hat die Karatsuba Multiplikation auf der GPU implementiert. Er kann jedoch keinen Performancegewinn gegenüber der CIOS Variante feststellen[85] [178].
Während der Anfertigung dieser Arbeit wurde der Karatsuba Algorithmus auf einer GeForce GTX 580 implementiert. Es konnte zwar eine Performanceverbesserung der Implementierung von Zhao erreicht werden, durch die gestiegene Komplexität und durch den erhöhten Bedarf an temporärem Speicher [110] ist die Anwendung des Karatsuba Algorithmus allerdings weiterhin auf Grafikkarten bei Operandenlängen zwischen $\texttt{NO\_BITS} = 512$ bis $\texttt{NO\_BITS} = 4096$ ineffizient.

[85] Zhao verwendet eine GeForce GTX 280.

6 Zusammenfassung

Crowdsourcing-Szenarien nehmen in der Praxis immer mehr an Bedeutung zu. DCUs senden beim nicht-interaktiven Crowdsourcing Daten automatisch und ohne Benutzerinteraktion an einen Provider. Da sich diese Daten häufig mit den Lebensgewohnheiten der Gerätebesitzer verknüpfen lassen können, ergeben sich daraus Anforderungen an den Datenschutz. Gleichzeitig ist der Provider auf korrekte Daten angewiesen. In dieser Arbeit werden erstmals diese beiden unterschiedlichen Anforderungen für nicht-interaktive Crowdsourcing-Szenarien beschrieben und ihre praktische Relevanz anhand von zwei Szenarien erläutert.
Im FCD-Szenario berechnet ein Dienstanbieter aus den Positionsdaten mobiler Endgeräte Echtzeitverkehrsdaten. Es zeigt sich am Beispiel von Google, dass ein Tracking einzelner Endgeräte durch den Dienstanbieter (Google) möglich ist. Gleichzeitig kann die Generierung der Echtzeitverkehrsdaten manipuliert werden, ohne durch den Dienstanbieter erkannt zu werden.
Im Smart Metering Szenario verwendet ein VNB die Verbrauchsdaten von Smart Metern zur Optimierung der Kraftwerksauslastung. Aus den Daten lassen sich Lastprofile erzeugen, aus denen sich Rückschlüsse auf die Lebensgewohnheiten der Kunden ziehen lassen können. Die Anonymisierung der Daten ist zwar möglich, gleichzeitig müssen die Auswirkungen eines Angreifers, der falsche Verbrauchsdaten an den VNB sendet, aber gering sein.

Die in dieser Arbeit vorgestellten „Ticket-Chain“ und „Ticket-Spender“ Protokolle erfüllen die Sicherheits- und Datenschutzanforderungen in nicht-interaktiven Crowdsourcing-Szenarien. Die beiden auf Zero-Knowledge Proofs basierenden Protokolle wurden auf unterschiedlichen Hardwareplattformen implementiert und ihre Praxistauglichkeit gezeigt.
Bei einem Vergleich mit anderen Lösungsansätzen zeigt sich, dass die beiden Protokolle nach aktuellem Kenntnisstand die ersten Protokolle sind, die alle

Sicherheits- und Datenschutzanforderungen berücksichtigen. Ein wesentlicher Unterschied zu anderen Verfahren ist, dass keine TTP erforderlich ist.

Aufgrund der hohen Implementierungskomplexität von Protokollen mit Zero-Knowledge Proofs ist im Rahmen dieser Arbeit eine domänenspezifische Eingabesprache und ein Compiler entwickelt worden, mit denen sich auf zahlentheoretischen Problemen basierende Protokolle mithilfe von Crypto-Objekten implementieren lassen können.
Der Crypto-Compiler vereint Framework, Interpreter und Compiler, ohne die Funktionalität der Crypto-Objekte mehrfach beschreiben zu müssen. Die Beschreibung der semantischen Aktionen des Compilers erfolgt anhand von durch CallMI-Makros markiertem C++ Code und nicht wie bei klassischen Compilern in einer Zwischensprache. Damit lässt sich das Crypto-Framework zu einem Compiler erweitern. Der zusätzliche Aufwand zur Markierung der Variablendeklarationen und Funktionsaufrufe ist begrenzt und die Komplexität des CMF bleibt dem Programmierer verborgen. Das CMF basiert auf C++, weshalb es sich nahtlos in das Crypto-Framework integriert, so dass in modernen Entwicklungsumgebungen selbst die fehlerhafte Anwendung des CMF bereits während des Programmierens erkennbar ist.
Ein weiterer Vorteil des CMF ist die Unabhängigkeit vom Rest des Crypto-Compilers und die Turing-vollständige Zwischensprache. Das Crypto-Framework lässt sich um neue Funktionen erweitern, ohne das Backend anpassen zu müssen. Die Trennung zwischen Code-Generatoren und dem Rest des CMF vereinfacht die Erweiterung des Crypto-Compilers um eine neue Zielsprache.
Die Eingabesprache verfügt im Gegensatz zu anderen Sprachen über eine hohe Ausdrucksmächtigkeit bei einer gleichzeitig geringen Anzahl von Sprachelementen. Ein modularer Aufbau des Crypto-Frameworks gewährleistet eine einfache Austauschbarkeit und Erweiterbarkeit. Durch Konfigurationsparameter lassen sich Performancemessungen von Protokollen einfach durchführen. Die Tupel-Schreibweise vereinfacht die Beschreibung komplexer Zero-Knowledge Proofs durch Gruppierung von Sprachelementen.
Im Vergleich zu anderen Compilern und Frameworks führt der Crypto-Compiler nicht nur High-Level-Optimierungen wie z. B. die Erkennung von Redundanzen in ZPK-Statements aus, sondern kann vor der Kompilierungsphase auch den Zwischencode optimieren. Im Rahmen dieser Arbeit wurde ein Algorithmus implementiert, der nicht relevanten Code entfernt. Da alle Variablen (und somit auch Variablen zur Beschreibung von Langzahlen) durch CCRs repräsentiert werden, ist eine Optimierung der in der Regel rechenintensiven Langzahloperationen möglich. Der Crypto-Compiler kann damit effizienteren Code erzeugen als klassische Compiler wie der GCC C-Compiler [72] oder der Oracle Java Compiler [132].

In nicht-interaktiven Crowdsourcing-Szenarien stellt der Provider aufgrund der zu verarbeitenden Datenmengen einen Flaschenhals dar. Die Ausführung des

„Ticket-Chain“ und „Ticket-Spender“ Protokolls erfordert die Berechnung einer Vielzahl von modularen Exponentiationen mit festen Basen. Diese Arbeit zeigt, wie solche Exponentiationen auf Grafikkarten effizienter ausgeführt werden können. Dazu werden verschiedene Varianten des Montgomery-Verfahrens auf einer Grafikkarte implementiert und miteinander verglichen.
Berechnet jeweils ein Thread auf der GPU eine Exponentiation, zeigt sich, dass mit geändertem Caching-Schema und einem nicht-coalesced Speicherlayout die CIOS und eine erweiterte FIOS Variante die besten Ergebnisse liefern. Durch die Verwendung von LUTs kann für alle untersuchten Bitlängen eine höhere Performance auf der GPU als auf einer vergleichbaren CPU erreicht werden.
Wird die Berechnung einer modularen Exponentiation auf 16 bzw. 32 Threads verteilt und Shared Memory sowie Kogge-Stone Addierer verwendet, lässt sich die Leistung weiter erhöhen. Bei relevanten Modulus-Bitlängen von 1024, 2048, 3072 und 4096 Bits ergeben Messungen eine um ca. Faktor 10 höhere Anzahl an modularen Exponentiationen pro Sekunde als bei Ausführung der Berechnungen auf einer CPU.
Ein Vergleich mit den Veröffentlichungen anderer Autoren zeigt, dass sich die Ergebnisse dieser Arbeit auch zur Beschleunigung von RSA-Operationen auf Grafikkarten einsetzen lassen können.

6.1 Ausblick

Camenisch und Lysyanskaya haben eine Variante des CL-Verfahrens auf Grundlage von Pairings vorgestellt [34]. Dadurch können Gruppen basierend auf elliptischen Kurven verwendet werden, so dass Gruppenelemente bei demselben Sicherheitslevel eine geringere Bitlänge aufweisen als beim CL-Verfahren basierend auf multiplikativen Gruppen [44].
In einem ersten Ansatz wurde der Crypto-Compiler bereits erweitert, um einfache Pairings in der Eingabesprache zu beschreiben und das erweiterte CL-Verfahren zu implementieren [158]. Messungen haben allerdings ergeben, dass ein Performancegewinn trotz der geringeren Gruppenordnungen nicht erreicht werden kann. Ein Grund hierfür ist die höhere Berechnungskomplexität einzelner Pairing-Operationen und die ineffiziente Implementierung der verwendeten Pairingbibliothek.

Die Eingabesprache des Crypto-Compilers lässt sich nicht nur um Pairing-Operationen erweitern, sondern auch um weitere Sprachelemente. Dazu gehört z. B. die Unterstützung von Verschlüsselungs- oder Signaturalgorithmen oder die Implementierung von Kontrollstrukturen. Dabei liefert der Compiler in Form des Crypto-Frameworks und des CMF bereits alle Voraussetzungen.

In dieser Arbeit wird gezeigt, wie die Berechnung modularer Exponentiationen durch GPUs beschleunigt werden kann. Für die Validierung eines ZPK sind

allerdings weitere Langzahloperationen wie Additionen und Multiplikationen erforderlich. Weitere Forschungsarbeit ist notwendig, um zu untersuchen, welche Operationen bei einem komplettem ZPK effizienter auf der GPU ausgeführt werden können.

Anhang

A

Die folgenden Seiten zeigen die Grammatik der Eingabesprache des Crypto-Compilers in Bison, sowie die Beschreibung des „Ticket-Chain“ und „Ticket-Spender“ Protokolls in der Eingabesprache des Crypto-Compilers für $\ell_n = 1024$.

