Institut für Informationssicherheit

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

# Untersuchung der Durchführung der Offline Phase von MPC Protokollen

Christina Bauer

| | |
|---|---|
| **Studiengang:** | Informatik |
| **Prüfer/in:** | Prof. Dr. Ralf Küsters |
| **Betreuer/in:** | Julian Liedtke, M.Sc. |
| **Beginn am:** | 11. April 2019 |
| **Beendet am:** | 10. Oktober 2019 |

## Kurzfassung

Multiparty Computation Protokolle erlauben es einer Menge von Parteien gemeinsam eine Funktion zu berechnen. Diese Protokolle müssen die Korrektheit des Ergebnisses sowie die Geheimhaltung der Eingaben (Privacy) sicherstellen. Oft teilen sich solche Protokolle in eine Vorberechnungsphase (Offline Phase), welche von den Eingaben der Parteien unabhängig ist, und eine Phase für die tatsächliche Berechnung (Online Phase).

In dieser Arbeit wird die Offline Phase für Multiparty Computation Protokolle basierend auf Paillier-Verschlüsselung betrachtet. Es wird ein Protokoll für die Erzeugung eines Schlüsselpaars des Paillier-Kryptosystems mit einem unter den Parteien aufgeteilten geheimen Schlüssel untersucht, ohne dass eine Partei, welcher alle anderen vertrauen, benötigt wird. Zudem wird betrachtet, wie Ciphertexte von zufälligen Werten in der Offline Phase vorberechnet werden können, ohne dass Parteien Kenntnis über den zugrundeliegenden Wert erlangen. Dazu werden Protokolle für die Generierung eines Zufallswerts, Zufallsbits sowie eines zufälligen Werts in einem eingegrenzten Bereich vorgestellt.

# Inhaltsverzeichnis

# Verzeichnis der Protokolle

# Abkürzungsverzeichnis

**MPC** Multiparty Computation. 11

**ZK** Zero-Knowledge. 17

# 1 Einleitung

In einem IT Unternehmen wollen die Mitglieder des Betriebsrats einen neuen Vorsitzenden aus ihrer Mitte wählen. Der Betriebsrat ist zerstritten und so befürchten die Mitglieder, dass einzelne Mitglieder sich verbünden und versuchen, die Wahl über ihre jeweilige Stimme hinaus zu beeinflussen. Die Stimmen der einzelnen Mitglieder sollen zudem geheim bleiben.

Ein Ansatz, um eine solche Wahl durchzuführen, ist Multiparty Computation (MPC). Bei MPC wollen mehrere Parteien gemeinsam eine Funktion berechnen. Dabei soll sichergestellt werden, dass die Berechnung korrekt durchgeführt wird und die Eingaben der Parteien geheim bleiben. Der Ursprung der MPC geht auf Arbeiten in den 1980er Jahren zurück ([19], [13], [2]). Seitdem wurden die Theorien zu MPC in diversen Arbeiten weiterentwickelt. Neben elektronischen Abstimmungen (E-Voting) sind auch Auktionen und Privacy-Preserving Data Mining mögliche Anwendungsfelder für MPC. Eine erste Anwendung fand 2008 für die Berechnung von Zuckerrübenpreisen in Dänemark statt [3].

Da MPC Protokolle oft nach Erhalt der Eingabe die Ausgabe schnell berechnen sollen, in obigem Beispiel noch während der Betriebsratssitzung, werden sie oft in eine Vorberechnungsphase (Offline Phase) und eine Phase für die übrige Berechnung (Online Phase) unterteilt. In der Offline Phase werden von der Eingabe unabhängige Vorberechnungen durchgeführt. Diese Vorberechnungen werden genutzt, um in der Online Phase die Berechnung mit den Eingaben durchzuführen. Dadurch kann die Berechnung in der Online Phase effizienter durchgeführt werden, da die Vorberechnungen bereits in der Offline Phase durchgeführt wurden. Für den Fall der Wahl des Betriebsratsvorsitzenden kann die Offline Phase durchgeführt werden, nachdem bekannt wird, dass ein neuer Vorsitzender gewählt wird. Die Online Phase wird während der Betriebsratssitzung durchgeführt, sobald alle Betriebsratsmitglieder ihre Stimme abgegeben haben.

## 1.1 Ziel dieser Arbeit

Diese Arbeit befasst sich mit Protokollen für die Offline Phase. Da MPC Protokolle oft homomorphe Verschlüsselungsverfahren verwenden, wird in dieser Arbeit MPC basierend auf dem Paillier-Verschlüsselungsverfahren betrachtet. Dafür benötigen die Parteien ein Paillier-Schlüsselpaar mit geteiltem Secret Key. Diese Arbeit soll beschreiben, wie ein solcher geheimer Schlüssel generiert werden kann, ohne dass die Parteien dafür eine vertrauenswürdige Partei heranziehen müssen. Mit diesem Schlüsselpaar können die Parteien dann Ciphertexte berechnen, ohne dass einzelne Parteien Kenntnis von den zugrundeliegenden Werten erlangen. In dieser Arbeit soll zudem aufgezeigt werden, wie zufällige Werte in der Offline Phase vorberechnet werden können. Für jedes der vorgestellten Protokolle wird die Korrektheit und die Wahrung der Privacy der Werte untersucht. Das Ziel dieser Arbeit ist somit die Erstellung einer Übersicht über Werkzeuge für die Offline Phase.

## 1.2 Aufbau der Arbeit

Diese Arbeit ist wie folgt aufgebaut: In Kapitel 2 werden zunächst die Grundlagen dieser Arbeit beschrieben. Die verteilte Schlüsselerzeugung sowie die Ver- und Entschlüsselung mit dem generierten Schlüssel werden in Kapitel 3 vorgestellt. Darauf aufbauend wird in Kapitel 4 gezeigt, wie mit dem erstellten Schlüssel Berechnungen auf Ciphertexten ausgeführt und damit Ciphertexte für die Online Phase vorberechnet werden können. Die Arbeit schließt mit einer Zusammenfassung in Kapitel 5.

# 2 Grundlagen

In diesem Kapitel wird in die Grundlagen dieser Arbeit eingeführt. Dazu werden in Abschnitt 2.1 die Annahmen über das zugrundeliegende Netzwerk und in Abschnitt 2.2 die betrachteten Sicherheitseigenschaften vorgestellt. Danach werden von den Protokollen der folgenden Kapitel verwendete Werkzeuge beschrieben, in Abschnitt 2.3 homomorphe Verschlüsselung, in Abschnitt 2.4 Secret Sharing und in Abschnitt 2.5 Zero-Knowledge Proofs.

Die Protokolle in dieser Arbeit werden jeweils von $n$ Parteien $P_1, ..., P_n$ ausgeführt.

## 2.1 Kommunikationskanäle

Die Parteien sind jeweils über sichere Kanäle verbunden, welche die Integrität und Authentifikation der versendeten Nachrichten sicherstellen. Dies kann über eine Public Key Infrastruktur realisiert werden. Dabei wird die Existenz folgender Kommunikationskanäle angenommen:

**Nachrichten senden** Die Partei $P_i$ kann Partei $P_j$ eine Nachricht $m$ zusenden. Dies wird mit

$$\text{SEND}\,(i, j, m)$$

notiert.

**Broadcast Kanal** Die Partei $i$ kann eine Nachricht $m$ an alle anderen Parteien broadcasten. Dies wird mit

$$\text{BROADCAST}\,(i, m)$$

notiert.

Es wird zudem davon ausgegangen, dass alle Kommunikationskanäle nach endlicher Zeit die Nachrichten übermitteln.

## 2.2 Sicherheit

Sicherheit im Kontext von MPC bedeutet, dass die ehrlichen Parteien, also jene Parteien, die sich an das Protokoll halten, das Protokoll, auch in Anwesenheit eines potenziellen Angreifers, des Adversary, sicher ausführen können. Sicher bedeutet hierbei, dass die Eigenschaften Korrektheit und Privacy garantiert werden. Zudem ist es wünschenswert, dass MPC Protokolle Guaranteed Output Delivery erfüllen.

#### Adversary

Der Adversary kontrolliert eine Teilmenge der Parteien. Er kann semi-honest oder malicious sein. Ein semi-honest Adversary folgt dem Protokoll, versucht aber geheime Informationen abzuleiten. In der Literatur wird ein solcher Adversary auch 'honest-but-curious' oder 'passiv' genannt. Ein malicious Adversary kann zusätzlich beliebig vom Protokoll abweichen. Die Protokolle in dieser Arbeit streben Sicherheit gegen einen malicious Adversary an.

In dieser Arbeit wird davon ausgegangen, dass die Menge der vom Adversary kontrollierten Parteien sich während einer Protokollausführung nicht ändert. Ein solcher Adversary heißt statisch. Zudem wird angenommen, dass kryptografische Sicherheitsannahmen gelten, der Adversary hat also nur begrenzte Rechenleistung.

#### Korrektheit

Die MPC Protokolle müssen korrekt sein, das heißt, dass jede ehrliche Partei die korrekten Ausgaben des Protokolls in Anwesenheit eines Adversary erhält. Bei Protokollen mit Abbruch erhält jede ehrliche Partei die korrekten Ausgaben oder erfährt, dass das Protokoll abgebrochen wurde.

#### Privacy

Die Privacy eines Werts bedeutet, dass durch das Protokoll keine Informationen, mit denen Rückschlüsse auf diesen Wert gemacht werden könnten, preisgegeben werden.

#### Guaranteed Output Delivery

Ein MPC Protokoll bietet Guaranteed Output Delivery, wenn der Adversary ehrliche Parteien nicht davon abhalten kann ihre Ausgabe zu erhalten.

## 2.3 Homomorphe Verschlüsselung

Für die Sicherheit gegen einen malicious Adversary benötigen die Protokolle aus Kapitel 3 Verschlüsselungsverfahren. Mit ihnen kann sich eine Partei $P_i$ in den Protokollen aus dieser Arbeit auf einen geheimen Wert festlegen, indem sie einen Ciphertext dazu broadcastet. Die Parteien können diese Ciphertexte nutzen, um zu prüfen, ob Berechnungen korrekt durchgeführt wurden.

Bei den homomorphen Verschlüsselungsverfahren in dieser Arbeit handelt sich um Public Key Verfahren, der Public Key wird mit $\mathsf{pk}$ notiert, der Secret Key mit $\mathsf{sk}$. Verschlüsselungen eines Klartexts $m$ werden mit $\mathsf{Enc}_{\mathsf{pk}}(m)$ notiert oder mit $\mathsf{Enc}_{\mathsf{pk}}(m,r)$, falls der bei der Verschlüsselung verwendete Zufallswert $r$ hervorgehoben werden soll. Die Entschlüsselung eines Ciphertexts $c$ wird $\mathsf{Dec}_{\mathsf{sk}}(c)$ geschrieben.

Die verwendeten Verfahren sind additiv homomorph, das heißt es gibt eine Multiplikation $\cdot$ auf den Ciphertexten, sodass

$$\mathsf{Dec}_{\mathsf{sk}}\left(\mathsf{Enc}_{\mathsf{pk}}(m_1) \cdot \mathsf{Enc}_{\mathsf{pk}}(m_2)\right) = m_1 + m_2$$

für alle Schlüsselpaare $(\mathsf{pk}, \mathsf{sk})$ und alle Klartexte $m_1$ und $m_2$ gilt.

### 2.3.1 Additiv homomorphes ElGamal-Kryptosystem

Das in diesem Abschnitt vorgestellte Kryptosystem ist eine additiv homomorphe Variante der ElGamal-Verschlüsselung [12] wie von Hazay et al. [14] beschrieben.

**Definition 2.3.1**
*Ein Element $g$ einer Gruppe $\mathbb{G}$, für welches für alle $h$ aus $\mathbb{G}$ ein $a \in \mathbb{G}$ existiert, sodass $h = g^a$, heißt Generator.*

Der additiv homomorphen ElGamal-Verschlüsselung liegt eine Restklassengruppe $\mathbb{Z}_Q$ bezüglich einer Primzahl $Q$ sowie ein Generator $g$ zugrunde. Diese Parameter sind öffentlich bekannt. Der Public Key dieses Verfahrens $\mathsf{pk}_{EG}$ ist das Tupel $(g, h)$ und der Secret Key $\mathsf{sk}_{EG} = \log_g(h)$ für ein $h \in \mathbb{Z}_Q$. Die Verschlüsselung wird berechnet als $\mathsf{Enc}_{\mathsf{pk}_{EG}}(m, r) = (g^r, h^r g^m)$ für $m \in \mathbb{Z}_Q$ und ein zufälliges $r \in \mathbb{Z}_Q$. Es wird hier also $g^m$ im klassischen ElGamal verschlüsselt. Dadurch ergibt sich die Eigenschaft der additiven Homomorphie. Die Entschlüsselung von $(a, b)$ ist $\mathsf{Dec}_{\mathsf{sk}_{EG}}(a, b) = b \cdot a^{-\mathsf{sk}_{EG}}$. Damit ergibt sich allerdings nur $\mathsf{Dec}_{\mathsf{sk}_{EG}}(g^r, h^r g^m) = g^m$. Für die folgenden Anwendungen ist dies allerdings ausreichend.