A.1 Parser

```
1  /* Program */
2  program       : /* empty */                                { DBG("program : /* empty */", @$);                                  }
3                | statementlist                               { DBG("program : statementlist ';'", @$);                            }
4  ;
5
6
7  /* Statement */
8  statementlist : statement ';'                               { DBG("statementlist : statement ';'", @$);                          }
9                | statementlist statement ';'                 { DBG("statementlist : statementlist statement ';'", @$);            }
10 ;
11               /* Block */
12 statement     : block                                       { DBG("statement : block", @$);                                      }
13               /* ConfigParam */
14               | KW_CONFIGPARAM IDENT                        { DBG("statement : KW_CONFIGPARAM IDENT", @$);
15                                                               ASRT(p->createConfigParamBnz($2, STR("0")), @$);                  }
16               | KW_CONFIGPARAM IDENT ':' '=' number         { DBG("statement : KW_CONFIGPARAM IDENT ':' '=' number", @$);
17                                                               ASRT(p->createConfigParamBnz($2, $5), @$);                        }
18               | KW_CONFIGPARAM IDENT ':' '=' QUOTED_STRING  { DBG("statement : KW_CONFIGPARAM IDENT ':' '=' QUOTED_STRING", @$);
19                                                               ASRT(p->createConfigParamString($2, $5), @$);                     }
20               | KW_EXPORT KW_CONFIGPARAM IDENT              { DBG("statement : KW_EXPORT KW_CONFIGPARAM IDENT", @$);
21                                                               ASRT(p->createConfigParamBnz($3, STR("0"), true), @$);            }
22               | KW_EXPORT KW_CONFIGPARAM IDENT ':' '=' number { DBG("statement : KW_EXPORT KW_CONFIGPARAM IDENT ':' '=' NUMBER", @$);
23                                                               ASRT(p->createConfigParamBnz($3, $6, true), @$);                  }
24               | KW_EXPORT KW_CONFIGPARAM IDENT ':' '='
25                 QUOTED_STRING                               { DBG("statement : KW_EXPORT KW_CONFIGPARAM IDENT ':' '=' QUOTED_STRING",
26                                                                    @$);
27                                                               ASRT(p->createConfigParamString($3, $6, true), @$);               }
28               | confparambnz ':' '=' number                 { DBG("statement : confparambnz ':' '=' number", @$);
29                                                               ASRT(p->setConfigParamBnz($1, $4), @$);                           }
30               | confparamstr ':' '=' QUOTED_STRING          { DBG("statement : confparambnz ':' '=' QUOTED_STRING", @$);
31                                                               ASRT(p->setConfigParamString($1, $4), @$);                        }
32               | KW_SET_CONFIG_PARAM '(' confparambnz ','
33                 confparambnz ',' KW_PRIME ')'               { DBG("statement: KW_SET_CONFIG_PARAM '(' confparambnz ',' confparambnz, \
34                                                                      ',' KW_PRIME ')'", @$);
35                                                               ASRT(p->setConfigParamPrime($3, $5), @$);                         }
```

```
36                | KW_SET_CONFIG_PARAM '(' confparambnz ','
37                  confparambnz ',' confparambnz ','
38                  KW_SAFE_PRIME ')'                           { DBG("statement: KW_SET_CONFIG_PARAM '(' confparambnz ',' ... ',' \
39                                                                      KW_SAFE_PRIME ')'", @$);
40                                                                ASRT(p->setConfigParamSafePrime($3, $5, $7), @$);                    }
41                | KW_SET_CONFIG_PARAM '(' confparambnz ','
42                  confparambnz ',' confparambnz ','
43                  confparambnz ',' KW_SCHNORR ')'             { DBG("statement: KW_SET_CONFIG_PARAM '(' confparambnz ',' ... ',' \
44                                                                      KW_SCHNORR ')'", @$);
45                                                                ASRT(p->setConfigParamSchnorr($3, $5, $7, $9), @$);                  }
46                | KW_SET_CONFIG_PARAM '(' confparambnz ','
47                  confparambnz ',' confparambnz ','
48                  confparambnz ',' KW_RSA ')'                 { DBG("statement: KW_SET_CONFIG_PARAM '(' confparambnz ',' ... ',' \
49                                                                      KW_RSA ')'", @$);
50                                                                ASRT(p->setConfigParamRSA($3, $5, $7, $9), @$);                      }
51                | KW_SET_CONFIG_PARAM '(' confparambnz ','
52                  confparambnz ',' confparambnz ','
53                  confparambnz ',' confparambnz ','
54                  confparambnz ','
55                  KW_SAFE_PRIME ',' KW_RSA ')'                { DBG("statement: KW_SET_CONFIG_PARAM '(' confparambnz ',' ... ',' \
56                                                                      KW_SAFE_PRIME ',' KW_RSA ')'", @$);
57                                                                ASRT(p->setConfigParamSafeRSA($3, $5, $7, $9, $11, $13), @$);        }
58                /* Group */
59                | IDENT ':' '=' KW_GROUP_Z '(' ')' bitlength   { DBG("statement : IDENT ':' '=' KW_GROUP_Z '(' ')' bitlength", @$);
60                                                                ASRT(p->createZGroup($1, NULL, NULL, $7), @$);                       }
61                | IDENT ':' '=' KW_GROUP_Z '('
62                  confparambnzin ')' bitlength                { DBG("statement : IDENT ':' '=' KW_GROUP_Z '(' confparambnzin ')' \
63                                                                      bitlength", @$);
64                                                                ASRT(p->createZGroup($1, $6, NULL, $8, CryptoComp::exp), @$);        }
65                | IDENT ':' '=' KW_GROUP_Z '('
66                  confparambnzin ',' KW_EXACT ')' bitlength    { DBG("statement : IDENT ':' '=' KW_GROUP_Z '(' confparambnzin ',' \
67                                                                      KW_EXACT ')' bitlength", @$);
68                                                                ASRT(p->createZGroup($1, $6, NULL, $10, CryptoComp::exp_exact), @$); }
69                | IDENT ':' '=' KW_GROUP_Z '('
70                  confparambnzin ',' confparambnzin ')'
71                  bitlength                                   { DBG("statement : IDENT ':' '=' KW_GROUP_Z '(' confparambnzin ',' \
72                                                                      confparambnzin ')' bitlength", @$);
73                                                                ASRT(p->createZGroup($1, $6, $8, $10), @$);                          }
```

```
74              | IDENT ':' '=' KW_GROUP_Z_ADD '('
75                confparambnzin ')' bitlength                { DBG("statement : IDENT ':' '=' KW_GROUP_Z_ADD '(' confparambnzin ')' \
76                                                                  bitlength", @$);
77                                                            ASRT(p->createZaddGroup($1, $6, $8), @$);                                 }
78              | IDENT ':' '=' KW_GROUP_Z_ADD '('
79                confparambnzin ',' KW_PRIME ')' bitlength    { DBG("statement : IDENT ':' '=' KW_GROUP_Z_ADD '(' confparambnzin ',' \
80                                                                  KW_PRIME ')' bitlength", @$);
81                                                            ASRT(p->createZaddGroup($1, $6, $10, CryptoComp::prime), @$);             }
82              | IDENT ':' '=' KW_GROUP_EC_PRIME '(' IDENT ')' { DBG("statement : IDENT ':' '=' KW_GROUP_EC_PRIME '(' IDENT ')' \
83                                                                 ", @$);
84                                                            ASRT(p->createECprimeGroup($1, $6), @$);                                  }
85              | IDENT ':' '=' KW_GROUP_EC_PRIME '('
86                confparambnzin ',' confparambnzin ','
87                confparambnzin ',' confparamstrin ','
88                confparambnzin ',' confparambnzin ')'       { DBG("statement : IDENT ':' '=' KW_GROUP_EC_PRIME '(' confparambnzin \
89                                                                  ',' ... ')'", @$);
90                                                            ASRT(p->createECprimeGroup($1, $6, $8, $10, $12, $14, $16), @$);          }
91              | IDENT ':' '=' KW_GROUP_Z_MUL '('
92                confparambnzin ',' KW_PRIME ')' bitlength    { DBG("statement : IDENT ':' '=' KW_GROUP_Z_MUL '(' confparambnzin ',' \
93                                                                  KW_PRIME ')' bitlength", @$);
94                                                            ASRT(p->createZmulGroup($1, $6, NULL, $10, CryptoComp::prime), @$);       }
95              | IDENT ':' '=' KW_GROUP_Z_MUL '('
96                confparambnzin ',' confparambnzin ','
97                KW_SAFE_PRIME ')' bitlength                 { DBG("statement : IDENT ':' '=' KW_GROUP_Z_MUL '(' confparambnzin ',' \
98                                                                  confparambnzin ',' KW_SAFE_PRIME ')' bitlength", @$);
99                                                            ASRT(p->createZmulGroup($1, $6, $8, $12, CryptoComp::safe_prime), @$);    }
100             | IDENT ':' '=' KW_GROUP_Z_MUL '('
101               confparambnzin ',' confparambnzin ','
102               KW_SAFE_PRIME ',' KW_QR ')' bitlength       { DBG("statement : IDENT ':' '=' KW_GROUP_Z_MUL '(' confparambnzin \
103                                                                 ',' ... ',' KW_SAFE_PRIME ',' KW_QR ')' bitlength", @$);
104                                                           ASRT(p->createZmulGroup($1, $6, $8, $14, CryptoComp::safe_prime_qr), @$);}
105             | IDENT ':' '=' KW_GROUP_Z_MUL '('
106               confparambnzin ',' confparambnzin ','
107               KW_SCHNORR ')' bitlength                    { DBG("statement : IDENT ':' '=' KW_GROUP_Z_MUL '(' confparambnzin ',' \
108                                                                 confparambnzin ',' KW_SCHNORR ')' bitlength", @$);
109                                                           ASRT(p->createZmulSchnorrGroup($1, $6, $8, $12), @$);                     }
110             | IDENT ':' '=' KW_GROUP_Z_MUL '('
111               confparambnzin ',' confparambnzin ','
```

```
112                 confparambnzin ',' KW_PRIME ')' bitlength     { DBG("statement : IDENT ':' '=' KW_GROUP_Z_MUL '(' confparambnzin \
113                                                                       ',' ... ',' confparambnzin ',' KW_PRIME ')' bitlength", @$);
114                                                                 ASRT(p->createZmulnGroup($1, $6, $8, $10, NULL, NULL, $14,
115                                                                      CryptoComp::prime), @$);                                  }
116               | IDENT ':' '=' KW_GROUP_Z_MUL '('
117                 confparambnzin ',' confparambnzin ','
118                 confparambnzin ',' confparambnzin ','
119                 confparambnzin ',' KW_SAFE_PRIME ')'
120                 bitlength                                     { DBG("statement : IDENT ':' '=' KW_GROUP_Z_MUL '(' confparambnzin \
121                                                                       ',' ... ',' KW_SAFE_PRIME ')' bitlength", @$);
122                                                                 ASRT(p->createZmulnGroup($1, $6, $8, $10, $12, $14, $18,
123                                                                      CryptoComp::safe_prime), @$);                             }
124               | IDENT ':' '=' KW_GROUP_Z_MUL '('
125                 confparambnzin ',' confparambnzin ','
126                 confparambnzin ',' confparambnzin ','
127                 confparambnzin ',' KW_SAFE_PRIME ','
128                 KW_QR ')' bitlength                           { DBG("statement : IDENT ':' '=' KW_GROUP_Z_MUL '(' confparambnzin \
129                                                                       ',' ... ',' KW_SAFE_PRIME ',' KW_QR ')' bitlength", @$);
130                                                                 ASRT(p->createZmulnGroup($1, $6, $8, $10, $12, $14, $20,
131                                                                      CryptoComp::safe_prime_qr), @$);                          }
132               | IDENT ':' '=' KW_GROUP_Z_MUL_PHI
133                 '(' group ')'                                 { DBG("statement : IDENT ':' '=' KW_GROUP_Z_MUL_PHI '(' group ')'", @$);
134                                                                 ASRT(p->createZmulPhiGroup($1, $6), @$);                    }
135               | IDENT ':' '=' '[' grouplist ']'               { DBG("statement : IDENT ':' '=' '[' grouplist ']'", @$);
136                                                                 ASRT(p->createTupleGroup($1, $5), @$);                      }
137               /* Pairing */
138               | IDENT ':' '=' KW_PAIRING_A
139                 '(' group ',' group ',' IDENT ')'             { DBG("statement : IDENT ':' '=' KW_PAIRING '(' group ',' group ',' \
140                                                                       IDENT ')'", @$);
141                                                                 ASRT(p->createPairing($1, $6, $8, $10), @$);                }
142               /* Var definition */
143               | group ':'                                     { DBG("statement : group ':' <midrule>", @$);
144                                                                 ps->pVarGrp = $1;                                           }
145                 vardeflist                                    { DBG("statement : group ':' vardeflist", @$);              }
146               /* VarAssignment */
147               | variable '='                                  { DBG("statement : variable '=' <midrule>", @$);
148                                                                 ASRT(ps->exprStack.empty(), @$);                            }
149                 expr                                          { DBG("statement : variable '=' expr", @$);
```

```
150                                                     ASRT(ps->exprStack.empty(), @$);
151                                                     ASRT(p->createVarAssignment($1, $4), @$);                       }
152             /* Homomorphism */
153             | IDENT '[' group '-' '>' group ']' '='         { DBG("statement : IDENT '[' group '-' '>' group ']' '=' <midrule>", @$);
154                                                     ps->pSrcGrp = $3; ASRT(ps->exprStack.empty(), @$);              }
155               expr                                  { DBG("statement : IDENT '[' group '-' '>' group ']' '=' expr", @$);
156                                                     ASRT(ps->exprStack.empty(), @$);
157                                                     ASRT(p->createHom($1, $3, $6, $10), @$);
158                                                     // clear source group
159                                                     ps->pSrcGrp = NULL;                                             }
160             /* Sigma */
161             | IDENT '=' KW_SIGMA_PHI '[' hom ','
162               variable ',' variable ',' constnumhash ']'    { DBG("statement : IDENT '=' KW_SIGMA_PHI '[' hom ',' variable ',' \
163                                                           variable ',' constnumhash ']'", @$);
164                                                     ASRT(p->createSigmaPhi($1, $5, $7, $9, $11), @$);               }
165             | IDENT '=' KW_SIGMA_GSP '[' hom ','
166               variable ',' variable ','
167               constnum ',' constnumhash ']'                 { DBG("statement : IDENT '=' KW_SIGMA_GSP '[' hom ',' variable \
168                                                           ',' ... ',' constnumhash ']'", @$);
169                                                     ASRT(p->createSigmaGsp($1, $5, $7, $9, $11, $13), @$);          }
170             | IDENT '=' KW_SIGMA_AND '[' sigmalist ']'      { DBG("statement : IDENT '=' KW_SIGMA_AND '[' sigmalist ']'", @$);
171                                                     ASRT(p->createSigmaAnd($1, $5), @$);                            }
172             | IDENT '=' KW_SIGMA_OR '[' sigmalist ']'       { DBG("statement : IDENT '=' KW_SIGMA_OR '[' sigmalist ']'", @$);
173                                                     ASRT(p->createSigmaOr($1, $5), @$);                             }
174             /* ZPK statement */
175             | IDENT '=' KW_PK '['                           { DBG("statement : IDENT '=' KW_PK '[' <midrule>", @$);
176                                                     p->beginZPK();                                                  }
177               '(' varidentlist ')' ':' preddisjop ','
178               KW_SIGMA_PHI ',' constnum ']'                 { DBG("statement : IDENT '=' KW_PK '[' '(' varidentlist ')' ':' \
179                                                           preddisjop ',' KW_SIGMA_PHI ',' constnum ']'", @$);
180                                                     ASRT(p->createZPK(zpk_sigma_phi, $1, $7, $10, true, $14,
181                                                           new hashFunctions_t(HASH_DEFAULT)), @$);                  }
182             | IDENT '=' KW_PK '['                           { DBG("statement : IDENT '=' KW_PK '[' <midrule>", @$);
183                                                     p->beginZPK();                                                  }
184               '(' varidentlist ')' ':' preddisjop
185               optintcheck ',' KW_SIGMA_GSP ','
186               constnum ',' constnum ']'                     { DBG("statement : IDENT '=' KW_PK '[' '(' varidentlist ')' ':' \
187                                                           preddisjop optintcheck ',' KW_SIGMA_GSP ',' constnum ',' constnum \
```

```
188                                                                          ']'", @$);
189                                                                        ASRT(p->createZPK(zpk_sigma_gsp, $1, $7, $10, true, $15,
190                                                                             new hashFunctions_t(HASH_DEFAULT), $17, $11), @$);                  }
191               | IDENT '=' KW_SPK '['                                   { DBG("statement : IDENT '=' KW_SPK '[' <midrule>", @$);
192                                                                          p->beginZPK();                                                      }
193                 '(' varidentlist ')' ':' preddisjop ','
194                 KW_SIGMA_PHI ',' consthash binary ']'                  { DBG("statement : IDENT '=' KW_SPK '[' '(' varidentlist ')' ':' \
195                                                                             preddisjop ',' KW_SIGMA_PHI ',' consthash binary ']'", @$);
196                                                                        ASRT(p->createZPK(zpk_sigma_phi, $1, $7, $10, false, 0, $14, 0, NULL,
197                                                                             $15), @$);                                                      }
198               | IDENT '=' KW_SPK '['                                   { DBG("statement : IDENT '=' KW_PK '[' <midrule>", @$);
199                                                                          p->beginZPK();                                                      }
200                 '(' varidentlist ')' ':' preddisjop
201                 optintcheck ',' KW_SIGMA_GSP ','
202                 constnum ',' consthash ']'                             { DBG("statement : IDENT '=' KW_SPK '[' '(' varidentlist ')' ':' \
203                                                                             preddisjop optintcheck ',' KW_SIGMA_GSP ',' constnum ',' \
204                                                                             consthash ']'", @$);
205                                                                        ASRT(p->createZPK(zpk_sigma_gsp, $1, $7, $10, false, 0, $17, $15,
206                                                                             $11), @$);                                                      }
207               /* Check */
208               | KW_CHECK '(' variable comparator expr ')'              { DBG("statement: KW_CHECK '(' variable comparator expr ')'", @$);
209                                                                          ASRT(p->createCheck($3, $4, $5), @$);                               }
210               /* print */
211               | KW_PRINT '(' IDENT ')'                                 { DBG("statement: KW_PRINT '(' IDENT ')'", @$);
212                                                                          ASRTMSG(p->print(print_decimal, $3), @$, ERR("Identifier unknown!"));    }
213               | KW_PRINT '(' IDENT ',' KW_HEX ')'                      { DBG("statement: KW_PRINT '(' IDENT ',' KW_HEX ')'", @$);
214                                                                          ASRTMSG(p->print(print_hex, $3), @$, ERR("Identifier unknown!"));        }
215               | KW_PRINT '(' QUOTED_STRING ',' IDENT ')'               { DBG("statement: KW_PRINT '(' QUOTED_STRING ',' IDENT ')'", @$);
216                                                                          ASRTMSG(p->print(print_decimal, $5, $3), @$, ERR("Identifier unknown!"));}
217               | KW_PRINT '(' QUOTED_STRING ',' IDENT ','
218                 KW_HEX ')'                                             { DBG("statement: KW_PRINT '(' QUOTED_STRING ',' IDENT ',' KW_HEX ')'", @$);
219                                                                          ASRTMSG(p->print(print_hex, $5, $3), @$, ERR("Identifier unknown!"));    }
220 ;
221
222
223 /* Block */
224 block         : beginblock                                             { DBG("block : beginblock <midrule>", @$);                          }
225                 '(' blckparamlist ')'                                  { DBG("block : beginblock '(' blckparamlist ')' <midrule>", @$);
```

```
226                                                     ASRT(p->addBlockParams($4), @$);                                    }
227                 '{' blockcode '}'                 { DBG("block : beginblock '(' blckparamlist ')' '{' blockcode '}' ';'", @$);
228                                                     p->endBlock();                                                      }
229 ;
230 beginblock    : IDENT                             { DBG("beginblock : IDENT", @$);
231                                                     ASRT(p->beginBlock($1), @$);                                        }
232               | KW_BLOCK_CRITICAL IDENT           { DBG("beginblock : KW_BLOCK_CRITICAL IDENT", @$);
233                                                     ASRT(p->beginBlock($2, true), @$);                                  }
234 ;
235 blockcode     : /* empty */                       { DBG("blockcode : /* empty */", @$);                               }
236               | statementlist                     { DBG("blockcode : statementlist", @$);                             }
237 ;
238 blckparam     : KW_BLOCK_PARAM_IN group ':' IDENT { DBG("blckparam : KW_BLOCK_PARAM_IN group ':' IDENT", @$);
239                                                     ASRT($$ = p->createBlockParam(CryptoComp::blockparam_in, $2, $4), @$);  }
240               | KW_BLOCK_PARAM_OUT group ':' IDENT { DBG("blckparam : KW_BLOCK_PARAM_OUT group ':' IDENT", @$);
241                                                     ASRT($$ = p->createBlockParam(CryptoComp::blockparam_out, $2, $4), @$);  }
242 ;
243
244
245 /* Group definition */
246 bitlength     : /* empty */                         { DBG("bitlength : /* empty */", @$);
247                                                     $$ = 0;                                                             }
248               | '[' constnum ']'                  { DBG("bitlength : '[' constnum ']'", @$);
249                                                     $$ = $2;                                                            }
250 ;
251
252
253 /* Var definition */
254 /* we can declare several variables for the same group in one statement */
255 vardeflist    : vardef                            { DBG("vardeflist : vardef", @$);                                   }
256               | vardeflist ',' vardef             { DBG("vardeflist : vardeflist ',' vardef", @$);                    }
257 ;
258 /* a variable is represented by an identifier and could be initialized by a number or tuple, respectively */
259 vardef        : IDENT                             { DBG("vardef : IDENT", @$);
260                                                     ASRT(p->createVar(ps->pVarGrp, $1,  NULL), @$);                     }
261               | IDENT '=' varinit                 { DBG("vardef : IDENT '=' varinit", @$);
262                                                     ASRT(p->createVar(ps->pVarGrp, $1, $3), @$);                        }
263 ;
```

```
varinit       : number                                  { DBG(”varinit : number”, @$);
                                                          ASRT($$ = p->createAnonAtomicVar(p->vectorNew<std::string>($1)), @$);    }
              | KW_POINT_OF_INF                         { DBG(”varinit : KW_POINT_OF_INF”, @$);
                                                          ASRT($$ = p->createAnonAtomicVar(p->vectorNew<std::string>(STR(”0”))),
                                                                                                                                   @$);    }
              | ’<’ numberlist ’>’                      { DBG(”varinit :’<’ numberlist ’>’”, @$);
                                                          ASRT($$ = p->createAnonAtomicVar($2), @$);                               }
              | ’<’ KW_POINT_OF_INF ’>’                 { DBG(”varinit :’<’ KW_POINT_OF_INF ’>’”, @$);
                                                          ASRT($$ = p->createAnonAtomicVar(p->vectorNew<std::string>(STR(”0”))),
                                                                                                                                   @$);    }
              | ’[’ varinitlist ’]’                     { DBG(”varinit :’[’ varinitlist ’]’”, @$);
                                                          ASRT($$ = p->createAnonTupleVar($2), @$);                                }
;


/* VarAssignment / Homomorphism */
/* an expression could be a composition of expressions or an addexpr */
expr          : expr ’:’                                { DBG(”expr : expr ’:’ <midrule>”, @$);
                                                          ps->exprStack.push($1); /* push $1 onto stack */                         }
                expr                                    { DBG(”expr : expr ’:’ expr”, @$);
                                                          ps->exprStack.pop(); $$ = $4; /* pop $1 from stack */                   }
              | addexpr                                 { DBG(”expr : addexpr”, @$);
                                                          $$ = $1;                                                                 }
;
addexpr       : addexpr ’+’ addexpr                     { DBG(”addexpr : addexpr ’+’ addexpr”, @$);
                                                          ASRTDESTR($$ = $1->createOpExpression($3), @$, $1->destruct();
                                                                    $3->destruct());                                               }
              | addexpr ’-’ addexpr                     { DBG(”addexpr : addexpr ’-’ addexpr”, @$);
                                                          ASRTDESTR($$ = $1->createIopExpression($3), @$, $1->destruct();
                                                                    $3->destruct());                                               }
              | powexpr                                 { DBG(”addexpr : powexpr”, @$);
                                                          $$ = $1;                                                                 }
;
powexpr       : powexpr ’^’ powexpr                     { DBG(”powexpr : powexpr ’^’ powexpr”, @$);
                                                          ASRTDESTR($$ = $1->createPowExpression($3), @$, $1->destruct();
                                                                    $3->destruct());                                               }
              | pairingexpr                             { DBG(”powexpr : pairingexpr”, @$);
                                                          $$ = $1;                                                                 }
```

```
302 ;
303 pairingexpr   : pairing '(' expr ',' expr ')'          { DBG("pairingexpr : '(' expr ',' expr ')'", @$);
304                                                          ASRTDESTR($$ = $3->createPairingExpression($1, $5), @$, $1->destruct();
305                                                                    $3->destruct()); }
306               | unaryexpr                               { DBG("pairingexpr : unaryexpr", @$);
307                                                          $$ = $1; }
308 ;
309 unaryexpr     : '-' unaryexpr                           { DBG("unaryexpr : '-' unaryexpr", @$);
310                                                          ASRTDESTR($$ = $2->createInvExpression(), @$, $2->destruct()); }
311               | dotexpr                                 { DBG("unaryexpr : dotexpr", @$);
312                                                          $$ = $1; }
313 ;
314 dotexpr       : dotexpr '.' number                      { DBG("dotexpr : dotexpr '.' number", @$);
315                                                          ASRTDESTR($$ = CryptoComp::TupleExpression::createDotExpression($1, $3),
316                                                                    @$, $1->destruct(); delete($3)); }
317               | atomexpr                                { DBG("dotexpr : atomexpr", @$);
318                                                          $$ = $1; }
319 ;
320 atomexpr      : '(' expr ')'                            { DBG("atomexpr : '(' expr ')'", @$);
321                                                          $$ = $2; }
322               | '$'                                     { DBG("atomexpr : '$'", @$);
323                                                          ASRTMSG(ps->pSrcGrp != NULL, @$,
324                                                                  ERR("$ operator only allowed in hom expressions!"));
325                                                          ASRT($$ = ps->pSrcGrp->createSrcVarExpression(), @$); }
326               | tmpvar                                  { DBG("atomexpr : tmpvar", @$);
327                                                          ASRTDESTR($$ = $1->copyExpression(NULL), @$, $1->destruct()); }
328               | variable                                { DBG("atomexpr : variable", @$);
329                                                          ASRTDESTR($$ = $1->createVarExpression(), @$, $1->destruct()); }
330               | IDENT '(' expr ')'                      { DBG("atomexpr : IDENT '(' expr ')'", @$);
331                                                          CryptoComp::Group *pGrp = p->getGroup(*$1);
332                                                          CryptoComp::Hom *pHom = p->getHom(*$1);
333                                                          delete($1);
334                                                          ASRT(pGrp != NULL || pHom != NULL, @$);
335                                                          if (pGrp != NULL) {
336                                                              ASRTDESTR($$ = $3->createCastExpression(pGrp), @$, $3->destruct());
337                                                          } else {
338                                                              ASRTDESTR($$ = pHom->createHomExpression($3), @$, $3->destruct());
339                                                          }
```

```
340                 | '?' group                                 { DBG("atomexpr : '?' group", @$);
341                                                               ASRTDESTR($$ = $2->createRndExpression(),@$, $2->destruct());           }
342                 | '~' group                                 { DBG("atomexpr : '~' group", @$);
343                                                               ASRTDESTR($$ = $2->createIdExpression(), @$, $2->destruct());           }
344                 | KW_GENERATOR '(' group ')'                { DBG("atomexpr : KW_GENERATOR '(' group ')'", @$);
345                                                               ASRTDESTR($$ = $3->createGeneratorExpression(), @$, $3->destruct());    }
346                 | number                                    { DBG("atomexpr : number", @$);
347                                                               CryptoComp::AtomicVar *pAtomicVar = NULL;
348                                                               ASRT(pAtomicVar = p->createAnonAtomicVar(p->vectorNew<std::string>($1)),
349                                                                                                        @$);
350                                                               ASRT($$ = pAtomicVar->createVarExpression(), @$);                       }
351                 | KW_POINT_OF_INF                           { DBG("atomexpr : KW_POINT_OF_INF", @$);
352                                                               CryptoComp::AtomicVar *pAtomicVar = NULL;
353                                                               ASRT(pAtomicVar = p->createAnonAtomicVar(
354                                                                    p->vectorNew<std::string>(STR("0"))), @$);
355                                                               ASRT($$ = pAtomicVar->createVarExpression(), @$);                       }
356                 | '<' numberlist '>'                        { DBG("atomexpr : '<' numberlist '>'", @$);
357                                                               ASRT($$ = (p->createAnonAtomicVar($2))->createVarExpression(), @$);     }
358                 | '<' KW_POINT_OF_INF '>'                   { DBG("atomexpr : '<' KW_POINT_OF_INF '>'", @$);
359                                                               CryptoComp::AtomicVar *pAtomicVar = NULL;
360                                                               ASRT(pAtomicVar = p->createAnonAtomicVar(
361                                                                    p->vectorNew<std::string>(STR("0"))), @$);
362                                                               ASRT($$ = pAtomicVar->createVarExpression(), @$);                       }
363                 | '[' exprlist ']'                          { DBG("atomexpr : '[' exprlist ']'", @$);
364                                                               $$ = $2;                                                                }
365 ;
366 tmpvar          : '#'                                       { DBG("tmpvar : '#'", @$);
367                                                               ASRTMSG($$ = ps->exprStack.top(), @$, ERR("no tmpvar available"));       }
368                 | tmpvar '#'                                { DBG("tmpvar : tmpvar '#'", @$);
369                                                               $$ = $1; /* dummy statement to get rid off warning */
370                                                               ASRTMSG($$ = ps->exprStack.decStackPointer(), @$,
371                                                               ERR("No tmpvar available!"));                                           }
372 ;
373
374
375 /* ZPK statement */
376 varidentlist    : variable                                  { DBG("varidentlist : variable", @$);
377                                                               ASRT($$ = p->vectorNew<CryptoComp::Var>($1), @$);                        }
```

```
378                | varidentlist ',' variable                  { DBG("varidentlist : varidentlist ',' variable", @$);
379                                                               ASRT($$ = p->vectorAppend<CryptoComp::Var>($1, $3), @$);              }
380  ;
381  preddisjop    : preddisjop '|' preddisjop                  { DBG("preddisjop : preddisjop '|' preddisjop", @$);
382                                                               ASRT($$ = $1->createZPKExprOpDisj($3), @$);                           }
383                | predkonjop                                 { DBG("preddisjop : predkonjop", @$);
384                                                               $$ = $1;                                                              }
385  ;
386  predkonjop    : predkonjop '&' predkonjop                  { DBG("predkonjop : predkonjop '&' predkonjop", @$);
387                                                               ASRT($$ = $1->createZPKExprOpKonj($3), @$);                           }
388                | predop                                     { DBG("predkonjop : predop", @$);
389                                                               $$ = $1;                                                              }
390  ;
391  predop        : '(' preddisjop ')'                         { DBG("predop : '(' preddisjop ')'", @$);
392                                                               $$ = $2;                                                              }
393                | '(' group ',' variable ',' variable ')'
394                  variable '=' variable '*' variable         { DBG("predop : '(' group ',' variable ',' variable ')' variable '=' \
395                                                                     variable '*' variable", @$);
396                                                               ASRT($$ = CryptoComp::PredicateOp::createMultPredOp($2, $4, $6, $8, $10,
397                                                                                                                   $12), @$);        }
398
399                | '(' group ',' variable ',' variable ')'
400                  expr '<' '=' variable '<' '=' expr         { DBG("predop : '(' group ',' variable ',' variable ')'  expr '<' '=' \
401                                                                     variable '<' '=' expr", @$);
402                                                               ASRT($$ = p->createIntervalPredicate($2, $4, $6, $8, $11, $14), @$);  }
403                | '(' group ',' variable ',' variable ')'
404                  expr '<' '=' variable                      { DBG("predop : '(' group ',' variable ',' variable ')' expr '<' '=' \
405                                                                     variable ", @$);
406                                                               ASRT($$ = p->createIntervalPredicate($2, $4, $6, $8, $11, NULL), @$);  }
407                | '(' group ',' variable ',' variable ')'
408                  expr '>' '=' variable                      { DBG("predop : '(' group ',' variable ',' variable ')' expr '>' '=' \
409                                                                     variable ", @$);
410                                                               ASRT($$ = p->createIntervalPredicate($2, $4, $6, NULL, $11, $8), @$);  }
411                | expr '=' expr                              { DBG("predop : expr '=' expr", @$);
412                                                               ASRT($$ = CryptoComp::PredicateOp::createPredOp("=", $1, $3), @$);    }
413                | expr                                       { DBG("predop : expr", @$);
414                                                               ASRT($$ = CryptoComp::PredicateOp::createPredOp("=", NULL, $1), @$);  }
415  ;
```

```
416  optintcheck    : /* empty */                                  { DBG("optintcheck : /* empty */", @$);
417                                                                   ASRT($$ = new CryptoComp::IntChecks(), @$);                    }
418                 | ':' intchecklist                             { DBG("optintcheck : : intchecklist", @$);
419                                                                   ASRT($$ = $2, @$);                                             }
420  ;
421  intcheck       : KW_INTCHECK '(' variable ',' constnum')'     { DBG("intcheck : KW_INTCHECK '(' variable ',' constnum ')'", @$);
422                                                                   ASRT($$ = new CryptoComp::IntCheck($3, $5), @$);               }
423  ;
424  binary         : /* empty */                                  { DBG("binary : /* empty */", @$);
425                                                                   $$ = false;                                                    }
426                 | ',' KW_BINARY                                { DBG("binary : ',' KW_BINARY", @$);
427                                                                   $$ = true;                                                     }
428  ;
429
430
431  /* Check */
432  comparator     : '=' '='                                      { $$ = STR("==");                                              }
433                 | '!' '='                                      { $$ = STR("!=");                                              }
434                 | '>' '='                                      { $$ = STR(">=");                                              }
435                 | '<' '='                                      { $$ = STR("<=");                                              }
436  ;
437
438
439  /* identifier */
440  confparambnz   : IDENT                                        { DBG("confparambnz : IDENT", @$);
441                                                                   ASRTDESTRMSG($$ = p->getConfigParamBnz(*$1), @$,
442                                                                                ERR("Unknown ConfigParamBnz '%s'", $1->c_str()),
443                                                                   delete($1));                                                   }
444  ;
445  confparamstr   : IDENT                                        { DBG("confparamstr : IDENT", @$);
446                                                                   ASRTDESTRMSG($$ = p->getConfigParamString(*$1), @$,
447                                                                   ERR("Unknown ConfigParamString '%s'", $1->c_str()),
448                                                                   delete($1));                                                   }
449  ;
450  group          : IDENT                                        { DBG("group : IDENT", @$);
451                                                                   ASRTDESTRMSG($$ = p->getGroup(*$1), @$,
452                                                                                ERR("Unknown Group '%s'", $1->c_str()), delete($1)); }
453  ;
```

```
454 variable      : IDENT                          { DBG("variable : IDENT", @$);
455                                                  ASRTDESTRMSG($$ = p->getVar(*$1), @$,
456                                                               ERR("Unknown variable '%s'", $1->c_str()), delete($1));     }
457 ;
458 hom           : IDENT                          { DBG("hom : IDENT", @$);
459                                                  ASRTDESTRMSG($$ = p->getHom(*$1), @$,
460                                                               ERR("Unknown homomorphism '%s'", $1->c_str()), delete($1)); }
461 ;
462 sigma         : IDENT                          { DBG("sigma : IDENT", @$);
463                                                  ASRTDESTRMSG($$ = p->getSigma(*$1), @$,
464                                                               ERR("Unknown sigma '%s'", $1->c_str()), delete($1));        }
465 ;
466 pairing       : IDENT                          { DBG("pairing : IDENT", @$);
467                                                  ASRTDESTRMSG($$ = p->getPairing(*$1), @$,
468                                                               ERR("Unknown Pairing '%s'", $1->c_str()), delete($1));      }
469 ;
470
471
472 /* Lists */
473 blckparamlist : /* empty */                    { DBG("blckparamlist : /* empty */", @$);
474                                                  ASRT($$ = p->vectorNew<CryptoComp::BlockParam>(), @$);                   }
475               | blckparam                      { DBG("blckparamlist : blckparam", @$);
476                                                  ASRT($$ = p->vectorNew<CryptoComp::BlockParam>($1), @$);                 }
477               | blckparamlist ',' blckparam    { DBG("blckparamlist : blckparamlist ',' blckparam", @$);
478                                                  ASRT($$ = p->vectorAppend<CryptoComp::BlockParam>($1, $3), @$);          }
479 ;
480 numberlist    : number                         { DBG("numberlist : number", @$);
481                                                  ASRT($$ = p->vectorNew<std::string>($1), @$);                            }
482               | numberlist ',' number          { DBG("numberlist : numberlist ',' number", @$);
483                                                  ASRT($$ = p->vectorAppend<std::string>($1, $3), @$);                     }
484 ;
485 varinitlist   : varinit                        { DBG("varinitlist : varinit", @$);
486                                                  ASRT($$ = p->vectorNew<CryptoComp::Var>($1), @$);                        }
487               | varinitlist ',' varinit        { DBG("varinitlist : varinitlist ',' varinit", @$);
488                                                  ASRT($$ = p->vectorAppend<CryptoComp::Var>($1, $3), @$);                 }
489 ;
490 grouplist     : group                          { DBG("grouplist : group", @$);
491                                                  ASRT($$ = p->vectorNew<CryptoComp::Group>($1), @$);                      }
```

```
492                 | grouplist ',' group                       { DBG("grouplist : grouplist ',' group", @$);
493                                                               ASRT($$ = p->vectorAppend<CryptoComp::Group>($1, $3), @$);              }
494 ;
495 sigmalist       : sigma                                     { DBG("sigmalist : sigma", @$);
496                                                               ASRT($$ = p->vectorNew<CryptoComp::Sigma>($1), @$);                     }
497                 | sigmalist ',' sigma                       { DBG("sigmalist : sigmalist ',' sigma", @$);
498                                                               ASRT($$ = p->vectorAppend<CryptoComp::Sigma>($1, $3), @$);              }
499 ;
500 exprlist        : expr                                      { DBG("exprlist : expr", @$);
501                                                               ASRT($$ = p->createNewTupleExpression($1), @$);                         }
502                 | exprlist ',' expr                         { DBG("exprlist : exprlist ',' expr", @$);
503                                                               ASRT($$ = p->extendTupleExpression($1, $3), @$);                        }
504 ;
505 intchecklist    : intcheck                                  { DBG("intchecklist : intcheck", @$);
506                                                               CryptoComp::IntChecks *pIntChecks = new CryptoComp::IntChecks();
507                                                               ASRT($$ = pIntChecks->insert($1), @$);                                  }
508                 | intchecklist ',' intcheck                 { DBG("intchecklist : intchecklist ',' intcheck", @$);
509                                                               ASRT($$ = $1->insert($3), @$);                                          }
510 ;
511
512
513 /* IDENT     = string which begins with a-zA-Z
514  * POSNUMBER = positive decimal number
515  * NEGNUMBER = negative decimal number with leading '-'
516  * STRING    = all other strings
517  */
518 confparambnzin: IDENT                                       { $$ = $1;                                                              }
519                 | number                                    { $$ = $1;                                                              }
520 ;
521 confparamstrin: IDENT                                       { $$ = $1;                                                              }
522                 | QUOTED_STRING                             { $$ = $1;                                                              }
523 ;
524 confparamin     : IDENT                                     { $$ = $1;                                                              }
525                 | number                                    { $$ = $1;                                                              }
526                 | QUOTED_STRING                             { $$ = $1;                                                              }
527 ;
528 number          : POSNUMBER                                 { $$ = $1;                                                              }
529                 | NEGNUMBER                                 { $$ = $1;                                                              }
```

```
530 ;
531 constnum      : POSNUMBER                                { DBG("constnum : POSNUMBER", @$);
532                                                           $$ = atoi($1->c_str()); delete($1);                                  }
533                | KW_EXTRACT '(' confparambnzin ')'       { DBG("constnum : KW_EXTRACT '(' confparambnzin ')'", @$);
534                                                           ASRT($$ = p->getConstNumFromConfigParamBnz($3), @$);                 }
535 ;
536 constnumhash  : POSNUMBER                                { DBG("constnumhash : POSNUMBER", @$);
537                                                           ASRT($$ = new CryptoComp::ConstNumOrHash(atoi($1->c_str())), @$);
538                                                           delete($1);                                                          }
539                | hash                                    { DBG("constnumhash : hash", @$);
540                                                           ASRT($$ = new CryptoComp::ConstNumOrHash(*$1), @$); delete($1);      }
541                | KW_EXTRACT '(' confparamin ')'          { DBG("constnumhash : KW_EXTRACT '(' confparamin ')'", @$);
542                                                           ASRT($$ = p->getConstNumOrHashFromConfigParamBnz($3), @$);           }
543 ;
544 consthash     : hash                                     { DBG("consthash : hash", @$);
545                                                           ASRT($$ = $1, @$);                                                   }
546                | KW_EXTRACT '(' confparamstrin ')'       { DBG("consthash : KW_EXTRACT '(' confparamstrin ')'", @$);
547                                                           ASRT($$ = p->getHashFuncFromConfigParamStr($3), @$); delete($3);     }
548 ;
549 hash          : KW_HASH_MD4                              { DBG("hash : KW_HASH_MD4", @$);
550                                                           ASRT($$ = new hashFunctions_t(HASH_MD4), @$);                        }
551                | KW_HASH_MD5                             { DBG("hash : KW_HASH_MD5", @$);
552                                                           ASRT($$ = new hashFunctions_t(HASH_MD5), @$);                        }
553                | KW_HASH_SHA1                            { DBG("hash : KW_HASH_SHA1", @$);
554                                                           ASRT($$ = new hashFunctions_t(HASH_SHA1), @$);                       }
555                | KW_HASH_SHA224                          { DBG("hash : KW_HASH_SHA224", @$);
556                                                           ASRT($$ = new hashFunctions_t(HASH_SHA224), @$);                     }
557                | KW_HASH_SHA256                          { DBG("hash : KW_HASH_SHA256", @$);
558                                                           ASRT($$ = new hashFunctions_t(HASH_SHA256), @$);                     }
559                | KW_HASH_SHA384                          { DBG("hash : KW_HASH_SHA384", @$);
560                                                           ASRT($$ = new hashFunctions_t(HASH_SHA384), @$);                     }
561                | KW_HASH_SHA512                          { DBG("hash : KW_HASH_SHA512", @$);
562                                                           ASRT($$ = new hashFunctions_t(HASH_SHA512), @$);                     }
563 ;
```