Der Term $c_1 \cdot c_2$ meint in dieser Arbeit die elementweise Multiplikation von Ciphertexten $c_1$ und $c_2$ und die elementweise Exponentiation mit $a$ wird als $c_1^a$ notiert. Diese Multiplikation führt zur Homomorphie des Kryptosystems.

Die Sicherheit des Verfahrens wird durch das Decisional Diffie Hellman-Problem bestimmt. Die zugehörige Decisional Diffie Hellman-Annahme ist, dass es keinen Polynomialzeit-Algorithmus gibt, welcher für gegebene $g^x$ und $g^y$ für Elemente $x, y \in \mathbb{Z}_Q^*$ und einem Generator $g$ der Gruppe $\mathbb{Z}_Q^*$ das Tupel $(g^x, g^y, g^{xy})$ von dem Tupel $(g^x, g^y, g^z)$ mit einem zufälligen Element $z$ aus der Gruppe unterscheidet [14].

### 2.3.2 Paillier-Kryptosystem

Das Paillier-Kryptosystem wurde in der Arbeit von Paillier [15] erstmals vorgestellt. Der Public Key $\mathsf{pk}_{Pa}$ des Kryptosystems ist $N$, wobei $N$ das Produkt zweier Primzahlen $p$ und $q$ mit gleicher Bitlänge ist. Der Secret Key $\mathsf{sk}_{Pa}$ ist $(N, \phi(N))$. $\phi$ bezeichnet dabei die Eulersche Phi-Funktion. Die Verschlüsselung von $m \in \mathbb{Z}_N$ wird als

$$\mathsf{Enc}_{\mathsf{pk}_{Pa}}(m, r) = (1 + N)^m r^N \mod N^2$$

für ein zufälliges $r \in \mathbb{Z}_N^*$ berechnet. Ein Ciphertext $c$ wird mit

$$\mathsf{Dec}_{\mathsf{sk}_{Pa}}(c) = \frac{c^{\phi(N)} \mod N^2 - 1}{N} \cdot \phi(N)^{-1} \mod N$$

entschlüsselt.

Die Sicherheit von Paillier ist von der Schwierigkeit des Decisional Composite Residuosity-Problems abhängig. Die zugehörige Decisional Composite Residuosity-Annahme ist, dass es keinen Polynomialzeit-Algorithmus gibt, welcher entscheidet, ob ein Element aus $\mathbb{Z}_{N^2}$ eine $N$-te Wurzel modulo $N^2$ besitzt [15].

## 2.4 Secret Sharing

Beim Secret Sharing wird ein geheimer Wert in Anteile für jede einzelne Partei, sogenannte Shares, aufgeteilt. Die Menge aller Shares eines Werts $a$ wird Sharing von $a$ genannt. Das Geheimnis $a$ kann dann nur von einer qualifizierten Menge von Parteien rekonstruiert werden, während andere Mengen von Parteien aus ihren Shares keine Informationen über das Geheimnis erfahren. Typischerweise sind qualifizierte Mengen über einen Threshold $t$ festgelegt. Mengen von mindestens $t + 1$ Parteien können das Geheimnis aufdecken, während Mengen von höchstens $t$ Parteien keine Informationen über den geheimen Wert lernen. Somit ist das Geheimnis vor einem Adversary der höchstens $t$ Parteien kontrolliert sicher.

### 2.4.1 Additives Secret Sharing

Für die Protokolle aus Abschnitt 3.3 und Abschnitt 3.4 wird ein additives Secret Sharing Scheme benötigt. Ein Sharing von $a$ in diesem System wird mit $[a]_+$ notiert. Jede Partei $P_i$ kennt eine Share $[a]_{+,i}$, sodass $a = \sum_{i=1}^{n} [a]_{+,i}$ ist. Folglich ist für dieses Verfahren $t = n - 1$: Es wird die Share jeder Partei benötigt, um den geteilten Wert wiederherzustellen.

### 2.4.2 Shamir Sharing in den ganzen Zahlen

Das Shamir Secret Sharing Scheme wurde von Shamir [18] erstmals vorgestellt. In Kapitel 3 wird Shamir Secret Sharing in den ganzen Zahlen $\mathbb{Z}$ für das Teilen des Secret Keys verwendet. Ein Sharing für einen Wert aus $a \in \mathbb{Z}$ wird mit $[a]_{\mathbb{Z}}$ notiert. Die Share der Partei $P_i$ wird $[a]_{\mathbb{Z},i}$ geschrieben. Ein Shamir Sharing kann für ein beliebiges $t \in \{1, \ldots, n-1\}$ konstruiert werden.

Um einen Wert $a$ zu teilen, wählt die teilende Partei ein zufälliges Polynom $f$ des Grades $t$ für welches $f(0) = a$ gilt. Jede Partei $P_i$ erhält dann den Wert $f(i)$ als ihre Share $[a]_{\mathbb{Z},i}$.

Das Polynom $f$ ist mit $t + 1$ Punkten vollständig bestimmt, somit kann eine Menge $S$ von $t + 1$ Parteien das Geheimnis gemeinsam rekonstruieren. Dazu wenden sie Lagrange Interpolation an: Jede Partei $P_i$ aus $S$ trägt ihre Share $[a]_{\mathbb{Z},i}$ bei und die Parteien berechnen

$$a = \sum_{i \in S} [a]_{\mathbb{Z},i} \, \lambda_{0,i}^{S}$$

mit $\lambda_{0,i}^{S} = \prod_{j \in S \setminus \{i\}} \frac{-i}{j-i}$.

## 2.5 Zero-Knowledge Proofs

Zero-Knowledge (ZK) Protokolle sind Protokolle, bei welchen ein Prover einem Verifier Kenntnis über ein Geheimnis beweist. ZK Protokolle erfüllen die folgenden drei Eigenschaften:

- Vollständigkeit: Wenn der Prover das Geheimnis kennt und dem Protokoll folgt, akzeptiert der Verifier.
- Zuverlässigkeit: Wenn der Prover das Geheimnis nicht kennt, akzeptiert der Verifier nicht.
- Zero-Knowledge: Die Privacy des Geheimnisses des Provers bleibt gewahrt. Es wird keine Information preisgeben außer, ob der Prover das Geheimnis kennt.

Typischerweise werden Vollständigkeit und Zuverlässigkeit nur mit einer hohen Wahrscheinlichkeit erreicht. Somit müssen die Protokolle mehrmals hintereinander ausgeführt werden, bis der Verifier sich hinreichend sicher sein kann.

In diesem Abschnitt werden ZK Protokolle vorgestellt, die von Protokollen in dieser Arbeit benötigt werden, um Korrektheit und Privacy bei einem malicious Adversary zu garantieren. Dabei wird nur die bereitgestellte Funktionalität des ZK Protokolls beschrieben, da eine ausführliche Behandlung den Umfang dieser Arbeit sprengen würde.

#### ZK-DL

Der Prover kann dem Verifier mit dem Protokoll ZK-DL beweisen, dass er einen diskreten Logarithmus $\alpha$ eines Werts $h$ bezüglich der Basis $g$ kennt. Beide Parteien kennen $h$ und $g$, $\alpha$ ist nur dem Prover bekannt. Das Protokoll hierfür folgt aus dem Paper von Schnorr [16].

#### ZK-DH

Mithilfe des Protokolls ZK-DH beweist der Prover dem Verifier, dass ein beiden bekanntes Tupel $(g_0, g_1, h_0, h_1)$ ein sogenanntes Diffie-Hellman-Tupel ist, also dass $\log_{g_0}(h_0) = \log_{g_1}(h_1)$ gilt. Dabei kennt der Prover $w = \log_{g_0}(h_0) = \log_{g_1}(h_1)$. Ein solches Protokoll stellen Chaum und Pedersen [7] vor.

#### ZK-BOUND

Mit dem Protokoll ZK-BOUND kann der Prover dem Verifier beweisen, dass ein verschlüsselter Wert $\alpha$ kleiner als ein Wert $B$ ist. Der Prover kennt dabei die Werte $\alpha$ und den bei der Verschlüsselung verwendeten Zufallswert $r$ und beide Parteien kennen $B$, den verwendeten öffentlichen Schlüssel $\mathsf{pk}$ sowie $\mathsf{Enc}_{\mathsf{pk}}(\alpha, r)$. Eine Skizze eines solchen Protokolls wird von Hazay et al. [14] beschrieben.

#### ZK-ENC

Der Prover kann mit dem Protokoll ZK-ENC dem Verifier beweisen, dass er den Klartext $\alpha$ zu einem Ciphertext $c$ kennt. Beide Parteien kennen dabei $c$ und den verwendeten öffentlichen Schlüssel $\mathsf{pk}$. Der Prover verfügt zusätzlich über Kenntnis von $\alpha$ und $r$, sodass $c = \mathsf{Enc}_{\mathsf{pk}}(\alpha, r)$. Ein solches Protokoll für ElGamal-Verschlüsselung folgt aus dem Paper von Schnorr [16], für Paillier-Verschlüsselung wird ein Protokoll von Cramer, Damgard und Nielsen [9] vorgestellt.

#### ZK-MULT

Mit dem Protokoll ZK-MULT beweist der Prover dem Verifier, dass eine Multiplikation korrekt ausgeführt wurde. Beide Parteien kennen die Ciphertexte der Faktoren $c_0$ und $c_1$ sowie einen Ciphertext $c_2$ und den öffentlichen Schlüssel $\mathsf{pk}$. Der Prover kennt außerdem $\alpha$ und $r_\alpha$ sowie $r_0$, sodass er zeigen kann, dass $c_1 = \mathsf{Enc}_{\mathsf{pk}}(\alpha, r_\alpha)$ und $c_2 = c_0^\alpha \cdot \mathsf{Enc}_{\mathsf{pk}}(0, r_0)$ gilt. Dieses Protokoll geht auf Damgard und Jurik [11] zurück.

#### ZK-EXP

Mit dem Protokoll ZK-EXP kann der Prover dem Verifier beweisen, dass eine Exponentiation korrekt durchgeführt wurde. Die Werte $h, h'$ und $e$ sowie der verwendete ElGamal Public Key $\mathsf{pk}_{EG}$ sind dem Verifier und dem Prover bekannt. Der Prover will nun dem Verifier beweisen, dass für ein $\alpha$ gilt, dass $h^\alpha = h'$ und $e$ eine Verschlüsselung von $\alpha$ bezüglich $\mathsf{pk}_{EG}$ ist. Der Prover kennt $\alpha$ und den Wert $r$, sodass $\mathsf{Enc}_{\mathsf{pk}_{EG}}(\alpha, r) = e$. Das Protokoll hierfür wird von Hazay et al. [14] beschrieben.

#### ZK-RSA

Der Prover kann dem Verifier mit dem Protokoll ZK-RSA beweisen, dass $N$ und $\phi(N)$ teilerfremd sind. Dabei kennen beide Parteien $N$, $\phi(N)$ ist aber nur dem Prover bekannt. Ein solches Protokoll wird in dem Paper von Hazay et al. [14] vorgestellt.

#### ZK-VERLIN

Das Protokoll ZK-VERLIN erlaubt dem Prover zu zeigen, dass ein Paillier-Ciphertext aus einem dem Prover bekannten Wert und beiden Parteien bekannten Ciphertexten berechnet wurde. Die Parteien kennen dabei einen öffentlichen Paillier-Schlüssel $N_P$ und drei Ciphertexte $c, c'$ und $c_x$. Der Prover kennt zusätzlich $x, x', x''$ und $r_x$. Der Prover zeigt nun mithilfe des Protokolls, dass $c_x = c^x \cdot (c')^{x'} \cdot \mathsf{Enc}_{\mathsf{pk}}(x'', r_x)$ ist. Hazay et al. beschreiben dieses Protokoll in [14].

#### ZK-MOD

Das Protokoll ZK-MOD dient dem Beweis, dass für Klartexte $x$ und $y$ und eine öffentlich bekannte Primzahl $\alpha$ die Kongruenz $x \equiv y \mod \alpha$ gilt. Die Ciphertexte zu $x$ und $y$, sowie $\alpha$ sind beiden Parteien bekannt, der Prover kennt zusätzlich $x$, $y$ und die bei der Verschlüsselung verwendeten Zufallswerte $r_x$ und $r_y$. Das Protokoll wird von Hazay et al. [14] vorgestellt.

# 3 Paillier-Kryptosystem mit geteiltem Secret Key

Für die Protokolle in Kapitel 4 wird ein Paillier-Schlüsselpaar benötigt, bei welchem der Public Key allen Parteien bekannt ist, während der Secret Key unter den Parteien aufgeteilt ist. In diesem Kapitel wird nun beschrieben, wie ein solches Paillier-Schlüsselpaar von den Parteien generiert werden kann, sodass der Adversary keine Kenntnis über den Secret Key erlangt, wenn er nicht mehr Parteien als ein Threshold $t$ kontrolliert.

Alle Protokolle in diesem Kapitel erfordern die Sicherheit der verwendeten Verschlüsselungsverfahren und ZK Protokolle.

In den Abschnitten 3.1 und 3.2 wird zunächst das Threshold Paillier-Kryptosystem beschrieben. Die Schlüsselerzeugung erfordert ein RSA Modulus $N$. Die Erzeugung dieses Modulus wird in Abschnitt 3.3 vorgestellt und darauf aufbauend zeigt Abschnitt 3.4, wie ein Paillier-Schlüsselpaar von den Parteien berechnet werden kann. In Abschnitt 3.5 wird auf einige Schwächen der beschriebenen Schlüsselerzeugung eingegangen.

## 3.1 Threshold Kryptosystem

Ein Threshold Kryptosystem ist ein Public-Key-Kryptosystem, bei welchem der Secret Key von einer Menge von Parteien mit einem Threshold $t$ geteilt wird. Jede Partei kann Klartexte verschlüsseln und eine Menge von $t + 1$ Parteien kann mithilfe eines Decrypt-Protokolls Ciphertexte entschlüsseln. Für MPC sind homomorphe Threshold Kryptosysteme interessant, da mit ihnen auf den Ciphertexten Berechnungen ausgeführt werden können, deren Ergebnis durch kooperierende Parteien veröffentlicht werden kann.

## 3.2 Threshold Paillier-Kryptosystem

Das in diesem Abschnitt vorgestellte Threshold Paillier-Kryptosystem basiert auf Damgard und Jurik [11] und nutzt die Schlüsselerzeugung wie von Hazay et al. [14] beschrieben. Es bildet die Basis für die im nächsten Kapitel beschriebenen Protokolle.

### 3.2.1 Schlüsselerzeugung

Der Public Key $\mathsf{pk}_{Paillier}$ für das Threshold Paillier-Kryptosystem ist ein RSA Modulus $N$. Es gilt also $N = pq$ für zwei Primzahlen $p$ und $q$ mit gleicher Bitlänge. Der geteilte Secret Key $\mathsf{sk}_{Paillier}$ ist, anders als beim einfachen Paillier, ein $d$ mit $d \equiv 0 \mod \phi(N)$ und $d \equiv 1 \mod N$. Dieser andere Schlüssel ist notwendig, um die Privacy des Schlüssels beim Entschlüsseln sicherzustellen.

Der geteilte Schlüssel hat das Threshold $t$, das heißt, dass Mengen von $t + 1$ Parteien entschlüsseln können, während Mengen von $t$ oder weniger Parteien keine Informationen über den Schlüssel oder den Klartext zu einem Ciphertext erlangen können. In Abschnitt 3.3 wird gezeigt, wie ein RSA Modulus für den Public Key generiert werden kann und darauf aufbauend wird in Abschnitt 3.4 das Protokoll PAIL-KGEN für die Erstellung des Schlüsselpaars beschrieben.

### 3.2.2 Verschlüsselung

Jede Partei kann mithilfe des öffentlichen Schlüssels $N$ einen Wert $m \in \mathbb{Z}_N$ verschlüsseln. Dafür wählt sie einen zufälligen Wert $r \in \mathbb{Z}_N^*$ und berechnet den Ciphertext

$$c = (1 + N)^m r^N \mod N^2$$

wie bei der Verschlüsselung des einfachen Paillier.

### 3.2.3 Entschlüsselung

Der Algorithmus PAIL-DEC dient dem Entschlüsseln eines Paillier-Ciphertexts mithilfe des geteilten Paillier Secret Keys $d$. Eine Entschlüsselung des Ciphertexts $c$ des Klartexts $m$ kann mit diesem Schlüssel als

$$m = \frac{c^d \mod N^2 - 1}{N} \mod N$$

berechnet werden, wie Damgard und Jurik [11] für einen allgemeineren Fall zeigen.

Jede Partei $P_i$ kennt hierfür eine Shamir Share $[d]_{\mathbb{Z},i}$ des Schlüssels $d$, also $[d]_{\mathbb{Z},i} \equiv f(i) \mod \phi(N)N$, wobei $f$ ein Polynom des Grads $t$ mit $f(0) = d$ ist. Für die Entschlüsselung müssen nun $t + 1$ Parteien kooperieren. Die Menge dieser Parteien wird in diesem Protokoll mit $S$ bezeichnet. Zusätzlich kennt jede Partei den zu entschlüsselnden Ciphertext $c$, den öffentlichen Paillier-Schlüssel $N$ und für jede Partei $P_i$ eine ElGamal-Verschlüsselung von $[d]_{\mathbb{Z},i}$, welche für den Beweis der korrekten Exponentiation verwendet wird. Nach einer erfolgreichen Ausführung des Protokolls kennt jede Partei den Klartext $m$ zu dem Ciphertext $c$.

Die Parteien fügen $d$ im Exponenten durch Lagrange-Interpolation zusammen. Um sicherzustellen, dass der Exponent eine Ganzzahl ist, werden die Lagrange-Gewichte mit $n!$ multipliziert und sind damit

$$\lambda_{x,i}^S = n! \prod_{j \in S \setminus \{i\}} \frac{x - i}{j - i}.$$

Für die Interpolation trägt dann jede Partei $P_i$ aus $S$ den Faktor $c^{[d]_{\mathbb{Z},i}}$ bei. Damit kann jede Partei $c^{n!d}$ berechnen. Aufgrund der Homomorphie der Paillier-Verschlüsselung ist $c^{n!}$ eine Verschlüsselung von $n! \cdot m$. Entsprechend berechnen die Parteien zunächst die Entschlüsselung $m' = n! \cdot m$ wie oben beschrieben und bestimmen dann durch die Multiplikation mit $(n!)^{-1}$, dem Inversen von $n!$ modulo $N$, den ursprünglichen Klartext $m$.

**Protokoll 3.1** PAIL-DECRYPT

$m \leftarrow$ PAIL-DECRYPT$([d]_{\mathbb{Z}}, c, N, S)$

1 : **for** $P_i \in S$ **do**

2 : $\quad c_i = c^{[d]_{\mathbb{Z},i}} \mod N^2$

3 : $\quad$ BROADCAST $(i, c_i)$

4 : $\quad P_i$ beweist mittels ZK-EXP, dass die Exponentiation korrekt durchgeführt wurde

5 : **endfor**

6 : $c' = \prod_{i \in S} c_i^{\lambda^S_{0,i}} \mod N^2$

7 : $m' = \frac{c' - 1}{N}$

8 : $m = m' \cdot (n!)^{-1} \mod N$

**Lemma 3.2.1**
*Für jeden Ciphertext c zu einem Klartext m ist die Ausgabe des Protokolls PAIL-DECRYPT gleich m.*

**Beweis** Die Lagrange Interpolation von $d$ wird in den Exponenten als

$$\sum_{i \in S} [d]_{\mathbb{Z},i} \, \lambda^s_{0,i} = \sum_{i \in S} f(i) \lambda^s_{0,i} = n! f(0) = n! d$$

zusammengesetzt und somit gilt

$$c' \equiv \prod_{i \in S} \left( c_i^{[d]_{\mathbb{Z},i}} \right)^{\lambda^s_{0,i}} \equiv c^{\sum_{i \in S} [d]_{\mathbb{Z},i} \lambda^s_{0,i}} \equiv c^{n! d} \mod N^2.$$

Die Partei $P_i$ muss dabei jeweils den korrekten Wert $c^{[d]_{\mathbb{Z},i}}$ beitragen, da sie sonst den Beweis über die korrekte Exponentiation mittels ZKP-EXP nicht erbringen kann. Da $c^{n!}$ aufgrund der Homomorphie-Eigenschaft eine Verschlüsselung von $m \cdot n!$ ist, ergibt sich damit

$$m' = \frac{c' - 1}{N} \equiv n! m \mod n.$$

Damit können die Parteien $m$ durch die Multiplikation von $m'$ mit dem Inversen von $n!$ modulo $N$ berechnen. [1] □

Wenn der Adversary die Kontrolle über $t + 1$ oder mehr Parteien $P_i$ hat, kann er durch Interpolation auf deren Shares den Schlüssel $d$ berechnen.

**Lemma 3.2.2**
*Die Privacy des Secret Keys d bleibt durch das Protokoll PAIL-DECRYPT gewahrt, wenn der Adversary höchstens t Parteien kontrolliert.*

[1] $n!$ ist invertierbar modulo $N$, solange die Anzahl der Parteien $n$ und $N$ teilerfremd sind. Das ist der Fall, wenn $n$ kleiner als beide Primfaktoren von $N$ ist. Dass dies gilt, kann angenommen werden, da die Primfaktoren von $N$ sehr groß gewählt werden.

**Beweisskizze** Die Parteien können $d$ aus höchstens $t$ Shares nicht durch Interpolation rekonstruieren. Da aus $c^{[d]_{\mathbb{Z},i}} \mod N^2$ keine neue Information über die Share $[d]_{\mathbb{Z},i}$ der Partei $P_i$ gewonnen werden kann, wahrt das Protokoll die Privacy des Secret Keys $d$. □

Die Parteien können auch für nur eine einzelne Partei aus $S$ entschlüsseln. Dazu senden sie die $c_i$ an diese Partei und führen die Beweise für die Korrektheit nur gegenüber dieser Partei.

Als eine weitere Variation können alle $n$ Parteien zunächst ihr $c_i$ mit dem Beweis beitragen. Jede Partei wählt dann für die weitere Berechnung $t + 1$ der erhaltenen Werte $c_i \in T$ aus, wobei $T$ die Menge der $c_i$ ist, für welche der Beweis erfolgreich war. Diese Variante bietet Guaranteed Output Delivery, wenn $t$ kleiner als $n/2$ ist und der Adversary höchstens eine Minderheit der Parteien kontrolliert. In diesem Fall tragen $t + 1$ ehrliche Parteien korrekte Werte bei und der Adversary kann, auch wenn er keine oder falsche Werte beiträgt, die erfolgreiche Ausführung des Protokolls nicht verhindern.

## 3.3 Erzeugung eines RSA Modulus

Das Protokoll zur RSA Modulus Erzeugung basiert auf den Protokollen von Boneh und Franklin [4]. Es wurde von Hazay et al. [14] erweitert, sodass Korrektheit und Privacy der Shares der ehrlichen Parteien auch dann gewährleistet ist, wenn ein malicious Adversary maximal $n - 1$ Parteien, also alle Parteien bis auf eine kontrolliert. In diesem Abschnitt wird eine Übersicht über das Protokoll vorgestellt. Im Folgenden werden dann die einzelnen Bestandteile beschrieben.

Das Protokoll generiert ein RSA Modulus $N = pq$ mit Primzahlen $p$ und $q$ und je ein additives Sharing von $p$ und $q$. Es gilt dabei $p \equiv q \equiv 3 \mod 4$. Dies ist eine Anforderung aus dem Protokoll DPRIM. Jede Partei $P_i$ kennt nach der Ausführung also $N$, $[p]_{+,i}$ und $[q]_{+,i}$, sodass $N = \left(\sum_{j=1}^{n} [p]_{+,i}\right)\left(\sum_{j=1}^{n} [q]_{+,i}\right)$ ist.

Das Protokoll erhält als Eingabe eine natürliche Zahl $l$, welche die Größe der Primzahlen und damit die Größenordnung von $N$ bestimmt, sowie einen Schwellwert $B$ für das Abgleichen mit anderen Primzahlen in dem Trial Division Schritt.