Listing A.1: Grammatik der Eingabesprache des Crypto-Compilers in Bison.

A.2 „Ticket-Chain“ Protokoll

```
// bit length 1024 bit
ConfigParam ln := 1024;
ConfigParam lphi := 80;
ConfigParam lw := 80;       // tickets
ConfigParam lz := 390;      // lz > lw_tilde + 2 = lw + lphi + lc + 4 = 388
ConfigParam lz_hat := 80;   // lz_hat < lz - lphi - lc - 3 = 83
ConfigParam lv := 1490;     // lv = ln + lw_tilde + lphi = ln + lw + lphi + lc + 2 + lphi = 1490
ConfigParam lv_1 := 1104;   // lv_1 = ln + lphi = 1104
ConfigParam lr := 1104;     // lr = ln + lphi = 1104
ConfigParam lr_hat := 1104; // lr_hat = ln + lphi = 1104
ConfigParam lTS := 32;
ConfigParam hashfunc := ”SHA224”;


// RSA group
// bit length 1024 bit
ConfigParam n_rsa := 0x21d0f54e5f6cf6609c363b562cd2021a9a025e8a9ac9d86232edadf57d2212acfe5a0c03b1bd6ea856f2fbe3f0bf55194df35ec504d14a09f5
    ↳ cd08c2f13b121d16b0a04b17e00db44c989e1168ad2731023edb752efeefe896b53da911c8a4a43a290cfbe1c566cfb7b1eb2b2c50670e2c3d84fcd17500fc297d23
    ↳ dfd8399431;
ConfigParam p_rsa := 0x2f3b6674cce0facaf2011a2fc78f98cc8c721cd4933f9e727fa29059d84d039229999f13212f79fece4597ed991dd7a1dddf377fadc8010bff8
    ↳ 3848b33bf7987;
ConfigParam q_rsa := 0xb74926a1c3a65ef183a5fe5ad08dce28b5e17d593c3f166c376514928d4eaa770884ba0b052e3fb2760ad3b3e15b59f091014d637140f3ba84
    ↳ cf3c1b43e31287;
ConfigParam p_prime_rsa := 0x179db33a66707d6579008d17e3c7cc6646390e6a499fcf393fd1482cec2681c914cccf899097bcff6722cbf6cc8eebd0eeef9bbfd6e40
    ↳ 085ffc1c24599dfbcc3;
ConfigParam q_prime_rsa := 0x5ba49350e1d32f78c1d2ff2d6846e7145af0beac9e1f8b361bb28a4946a7553b84425d0582971fd93b0569d9f0adacf84880a6b1b8a07
    ↳ 9dd42679e0da1f18943;


GrpMulRSAQR := Zmul(n_rsa, p_rsa, q_rsa, p_prime_rsa, q_prime_rsa, safe prime, qr)[extract(ln)];
GrpMulRSAQRPhi := ZmulPhi(GrpMulRSAQR);

GrpMulRSAQR: S_rsa;
GrpMulRSAQR: Z_rsa;
GrpMulRSAQR: R_rsa_0;
```

```
GrpMulRSAQR: R_rsa_1;
GrpMulRSAQR: R_rsa_2;
GrpMulRSAQR: R_rsa_3;
GrpMulRSAQR: g_rsa;
GrpMulRSAQR: h_rsa;
GrpMulRSAQR: S_rsa_qr;
GrpMulRSAQR: Z_rsa_qr;
GrpMulRSAQR: R_rsa_qr_0;
GrpMulRSAQR: R_rsa_qr_1;
GrpMulRSAQR: R_rsa_qr_2;
GrpMulRSAQR: R_rsa_qr_3;
GrpMulRSAQR: g_rsa_qr;
GrpMulRSAQR: h_rsa_qr;

GrpMulRSAQRPhi: r_Z_rsa;
GrpMulRSAQRPhi: r_R_rsa_0;
GrpMulRSAQRPhi: r_R_rsa_1;
GrpMulRSAQRPhi: r_R_rsa_2;
GrpMulRSAQRPhi: r_R_rsa_3;
GrpMulRSAQRPhi: r_g_rsa;
GrpMulRSAQRPhi: r_h_rsa;


// Z
GrpZw := Z(lw)[extract(lw)];
GrpZz := Z(lz)[extract(lz)];
GrpZz_bar := Z(lz_hat)[extract(lz_hat)];
GrpZr := Z(lr)[extract(lr)];
GrpZr_hat := Z(lr_hat)[extract(lr_hat)];
GrpZv := Z(lv)[extract(lv)];
GrpZv_1 := Z(lv_1)[extract(lv_1)];
GrpZv_2 := Z(lv, exact)[extract(lv)];
GrpZTS := Z(lTS)[extract(lTS)];


// global variables
GrpZw: w;
```

```
68  GrpZv: v;
69  GrpZv_1: v_1;
70  GrpZv: v_hat;
71  GrpMulRSAQRPhi: d;
72  GrpMulRSAQR: A_hat_rsa_qr;
73  GrpZr_hat: r_hat;
74  GrpMulRSAQR: C_const_rsa_qr;
75  GrpZTS: TS2_hat_0;
76  GrpZTS: TS2_hat_1;
77  GrpZTS: TS2_hat_2;
78  GrpMulRSAQR: C_TS_hat_rsa_qr_0;
79  GrpMulRSAQR: C_TS_hat_rsa_qr_1;
80  GrpMulRSAQR: C_TS_hat_rsa_qr_2;
81  GrpZr: r_0;
82  GrpZr: r_1;
83  GrpZr: r_2;
84  GrpZr: p_0;
85  GrpZr: p_1;
86  GrpZr: p_2;
87
88
89
90  // ++++++++++++++++++ registration ++++++++++++++++++
91  __critical Registrierung_D1(out GrpMulRSAQR: U_rsa_qr)
92  {
93      w = ?GrpZw;
94      v_1 = ?GrpZv_1;
95
96      U_rsa_qr = S_rsa_qr^v_1 + R_rsa_qr_0^w;
97      Registrierung_SPK1 = SPK [ (w, v_1) : U_rsa_qr = S_rsa_qr^v_1 + R_rsa_qr_0^w, SigmaGsp, extract(lphi), extract(hashfunc) ];
98  };
99
100 __critical Registrierung_P1(out GrpMulRSAQR: A_rsa_qr, out GrpZv: v_2, in GrpMulRSAQR: U_rsa_qr,
101                             in GrpZTS: TS_from, in GrpZTS: TS_to, in GrpZTS: qual, in GrpMulRSAQRPhi: z)
102 {
103     v_2 = ?GrpZv;
104
105     d = z^(-1);
```

```
    A_rsa_qr = (Z_rsa_qr + U_rsa_qr^(-1) + S_rsa_qr^(-v_2) + R_rsa_qr_1^(-(TS_from^4)) + R_rsa_qr_2^(-TS_to) + R_rsa_qr_3^(-qual))^d;

    Registrierung_SPK2 = SPK [ (d) : A_rsa_qr = (Z_rsa_qr + U_rsa_qr^(-1) + S_rsa_qr^(-v_2) + R_rsa_qr_1^(-(TS_from^4)) +
                                     R_rsa_qr_2^(-TS_to) + R_rsa_qr_3^(-qual))^d, SigmaGsp, extract(lphi), extract(hashfunc) ];
};

__critical Registrierung_D2(in GrpMulRSAQR: A_rsa_qr, in GrpMulRSAQRPhi: z, in GrpZv: v_2,
                            in GrpZTS: TS_from, in GrpZTS: TS_to, in GrpZTS: qual)
{
    v = GrpZv(v_1) + GrpZv(v_2);

    GrpMulRSAQR: left_rsa_qr;
    GrpMulRSAQR: right_rsa_qr;

    left_rsa_qr = A_rsa_qr^z + S_rsa_qr^v + R_rsa_qr_0^w + R_rsa_qr_1^(TS_from^4) + R_rsa_qr_2^TS_to + R_rsa_qr_3^qual;
    right_rsa_qr = Z_rsa_qr;

    check(left_rsa_qr == right_rsa_qr);
};
// ++++++++++++++++++ registration ++++++++++++++++++


// ++++++++++++++++++  reporting   ++++++++++++++++++
__critical Reporting_D1(out GrpZz_bar: z_bar, in GrpMulRSAQR: A_rsa_qr, in GrpZz: expConst, in GrpZTS: TS, in GrpZTS: TS_to,
                        in GrpZTS: qual, in GrpZz: z, in GrpZTS: TS_hat_0, in GrpZTS: TS_hat_1, in GrpZTS: TS_hat_2)
{
    r_hat = ?GrpZr_hat;
    v_hat = GrpZv(v) + GrpZv(z)^r_hat;

    A_hat_rsa_qr = A_rsa_qr + S_rsa_qr^(-r_hat);
    z_bar = GrpZz_bar(z - expConst);

    C_const_rsa_qr = Z_rsa_qr + A_hat_rsa_qr^(-expConst) + R_rsa_qr_0^(-w) +
                     R_rsa_qr_1^(-(TS^4 + 1)) + R_rsa_qr_2^(-TS_to) + R_rsa_qr_3^(-qual);

    TS2_hat_0 = TS_hat_0^TS_hat_0;
    TS2_hat_1 = TS_hat_1^TS_hat_1;
```

```
144     TS2_hat_2 = TS_hat_2^TS_hat_2;
145
146     r_0 = ?GrpZr;
147     r_1 = ?GrpZr;
148     r_2 = ?GrpZr;
149     C_TS_hat_rsa_qr_0 = g_rsa_qr^TS_hat_0 + h_rsa_qr^r_0;
150     C_TS_hat_rsa_qr_1 = g_rsa_qr^TS_hat_1 + h_rsa_qr^r_1;
151     C_TS_hat_rsa_qr_2 = g_rsa_qr^TS_hat_2 + h_rsa_qr^r_2;
152     p_0 = r_0^TS_hat_0;
153     p_1 = r_1^TS_hat_1;
154     p_2 = r_2^TS_hat_2;
155
156     Reporting_SPK1 = SPK [ (z_bar, v_hat, TS_hat_0, TS2_hat_0, TS_hat_1, TS2_hat_1, TS_hat_2, TS2_hat_2, r_0, r_1, r_2, p_0, p_1, p_2) :
157                 C_const_rsa_qr = R_rsa_qr_1^(-TS2_hat_0 - TS2_hat_1 - TS2_hat_2) + A_hat_rsa_qr^(z_bar) + S_rsa_qr^v_hat &
158                             C_TS_hat_rsa_qr_0 = g_rsa_qr^TS_hat_0 + h_rsa_qr^r_0 &
159                             C_TS_hat_rsa_qr_1 = g_rsa_qr^TS_hat_1 + h_rsa_qr^r_1 &
160                             C_TS_hat_rsa_qr_2 = g_rsa_qr^TS_hat_2 + h_rsa_qr^r_2 &
161                             GrpMulRSAQR(1) = (C_TS_hat_rsa_qr_0)^TS_hat_0 + g_rsa_qr^(-TS2_hat_0) + h_rsa_qr^(-(p_0)) &
162                             GrpMulRSAQR(1) = (C_TS_hat_rsa_qr_1)^TS_hat_1 + g_rsa_qr^(-TS2_hat_1) + h_rsa_qr^(-(p_1)) &
163                             GrpMulRSAQR(1) = (C_TS_hat_rsa_qr_2)^TS_hat_2 + g_rsa_qr^(-TS2_hat_2) + h_rsa_qr^(-(p_2)) :
164                             intcheck(z_bar, extract(lz_hat)), SigmaGsp, extract(lphi), extract(hashfunc) ];
165 };
166 // ++++++++++++++++++  reporting   ++++++++++++++++++
167
168
169
170 // ++++++++++++++++++     init     ++++++++++++++++++
171 init()
172 {
173     setConfigParam(n_rsa, p_rsa, q_rsa, p_prime_rsa, q_prime_rsa, ln, safe prime, RSA);
174
175     print(n_rsa, hex);
176     print(p_rsa, hex);
177     print(q_rsa, hex);
178     print(p_prime_rsa, hex);
179     print(q_prime_rsa, hex);
180 };
181
```

```
182  initBases()
183  {
184      r_Z_rsa   = ?GrpMulRSAQRPhi;
185      r_R_rsa_0 = ?GrpMulRSAQRPhi;
186      r_R_rsa_1 = ?GrpMulRSAQRPhi;
187      r_R_rsa_2 = ?GrpMulRSAQRPhi;
188      r_R_rsa_3 = ?GrpMulRSAQRPhi;
189      r_g_rsa   = ?GrpMulRSAQRPhi;
190      r_h_rsa   = ?GrpMulRSAQRPhi;
191
192      S_rsa    = gen(GrpMulRSAQR);
193      Z_rsa    = S_rsa^r_Z_rsa;
194      R_rsa_0  = S_rsa^r_R_rsa_0;
195      R_rsa_1  = S_rsa^r_R_rsa_1;
196      R_rsa_2  = S_rsa^r_R_rsa_2;
197      R_rsa_3  = S_rsa^r_R_rsa_3;
198      g_rsa    = S_rsa^r_g_rsa;
199      h_rsa    = S_rsa^r_h_rsa;
200
201      // both provider and DCUs square bases so that DCUs can be assured that
202      // we are working in QR
203      S_rsa_qr   = S_rsa^2;
204      Z_rsa_qr   = Z_rsa^2;
205      R_rsa_qr_0 = R_rsa_0^2;
206      R_rsa_qr_1 = R_rsa_1^2;
207      R_rsa_qr_2 = R_rsa_2^2;
208      R_rsa_qr_3 = R_rsa_3^2;
209      g_rsa_qr   = g_rsa^2;
210      h_rsa_qr   = h_rsa^2;
211
212      // SPK to show that Z_rsa = <S_rsa>
213      // only for benchmarking!
214      SPK_Z_rsa   = SPK [ (r_Z_rsa, r_R_rsa_0, r_R_rsa_1, r_R_rsa_2, r_R_rsa_3, r_g_rsa, r_h_rsa) :
215                                          Z_rsa_qr   = S_rsa_qr^r_Z_rsa &
216                                          R_rsa_qr_0 = S_rsa_qr^r_R_rsa_0 &
217                                          R_rsa_qr_1 = S_rsa_qr^r_R_rsa_1 &
218                                          R_rsa_qr_2 = S_rsa_qr^r_R_rsa_2 &
219                                          R_rsa_qr_3 = S_rsa_qr^r_R_rsa_3 &
```

```
220                                     g_rsa_qr   = S_rsa_qr^r_g_rsa &
221                                     h_rsa_qr   = S_rsa_qr^r_h_rsa, SigmaPhi, extract(hashfunc), binary ];
222
223       print(S_rsa, hex);
224       print(Z_rsa, hex);
225       print(R_rsa_0, hex);
226       print(R_rsa_1, hex);
227       print(R_rsa_2, hex);
228       print(R_rsa_3, hex);
229       print(g_rsa, hex);
230       print(h_rsa, hex);
231   };
232   // ++++++++++++++++++     init     ++++++++++++++++++
```