Das Protokoll RSA-MOD-GEN beginnt mit der Erstellung von Schlüsseln für die Verifikation der weiteren Berechnungen mittels K-SETUP. Dann werden mit CANDIDATE-GEN zwei Sharings von zufälligen Werten $p$ und $q$ generiert. Das Produkt dieser Werte $N$ wird mit dem Protokoll MULTIPLY berechnet. Anschließend wird mit DPRIM geprüft, ob es sich bei $N$ um das Produkt von zwei Primzahlen handelt. Wenn dies der Fall ist, müssen diese Primzahlen $p$ und $q$ sein. Das Protokoll liefert bei einer erfolgreichen Ausführung mit einer hohen Wahrscheinlichkeit, abhängig von der Anzahl der Ausführungen von DPRIM, einen korrekten RSA Modulus $N$ und je ein additives Sharing der Primfaktoren $p$ und $q$ von $N$.

Das Protokoll wahrt die Privacy der geheimen Ausgabewerte, wenn die Subprotokolle dies tun. Wenn das Protokoll abgebrochen wird, können alle zufälligen, für das Protokoll generierte Werte sowie alle versendeten Nachrichten offengelegt werden, da es keine Eingaben hat, deren Privacy geschützt werden muss.

**Protokoll 3.2** RSA-MOD-GEN

$(N, [p]_+, [q]_+) \leftarrow$ RSA-MOD-GEN$(B, l)$

1 : $\mathsf{pk}_{EG}, \mathsf{pk}^1_{Pa}, \mathsf{pk}^2_{Pa}, \ldots, \mathsf{pk}^n_{Pa} \leftarrow$ K-SETUP()
2 : $[p]_+ \leftarrow$ CANDITATE-GEN$(B, l)$
3 : $[q]_+ \leftarrow$ CANDITATE-GEN$(B, l)$
4 : $N \leftarrow$ MULTIPLY$([p]_+, [q]_+)$
5 : **if** DPRIM$(N, [p]_+, [q]_+)$ fails **then**
6 : **abort**
7 : **endif**

Da das verwendete additive Secret Sharing die Kooperation aller Parteien benötigt, kann der Adversary die Ausführung des Protokolls verhindern, indem er sich weigert korrekte Werte beizutragen. In diesem Fall muss das Protokoll abgebrochen werden. Das Protokoll erfüllt die Eigenschaft Guaranteed Output Delivery also nicht.

### 3.3.1 Vorbereitung von Schlüsseln

Mit dem Protokoll K-SETUP einigen sich die Parteien auf einige Parameter für die Berechnung des RSA Modulus.

Für die folgenden Protokolle benötigen die Parteien eine Gruppe $\mathbb{Z}_Q$ für eine große Primzahl Q und einen ElGamal Public Key $\mathsf{pk}_{EG}$ sowie einen additiv geteilten Secret Key $\mathsf{sk}_{EG}$ für Verschlüsselungen in dieser Gruppe. Dafür wählen die Parteien eine große Primzahl $Q$ und einen Generator $g$ für die Gruppe $\mathbb{Z}_Q$. Zusätzlich hat jede Partei $P_i$ einen Paillier Public Key $\mathsf{pk}^i_{Pa}$, der den anderen Parteien bekannt ist, und zu dem nur $P_i$ den geheimen Schlüssel $\mathsf{sk}^i_{Pa}$ kennt. Diese Schlüssel werden mit dem Protokoll K-SETUP generiert und ausgetauscht.

**Lemma 3.3.1**
*Die Geheimhaltung des geteilten ElGamal Secret Keys und der Paillier Secret Keys der Parteien wird durch das Protokoll K-SETUP sichergestellt.*

**Beweis** Die Parteien broadcasten jeweils $g^{[\mathsf{sk}_{EG}]_{+,i}}$. Unter der Annahme, dass die Bestimmung des diskreten Logarithmus in $\mathbb{Z}_Q$ hinreichend schwer ist, kann daraus keine Information zu den Shares des Secret Key $\left[\mathsf{sk}_{EG}\right]_{+,i}$ gewonnen werden. Die Geheimhaltung des ElGamal Secret Keys ist also sichergestellt. Die Parteien broadcasten jeweils ihren Paillier Public Key. Unter der Annahme, dass die Faktorisierung der Public Keys $N_i$ schwer ist, kann ein Adversary daraus nicht den zugehörigen Secret Key $\mathsf{pk}^i_{Pa}$ berechnen. □

Jede Partei $P_i$ hält nach der Ausführung von K-SETUP eine Share $\left[\mathsf{sk}_{EG}\right]_{+,i}$, sodass $\mathsf{sk}_{EG} = \sum_{i=1}^n x_i$ ist. Damit wird die Entschlüsselung von $(a, b)$ wie folgt berechnet: Jede Partei $P_i$ broadcastet $s_i = (a)^{x_i}$ und beweist die Korrektheit der Berechnung mit ZK-DH. Dann berechnen die Parteien jeweils $(a)^{\mathsf{sk}_{EG}} = \prod_{i=1}^n s_i$ und berechnen damit $\mathsf{Dec}_{\mathsf{sk}_{EG}}(a, b) = b \cdot \left(a^{\mathsf{sk}_{EG}}\right)^{-1}$. Die Entschlüsselung des Ciphertexts $c = (a, b)$ wird im Folgenden mit

$$m \leftarrow \text{EG-DEC}(c)$$

**Protokoll 3.3** K-SETUP

$\mathsf{pk}_{EG}, \mathsf{pk}^1_{Pa}, \mathsf{pk}^2_{Pa}, \ldots, \mathsf{pk}^n_{Pa} \leftarrow$ K-SETUP()

1 : Parteien einigen sich auf eine Gruppe mit großer Primzahl $Q$ als Ordnung und einen Generator $g$
2 : **for** $i = 1, \ldots, n$ **do**
3 : $P_j$ wählt $\left[\mathsf{sk}_{EG}\right]_{+,i}$ gleichverteilt zufällig aus $\mathbb{Z}_Q$
4 : $h_i = g^{\left[\mathsf{sk}_{EG}\right]_{+,i}}$
5 : BROADCAST $(i, h_i)$
6 : $P_i$ beweist jeder anderen Partei Kenntnis eines diskreten Logarithmus von $h_i$ mittels ZK-DL
7 : **endfor**
8 : $h = \prod_{i=1}^{n} h_i$
9 : $\mathsf{pk}_{EG} = (g, h)$
10 : **for** $i = 1, \ldots, n$ **do**
11 : $P_i$ wählt ein Paillier-Schlüsselpaar mit $\mathsf{pk}^i_{Pa} = N_i >> Q^2$
12 : BROADCAST $\left(i, \mathsf{pk}^i_{Pa}\right)$
13 : $P_i$ beweist jeder anderen Partei mit ZK-RSA, dass $\mathsf{pk}^i_{Pa}$ wohlgeformt ist
14 : **endfor**

notiert.

### 3.3.2 Kandidaten bestimmen

Das Protokoll CANDIDATE-GEN dient dem Erstellen eines additiven Sharings für einen zufälligen Wert $p$, einer potenziellen Primzahl.

Als Eingabe erhält das Protokoll eine minimale Bitlänge $l$ für die gewünschte Primzahl und eine obere Schranke $B$ für das Testen gegen bekannte Primzahlen im Trial Division Protokoll. Nach Ausführung des Protokolls kennt jede Partei $P_i$ eine Share $[p]_{+,i}$, sodass der Primzahlkandidat $p$ kongruent $3 \mod 4$ ist. Zudem kennt sie für alle Parteien $P_j$ eine ElGamal-Verschlüsselung von $[p]_{+,j}$.

Jede Partei wählt in dem Protokoll zunächst einen zufälligen Wert $\overline{p_i}$. Die Share $[p]_{+,i}$ für Partei $P_1$ ist der Wert $4\overline{p_1} + 3$, während sich für die anderen Parteien $P_i$ die Share $4\overline{p_i}$ ergibt. Dadurch wird sichergestellt, dass $[p]_{+,1} \equiv 3 \mod 4$ und die übrigen $[p]_{+,i} \equiv 0 \mod 4$ sind.In den Zeilen 6 und 13 berechnet jede Partei jeweils $\mathsf{Enc}_{\mathsf{pk}_{EG}}\left([p]_{+,i}\right)$ mithilfe der Homomorphie des ElGamal-Verfahrens. Anschließend werden mittels des TRIAL-DIV-Protokolls Kandidaten aussortiert, die von einer Primzahl kleiner $B$ geteilt werden.

**Lemma 3.3.2**
*Nach Ausführung des Protokolls CANDIDATE-GEN kennt jede Partei eine additive Share eines Primzahlkandidaten p mit* $p \equiv 3 \mod 4$.

**Protokoll 3.4** CANDITATE-GEN

$[p]_+ \leftarrow$ CANDIDATE-GEN$(B, l)$

1 : $P_1$ wählt eine zufällige $(l-2)$-Bit lange Zahl $\overline{p}_1$
2 : BROADCAST $\left(1, \mathsf{Enc}_{\mathsf{pk}_{EG}}(\overline{p_1})\right)$
3 : $P_1$ beweist jeder anderen Partei mit ZK-BOUND, dass $\overline{p_1} < 2^{l-2}$
4 : $P_1$ beweist jeder anderen Partei mit ZK-ENC, dass sie den Klartext zu $\mathsf{Enc}_{\mathsf{pk}_{EG}}(\overline{p_1})$ kennt
5 : $[p]_{+,1} = 4\overline{p}_1 + 3$
6 : $e_1 = (\mathsf{Enc}_{\mathsf{pk}_{EG}}(\overline{p_1}))^4 \cdot \mathsf{Enc}_{\mathsf{pk}_{EG}}(3)$
7 : **for** $i = 2, \ldots, n$ **do**
8 : $P_i$ wählt eine zufällige $(l-2)$-Bit lange Zahl $\overline{p}_i$
9 : BROADCAST $\left(i, \mathsf{Enc}_{\mathsf{pk}_{EG}}(\overline{p_i})\right)$
10 : $P_i$ beweist jeder anderen Partei mit ZK-BOUND, dass $\overline{p_i} < 2^{l-2}$
11 : $P_i$ beweist jeder anderen Partei mit ZK-ENC, dass sie den Klartext zu $\mathsf{Enc}_{\mathsf{pk}_{EG}}(\overline{p_i})$ kennt
12 : $[p]_{+,i} = 4\overline{p_i}$
13 : $e_i = (\mathsf{Enc}_{\mathsf{pk}_{EG}}(\overline{p_i}))^4$
14 : **endfor**
15 : **if** TRIAL-DIV(B, $[p]_+$) rejects $p$ **then**
16 : **abort**
17 : **endif**

**Beweis** Die Share von $P_1$ ist $4\overline{p_1} + 3$ und die Share jeder anderer Partei $P_i$ wird als $4\overline{p_i}$ berechnet, wobei $\overline{p_i}$ jeweils ein zufälliger von $P_i$ gewählter Wert ist. Somit gilt für den geteilten Wert $p$:

$$p = 4\overline{p_1} + 3 + \sum_{i=2}^{n} 4\overline{p_i} \equiv 3 \mod 4. \qquad \square$$

Es werden nur sichere Verschlüsselungen versendet, allerdings ist die Zufallsvariable zu dem gewählten Primzahlkandidaten $p$ nicht gleichverteilt, obwohl die einzelnen Summanden gleichverteilt zufällig gewählt werden. Die $l$ niedrigwertigsten Bits sind mit Ausnahme der 2 niedrigwertigsten Bits von $p$ jedoch gleichverteilt zufällig. Zudem weiß jede Partei $P_i$, dass $p > p_i$ und $p \equiv 3 \mod 4$ ist. Diese Einschränkungen von $p$ sind allerdings nicht ausreichend, um es dem Adversary zu erlauben den generierten Wert $N$ zu faktorisieren. Boneh und Franklin zeigen dies in ihrer Arbeit [4].

### 3.3.3 Trial Division

Mit dem Protokoll TRIAL-DIV werden Primzahl-Kandidaten gegen alle Primzahlen kleiner als einer Schranke $B$ geprüft. Teilt eine Primzahl $\alpha < B$ den Kandidaten $p$, so wird er verworfen. Dieses Protokoll dient der Effizienzsteigerung, da so Kandidaten, wenn kleine Primzahlen sie teilen, frühzeitig aussortiert werden.

**Protokoll 3.5** TRIAL-DIV

TRIAL-DIV$(B, [p]_+)$

1 : **for** Primzahl $\alpha < B$ **do**
2 : **for** $i = 1, \ldots, n$ **do**
3 : $c_{\alpha,i} = \mathsf{Enc}_{\mathsf{pk}_{EG}}\left([p]_{+,i} \mod \alpha\right)$
4 : BROADCAST $\left(i, c_{\alpha,i}\right)$
5 : $P_i$ beweist jeder Partei mittels ZK-MOD, dass die Berechnung korrekt durchgeführt wurde
6 : **endfor**
7 : $c_\alpha = \mathsf{Enc}_{\mathsf{pk}_{EG}}\left(\sum_{i=1}^{n}([p]_{+,i} \mod \alpha)\right)$
8 : **for** $k = 0, \ldots, n-1$ **do**
9 : $c_\alpha^{(k)} = c_\alpha \cdot \mathsf{Enc}_{\mathsf{pk}_{EG}}(-k\alpha, 0)$
10 : $\overline{c_\alpha^{(k)}} = \left(c_\alpha^{(k)}\right)^r$ // $r$ ist hier ein zufälliger geheimer Wert aus $\mathbb{Z}_Q \setminus \{0\}$
11 : **if** EG-DEC $\left(\overline{c_\alpha^{(k)}}\right) = 1$ **then**
12 : reject $p$
13 : **endif**
14 : **endfor**
15 : **endfor**

Die Parteien broadcasten für Primzahlen $\alpha < B$ jeweils einen ElGamal-Ciphertext des Rests ihrer Share nach Modulo Reduktion bezüglich der Primzahl $\alpha$. Mittels Homomorphie wird sie einen Ciphertext der Summe der Reste, maskieren diesen mit einem Zufallswert und decken ihn auf. Wenn die Entschlüsselung der Werte einem Vielfachen von $\alpha$ entspricht, wird der Primzahl-Kandidat verworfen.

Die Exponentiation mit einen zufälligen, geheimen Wert in Zeile 10 kann realisiert werden, indem jede Partei $c_\alpha^{(k)}$ mit einem privaten zufälligen Wert $r_i \in \mathbb{Z}_Q$ ungleich 0 potenziert und das Ergebnis broadcastet. Zusätzlich beweist jede Partei mittels ZK-DH jeder anderen, dass sie über Kenntnis eines diskreten Logarithmus der Elemente des Ciphertext-Tupels verfügt. Der Ciphertext $\overline{c_\alpha^{(k)}}$ ist dann das Produkt all dieser Nachrichten. Solange eine Partei einen zufälligen Wert ungleich 0 beiträgt, ist damit die Wahrscheinlichkeit, dass der Exponent $r = \sum_{i=1}^{n} r_i \in \mathbb{Z}_Q$ gleich 0 ist vernachlässigbar. Sollte dies dennoch eintreten, kann ein Kandidat fälschlicherweise verworfen werden.

Der Ciphertext $c_\alpha$ kann in Zeile 7 von jeder Partei über Homomorphie aus den $c_{\alpha,i}$ berechnet werden.

**Lemma 3.3.3**
*Das Protokoll TRIAL-DIV verwirft einen Kandidaten p, der von einer Primzahl $\alpha < B$ geteilt wird, immer.*

**Beweis** In Zeile 11 wird geprüft, ob die Entschlüsselung von $\overline{c_\alpha^{(k)}}$ gleich 1 ist, also ob

$$r\left(\sum_{i=1}^{n}\left([p]_{+,i} \mod \alpha\right) - k\alpha\right) = 0$$

für $k \in \{0, \ldots, n-1\}$ gilt. Ist $p \equiv 0 \mod \alpha$, so existiert für beliebige $r$ ein solches $k$ und $p$ wird immer verworfen. □

**Lemma 3.3.4**
*Das Protokoll TRIAL-DIV wahrt die Privacy des Eingabewerts p.*

**Beweis** Bis auf die Entschlüsselung von $\overline{c_\alpha^{(k)}}$ erhalten die Parteien nur sichere Verschlüsselungen und die Informationen aus den ZK-Protokollen. Die Entschlüsselung ergibt $g^{r\left(\sum_{i=1}^{n}([p]_{+,i} \mod \alpha) - k\alpha\right)}$ für ein geheimes $r \in \mathbb{Z}_Q$. Wenn $\sum_{i=1}^{n}([p]_{+,i} \mod \alpha) - k\alpha \neq 0$ ist, ist dieser Wert ebenfalls zufällig. Somit werden keine Informationen preisgegeben, außer ob $p \equiv 0 \mod \alpha$ für eine Primzahl $\alpha < B$.□

### 3.3.4 Multiplikation auf additiven Sharings

Das Protokoll MULTIPLY dient der Berechnung des Produkts der Werte von additiven Sharings. Es erhält additive Sharings der Faktoren als Eingabe zusammen mit deren ElGamal-Verschlüsselungen. Nach der Ausführung ist das Produkt allen Parteien bekannt.

Das Produkt $N$ wird als $pq = \sum_{i=1}^{n}\left(\sum_{j=1}^{n}\left([p]_{+,i}\,[p]_{+,j}\right)\right)$ berechnet. Dazu berechnet jedes Paar von Parteien $P_i$ und $P_j$ zunächst ein additives Sharing des Produkts ihrer Shares, wobei nur diese zwei Parteien je eine Share des Werts kennen. Aus den bekannten Shares berechnet $P_i$ dann eine additive Share $[N]_{+,i}$ von $N$, welche abschließend für alle Parteien aufgedeckt wird. Damit können alle Parteien den Wert $N$ als Summe der $[N]_{+,i}$ berechnen.

**Lemma 3.3.5**
*Auf Eingabe von additiven Sharings $[p]_+$ und $[q]_+$ gibt das Protokoll MULTIPLY den Wert $N = pq$ aus.*

**Beweis** In den Zeilen 1 bis 8 berechnen alle Paare von Parteien $P_i$ und $P_j$ je ein additives Sharing von $[p]_{+,i}\,[q]_{+,j}$. Danach kennt Partei $P_i$ den Wert $s_{i,j}^{(i)}$ und Partei $P_j$ den Wert $s_{i,j}^{(j)}$, sodass $[p]_{+,i}\,[q]_{+,j} = s_{i,j}^{(i)} + s_{i,j}^{(j)}$. Mithilfe der Homomorphie der Verschlüsselung wird nun in Zeile 10 jeweils eine Verschlüsselung von $[p]_{+,i}\,[q]_{+,j}$ berechnet. Darauffolgend berechnen die einzelnen Parteien eine Verschlüsselung von $[p]_{+,i}\,[q]_{+,j} - (s_{i,j}^{(i)} + s_{i,j}^{(j)})$ und entschlüsseln diese. Sie prüfen, ob die Differenz 0 ist, also ob die berechnete Entschlüsselung $g^0 = 1$ ergibt. Ist dies nicht der Fall, wird das Protokoll abgebrochen, da eine Partei vom Protokoll abgewichen sein muss. So kann jede Partei sicher sein, dass $[p]_{+,i}\,[q]_{+,j} = s_{i,j}^{(i)} + s_{i,j}^{(j)}$ ist. Aus den ihr bekannten Werten $s_{i,j}^{(i)}$ und $s_{j,i}^{(i)}$ für jede Partei $P_j$ kann $P_i$ nun $[N]_{+,i} = \sum_{j=1}^{n} s_{i,j}^{(i)} + s_{j,i}^{(i)}$ berechnen und broadcasten. Alle Parteien können eine Verschlüsselung von $[N]_{+,i}$ mittels Homomorphie berechnen, entschlüsseln diese gemeinsam und prüfen sie gegen $g^{[N]_{+,i}}$. Wenn diese Prüfung fehlschlägt, muss eine Partei vom Protokoll abgewichen sein und die Parteien brechen das Protokoll ab. Abschließend kann jede Partei

$$N = \sum_{i=1}^{n}[N]_{+,i} = \sum_{i=1}^{n}\sum_{j=1}^{n} s_{i,j}^{(i)} + s_{j,i}^{(i)} = \sum_{i=1}^{n}\sum_{j=1}^{n} s_{i,j}^{(i)} + s_{i,j}^{(j)} = \sum_{i=1}^{n}\sum_{j=1}^{n}[p]_{+,i}\,[q]_{+,j} = pq$$

berechnen. □

**Protokoll 3.6** MULTIPLY

$N \leftarrow \text{MULTIPLY}([p]_+, [q]_+)$

1 : **for** $i = 1, \ldots, n; j = 1, \ldots, n$ **do**
2 : $P_j$ wählt einen zufälligen Wert $-s_{i,j}^{(j)} \in \mathbb{Z}_Q$
3 : BROADCAST $\left(j, \mathsf{Enc}_{\mathsf{pk}_{EG}}\left(-s_{i,j}^{(j)}\right)\right)$
4 : $P_j$ beweist jeder anderen Partei mittels ZK-ENC, dass sie den Klartext kennt
5 : **endfor**
6 : **for** $i = 1, \ldots, n; j = 1, \ldots, n$ **do**
7 : $s_{i,j}^{(i)} \leftarrow \text{MULT-SHARES}(i, j, -s_{i,j}^{(j)}, [p]_+, [q]_+)$
8 : **endfor**
9 : **for** $i = 1, \ldots, n; j = 1, \ldots, n$ **do**
10 : BROADCAST $\left(i, \mathsf{Enc}_{\mathsf{pk}_{EG}}\left(q_j\right)^{p_i} \cdot \mathsf{Enc}_{\mathsf{pk}_{EG}}(0)\right)$
11 : $P_i$ beweist jeder anderen Partei die Korrektheit der neuen Verschlüsselung mit ZK-MULT
12 : $f_{i,j} = \mathsf{Enc}_{\mathsf{pk}_{EG}}\left(p_i q_j - \left(s_{i,j}^{(i)} + s_{i,j}^{(j)}\right)\right)$
13 : **if** EG-DEC$(f_{i,j}) \neq 1$ **then**
14 : **abort**
15 : **endif**
16 : **endfor**
17 : **for** $i = 1, \ldots, n$ **do**
18 : $[N]_{+,i} = \sum_{j=1}^{n} s_{i,j}^{(i)} + s_{j,i}^{(i)}$
19 : $c_i = \mathsf{Enc}_{\mathsf{pk}_{EG}}\left([N]_{+,i}\right)$
20 : BROADCAST $\left(i, [N]_{+,i}\right)$
21 : **if** EG-DEC$(c_i) \neq g^{[N]_{+,i}}$ **then**
22 : **abort**
23 : **endif**
24 : **endfor**
25 : $N = \sum_{i=1}^{n} [N]_{+,i} \mod Q$

**Lemma 3.3.6**
*Die Privacy der Eingabewerte p und q wird durch das Protokoll MULTIPLY gewahrt, wenn das Protokoll nicht abbricht.*

**Beweis** Die Werte $[N]_{+,i}$ geben keine neuen Informationen über die $[p]_{+,i}\,[q]_{+,j}$ preis, da die Shares von $[p]_{+,i}\,[q]_{+,j}$ jeweils einen beliebigen Wert aus $\mathbb{Z}_Q$ annehmen können. Solange eine Partei $P_i$ ihre Shares $s_{i,j}^{(i)}$ und $s_{j,i}^{(i)}$ geheim hält, erhalten die anderen Parteien keine Informationen über $[p]_{+,i}$ und $[q]_{+,i}$. Ansonsten werden nur verschlüsselte Werte versendet, die der Adversary nicht entschlüsseln kann. In Zeile 13 wird jeweils $[p]_{+,i}\,[q]_{+,j} - (s_{i,j}^{(i)} + s_{i,j}^{(j)})$ aufgedeckt. Das Protokoll bricht ab, wenn die Entschlüsselung nicht 1 ergibt. Ansonsten ist der Wert

$$[p]_{+,i}\,[q]_{+,j} - (s_{i,j}^{(i)} + s_{i,j}^{(j)}) = [p]_{+,i}\,[q]_{+,j} - [p]_{+,i}\,[q]_{+,j} = 0,$$

wie für alle möglichen Eingaben gelten muss. Somit gibt das Protokoll keine Informationen über $p$ und $q$ preis. □

In späteren Protokollen wird auch

$$[c]_+ \leftarrow \text{MULTIPLY}([a]_+, [b]_+)$$

geschrieben. Das Protokoll entspricht dem vorgestellten MULTIPLY Protokoll ohne die Zeilen 20 bis 25. Die Ausgabe ist dann ein Sharing des Produkts zusammen mit ElGamal-Verschlüsselungen für jede Share.