Listing A.2: Beschreibung des „Ticket-Chain“ Protokolls in der Eingabesprache des Crypto-Compilers.

A.3 „Ticket-Spender“ Protokoll

```
// bit length 1024 bit
ConfigParam ln := 1024;
ConfigParam lp_hat := 1024;
ConfigParam lq_hat := 160;
ConfigParam lphi := 80;
ConfigParam lw := 80;       // tickets
ConfigParam lz := 390;      // lz > lw_tilde + 2 = lw + lphi + lc + 4 = 388
ConfigParam lz_hat := 80;   // lz_hat < lz - lphi - lc - 3 = 83
ConfigParam lv := 1490;     // lv = ln + lw_tilde + lphi = ln + lw + lphi + lc + 2 + lphi = 1490
ConfigParam lv_1 := 1104;   // lv_1 = ln + lphi = 1104
ConfigParam lr := 1104;     // lr = ln + lphi = 1104
ConfigParam lr_hat := 1104; // lr_hat = ln + lphi = 1104
ConfigParam lTS := 32;
ConfigParam hashfunc := ”SHA224”;


// RSA group
// bit length 1024 bit
ConfigParam n_rsa := 0x21d0f54e5f6cf6609c363b562cd2021a9a025e8a9ac9d86232edadf57d2212acfe5a0c03b1bd6ea856f2fbe3f0bf55194df35ec504d14a09f5
    ↳ cd08c2f13b121d16b0a04b17e00db44c989e1168ad2731023edb752efeefe896b53da911c8a4a43a290cfbe1c566cfb7b1eb2b2c50670e2c3d84fcd17500fc297d23
    ↳ dfd8399431;
ConfigParam p_rsa := 0x2f3b6674cce0facaf2011a2fc78f98cc8c721cd4933f9e727fa29059d84d039229999f13212f79fece4597ed991dd7a1dddf377fadc8010bff8
    ↳ 3848b33bf7987;
ConfigParam q_rsa := 0xb74926a1c3a65ef183a5fe5ad08dce28b5e17d593c3f166c376514928d4eaa770884ba0b052e3fb2760ad3b3e15b59f091014d637140f3ba84
    ↳ cf3c1b43e31287;
ConfigParam p_prime_rsa := 0x179db33a66707d6579008d17e3c7cc6646390e6a499fcf393fd1482cec2681c914cccf899097bcff6722cbf6cc8eebd0eeef9bbfd6e40
    ↳ 085ffc1c24599dfbcc3;
ConfigParam q_prime_rsa := 0x5ba49350e1d32f78c1d2ff2d6846e7145af0beac9e1f8b361bb28a4946a7553b84425d0582971fd93b0569d9f0adacf84880a6b1b8a07
    ↳ 9dd42679e0da1f18943;


GrpMulRSAQR := Zmul(n_rsa, p_rsa, q_rsa, p_prime_rsa, q_prime_rsa, safe prime, qr)[extract(ln)];
GrpMulRSAQRPhi := ZmulPhi(GrpMulRSAQR);

GrpMulRSAQR: S_rsa;
```

```
30  GrpMulRSAQR: Z_rsa;
31  GrpMulRSAQR: R_rsa_0;
32  GrpMulRSAQR: R_rsa_1;
33  GrpMulRSAQR: S_rsa_qr;
34  GrpMulRSAQR: Z_rsa_qr;
35  GrpMulRSAQR: R_rsa_qr_0;
36  GrpMulRSAQR: R_rsa_qr_1;
37
38  GrpMulRSAQRPhi: r_Z_rsa;
39  GrpMulRSAQRPhi: r_R_rsa_0;
40  GrpMulRSAQRPhi: r_R_rsa_1;
41
42
43  // Z*_p (Schnorr group)
44  // bit length 1024 bit
45  ConfigParam p_hat := 0x8022ae0d3c9b94b5331a67fa8d1b624fd540f7853480d071920597e814c49f2fbb4fe13dd79ca0222f0f216d7677fab1d5605993d10c0e76188
       ↳ 827a4f599d9eb5252258d27f90fea6bb179164973d2fdb3513f49ec59076beccc62cdcca6e068855d945c261c0c36bf35adaaade57c7883da5c4e3e9506987f81483
       ↳ 05ea92d19;
46  ConfigParam q_hat := 0xedc0cfc2b53a6a26bd024fcaafe4fcb6fe503053;
47
48  GrpMulphat := Zmul (p_hat, q_hat, Schnorr)[extract(lp_hat)];
49  GrpMulqhat := Zmul (q_hat, prime)[extract(lq_hat)];
50  GrpAdd := Zadd (q_hat)[extract(lq_hat)];
51
52  GrpMulphat: g;
53  GrpMulphat: h;
54
55
56  // Z
57  GrpZ := Z()[extract(ln)];
58  GrpZs := Z(0, q_hat)[extract(lp_hat)];
59  GrpZz := Z(lz)[extract(lz)];
60  GrpZz_bar := Z(lz_hat)[extract(lz_hat)];
61  GrpZr := Z(lr)[extract(lr)];
62  GrpZr_hat := Z(lr_hat)[extract(lr_hat)];
63  GrpZv := Z(lv)[extract(lv)];
64  GrpZv_1 := Z(lv_1)[extract(lv_1)];
65  GrpZv_2 := Z(lv, exact)[extract(lv)];
```

```
66  GrpZTS := Z(lTS)[extract(lTS)];
67
68
69
70  // global variables
71  GrpZs: s;
72  GrpZs: s_1;
73  GrpZv: v;
74  GrpZv_1: v_1;
75  GrpZv: v_hat;
76  GrpMulRSAQRPhi: d;
77  GrpMulRSAQR: A_hat_rsa_qr;
78  GrpZr_hat: r_hat;
79  GrpMulRSAQR: C_const_rsa_qr;
80  GrpAdd: a;
81  GrpAdd: r_s;
82  GrpAdd: p_s;
83  GrpMulphat: C_s;
84  GrpMulphat: w;
85
86
87  // ++++++++++++++++++ registration ++++++++++++++++++
88  __critical Registrierung_D1(out GrpMulRSAQR: U_rsa_qr)
89  {
90      s_1 = ?GrpZs;
91      v_1 = ?GrpZv_1;
92
93      U_rsa_qr = S_rsa_qr^v_1 + R_rsa_qr_0^s_1;
94      Registrierung_SPK1 = SPK [ (s_1, v_1) : U_rsa_qr = S_rsa_qr^v_1 + R_rsa_qr_0^s_1, SigmaGsp, extract(lphi), extract(hashfunc) ];
95  };
96
97  __critical Registrierung_P1(out GrpMulRSAQR: A_rsa_qr, out GrpZs: s_2, out GrpZv: v_2, in GrpMulRSAQR: U_rsa_qr, in GrpZTS: TS_to, in
        ↳ GrpMulRSAQRPhi: z)
98  {
99      s_2 = ?GrpZs;
100     v_2 = ?GrpZv;
101
102     d = z^(-1);
```

```
103      A_rsa_qr = (Z_rsa_qr + U_rsa_qr^(-1) + S_rsa_qr^(-v_2) + R_rsa_qr_0^(-s_2) + R_rsa_qr_1^(-TS_to))^d;
104
105      Registrierung_SPK2 = SPK [ (d) : A_rsa_qr = (Z_rsa_qr + U_rsa_qr^(-1) + S_rsa_qr^(-v_2) + R_rsa_qr_0^(-s_2) + R_rsa_qr_1^(-TS_to))^d,
106                                       SigmaGsp, extract(lphi), extract(hashfunc) ];
107  };
108
109  __critical Registrierung_D2(in GrpMulRSAQR: A_rsa_qr, in GrpZ: z, in GrpZs: s_2, in GrpZv: v_2, in GrpZTS: TS_to)
110  {
111      s = s_1 + s_2;
112      v = GrpZv(v_1) + GrpZv(v_2);
113
114      GrpMulRSAQR: left_rsa_qr;
115      GrpMulRSAQR: right_rsa_qr;
116
117      left_rsa_qr = A_rsa_qr^z + S_rsa_qr^v + R_rsa_qr_0^s + R_rsa_qr_1^TS_to;
118      right_rsa_qr = Z_rsa_qr;
119
120      check(left_rsa_qr == right_rsa_qr);
121  };
122  // ++++++++++++++++++ registration ++++++++++++++++++
123
124
125  // ++++++++++++++++++  reporting   ++++++++++++++++++
126  __critical Reporting_D1(out GrpZz_bar: z_bar, in GrpMulRSAQR: A_rsa_qr, in GrpZz: expConst, in GrpZTS: TS, in GrpZTS: TS_to, in GrpZz: z)
127  {
128      r_hat = ?GrpZr_hat;
129      v_hat = GrpZv(v) + GrpZv(z)^r_hat;
130
131      A_hat_rsa_qr = A_rsa_qr + S_rsa_qr^(-r_hat);
132      z_bar = GrpZz_bar(z - expConst);
133
134      C_const_rsa_qr = Z_rsa_qr + A_hat_rsa_qr^(-expConst) + R_rsa_qr_1^(-TS_to);
135
136      a = GrpAdd(GrpMulqhat(GrpZ(s) + GrpZ(TS))^(-1));
137
138      r_s = ?GrpAdd;
139
140      p_s = a^r_s;
```

```
141
142     w = g^a;
143     C_s = g^s + h^r_s;
144
145     Reporting_SPK1 = SPK [ (s, z_bar, v_hat, a, r_s, p_s) : C_const_rsa_qr = R_rsa_qr_0^s + A_hat_rsa_qr^(z_bar) + S_rsa_qr^v_hat &
146                            w = g^a & g = (C_s + g^TS)^a + h^(-p_s) & C_s = g^s + h^r_s : intcheck(z_bar, extract(lz_hat)),
147                            SigmaGsp, extract(lphi), extract(hashfunc) ];
148 };
149 // ++++++++++++++++++  reporting   ++++++++++++++++++
150
151
152
153 // ++++++++++++++++++     init     ++++++++++++++++++
154 init()
155 {
156     setConfigParam(n_rsa, p_rsa, q_rsa, p_prime_rsa, q_prime_rsa, ln, safe prime, RSA);
157
158     print(n_rsa, hex);
159     print(p_rsa, hex);
160     print(q_rsa, hex);
161     print(p_prime_rsa, hex);
162     print(q_prime_rsa, hex);
163
164     setConfigParam(p_hat, q_hat, lp_hat, lq_hat, Schnorr);
165
166     print(p_hat, hex);
167     print(q_hat, hex);
168 };
169
170 initBases()
171 {
172     r_Z_rsa   = ?GrpMulRSAQRPhi;
173     r_R_rsa_0 = ?GrpMulRSAQRPhi;
174     r_R_rsa_1 = ?GrpMulRSAQRPhi;
175
176     S_rsa   = gen(GrpMulRSAQR);
177     Z_rsa   = S_rsa^r_Z_rsa;
178     R_rsa_0 = S_rsa^r_R_rsa_0;
```

```
179       R_rsa_1 = S_rsa^r_R_rsa_1;
180
181       // both provider and DCUs square bases so that DCUs can be assured that
182       // we are working in QR
183       S_rsa_qr   = S_rsa^2;
184       Z_rsa_qr   = Z_rsa^2;
185       R_rsa_qr_0 = R_rsa_0^2;
186       R_rsa_qr_1 = R_rsa_1^2;
187
188       // SPK to show that Z_rsa = <S_rsa>
189       // only for benchmarking!
190       SPK_Z_rsa   = SPK [ (r_Z_rsa, r_R_rsa_0, r_R_rsa_1) :
191                                       Z_rsa_qr = S_rsa_qr^r_Z_rsa &
192                                       R_rsa_qr_0 = S_rsa_qr^r_R_rsa_0 &
193                                       R_rsa_qr_1 = S_rsa_qr^r_R_rsa_1, SigmaPhi, extract(hashfunc), binary ];
194       print(S_rsa, hex);
195       print(Z_rsa, hex);
196       print(R_rsa_0, hex);
197       print(R_rsa_1, hex);
198
199       g = gen(GrpMulphat);
200       h = gen(GrpMulphat);
201
202       print(g, hex);
203       print(h, hex);
204   };
205   // ++++++++++++++++++     init     ++++++++++++++++++
```

Listing A.3: Beschreibung des „Ticket-Spender“ Protokolls in der Eingabesprache des Crypto-Compilers.

Abkürzungsverzeichnis

ACS	Anonymous Credential System
AES	Advanced Encryption Standard
ANSI	American National Standards Institute
ANTLR	ANother Tool for Language Recognition
ASCII	American Standard Code for Information Interchange
CA	Certification Authority
CACE	Computer Aided Cryptography Engineering
CallMI	Call Macro-Interface
CC	Control Center
CCR	Collected Code Representative
CCW	Collected Code Wrapper
CDH	Computational-Diffie-Hellman
CFL	Context Free Language
CIOS	Coarsely Integrated Operand Scanning
CL	Camenisch und Lysyanskaya
CMF	Code Management Framework
CPU	Central Processing Unit
CS	Client System
CSL	Context Sensitive Language
CSN	Camenisch-Stadler Notation
CUDA	Compute Unified Device Architecture
CYK	Cocke-Younger-Kasami

DCU	Data Collecting Unit
DDH	Decisional-Diffie-Hellman
DefMI	Definition Macro-Interface
DLP	Diskreter-Logarithmus-Problem
DRAM	Dynamic Random Access Memory
DSL	Digital Subscriber Line
EBNF	Erweiterte Backus-Naur-Form
ECC	Elliptic Curve Cryptography
ECDSA	Elliptic Curve Digital Signature Algorithm
EVU	Energieversorgungsunternehmen
FCD	Floating Car Data
FIOS	Finely Integrated Operand Scanning
FM	Frequenzmodulation
GCC	GNU Compiler Collection
GPGPU	General Purpose Computing on Graphics Processing Unit
GPRS	General Packet Radio Service
GPS	Global Positioning System
GPU	Graphics Processing Unit
HVZK	Honest-Verifier Zero-Knowledge
ID	Identifikator
IP	Internet Protocol
KA	Key Aggregator
LBS	Location-Based Services
LUT	Lookup Tabelle
MAC	Message Authentication Code
MASF	Mobile Application Sensing Framework
MI	Macro-Interface
MPIR	Multiple Precision Integers and Rationals
MRC	Mixed Radix Conversion
MUC	Multi-Utility-Communication

OBU	On-Board Unit
PBC	Pairing Based Cryptography
PC	Personal Computer
PIL	Protocol Implementation Language
PK	Proof of Knowledge
PLC	Powerline Communication
PRF	Pseudo-Random-Function Family
PSL	Protocol Specification Language
PTX	Parallel Thread Execution
PVT	Protokoll Verification Toolbox
RAD	Rapid Application Development
RSA	Rivest, Shamir und Adleman
SC	Service Center
SDDHI	Strong Decision-Diffie-Hellman Inversion
SHA-2	Secure Hash Algorithm 2
SID	Sample-ID
SIMD	Single Instruction, Multiple Data
SIMT	Single Instruction, Multiple Threads
SM	Smart Meter
SMP	Streaming Multiprocessor
SMS	Short Message Service
SOS	Separated Operand Scanning
SPK	Signature Proof of Knowledge
SRC	Single Radix Conversion
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TID	Trip-ID
TLS	Transport Layer Security
TMC	Traffic Message Channel
TTP	Trusted Third Party

VHDL	Very High Speed Integrated Circuit Hardware Description Language
VNB	Verteilnetzbetreiber
VRF	Verifiable Random Function
ZKPDL	Zero-Knowledge Proof Description Language
ZPK	Zero-Knowledge Proof of Knowledge

Symbolverzeichnis

Γ	(Terminal-)Alphabet
α	String von Symbolen
β	String von Symbolen
$\emptyset$	Regulärer Ausdruck, der eine leere Sprache beschreibt
ϵ	Leerer String
η	Wahrscheinlichkeit einen Verifier von der Kenntnis eines Geheimnisses zu überzeugen
γ	Exponent in der Dispenser Funktion des „Ticket-Spender“ Protokolls
$\varkappa$	Knowledge Error in Proof of Knowledges
φ	Eulersche Phi-Funktion
ϕ	Homomorphismus
$\mathbf{A}$	Teil einer CL-Signatur
$\mathcal{A}$	Angreifer
$\mathtt{A}$	Nicht-Terminalsymbol
A	Summand im Kogge-Stone Addierer
a	Ganze Zahl
$\mathtt{a}$	Symbol
$\overline{a}$	Montgomery-Darstellung der ganzen Zahl a
$\hat{\mathbf{A}}$	Verschleierung von $\mathbf{A}$
$\mathtt{B}$	Nicht-Terminalsymbol
B	Summand im Kogge-Stone Addierer
b	Ganze Zahl

$\mathtt{b}$	Symbol
$\mathtt{b}$	Basis (Verfahren von Brickell und Gordon)
$\overline{b}$	Montgomery-Darstellung der ganzen Zahl b
BL	Blacklist
$\mathtt{C}$	Algorithmus, der während eines Σ-Protokolls vom Verifier zur Bestimmung der Challenge ausgeführt wird
$\mathcal{C}$	Menge aus der die Challenges in einem Σ-Protokoll gewählt werden
C	Commitment
$\mathtt{C}$	Carry bei Bignum-Operationen
c	Challenge in Σ-Protokollen
$\mathtt{c}$	Multiplikatives Inverse von $\mathtt{Q}$
C_t	„Commitment“ in Σ-Protokollen
D	Daten, die eine DCU an den Provider überträgt
$\mathcal{D}$	DCU
d	Ganze Zahl
E	Verschlüsselungsoperation
e	Neutrales Gruppenelement
$\mathtt{e}$	Ganze Zahl (Exponent)
f	Dispenser Funktion
G	Gruppe
$\mathtt{G}$	Grammatik
$\mathtt{G}$	„Generate“-Bit im Kogge-Stone Addierer
g	Gruppenelement
$\overline{\mathtt{g}}$	Montgomery-Darstellung von g
$\mathbf{g}$	Generatorelement einer RSA-Gruppe
H	Gruppe
$\mathtt{H}$	Hashfunktion
h	Gruppenelement
$\mathbf{h}$	Generatorelement einer RSA-Gruppe

i	Indexvariable
j	Indexvariable
K	Schlüssel in einem symmetrischen Kryptosystem
k	„Blinding-Wert“, um Verbrauchsdaten in bestimmten Smart Metering Protokollen zu verschleiern
k	Lookahead
k	Koeffizienten (Verfahren von Brickell und Gordon)
$\tilde{k}$	„Blinding-Wert“, um den MAC im Protokoll von Vetter zu verschleiern
$\tilde{K}_{pub}$	Öffentlicher Schlüssel in einem asymmetrischen Kryptosystem
$\tilde{K}_{sec}$	Geheimer Schlüssel in einem asymmetrischen Kryptosystem
L	Sprache
ℓ	Bitlänge
ℓ_ϕ	Sicherheitsparameter
$\tilde{\ell}$	Erweiterte Bitlänge für implizite Intervall-Proofs
LUT_DEPTH	Anzahl der Bits eines Segments bei der LUT-optimierten Montgomery-Exponentiation
LUT_SIZE	Speicherbedarf einer LUT bei der LUT-optimierten Montgomery-Exponentiation
LZ_{GPU}	Anzahl der Operationen auf der GPU (mit Branch Divergence)
LZ_{ideal}	Anzahl der Operationen auf der GPU (ohne Branch Divergence)
M	Menge mit „Multiplizierern“ (Verfahren von Brickell und Gordon)
$\mathcal{M}$	Knowledge Extractor
M	Pseudo-Zufallsfunktion
m	Variable im Montgomery-Multiplikations-Algorithmus
m	Nachricht
$\hat{m}$	Dispenser-Modul
N	Gesamtanzahl von etwas
n	Länge eines geparsten Strings
n	Ganze Zahl, Modulus

NO_BITS	Operandengröße in Bits
NO_WORDS	Anzahl der Wörter einer ganzen Zahl
$\mathcal{O}$	Zufallsorakel in nicht-interaktiven Σ-Protokolls
P	Menge mit Produktionsregeln
P	„Propagate“-Bit im Kogge-Stone Addierer
$\mathcal{P}$	Provider
p	„Multiplizierer“ (Verfahren von Brickell und Gordon)
p	Primzahl
$\hat{\mathsf{P}}$	Algorithmus, der während eines Σ-Protokolls vom Prover ausgeführt wird
$\hat{\mathcal{P}}$	Prover in einem Σ-Protokoll
$\hat{p}$	Primzahl
$\tilde{p}$	Beliebiges Polynom
$\tilde{\mathsf{P}}$	Algorithmus, den der Prover in einem nicht-interaktiven Σ-Protokoll zur Signierung ausführt
Q	Produkt von Primzahlen im SRC Algorithmus
q	Primzahl
$\hat{q}$	Primzahl
R	Generatorelement einer RSA-Gruppe im CL-Verfahren
R	Relation
R	Algorithmus, den der Verifier in einem nicht-interaktiven Σ-Protokoll ausführt, um das Commitment zu berechnen
r	Regulärer Ausdruck
r	Ganze Zahl mit $\mathsf{r} \geq 2^{\ell_n}$
r	Zufallszahl
$\hat{r}$	„Blinding-Wert“, um **A** zu verschleiern
S	Generatorelement einer RSA-Gruppe im CL-Verfahren
S	Startsymbol
S	Summe bei Bignum-Operationen
$\mathcal{S}$	Signierer
S	Signieroperation

$\mathtt{s}$	Regulärer Ausdruck
s	Response in Σ-Protokollen
$\hat{S}$	Signatur
state	Zustandsvariable des Provers in Σ-Protokollen
$\tilde{s}$	Tupel mit Responses im Σ^{OR}-Protokoll
$\mathtt{T}$	Algorithmus, den ein Simulator während der Simulation eines Zero-Knowledge Proof of Knowledge ausführt
$\mathcal{T}$	Simulator, der einen Prover in einem Zero-Knowledge Proof of Knowledge simuliert
$\mathtt{t}$	Temporärer Speicher zur Berechnung des Montgomery-Produktes
t	Zufallswert in Σ-Protokollen
$\bar{\mathtt{t}}$	Montgomery-Darstellung von $\mathtt{t}$
TS	Zeitstempel
$\overleftarrow{TS}$	Zeitstempel (Gültigkeitsbeginn eines Tickets)
$\hat{TS}$	Differenz von Zeitstempeln
$\overrightarrow{TS}$	Zeitstempel (Ablauf eines Tickets / Ticket-Dispensers)
$\mathbf{U}$	Damgård-Fujisaki-Commitment
U	Untergruppe
$\mathcal{U}$	Benutzer
$\mathtt{u}$	Anzahl der Segmente in die ein Exponent bei der LUT-optimierten Montgomery-Exponentiation unterteilt wird
u	Qualitätsparameter im „Ticket-Chain“ Protokoll
$\mathtt{V}$	Menge von Nicht-Terminalsymbolen
v	Teil einer CL-Signatur
$\hat{\mathtt{V}}$	Algorithmus, der während eines Σ-Protokolls vom Verifier ausgeführt wird
$\hat{\mathcal{V}}$	Verifier in einem Σ-Protokoll
$\hat{v}$	Transformation von v für Zero-Knowledge Proof
$\tilde{v}$	Notwendig zur Berechnung von v_2

$\tilde{\mathtt{V}}$	Algorithmus, den der Verifier in einem nicht-interaktiven Σ-Protokolls zur Verifikation der Signatur ausgeführt
W	Gruppe
$\mathcal{W}$	Menge mit Geheimnissen
w	Geheimnis oder Ticket im „Ticket-Chain“ und „Ticket-Spender“ Protokoll
`WORDSIZE`	Wortbreite
$\tilde{\mathcal{W}}$	Menge der Komponenten eines Geheimnistupels
$\tilde{w}$	Tupel mit Geheimnissen
x	String
x	Ganze Zahl
x	Öffentlicher Wert in Σ-Protokollen
$\bar{\mathtt{x}}$	Montgomery-Darstellung von x
$\mathbf{Z}$	Generatorelement einer RSA-Gruppe im CL-Verfahren
z	Teil einer CL-Signatur
$\hat{z}$	Transformation von z

Literatur

[1] Divesh Aggarwal und Ueli Maurer. „Breaking RSA Generically Is Equivalent to Factoring“. *Proceedings of the 28th Annual International Conference on Advances in Cryptology: The Theory and Applications of Cryptographic Techniques*. EUROCRYPT '09. Cologne, Germany: Springer-Verlag, 2009, S. 36–53.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi und Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[3] Joseph A. Akinyele, Matthew D. Green und Avi D. Rubin. *Charm: A Framework for Rapidly Prototyping Cryptosystems*. Cryptology ePrint Archive, Report 2011/617. http://eprint.iacr.org/. 2011.

[4] José Bacelar Almeida, Endre Bangerter, Manuel Barbosa, Stephan Krenn, Ahmad-Reza Sadeghi und Thomas Schneider. „A certifying compiler for zero-knowledge proofs of knowledge based on Sigma-protocols“. *Proceedings of the 15th European conference on Research in computer security*. ESORICS'10. Athens, Greece: Springer-Verlag, 2010, S. 151–167.

[5] American National Standards Institute und Computer and Business Equipment Manufacturers Association and Secretariat. *American National Standard for information systems: programming language: C: ANSI X3.159-1989*. 14. Dez. 1989.

[6] Julia Angwin und Jennifer Valentino-Devries. *Apple, Google Collect User Data*. Apr. 2011. URL: http://online.wsj.com/article/SB10001424052748703983704576277101723453610.html.

[7] Endre Bangerter, Stefania Barzan, Stephan Krenn, Ahmad-Reza Sadeghi und Thomas Schneider. „Bringing zero-knowledge proofs of knowledge to practice“. *In 17th International Workshop on Security Protocols*. 2009.

[8] Endre Bangerter, Thomas Briner, Wilko Henecka, Stephan Krenn, Ahmad-Reza Sadeghi und Thomas Schneider. „Automatic Generation of Sigma-Protocols“. *Public Key Infrastructures, Services and Applications*. Hrsg. von Fabio Martinelli und Bart Preneel. Bd. 6391. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, S. 67–82.

[9] Endre Bangerter, Essam Ghadafi, Stephan Krenn, Ahmad-Reza Sadeghi, Thomas Schneider, Nigel P. Smart, Joe-Kai Tsay und Bogdan Warinschi. *Final Report on Unified Theoretical Framework of Efficient Zero-Knowledge Proofs of Knowledge*. Technischer Bericht. Technikon Forschungs- und Planungsgesellschaft GmbH, 2009.

[10] Endre Bangerter, Stephan Krenn, Ahmad-Reza Sadeghi, Thomas Schneider und Joe-Kai Tsay. „On the Design and Implementation of Efficient Zero-Knowledge Proofs of Knowledge". *ECRYPT workshop on Software Performance Enhancements for Encryption and Decryption and Cryptographic Compilers*. Okt. 2009.

[11] Endre Bangerter, Stephan Krenn, Matrial Seifriz und Ulrich Ultes-Nitsche. „cPLC - A Cryptographic Programming Language and Compiler". *ISSA*. Hrsg. von Hein S. Venter, Marijke Coetzee und Marianne Loock. ISSA, Pretoria, South Africa, 2011.

[12] Andrea Barisani und Daniele Bianco. „Injecting RDS-TMC Traffic Information Signals". *TELEMOBILITY 2007*. Nov. 2007.

[13] Mihir Bellare und Oded Goldreich. „On Defining Proofs of Knowledge". *Proceedings of the 12th Annual International Cryptology Conference on Advances in Cryptology*. CRYPTO '92. London, UK, UK: Springer-Verlag, 1993, S. 390–420.

[14] Ben Lynn. *PBC Library - The Pairing-Based Cryptography Library, Version 0.5.14*. 2013. URL: https://crypto.stanford.edu/pbc/.

[15] Berg Insight. *Worldwide installed base of smart electricity meters will reach 302.5 million units in 2015*. Aug. 2010. URL: http://www.berginsight.com/News.aspx?m%5C_m=6%5C&s%5C_m=1.

[16] L.T. Berger und K. Iniewski. *Smart Grid Applications, Communications, and Security*. Wiley, 2012. URL: https://encrypted.google.com/books?id=HPEbtwAACAAJ.

[17] A. Beutelspacher, J. Schwenk und K.D. Wolfenstetter. *Moderne Verfahren der Kryptographie: Von RSA zu Zero-Knowledge*. Vieweg+Teubner Verlag, 2010.

[18] K. Beuth. *Elektronik 4. Digitaltechnik*. Vogel Fachbuch. Vogel Business Media, 2006.

[19] Jens-Matthias Bohli, Christoph Sorge und Osman Ugus. „A Privacy Model for Smart Metering". *Proceedings of the First IEEE International Workshop on Smart Grid Communications (in conjunction with IEEE ICC 2010)*. 2010.

[20] Dan Boneh. „The Decision Diffie-Hellman Problem". *Proceedings of the Third International Symposium on Algorithmic Number Theory*. ANTS-III. London, UK, UK: Springer-Verlag, 1998, S. 48–63.

[21] Dan Boneh und Ramarathnam Venkatesan. „Breaking RSA May Not Be Equivalent to Factoring“ *EUROCRYPT*. Hrsg. von Kaisa Nyberg. Bd. 1403. Lecture Notes in Computer Science. Springer, 1998, S. 59–71.