#### MULT-SHARES

Mit dem Subprotokoll MULT-SHARES von MULTIPLY berechnen $P_i$ und $P_j$ zusammen zu dem $-s_{i,j}^{(j)} \in \mathbb{Z}_Q$ von $P_j$ ein $s_{i,j}^{(i)} \in \mathbb{Z}_Q$, welches Partei $P_i$ erhält. Es soll gelten, dass $s_{i,j}^{(i)} + s_{i,j}^{(j)} = [p]_{+,i}\,[q]_{+,j}$ ist. $s_{i,j}^{(i)}$ und $s_{i,j}^{(j)}$ bilden also ein additives Sharing von $[p]_{+,i}\,[q]_{+,j}$ in $\mathbb{Z}_Q$. Partei $P_i$ legt sich am Ende des Protokolls durch den Broadcast einer Verschlüsselung von $s_{i,j}^{(i)}$ auf den von ihr berechneten Wert fest. In diesem Protokoll werden die jeweiligen Paillier-Schlüsselpaare der einzelnen Parteien verwendet, da es eine vollständige Entschlüsselung der gesendeten Werte durch entsprechende Parteien erfordert.

Wenn $P_i = P_j$ ist, simuliert die entsprechende Partei das Protokoll, indem sie ein zufälliges Sharing für den ihr bekannten Wert $[p]_{+,i}\,[q]_{+,i}$ wählt und die Verschlüsselungen der Werte broadcastet.

Der Ciphertext $c'_{i,j}$ wird von $P_j$ berechnet. Dabei ist wichtig, dass $P_j$ zuvor sichergestellt hat, dass $-s_{i,j}^{(j)} + Q \cdot r_{i,j}$ hinreichend groß ist, um $q_j$ zu maskieren. Die Bitlänge $l$ von $r_{i,j}$ muss dafür der für die Generierung der Primzahlen verwendeten entsprechen. Deshalb muss $P_i$ die Beschränktheit der Größe des Klartexts zu $c_{p_i,j}$ gegenüber $P_j$ beweisen. Für die Anwendung von ZK-Verlin in Zeile 8 kann $P_j$ als zusätzlichen Ciphertext eine beliebige Verschlüsselung von 0 an $P_i$ senden.

**Lemma 3.3.7**
*Nach Ausführung des Protokolls MULT-SHARES kennt $P_i$ den Wert $s_{i,j}^{(i)}$ und $P_j$ den Wert $s_{i,j}^{(j)}$, sodass $s_{i,j}^{(i)} + s_{i,j}^{(j)} = [p]_{+,i}\,[q]_{+,j}$ ist.*

**Protokoll 3.7** MULT-SHARES

$s_{i,j}^{(i)} \leftarrow$ MULT-SHARES$(i, j, -s_{i,j}^{(j)}, [p]_+, [q]_+)$

1 : $c_{p_i,j} = \mathsf{Enc}_{\mathsf{pk}_{Pa}^i}\left([p]_{+,i}\right)$

2 : SEND $\left(i, j, c_{p_i,j}\right)$

3 : $P_i$ beweist $P_j$ Kenntnis des Klartexts mit ZK-ENC

4 : $P_i$ beweist $P_j$ mit ZK-BOUND, dass der Klartext kleiner als $2^l$ ist

5 : $P_j$ wählt einen zufälligen Wert $r_{i,j}$

6 : $c'_{i,j} = (c_{p_i,j})^{[q]_{+,j}} \cdot \mathsf{Enc}_{\mathsf{pk}_{Pa}^i}\left(-s_{i,j}^{(j)} + Q \cdot r_{i,j}\right)$

7 : SEND $\left(j, i, c'_{i,j}\right)$

8 : $P_j$ beweist $P_i$ mit ZK-VERLIN, dass $c'_{i,j}$ aus den bekannten Werten berechnet wurde

9 : $s_{i,j}^{(i)} = \mathsf{Dec}_{\mathsf{sk}_{Pa}^i}\left(c'_{i,j}\right) \mod Q$

10 : BROADCAST $\left(i, \mathsf{Enc}_{\mathsf{pk}_{EG}}\left(s_{i,j}^{(i)}\right)\right)$

11 : $P_i$ beweist jeder Partei mittels ZK-ENC, dass sie den Klartext kennt

**Beweis** $P_j$ kennt zu Begin des Protokolls $s_{i,j}^{(j)}$. Den Wert $s_{i,j}^{(i)}$ erhält $P_i$ aus der Entschlüsselung von $c_{i,j}$ gefolgt von einer Modulo Reduktion um $Q$. Dieser Ciphertext verschlüsselt den Wert $[p]_{+,i}[q]_{+,j} - s_{i,j}^{(j)} + Q \cdot r_{i,j}$. Folglich ist

$$s_{i,j}^{(i)} + s_{i,j}^{(j)} = \left([p]_{+,i}[q]_{+,j} - s_{i,j}^{(j)} + Q \cdot r_{i,j}\right) \mod Q + s_{i,j}^{(j)} = [p]_{+,i}[q]_{+,j}\,. \qquad \square$$

**Lemma 3.3.8**
*Das Protokoll MULT-SHARES gibt keine Informationen zu den Werten p und q preis.*

**Beweis** Bis auf $c'_{i,j}$ in Zeile 7 werden nur Ciphertexte versendet, welche der Empfänger nicht entschlüsseln kann. Der Ciphertext $c'_{i,j}$ kann von $P_i$ entschlüsselt werden, da er mit dem öffentlichen Paillier-Schlüssel der Partei verschlüsselt ist. Allerdings ist die Share $[q]_{+,j}$ von $j$ mit $-s_{i,j}^{(j)} + Q \cdot r_{i,j}$ hinreichend maskiert, sodass die Partei keine Informationen ableiten kann. $\square$

### 3.3.5 Biprimality Test

Das in diesem Abschnitt vorgestellte Protokoll DPRIM testet ein Produkt $N$ zweier Zahlen darauf, ob es sich um das Produkt zweier Primzahlen handelt. Das Protokoll akzeptiert Zahlen mit exakt zwei Primfaktoren $p$ und $q$ immer und Zahlen mit mehr als zwei Primfaktoren mit einer Wahrscheinlichkeit von maximal $\frac{1}{2}$. Um eine hinreichend geringe Wahrscheinlichkeit für einen Irrtum zu erreichen, muss das Protokoll folglich mehrmals ausgeführt werden.

Das Protokoll erhält als Eingabe ein öffentliches $N$ und die Partei $P_i$ kennt die Shares $[p]_{+,i}$ und $[q]_{+,i}$, sodass $N = \left(\sum [p]_{+,i}\right)\left(\sum [q]_{+,i}\right)$. Zudem kennt jede Partei für alle Parteien $P_i$ je eine ElGamal-Verschlüsselung von deren Shares $[p]_{+,i}$ beziehungsweise $[q]_{+,i}$. Für das Protokoll DPRIM

muss $p \equiv q \equiv 3 \mod 4$ gelten, wie mithilfe der vorigen Protokolle sichergestellt wird. Dabei sind $[p]_{+,1}$ und $[q]_{+,1}$, die Shares von Partei $P_1$, kongruent 3 modulo 4, für die übrigen Shares gilt $[p]_{+,i} \equiv [q]_{+,i} \equiv 0 \mod 4$.

**Definition 3.3.1**
*Das Jacobi-Symbol $\mathcal{J}_p$ für eine Primzahl $p > 2$ und $x \in \mathbb{F}_p^*$ ist definiert als*

$$\mathcal{J}_p(x) = \begin{cases} 1 & \text{wenn } x \text{ ein quadratischer Rest modulo } p \text{ ist.} \\ -1 & \text{wenn } x \text{ kein quadratischer Rest modulo } p \text{ ist.} \end{cases}$$

*Für ein $N$ mit $N = p_1^{a_1} p_2^{a_2} \cdots p_m^{a_m}$ für Primzahlen $p_i$ ist das Jacobi-Symbol*

$$\mathcal{J}_N(x) = \prod_{i=1}^{m} \left(\mathcal{J}_{p_i}(x)\right)^{a_i}.$$

Für den Test in den Zeilen 1 bis 12 wird ein $\gamma$ mit $\mathcal{J}_N(\gamma) = 1$ benötigt.

Der Test in den Zeilen 15 bis 26 wird auf der Gruppe $\mathbb{T}_N = (\mathbb{Z}_N[x]/(x^2+1))^*/\mathbb{Z}_N^*$ durchgeführt. Elemente aus $\mathbb{T}_N$ können als lineare Polynome $f(x) = ax + b$ aus $\mathbb{Z}_N[x]$ mit $\gcd(a,b) = 1$ aufgefasst werden, wobei Polynome $f, g \in \mathbb{Z}_N[x]$ dasselbe Element in $\mathbb{T}_N$ repräsentieren, wenn $f = cg$ für ein $c \in \mathbb{Z}_N^*$.

Die Wahl von $\gamma$ in Zeile 1 kann wie folgt geschehen: Jede Partei $P_i$ wählt einen zufälligen Wert $\overline{\gamma_i} \in \mathbb{Z}_N^*$ und broadcastet eine ElGamal-Verschlüsselung des Werts. Die Partei $P_i$ broadcastet dann jeweils den Wert $\overline{\gamma_i}$, die Parteien entschlüsseln den zugehörigen Ciphertext und prüfen, ob er mit dem Wert konsistent ist. Anschließend wird $\gamma = \prod_{i=1}^{n} \overline{\gamma_i}$ gesetzt und jede Partei berechnet $\mathcal{J}_N(\gamma)$. Ist $\mathcal{J}_N(\gamma) \neq 1$, wird $\gamma$ verworfen und der Vorgang wiederholt. Da die Parteien sich zunächst auf einen Wert festlegen, ist der Wert auch bei einem malicious Adversary zufällig, solange mindestens eine Partei einen zufälligen Wert beiträgt. Zudem können alle Parteien, da sie ihren Wert offenlegen müssen, nur gültige Werte beitragen. Analog kann auch ein zufälliger Wert aus $\mathbb{T}_N$ in Zeile 15 generiert werden.

Aufgrund der Homomorphie des verwendeten ElGamal-Verschlüsselungsverfahrens können die Parteien jeweils die Verschlüsselungen in den Zeilen 2, 9, 18 und 23 aus den bekannten Verschlüsselungen $\mathsf{Enc}_{\mathsf{pk}_{EG}}([p]_{+,i})$ und $\mathsf{Enc}_{\mathsf{pk}_{EG}}([q]_{+,i})$ berechnen.

**Lemma 3.3.9**
*Ein $N$ der Form $N = pq$ mit verschiedenen Primzahlen $p$ und $q$ und $p \equiv q \equiv 3 \mod 4$ wird von dem Protokoll DPRIM immer akzeptiert.*

Der folgende Beweis stammt aus der Arbeit von Boneh und Franklin [4].

**Beweis** In Zeile 12 wird geprüft, ob $\gamma' = \gamma_1 \cdots \gamma_n = \gamma'^{(N-p-q+1)/4} = \gamma'^{\phi(N)/4} \equiv \pm 1 \mod N$.

Seien $p$ und $q$ verschiedene Primzahlen. Da $\mathcal{J}_N(\gamma') = 1$, ist $\mathcal{J}_p(\gamma') = \mathcal{J}_q(\gamma') = \pm 1$. Da $p \equiv q \equiv 3 \mod 4$ ist, sind $(p-1)/2$ und $(q-1)/2$ ungerade, folglich ist

$$\gamma'^{\phi(N)/4} = \left(\gamma'^{(p-1)/2}\right)^{(q-1)/2} \equiv \mathcal{J}_p(\gamma')^{(q-1)/2} = \mathcal{J}_p(\gamma') \mod p$$

**Protokoll 3.8** DPRIM

DPRIM$(N, [p]_+, [q]_+)$

1: Die Parteien erstellen zusammen ein $\gamma \in \mathbb{Z}_N^*$ mit $\mathcal{J}_N(\gamma) = 1$
2: $e_1 = \mathsf{Enc}_{\mathsf{pk}_{EG}}\left(\frac{N+1-([p]_{+,1}+[q]_{+,1})}{4}\right)$
3: $\gamma_1 = \gamma^{\frac{N+1-([p]_{+,1}+[q]_{+,1})}{4}} \mod N$
4: BROADCAST$(1, \gamma_1)$
5: $P_1$ beweist jeder anderen Partei die Konsistenz von $\gamma_1$ und $e_1$ mittels ZK-EXP
6: **for** $i = 2, \ldots, n$ **do**
7: $\quad \gamma_i = \gamma^{\frac{-([p]_{+,i}+[q]_{+,i})}{4}} \mod N$
8: $\quad$ BROADCAST$(i, \gamma_i)$
9: $\quad e_i = \mathsf{Enc}_{\mathsf{pk}_{EG}}\left(\frac{-([p]_{+,i}+[q]_{+,i})}{4}\right)$
10: $\quad P_i$ beweist jeder anderen Partei die Konsistenz von $\gamma_i$ und $e_i$ mittels ZK-EXP
11: **endfor**
12: **if** $\gamma_1\gamma_2\cdots\gamma_n \mod N \notin \{-1, 1\}$ **then**
13: $\quad$ reject $N$
14: **endif**
15: Die Parteien wählen zusammen ein zufälliges $h \in \mathbb{T}_N$
16: $u_1 = h^{N+[p]_{+,1}+[q]_{+,1}+1}$ // Diese Berechnung wird in $\mathbb{T}_N$ durchgeführt
17: BROADCAST$(1, u_1)$
18: $d_1 = \mathsf{Enc}_{\mathsf{pk}_{EG}}\left(N + [p]_{+,1} + [q]_{+,1} + 1\right)$
19: $P_1$ beweist jeder anderen Partei die Konsistenz von $u_1$ und $d_1$ mittels ZK-EXP
20: **for** $i = 2, \ldots, n$ **do**
21: $\quad u_i = h^{[p]_{+,i}+[q]_{+,i}}$ // Diese Berechnung wird in $\mathbb{T}_N$ durchgeführt
22: $\quad$ BROADCAST$(i, u_i)$
23: $\quad d_i = \mathsf{Enc}_{\mathsf{pk}_{EG}}\left([p]_{+,i} + [q]_{+,i}\right)$
24: $\quad P_i$ beweist jeder anderen Partei die Konsistenz von $u_i$ und $d_i$ mittels ZK-EXP
25: **endfor**
26: **if** $u_1u_2\cdots u_n = 1$ **then**
27: $\quad$ accept $N$
28: **else**
29: $\quad$ reject $N$
30: **endif**

und analog $\gamma'^{\phi(N)/4} \equiv \mathcal{J}_q(\gamma') \mod q$. Somit gilt $\gamma'^{\phi(N)/4} \equiv \pm 1 \mod N$. Der Test in Zeile 12 ist also immer erfolgreich.

In Zeile 26 wird geprüft, ob $u = h^{N+p+q+1} = h^{(p+1)(q+1)} = 1$.

Seien $p$ und $q$ verschiedene Primzahlen und $p \equiv q \equiv 3 \mod 4$. Das Polynom $x^2 + 1$ hat somit keine Nullstelle in den Körpern $\mathbb{F}_p$ und $\mathbb{F}_q$. Der Körper $\mathbb{F}_p[x]/(x^2+1)$ der Polynome über $\mathbb{F}_p$ modulo $(x^2 + 1)$ ist also eine Erweiterung von $\mathbb{F}_p$, das heißt $\mathbb{F}_p$ ist eine Untergruppe von $\mathbb{F}_p[x]/(x^2+1)$, welche $\left|\mathbb{F}_p\right|^2$ Elemente hat. Ebenso ist der Körper $\mathbb{F}_q[x]/(x^2+1)$ eine Erweiterung von $\mathbb{F}_q$. Die Gruppe $\mathbb{T}_p = (\mathbb{Z}_p[x]/(x^2+1))^*/\mathbb{Z}_p^*$ hat folglich $p+1$ Elemente und analog ist $|\mathbb{T}_p| = q+1$. Nach dem Chinesischen Restsatz ist $\mathbb{T}_N = (\mathbb{Z}_N[x]/(x^2+1))^*/\mathbb{Z}_N^*$ isomorph zu $\mathbb{T}_p \times \mathbb{T}_q$. Folglich ist die Ordnung von $\mathbb{T}_N = (p+1)(q+1)$. Für alle $h \in \mathbb{T}_N$ gilt damit $h^{(p+1)(q+1)} = 1$. Der Test in Zeile 26 ist also immer erfolgreich. □

**Lemma 3.3.10**
*Wenn N nicht das Produkt zweier verschiedener Primzahlen ist, lehnt das Protokoll DPRIM N mit Wahrscheinlichkeit mindestens* $1/2$ *ab.*

Für dieses Lemma wird in dieser Arbeit nur eine Skizze des Beweises von Boneh und Franklin [4] angegeben.

**Beweisskizze** Angenommen die Faktoren $p$ und $q$ mit $p \equiv q \equiv 3 \mod N$ von $N$ sind keine Primzahlen. Sei $e = (N - p - q + 1)/4$ der Exponent, der sich bei der Multiplikation der $\gamma_i$ in Zeile 12 ergibt. Für den Beweis werden folgende Subgruppen von $\mathbb{Z}_N^*$ betrachtet:

$$G = \{\gamma \in \mathbb{Z}_N \mid \mathcal{J}_\gamma(N) = 1\},$$

die Gruppe der möglichen $\gamma$ in Zeile 1, und

$$H = \{\gamma \in G \mid \gamma^e = \pm 1\},$$

die Gruppe der potenzierten $\gamma$ die von DPRIM akzeptiert werden. Boneh und Franklin zeigen nun, dass $H$ eine Untergruppe und ungleich $G$ ist, wenn $N$ drei oder mehr Primzahlen hat, $p$ und $q$ nicht teilerfremd sind oder $p = r^d$ (symmetrisch auch $q$) für eine Primzahl $r$ und $d > 1$ ist und $q \not\equiv 1 \mod r^{d-1}$. Daraus folgt, dass in diesen Fällen $|H| \leq |G|/2$ gilt, sodass die Parteien abhängig von der zufälligen Wahl des $\gamma$ das zu prüfende $N$ mindestens mit der Wahrscheinlichkeit $1/2$ ablehnen.

Wenn $p = r_1^{d_1}$ und $q = r_2^{d_2}$ mit $d_1 > 1$ und $q \equiv 1 \mod r_1^{d_1-1}$, kann $H = G$ gelten. Für diesen Fall wird die Gruppe

$$H' = \{h \in \mathbb{T}_N \mid h^{(p+1)(q+1)} = 1\},$$

also die Gruppe der $h \in \mathbb{T}_N$ für welche der Test in Zeile 26 akzeptiert, betrachtet. Boneh und Franklin zeigen, dass $H'$ eine echte Untergruppe von $\mathbb{T}_N$ ist, folglich ist $|H'| \leq |\mathbb{T}_N|/2$ und $N$ wird mit der Wahrscheinlichkeit von $1/2$ abgelehnt. □

**Lemma 3.3.11**
*Das Protokoll DPRIM wahrt die Privacy der Werte p und q, wenn das Protokoll N akzeptiert.*

**Beweis** Jede Partei erhält jeweils die Werte $\gamma_i$ und $u_i$ von jeder anderen Partei $P_i$. Auch wenn der Adversary alle Parteien bis auf eine Partei $P_i$ kontrolliert, erhält er daraus keine neue Information, da bei einer erfolgreichen Ausführung das Produkt $u_1 u_2 \cdots u_n$ gleich 1 in $\mathbb{T}_N$ ist. Folglich muss der Wert der Partei $P_i$ das Inverse des Produkts der dem Adversary bekannten $u_j$ sein. Das Produkt $\gamma_1 \gamma_2 \cdots \gamma_n$ muss gleich $-1$ sein, wenn $\gamma$ kein quadratisches Residuum modulo $N$ ist, oder 1, wenn $\gamma$ ein quadratisches Residuum modulo $N$ ist. Das fehlende $\gamma_i$ ist also das Inverse des Produkts der Werte des Adversary, wenn $\gamma$ ein quadratisches Residuum modulo $N$ ist und sonst das Inverse mal $-1$. Somit lernt der Adversary keine zusätzlichen Informationen zu $p$ und $q$, wenn das Protokoll $N$ akzeptiert. □

## 3.4 Erzeugung eines Paillier-Schlüssels

In diesem Abschnitt wird die Erzeugung eines Threshold Paillier-Schlüsselpaars nach Hazay et al. [14] vorgestellt.

Das Protokoll PAIL-KGEN erhält als Eingabe einen Sicherheitsparameter $l$, ein Threshold für die Entschlüsselung $t$ und eine Schranke $B$ für den Trial Division Schritt bei der RSA Modulus Erzeugung. Nach der Ausführung des Protokolls kennt jede Partei einen Paillier Public Key $N$, eine Share eines zugehörigen Secret Keys $d$ über den ganzen Zahlen sowie eine ElGamal-Verschlüsselung der Share jeder Partei.

Die Korrektheit der Berechnungen der einzelnen Parteien stellt jede Partei mithilfe von paarweisen ZK Beweisen sicher. Dafür erfordert die Generierung eines Paillier-Schlüssels in allen Subprotokollen die Sicherheit der für die Beweise verwendeten Verschlüsselungen. Das Protokoll verwendet, wie das RSA Modulus Protokoll, additives Secret Sharing, welches bei der Rekonstruktion den Beitrag jeder Partei verlangt. Folglich kann der Adversary das Protokoll jederzeit an einer erfolgreichen Ausführung hindern, indem er sich weigert den korrekten Wert oder überhaupt einen Wert beizutragen. In diesem Fall können die Parteien die gesamte Schlüsselerzeugung nur abbrechen. Das Protokoll erfüllt also nicht die Eigenschaft Guaranteed Output Delivery. Da das Protokoll keine geheimen Werte als Eingabe hat, können die Parteien bei einem Abbruch alle Werte aufdecken und Parteien, welche vom Protokoll abgewichen sind, identifizieren.

Das Protokoll RSA-MOD-GEN gibt ein RSA Modulus $N$ und additive Sharings der Primfaktoren von $N$ aus. Aus den additiven Shares der Primfaktoren berechnen die Parteien zunächst ein additives Sharing eines Secret Keys. Damit $t+1 \leq n$ Parteien gemeinsam entschlüsseln können, transformieren die Parteien das Sharing anschließend in ein Shamir Sharing. Das Protokoll bietet also Korrektheit für einen malicious Adversary der höchstens $n-1$ Parteien kontrolliert. Da aber am Ende des Protokolls der Schlüssel von einer Menge von $t+1$ Parteien rekonstruiert werden kann, ist die Geheimhaltung des Schlüssels nur gegen einen Adversary, der höchstens $t$ Parteien kontrolliert, gewährleistet.

**Protokoll 3.9** PAIL-KGEN

$N, [d]_{\mathbb{Z}} \leftarrow$ PAIL-KGEN$(B, l, t)$

1 : $(N, [p]_+, [q]_+) \leftarrow$ RSA-MOD-GEN$(B, l)$
2 : $([\phi(N)]_+, [d]_+) \leftarrow$ ADD-KGEN$(N, [p]_+, [q]_+, l)$
3 : $[d]_{\mathbb{Z}} \leftarrow$ THRES-KGEN$(N, [\phi(N)]_+, [d]_+, t, l)$

### 3.4.1 Berechnung eines additiven Sharings eines Schlüssels

In diesem Abschnitt wird das Protokoll ADD-KGEN zur Berechnung eines additiven Sharings des Secret Keys $d$ vorgestellt. Es erhält als Eingabe $N = pq$ für Primzahlen $p$ und $q$, additive Sharings von $p$ und $q$ sowie einen Sicherheitsparameter. Nach der Ausführung des Protokolls kennt jede Partei eine additive Share des Secret Keys und je eine ElGamal-Verschlüsselung für die Shares der Parteien.

Das Protokoll basiert darauf, dass

$$\phi(N)\left(\phi(N)^{-1} \mod N\right) \equiv \begin{cases} 0 & \mod \phi(N) \\ 1 & \mod N \end{cases}$$

ist. Eine additive Share von $\phi(N)$ kann jede Partei lokal berechnen. In diesem Protokoll wird zunächst ein additives Sharing $w \equiv \phi(N)^{-1} \mod N$ berechnet. Dafür nutzen Hazay et al. eine Variante des Inversionsprotokolls von Bar-Ilan und Beaver [1]. Aus der Multiplikation von $w$ mit $\phi(N)$ ergibt sich dann der Secret Key $d$.