[22] Mark G. J. van den Brand, Jeroen Scheerder, Jurgen J. Vinju und Eelco Visser. „Disambiguation Filters for Scannerless Generalized LR Parsers“. *Proceedings of the 11th International Conference on Compiler Construction*. CC '02. London, UK, UK: Springer-Verlag, 2002, S. 143–158.

[23] Ernest F. Brickell, Daniel M. Gordon, Kevin S. McCurley und David B. Wilson. „Fast exponentiation with precomputation“. *Proceedings of the 11th annual international conference on Theory and application of cryptographic techniques*. EUROCRYPT '92. Balatonfüred, Hungary: Springer-Verlag, 1993, S. 200–207.

[24] Ernie Brickell, Jan Camenisch und Liqun Chen. „Direct anonymous attestation“. *Proceedings of the 11th ACM conference on Computer and communications security*. CCS '04. Washington DC, USA: ACM, 2004, S. 132–145.

[25] David Brumley und Dan Boneh. „Remote timing attacks are practical“. *Comput. Netw.* Bd. 48(5) (Aug. 2005), S. 701–716.

[26] Bundesministeriums der Justiz. *Energiewirtschaftsgesetz, § 21*. 2011.

[27] *CACE - Computer Aided Cryptography Engineering*. 2013. URL: http://www.cace-project.eu/.

[28] J. Camenisch, S. Hohenberger, M. Kohlweiss, A. Lysyanskaya und M. Meyerovich. „How to win the clone wars: Efficient periodic n-times anonymous authentication“. *ACM Conference on Computer and Communications Security. ACM*. 2006.

[29] J. Camenisch und M. Stadler. *Proof Systems for General Statements about Discrete Logarithms*. Technischer Bericht 260. Institute for Theoretical Computer Science, ETH Zürich, 1997.

[30] Jan Camenisch. *Direct Anonymous Attestation Explained*. Technischer Bericht. IBM Research, Juli 2007.

[31] Jan Camenisch und Jens Groth. „Group Signatures: Better Efficiency and New Theoretical Aspects“. *Proceedings of the 4th International Conference on Security in Communication Networks*. SCN'04. Amalfi, Italy: Springer-Verlag, 2005, S. 120–133.

[32] Jan Camenisch, Markulf Kohlweiss und Claudio Soriente. „An Accumulator Based on Bilinear Maps and Efficient Revocation for Anonymous Credentials“. *Public Key Cryptography*. 2009, S. 481–500.

[33] Jan Camenisch und Anna Lysyanskaya. „A Signature Scheme with Efficient Protocols“. *Security in Communication Networks*. Hrsg. von Stelvio Cimato, Giuseppe Persiano und Clemente Galdi. Bd. 2576. Lecture Notes in Computer Science. 10.1007/3-540-36413-7_20. Springer Berlin / Heidelberg, 2003, S. 268–289.

[34] Jan Camenisch und Anna Lysyanskaya. „Signature Schemes and Anonymous Credentials from Bilinear Maps“. *Advances in Cryptology - CRYPTO 2004, 24th Annual International CryptologyConference, Santa Barbara, California, USA, August 15-19, 2004, Proceedings*. Bd. 3152. Lecture Notes in Computer Science. Springer, 2004, S. 56–72. URL: http://www.iacr.org/cryptodb/archive/2004/CRYPTO/1035/1035.pdf.

[35] Jan Camenisch und Markus Michels. „Proving in zero-knowledge that a number is the product of two safe primes“. *Proceedings of the 17th international conference on Theory and application of cryptographic techniques*. EUROCRYPT'99. Prague, Czech Republic: Springer-Verlag, 1999, S. 107–122.

[36] Jan Camenisch und Victor Shoup. „Practical Verifiable Encryption and Decryption of Discrete Logarithms“. *Advances in Cryptology - CRYPTO 2003*. Hrsg. von Dan Boneh. Bd. 2729. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, S. 126–144.

[37] Jan Camenisch und Els Van Herreweghen. „Design and implementation of the idemix anonymous credential system“. *Proceedings of the 9th ACM conference on Computer and communications security*. CCS '02. Washington, DC, USA: ACM, 2002, S. 21–30.

[38] Sébastien Canard, Aline Gouget und Emeline Hufschmitt. „Handy Compact E-cash System“. *Proceedings of the 2nd Conference on Security in Network Architectures and Information Systems-SARSSI*. 2007.

[39] Ran Canetti, Oded Goldreich und Shai Halevi. „The random oracle methodology, revisited“. *J. ACM*, Bd. 51(4) (Juli 2004), S. 557–594.

[40] Nigel P. Chapman. *LR parsing: theory and practice*. New York, NY, USA: Cambridge University Press, 1987.

[41] David Chaum. „Security without identification: transaction systems to make big brother obsolete“. *Commun. ACM*, Bd. 28(10) (Okt. 1985), S. 1030–1044.

[42] David Chaum und Eugène Van Heyst. „Group Signatures“. *Proceedings of the 10th annual international conference on Theory and application of cryptographic techniques*. EUROCRYPT'91. Brighton, UK: Springer-Verlag, 1991, S. 257–265.

[43] Chi-Yin Chow und Mohamed F. Mokbel. „Privacy in Location-based Services: A System Architecture Perspective“. *SIGSPATIAL Special*, Bd. 1(2) (Juli 2009), S. 23–27.

[44] H. Cohen, G. Frey, R. Avanzi, C. Doche, T. Lange, K. Nguyen und F. Vercauteren. *Handbook of Elliptic and Hyperelliptic Curve Cryptography*. Discrete Mathematics and Its Applications. Taylor & Francis, 2010.

[45] Henri Cohen. *A Course in Computational Algebraic Number Theory*. New York, NY, USA: Springer-Verlag New York, Inc., 1993.

[46] S. Cook. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Applications of GPU Computing Series. Morgan Kaufmann, 2013.

[47] Ronald Cramer, Ivan Damgård und Berry Schoenmakers. „Proofs of Partial Knowledge and Simplified Design of Witness Hiding Protocols". *Proceedings of the 14th Annual International Cryptology Conference on Advances in Cryptology*. CRYPTO '94. London, UK, UK: Springer-Verlag, 1994, S. 174–187.

[48] Ivan Damgård. *On Sigma-protocols*. Vorlesung "Cryptologic Protocol Theory". Universität Aarhus, 2010.

[49] Ivan Damgård und Eiichiro Fujisaki. „A Statistically-Hiding Integer Commitment Scheme Based on Groups with Hidden Order". *Proceedings of the 8th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology*. ASIACRYPT '02. London, UK, UK: Springer-Verlag, 2002, S. 125–142.

[50] David Wang. *Stuck in traffic?* Feb. 2007. URL: http://googleblog.blogspot.de/2007/02/stuck-in-traffic.html.

[51] Thomas Denner. *Hardware based security for Smart Grids*. Vortragsfolien, Workshop über angewandte und eingebettete IT-Sicherheit, 5 Juli, Ludwigsburg, Deutschland. 2011.

[52] F.L. DeRemer. *Practical translators for LR(k) languages*. Project MAC, Mass. Inst. of Technology, 1969.

[53] T. Dierks und E. Rescorla. *RFC 5246 - The Transport Layer Security (TLS) Protocol Version 1.2*. Technischer Bericht. Aug. 2008. URL: http://tools.ietf.org/html/rfc5246.

[54] Yevgeniy Dodis und Aleksandr Yampolskiy. „A verifiable random function with short proofs and keys". *Proceedings of the 8th international conference on Theory and Practice in Public Key Cryptography*. PKC'05. Les Diablerets, Switzerland: Springer-Verlag, 2005, S. 416–431.

[55] Danny Dolev, Cynthia Dwork und Moni Naor. „Non-malleable Cryptography". *Proceedings of the Twenty-third Annual ACM Symposium on Theory of Computing*. STOC '91. New Orleans, Louisiana, USA: ACM, 1991, S. 542–552.

[56] Stephen Dussé und Burton Kaliski. „A Cryptographic Library for the Motorola DSP 56000“. *Advances in Cryptology – EUROCRYPT '90*. Hrsg. von Ivan Damgård. Bd. 473. Lecture Notes in Computer Science. 10.1007/3-540-46877-3_21. Springer Berlin / Heidelberg, 2006, S. 230–244.

[57] Jay Earley. „An efficient context-free parsing algorithm“. *Commun. ACM*, Bd. 26(1) (Jan. 1983), S. 57–61.

[58] Eclipse Foundation. *Eclipse 4.3*. 2013. URL: http://www.eclipse.org/.

[59] Mike Edgar. *Future Grid Realisation - Demand Response and Critical Peak Pricing in Europe*. Vortragsfolien, Realisation of the Future Smart Grid, 15–16 Juni, London, Großbritanien. 2011.

[60] S. Eichler. „Anonymous and Authenticated Data Provisioning for Floating Car Data Systems“. *Communication systems, 2006. ICCS 2006. 10th IEEE Singapore International Conference on*. 2006, S. 1–5.

[61] R. Farber. *CUDA Application Design and Development*. Applications of GPU computing. Morgan Kaufmann, 2011.

[62] Felix Disselhoff. *Japan: Strahlenmessung via Crowdsourcing*. März 2011. URL: http://meedia.de/internet/japan-strahlenmessung-via-crowdsourcing/2011/03/21.html.

[63] Amos Fiat und Adi Shamir. „How to prove yourself: Practical solutions to identification and signature problems“. *Advances in Cryptology – CRYPTO '86*. Hrsg. von A. M. Odlyzko. Bd. 263. Springer Verlag, 1987, S. 186–194.

[64] G. Fischer. *Lineare Algebra*. Vieweg-Studium : Grundkurs Mathematik. Vieweg, 2005.

[65] Sebastian Fleissner. „GPU-Accelerated Montgomery Exponentiation“. *Proceedings of the 7th international conference on Computational Science, Part I: ICCS 2007*. ICCS '07. Beijing, China: Springer-Verlag, 2007, S. 213–220.

[66] Forsa. „Erfolgsfaktoren von Smart Metering aus Verbrauchersicht“. Bd. (2010).

[67] Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides. *Design patterns: elements of reusable object-oriented software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[68] Flavio D. Garcia und Bart Jacobs. „Privacy-friendly Energy-metering via Homomorphic Encryption“. *Proceedings of the 6th International Conference on Security and Trust Management*. STM'10. Athens, Greece: Springer-Verlag, 2011, S. 226–238.

[69] Buğra Gedik und Ling Liu. „Protecting Location Privacy with Personalized k-Anonymity: Architecture and Algorithms“. *IEEE Transactions on Mobile Computing*, Bd. 7(1) (Jan. 2008), S. 1–18.

[70] Gabriel Ghinita, Panos Kalnis und Spiros Skiadopoulos. „MOBIHIDE: A Mobilea Peer-to-peer System for Anonymous Location-based Queries“. *Proceedings of the 10th International Conference on Advances in Spatial and Temporal Databases*. SSTD'07. Boston, MA, USA: Springer-Verlag, 2007, S. 221–238.

[71] Gladman, B. et al. *MPIR - Multiple Precision Integers and Rationals, Version 2.6.0*. 2012. URL: http://www.mpir.org/.

[72] GNU Project. *GNU Compiler Collection, Version 4.6.3*. März 2012. URL: http://gcc.gnu.org/.

[73] Shafi Goldwasser und Mihir Bellare. *Lecture Notes on Cryptography*. Juli 2008. URL: http://cseweb.ucsd.edu/~mihir/papers/gb.html.

[74] Shafi Goldwasser, Silvio Micali und Ronald L. Rivest. „A digital signature scheme secure against adaptive chosen-message attacks“. *SIAM J. Comput.* Bd. 17(2) (Apr. 1988), S. 281–308.

[75] Denis Graf. „Implementierung eines Frameworks zur Analyse von ausgeführtem C++ Code und seine Anwendung auf einen bestehenden Interpreter“. Bachelorarbeit. TU Hamburg-Harburg, Nov. 2010.

[76] Ulrich Greveler, Peter Glösekötter, Benjamin Justus und Dennis Loehr. „Multimedia Content Identification Through Smart Meter Power Usage Profiles“. *Computers, Privacy & Data Protection (CPDP)*, Bd. (Jan. 2012).

[77] Dick Grune und Ceriel J. H. Jacobs. *Parsing techniques: a practical guide*. Upper Saddle River, NJ, USA: Ellis Horwood, 1990.

[78] V.C. Gungor, D. Sahin, T. Kocak, S. Ergut, C. Buccella, C. Cecati und G.P. Hancke. „Smart Grid Technologies: Communication Technologies and Standards“. *Industrial Informatics, IEEE Transactions on*, Bd. 7(4) (2011), S. 529–539.

[79] Jan Habermann. „Erweiterung des Zero-Knowledge Compilers um elliptische Kurven“. Bachelorarbeit. TU Hamburg-Harburg, Aug. 2011.

[80] S. Hatje. *Telekom und Vodafone bestätigen dynamische IPv6-Adressen*. Mai 2011. URL: http://www.datenschutz.de/privo/news/detail/?nid=4902.

[81] Jörn Heissler. „Compiler für Zero-Knowledge Proofs of Knowledge“. Diplomarbeit. TU Hamburg-Harburg, Juli 2010.

[82] Jason I. Hong und James A. Landay. „An Architecture for Privacy-sensitive Ubiquitous Computing“. *Proceedings of the 2Nd International Conference on Mobile Systems, Applications, and Services*. MobiSys '04. Boston, MA, USA: ACM, 2004, S. 177–189.

[83] Jeff Howe. *Crowdsourcing: A Definition*. URL: http://www.crowdsourcing.com/cs/.

[84] Jeff Howe. „The Rise of Crowdsourcing“. *WIRED Magazine*, Bd. (Juni 2006). URL: http://www.wired.com/wired/archive/14.06/crowds.html.

[85] International Organization for Standardization, Hrsg. *ISO/IEC 14977:1996 Information Technology - Syntactic Metalanguage - Extended BNF*. 1996.

[86] K. Ireland und M.I. Rosen. *A Classical Introduction to Modern Number Theory*. Graduate Texts in Mathematics. Springer, 1990.

[87] ISO. *The ANSI C standard (C99)*. Technischer Bericht WG14 N1124. ISO/IEC, 1999. URL: http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1124.pdf.

[88] Keon Jang, Sangjin Han, Seungyeop Han, Sue Moon und KyoungSoo Park. „SSLShader: cheap SSL acceleration with commodity processors“. *Proceedings of the 8th USENIX conference on Networked systems design and implementation*. NSDI '11. Boston, MA: USENIX Association, 2011.

[89] Marek Jawurek, Martin Johns und Florian Kerschbaum. „Plug-in privacy for Smart Metering billing“. *CoRR*, Bd. abs/1012.2248 (2010).

[90] Tobias Jeske. „Datenschutzfreundliches Smart Metering“. *Datenschutz und Datensicherheit - DuD*, Bd. 35 (8 2011). 10.1007/s11623-011-0132-9, S. 530–534.

[91] Tobias Jeske. *Floating Car Data from Smartphones: What Google and Waze Know About You and How Hackers Can Control Traffic*. Black Hat 2013. März 2013. URL: https://media.blackhat.com/eu-13/briefings/Jeske/bh-eu-13-floating-car-data-jeske-wp.pdf.

[92] Tobias Jeske. „Privacy-Preserving Smart Metering Without a Trusted-Third-Party“. *SECRYPT 2011 - Proceedings of the International Conference on Security and Cryptography*. Hrsg. von Javier Lopez und Pierangela Samarati. SciTePress, 2011, S. 114–123.

[93] Tobias Jeske und Felix Kurth. „Big Number Modulo Exponentiations For Zero-Knowledge Protocols On GPUs“. *GPU Technology Conference (GTC 2012)*. 2012.

[94] Tao Jiang, Ming Li, Bala Ravikumar und Kenneth W. Regan. „Algorithms and theory of computation handbook“. Hrsg. von Mikhail J. Atallah und Marina Blanton. Chapman & Hall/CRC, 2010. Kap. Formal grammars and languages, S. 20–20.

[95] Richard Jones, Antony Hosking und Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. 1st. Chapman & Hall/CRC, 2011.

[96] Marcelo E. Kaihara. „An Implementation of RSA2048 on GPUs Using CUDA“. 4es Rencontres Arithmétique de l'Informatique Mathématique. Feb. 2011. URL: http://www.marcelokaihara.com/papers/An_Implementation_of_RSA2048_on_GPUs_using_CUDA.pdf.

[97] G. Kalogridis, C. Efthymiou, S.Z. Denic, T.A. Lewis und R. Cepeda. „Privacy for Smart Meters: Towards Undetectable Appliance Load Signatures“. *Smart Grid Communications (SmartGridComm), 2010 First IEEE International Conference on*. 2010, S. 232–237.

[98] Anatoly A. Karatsuba und Y. Ofman. „Multiplication of multidigit numbers on automata“. *Soviet Physics Doklady*, Bd. 7 (1963), S. 595–596.

[99] David B. Kirk und Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010.

[100] Donald Knuth. „On the Translation of Languages from Left to Right“. *Information and Control*, Bd. 8 (1965), S. 607–639.

[101] Donald E. Knuth. *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

[102] Çetin Kaya Koç. *High-Speed RSA Implementation*. Technischer Bericht 201. RSA Laboratories, 1994.

[103] Çetin Kaya Koç, Tolga Acar und Burton S. Kaliski Jr. „Analyzing and Comparing Montgomery Multiplication Algorithms“. *IEEE Micro*, Bd. 16 (3 Juni 1996), S. 26–33.

[104] Peter M. Kogge und Harold S. Stone. „A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations“. *IEEE Trans. Comput.* Bd. 22(8) (Aug. 1973), S. 786–793.

[105] Stefan Kuhn. „Smartphones rooten“. *com!*, Bd. (Aug. 2012), S. 122–125.

[106] Klaus Kursawe, George Danezis und Markulf Kohlweiss. „Privacy-friendly Aggregation for the Smart-grid“. *Proceedings of the 11th International Conference on Privacy Enhancing Technologies*. PETS'11. Waterloo, ON, Canada: Springer-Verlag, 2011, S. 175–191.

[107] Arjen K. Lenstra und Eric R. Verheul. „Selecting Cryptographic Key Sizes“. *Proceedings of the Third International Workshop on Practice and Theory in Public Key Cryptography: Public Key Cryptography*. PKC '00. London, UK, UK: Springer-Verlag, 2000, S. 446–465.

[108] Hsiao-Ying Lin, Wen-Guey Tzeng, Shiuan-Tzuo Shen und Bao-Shuh P. Lin. „A Practical Smart Metering System Supporting Privacy Preserving Billing and Load Monitoring“. *Proceedings of the 10th International Conference on Applied Cryptography and Network Security*. ACNS'12. Singapore: Springer-Verlag, 2012, S. 544–560.

[109] Helger Lipmaa. „On Diophantine Complexity and Statistical Zero-Knowledge Arguments". *Advances on Cryptology – ASIACRYPT 2003*. Bd. 2894. Lecture Notes in Computer Science. Springer-Verlag, 2003, S. 398–415.

[110] Roman Maeder. „Storage Allocation for the Karatsuba Integer Multipliation Algorithm". *Proceedings of the International Symposium on Design and Implementation of Symbolic Computation Systems*. DISCO '93. London, UK, UK: Springer-Verlag, 1993, S. 59–65.

[111] Mark Manulis, Nils Fleischhacker, Felix Günther, Franziskus Kiefer und Bertram Poettering. *Group Signatures: Authentication with Privacy*. Technischer Bericht. Cryptographic Protocols Group, Department of Computer Science Technische Universität Darmstadt, 2012.

[112] Félix Gómez Mármol, Christoph Sorge, Ronald Petrlic, Osman Ugus, Dirk Westhoff und Gregorio Martínez Pérez. „Privacy Enhanced Architecture for Smart Metering". *International Journal of Information Security*, Bd. 12(2) (2013), S. 67–82.

[113] Steve McConnell. *Rapid Development: Taming Wild Software Schedules*. 1st. Redmond, WA, USA: Microsoft Press, 1996.

[114] Patrick McDaniel und Stephen McLaughlin. „Security and Privacy Challenges in the Smart Grid". *IEEE Security and Privacy*, Bd. 7(3) (2009), S. 75–77.

[115] Stephen McLaughlin, Patrick McDaniel und William Aiello. „Protecting Consumer Privacy from Electric Load Monitoring". *Proceedings of the 18th ACM Conference on Computer and Communications Security*. CCS '11. Chicago, Illinois, USA: ACM, 2011, S. 87–98.

[116] Sarah Meiklejohn, C. Chris Erway, Alptekin Küpçü, Theodora Hinkle und Anna Lysyanskaya. „ZKPDL: a language-based system for efficient zero-knowledge proofs and electronic cash". *Proceedings of the 19th USENIX conference on Security*. USENIX Security'10. Washington, DC: USENIX Association, 2010, S. 13–13.

[117] Alfred J. Menezes, Scott A. Vanstone und Paul C. Van Oorschot. *Handbook of Applied Cryptography*. 1st. Boca Raton, FL, USA: CRC Press, Inc., 1996.

[118] Daniele Micciancio. „The RSA Group is Pseudo-Free". *J. Cryptol.* Bd. 23(2) (Apr. 2010), S. 169–186.

[119] Mohamed F. Mokbel, Chi-Yin Chow und Walid G. Aref. „The New Casper: Query Processing for Location Services Without Compromising Privacy". *Proceedings of the 32Nd International Conference on Very Large Data Bases*. VLDB '06. Seoul, Korea: VLDB Endowment, 2006, S. 763–774.

[120] Peter L. Montgomery. „Modular Multiplication without Trial Division". *Mathematics of Computation*, Bd. 44(170) (1985), S. 519–521.

[121] Nathan Morrow, Nancy Mock, Adam Papendieck und Nicholas Kocmich. „Independent Evaluation of the Ushahidi Haiti Project“. Bd. (Apr. 2011).

[122] Klaus J. Müller. „Gewinnung von Verhaltensprofilen am intelligenten Stromzähler“. *Datenschutz und Datensicherheit*, Bd. 6 (2010), S. 359–364.

[123] Birgit Niesing. „Mit neuer Energie“. *Fraunhofer Magazin weiter.vorn*, Bd. (2010), S. 5–9.

[124] NVIDIA Corporation. *CUDA 5.5*. 2013. URL: http://developer.nvidia.com/cuda.

[125] NVIDIA Corporation. *CUDA C Programming Guide*. Technischer Bericht PG-02829-001_v5.5. Juli 2013.

[126] NVIDIA Corporation. *NVIDIA Visual Profiler 5.5*. 2013. URL: http://developer.nvidia.com/cuda.

[127] NVIDIA Corporation. *Parallel Thread Execution ISA 3.2*. Technischer Bericht. 2013.

[128] Heise Online. *Angriff auf Playstation Network: Persönliche Daten von Millionen Kunden gestohlen*. Apr. 2011. URL: http://heise.de/-1233136.

[129] OpenMP Architecture Review Board. *OpenMP*. URL: http://www.openmp.org/.

[130] OpenSSL Development Team. *OpenSSL Project, Version 1.0.1e*. 2013. URL: https://www.openssl.org/.

[131] OpenStreetMap Foundation. *OpenStreetMap - The Free Wiki World Map*. URL: http://www.openstreetmap.org/.

[132] Oracle Corporation. *Java Standard Edition, Version 1.7.45*. Okt. 2013. URL: http://www.oracle.com/technetwork/java/.

[133] Terence J. Parr und Russell W. Quong. „ANTLR: A Predicated-LL(k) Parser Generator“. *Software Practice and Experience*, Bd. 25 (1994), S. 789–810.

[134] Torben P. Pedersen. „Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing“. *Proceedings of the 11th Annual International Cryptology Conference on Advances in Cryptology*. CRYPTO '91. London, UK, UK: Springer-Verlag, 1992, S. 129–140.

[135] Ronald Petrlic. „A privacy-preserving Concept for Smart Grids“. *18. DFN Workshop SSicherheit in vernetzten Systemen"*. 2011.

[136] C.P. Pfleeger und S.L. Pfleeger. *Security in Computing*. Prentice Hall professional technical reference. Prentice Hall PTR, 2003.

[137] S. Phoha, T.F.L. Porta und C. Griffin. *Sensor Network Operations*. Wiley, 2006.

[138] S. Pohlig und M. Hellman. „An improved algorithm for computing logarithms over GF(p) and its cryptographic significance (Corresp.)“ *IEEE Trans. Inf. Theor.* Bd. 24(1) (Jan. 1978), S. 106–110.

[139] The Flex Project. *Flex 2.5.37*. 2012. URL: http://flex.sourceforge.net/.

[140] J.-J. Quisquater und C. Couvreur. „Fast decipherment algorithm for RSA public-key cryptosystem“. *Electronics Letters*, Bd. 18 (21 1982), S. 905–907.

[141] J. O. Rabin und Jeffrey Shallit. *Randomized algorithms in number theory*. Technischer Bericht. Chicago, IL, USA, 1985.

[142] Stefan Rass, Simone Fuchs, Martin Schaffer und Kyandoghere Kyamakya. „How to Protect Privacy in Floating Car Data Systems“. *Proceedings of the Fifth ACM International Workshop on VehiculAr Inter-NETworking*. VANET '08. San Francisco, California, USA: ACM, 2008, S. 17–22.

[143] AT&T Labs Research und Contributors. *Graphviz, Version 2.30.0*. Jan. 2013. URL: http://graphviz.org/.

[144] Alfredo Rial und George Danezis. „Privacy-preserving Smart Metering“. *Proceedings of the 10th Annual ACM Workshop on Privacy in the Electronic Society*. WPES '11. Chicago, Illinois, USA: ACM, 2011, S. 49–60.

[145] R. L. Rivest, A. Shamir und L. Adleman. „A Method for Obtaining Digital Signatures and Public-key Cryptosystems“. *Commun. ACM*, Bd. 21(2) (Feb. 1978), S. 120–126.

[146] R.L. Rivest, A. Shamir und L.M. Adleman. *Cryptographic communications system and method*. US Patent 4,405,829. Sep. 1983. URL: http://www.google.de/patents/US4405829.

[147] Ron Rivest. *Lecture Notes 15 : Voting, Homomorphic Encryption*. Okt. 2002. URL: http://web.mit.edu/6.857/OldStuff/Fall02/handouts/L15-voting.pdf.

[148] Roy Williams. *Youve got better things to do than wait in traffic*. März 2011. URL: http://googlemobile.blogspot.de/2011/03/youve-got-better-things-to-do-than-wait.html.

[149] Jason Sanders und Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. 1st. Addison-Wesley Professional, 2010.

[150] Claus-Peter Schnorr. „Efficient Identification and Signatures for Smart Cards“. *Proceedings of the 9th Annual International Cryptology Conference on Advances in Cryptology*. CRYPTO '89. London, UK, UK: Springer-Verlag, 1990, S. 239–252.

[151] Berry Schoenmakers. *Cryptography 2 (2WC13) / Cryptographic Protocols 1 (2WC17) Lecture Notes*. Feb. 2014. URL: http://www.win.tue.nl/~berry/2WC13/LectureNotes.pdf.

[152] A. Sharma. *Theory of Automata and Formal Languages*. Laxmi Publications (P) Limited, 2006.

[153] A.K. Sharma. *Group Theory*. DPH Mathematics Series. Discovery Publishing House Pvt. Limited, 2010.

[154] Sahil Sharma. „Application of Smart Card Technology in a Smart Metering Gateway". Studienarbeit. TU Hamburg-Harburg, Okt. 2011.

[155] Victor Shoup. „Lower Bounds for Discrete Logarithms and Related Problems". *Proceedings of the 16th Annual International Conference on Theory and Application of Cryptographic Techniques*. EUROCRYPT'97. Konstanz, Germany: Springer-Verlag, 1997, S. 256–266.

[156] Victor Shoup und Joël Alwen. *Σ-Protocols Continued & Introduction to Zero Knowledge*. 2007. URL: http://cs.nyu.edu/courses/spring07/G22.3220-001/lec3.pdf.

[157] Jennifer G. Steiner, Clifford Neuman und Jeffrey I. Schiller. „Kerberos: An Authentication Service for Open Network Systems". *in Usenix Conference Proceedings*. 1988, S. 191–202.

[158] Simon Stoye. „Implementierung bilinearer Maps und die Unterstützung komplexer Sicherheitsprotokolle in einem Zero-Knowledge-Compiler". Diplomarbeit. TU Hamburg-Harburg, Mai 2013.

[159] Bjarne Stroustrup. *The C++ Programming Language*. 3rd. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

[160] Latanya Sweeney. „K-anonymity: A Model for Protecting Privacy". *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.* Bd. 10(5) (Okt. 2002), S. 557–570.

[161] Robert Szerwinski und Tim Güneysu. „Exploiting the Power of GPUs for Asymmetric Cryptography". *Cryptographic Hardware and Embedded Systems — CHES 2008*. 2008, S. 79–99.

[162] Till Tantau und Christian Feuersaenger. *PGF/TikZ, Version 2.10*. Aug. 2012. URL: http://sourceforge.net/projects/pgf/.

[163] OASIS Web Services Secure Exchange (WS-SX) TC. *WS-Trust 1.4*. Apr. 2012. URL: http://docs.oasis-open.org/ws-sx/ws-trust/v1.4/ws-trust.pdf.

[164] Terence Parr. *ANTLR - ANother Tool for Language Recognition, Version 4.1*. Juni 2013. URL: http://www.antlr.org/.

[165] The GNU Project. *GNU Bison, Version 3.0*. Juli 2013. URL: http://www.gnu.org/software/bison/.

[166] The Tor Project. *Tor: anonymity online*. 2013. URL: https://www.torproject.org/.

[167] H.C.A. van Tilborg und S. Jajodia. *Encyclopedia of Cryptography and Security*. Encyclopedia of Cryptography and Security Bd. 1. Springer, 2011.

[168] TISA. *The Traffic Message Channel (TMC)*. URL: http://www.tisa.org/technologies/tmc/.

[169] Masaru Tomita. *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Norwell, MA, USA: Kluwer Academic Publishers, 1985.

[170] David P. Varodayan und Ashish Khisti. „Smart meter privacy using a rechargeable battery: Minimizing the rate of information leakage". *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP 2011, May 22-27, 2011, Prague Congress Center, Prague, Czech Republic*. IEEE, 2011, S. 1932–1935.

[171] Benjamin Vetter, Osman Ugus, Dirk Westhoff und Christoph Sorge. „Homomorphic Primitives for a Privacy-Friendly Smart Metering Architecture". *Proceedings of the International Conference on Security and Cryptography (SECRYPT 2012)*. 2012.

[172] Markus Völter, Sebastian Benz, Christian Dietrich, Birgit Engelmann Mats Helander, Lennart C. L. Kats, Eelco Visser und Guido Wachsmuth. *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages*. http://dslbook.org. CreateSpace Independent Publishing Platform, 2013.

[173] VUB BrusSense group. *NoiseTube*. URL: http://noisetube.net/.

[174] Shuang Wang, Lijuan Cui, Jialan Que, Dae-Hyun Choi, Xiaoqian Jiang, S. Cheng und Le Xie. „A Randomized Response Model for Privacy Preserving Smart Metering". *Smart Grid, IEEE Transactions on*, Bd. 3(3) (2012), S. 1317–1324.

[175] Wikimedia Foundation. *Wikipedia - Die freie Enzyklopädie*. URL: http://www.wikipedia.org/.

[176] Man Lung Yiu, Christian S. Jensen, Xuegang Huang und Hua Lu. „SpaceTwist: Managing the Trade-Offs Among Location Privacy, Query Performance, and Query Accuracy in Mobile Services". *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*. ICDE '08. Washington, DC, USA: IEEE Computer Society, 2008, S. 366–375.

[177] Yvonne Ortmann. *Mobile Crowdsourcing als Lebensretter*. März 2012. URL: http://t3n.de/magazin/mobile-crowdsourcing-schwarmintelligenz-handynutzer-229492/.

[178] Kaiyong Zhao. *Implementation of Multiple-precision Modular Multiplication on GPU*. GPU Technology Conference. 2009.

[179] Alexander Zollondz. *Google: Mitarbeiter schnüffelte Jugendliche aus*. Sep. 2010. URL: http://www.netzwelt.de/news/84056-google-mitarbeiter-schnueffelte-jugendliche.html.