Die Berechnungen in den Zeilen 24 bis 32 führen die Parteien jeweils lokal auf den ihnen bekannten Werten aus. Die Verschlüsselungen $c_{w_i}$ können von jeder Partei berechnet werden, indem sie die Verschlüsselung von $t_i$ mit dem öffentlichen Wert $\overline{v}$ potenzieren. Das Protokoll bricht ab, wenn $v$ in $\mathbb{Z}_N$ nicht invertierbar ist. Dies ist der Fall, wenn $t \notin \mathbb{Z}_N^*$. In diesem Fall muss das Protokoll mit neuen Zufallswerten $t$ und $r$ wiederholt werden.

**Lemma 3.4.1**
*Das Protokoll ADD-KGEN gibt ein additives Sharing für einen Wert $d \equiv 0 \mod \phi(N) \wedge d \equiv 1 \mod N$ aus.*

**Beweis** Die Parteien berechnen zunächst ein additives Sharing von

$$\begin{aligned} w &= \sum_{i=1}^{n} [w]_{+,i} = \sum_{i=1}^{n} [t]_{+,i} \cdot \overline{v} \\ &= t \cdot \left( \left( \sum_{i=1}^{n} [u]_{+,i} + N \cdot [r]_{+,i} \right)^{-1} \mod N \right) \\ &= t \cdot \left( (t\phi(N) + N \cdot r)^{-1} \mod N \right) \end{aligned}$$

wobei $r \in \mathbb{Z}_{Nn2^l}$ und $t \in \mathbb{Z}_N^*$ zufällige Werte sind. Damit gilt

$$w = \left(\phi(N)^{-1}\right) \mod N + zN$$

**Protokoll 3.10** ADD-KGEN

$([\phi(n)]_+, [d]_+) \leftarrow \text{ADD-KGEN}(N, [p]_+, [q]_+, l)$

1: $[\phi(N)]_{+,1} = N + 1 - [p]_{+,1} - [q]_{+,1}$
2: **for** $i = 2, \ldots, n$ **do**
3: $\quad [\phi(N)]_{+,i} = -[p]_{+,i} - [q]_{+,i}$
4: **endfor**
5: **for** $i = 1, \ldots, n$ **do**
6: $\quad P_i$ wählt $[t]_{+,i}$ gleichverteilt zufällig aus $\mathbb{Z}_N$
7: $\quad c_{t_i} = \mathsf{Enc}_{\mathsf{pk}_{EG}}([t]_{+,i})$
8: $\quad \text{BROADCAST}(i, c_{t_i})$
9: $\quad P_i$ wählt $[r]_{+,i}$ gleichverteilt zufällig aus $\mathbb{Z}_{Nn2^l}$
10: $\quad c_{r_i} = \mathsf{Enc}_{\mathsf{pk}_{EG}}([r]_{+,i})$
11: $\quad \text{BROADCAST}(i, c_{r_i})$
12: $\quad P_i$ beweist allen Parteien Klartext-Kenntnis zu $c_{t_i}$ und $c_{r_i}$ mittels ZK-ENC
13: $\quad P_i$ beweist allen Parteien mit ZK-BOUND, dass die Werte im korrekten Bereich liegen
14: **endfor**
15: $[u]_+ \leftarrow \text{MULTIPLY}([t]_+, [\phi(N)]_+)$
16: **for** $i = 1, \ldots, n$ **do**
17: $\quad c_{u_i} = \mathsf{Enc}_{\mathsf{pk}_{EG}}([u]_{+,i})$
18: $\quad v_i = [u]_{+,i} + N \cdot [r]_{+,i}$
19: $\quad \text{BROADCAST}(i, v_i)$
20: $\quad$ **if** $\text{EG-DEC}\left(c_{u_i} c_{r_i}^N\right) \neq g^{v_i}$ **then**
21: $\quad\quad$ **abort**
22: $\quad$ **endif**
23: **endfor**
24: $v = \sum_{i=1}^{n} v_i$
25: **if** $v \notin \mathbb{Z}_N^*$ **then**
26: $\quad$ **abort**
27: **endif**
28: $\overline{v} = v^{-1} \mod N$
29: **for** $i = 1, \ldots, n$ **do**
30: $\quad [w]_{+,i} = [t]_{+,i} \cdot \overline{v}$
31: $\quad c_{w_i} = \mathsf{Enc}_{\mathsf{pk}_{EG}}([w]_{+,i})$
32: **endfor**
33: $[d]_+ \leftarrow \text{MULTIPLY}([\phi(N)]_+, [w]_+)$

**Protokoll 3.11** THRES-KGEN

[b] $[d]_{\mathbb{Z}} \leftarrow$ THRES-KGEN$(N, [\phi(N)]_+, [d]_+, t, l)$

1 : $[d_{\mathbb{Z}}]_+ \leftarrow$ INT-SHARE$([d]_+, l)$
2 : $([f(1)]_+, [f(2)]_+, \ldots, [f(n)]_+) \leftarrow$ POLY-SHARE$([d_{\mathbb{Z}}]_+, N, l, t)$
3 : $[d]_{\mathbb{Z}} \leftarrow$ ASSIGN-SHARE$(([f(1)]_+, [f(2)]_+, \ldots, [f(n)]_+), [\phi(N)]_+, N, l)$

für eine ganze Zahl $z$. Der Wert $d$ wird nun als Produkt aus $w$ und $\phi(N)$ berechnet, also ist

$$d = \phi(N) \cdot \left(\phi(N)^{-1} \mod N\right) + zN\phi(N).$$

Damit ist $d \equiv 0 \mod \phi(N)$ und $d \equiv 1 \mod N$. Folglich ist $d$ ein gültiger Threshold Paillier Secret Key zu dem Public Key $N$. □

**Lemma 3.4.2**
*Das Protokoll ADD-KGEN stellt die Privacy der Eingabe- und Ausgabewerte sicher.*

**Beweis** Mit Ausnahme der verschlüsselten Werte wird nur für jede Partei $v_i = [u]_{+,i} + N \cdot [r]_{+,i}$ gebroadcastet. Da $r$, wie $t$, ein geheimer zufälliger Wert ist und $u = t\phi(N)$ gilt, ist $v$ ein hinreichend zufälliger Wert. Die Entschlüsselung von $c_{u_i} c_{r_i}^N = g^{v_i}$ gibt dann ebenfalls keine Informationen preis. Somit wahrt das Protokoll die Privacy der Werte, solange MULTIPLY die Privacy der Werte wahrt. □

### 3.4.2 Transformation in ein Shamir Sharing

In diesem Abschnitt wird THRES-KGEN, das Protokoll für die Transformation des additiven Sharings von $d$ in ein Shamir Sharing mit dem Threshold $t$, vorgestellt.

Das Protokoll erhält als Eingabe den öffentlichen RSA Modulus $N$, die additiven Sharings $[\phi(N)]_+$ und $[d]_+$, ElGamal-Verschlüsselungen der Shares der Parteien sowie den Threshold $t$ und einen Sicherheitsparameter $l$. Nach der Ausführung kennt jede Partei $P_i$ eine Threshold Share $[d]_{\mathbb{Z},i}$ des Secret Keys $d$, sodass $t + 1$ Parteien entschlüsseln können, während $t$ Parteien keine Informationen über den Schlüssel oder die Entschlüsselungen erlangen können. Zusätzlich kennt jede Partei eine ElGamal-Verschlüsselung der Share jeder Partei.

Die Parteien berechnen zunächst ein additives Sharing von $d$ in den ganzen Zahlen und berechnen daraus ein Sharing von $d$ in den ganzen Zahlen mit dem Threshold $t$.

Im Folgenden sei $\ell_d$ die Bitlänge von $d$, also $\ell_d = \lceil \log(l) + 3\log(N) \rceil$.

#### Transformation in ein additives Sharing in den ganzen Zahlen

Das Subprotokoll INT-SHARE erlaubt den Parteien aus dem additiven Sharing von $d$ in $\mathbb{Z}_Q$ ein additives Sharing von $d$ in den ganzen Zahlen $\mathbb{Z}$ zu berechnen. Als Eingabe erhält das Protokoll zusätzlich einen Sicherheitsparameter $l$ und ElGamal-Verschlüsselungen der Shares von $d$. Das Protokoll gibt zusätzlich zu dem Sharing von $d$ Verschlüsselungen der ausgegebenen Shares aus.

**Protokoll 3.12** INT-SHARE

$[d_{\mathbb{Z}}]_+ \leftarrow$ INT-SHARE$([d]_+, l)$

1 : **for** $i = 1, \ldots, n$ **do**
2 : $P_i$ wählt $[r]_{+,i}$ gleichverteilt zufällig aus $\mathbb{Z}_{2^{\ell_d+l}}$
3 : $c_{r_i} = \mathsf{Enc}_{\mathsf{pk}_{EG}}\left([r]_{+,i}\right)$
4 : BROADCAST $\left(i, c_{r_i}\right)$
5 : $P_i$ beweist allen Parteien Klartext-Kenntnis zu $c_{r_i}$ mittels ZK-ENC
6 : $P_i$ beweist allen Parteien mit ZK-BOUND, dass $[r]_{+,i} < 2^{\ell_d+l}$
7 : **endfor**
8 : **for** $i = 1, \ldots, n$ **do**
9 : $[m]_{+,i} = [d]_{+,i} + [r]_{+,i}$
10 : BROADCAST $\left(i, [m]_{+,i}\right)$
11 : $e_i = \mathsf{Enc}_{\mathsf{pk}_{EG}}\left([m]_{+,i}\right)$
12 : **if** EG-DEC $(e_i) \neq g^{[m]_{+,i}}$ **then**
13 : **abort**
14 : **endif**
15 : **endfor**
16 : $[d_{\mathbb{Z}}]_{+,1} = \left(\sum_{j=1}^{n} [m]_{+,j} \mod Q\right) - [r]_{+,1}$
17 : $c_{d_{\mathbb{Z},1}} = \mathsf{Enc}_{\mathsf{pk}_{EG}}\left([d_{\mathbb{Z}}]_{+,1}\right)$
18 : **for** $i = 2, \ldots, n$ **do**
19 : $[d_{\mathbb{Z}}]_{+,i} = -[r]_{+,i}$
20 : $c_{d_{\mathbb{Z},i}} = \mathsf{Enc}_{\mathsf{pk}_{EG}}\left([d_{\mathbb{Z}}]_{+,i}\right)$
21 : **endfor**

Die Parteien maskieren jeweils ihre Share durch das Addieren mit einem von ihnen gewählten zufälligen Wert $[r]_{+,i} \in \mathbb{Z}_{2^{\ell_d+l}}$ und veröffentlichen den maskierten Wert $[m]_{+,i}$. Partei $P_1$ hat dann $\left(\sum_{j=1}^{n} [m]_{+,j} \mod Q\right) - [r]_{+,1}$ als Share, während die anderen Parteien $P_i$ jeweils ihre Share $[d_{\mathbb{Z}}]_{+,i} = -[r]_{+,i}$ kennen.

Die Verschlüsselungen in den Zeilen 11, 17 und 20 können die Parteien jeweils mithilfe der Homomorphie der Verschlüsselung aus bekannten Verschlüsselungen berechnen.

**Lemma 3.4.3**
*Der Wert des vom Protokoll INT-SHARE ausgegebenen Sharings $[d_{\mathbb{Z}}]_+$ entspricht dem Wert des Sharings aus der Eingabe.*

**Beweis** Es gilt

$$d_{\mathbb{Z}} = \left(\sum_{j=1}^{n} [m]_{+,j} \mod Q\right) + \sum_{j=1}^{n} -[r]_{+,j} = d + (r \mod Q) - r = d.$$

Folglich entspricht der Wert der Ausgabe dem Wert des Sharings aus der Eingabe. □

**Lemma 3.4.4**
*Das Protokoll INT-SHARE wahrt die Privacy der Eingabe- und Ausgabewerte.*

**Beweis** Die $[m]_{+,i}$ sind aufgrund der Addition mit $[r]_{+,i}$ zufällige Werte, daher geben sie keine Informationen über $d$ preis. Da die $[m]_{+,i}$ allen Parteien bekannt sind, können die Parteien aus der Entschlüsselung der $e_i$ keine Informationen gewinnen, außer ob $P_i$ den korrekten Wert gebroadcastet hat. Ansonsten werden nur sichere Verschlüsselungen veröffentlicht. Folglich bleibt die Privacy der Werte gewahrt. □

#### Erstellen des Polynoms für das Shamir Sharing

Mit dem Protokoll POLY-SHARE berechnen die Parteien je ein additives Sharing einer Shamir Share $f(j)$ von $d$ für jede Partei $P_j$ mit einem Polynom $f$ . Das Polynom $f$ hat Grad $t$ und es gilt $f(0) = d$.

Als Eingabe erhält das Protokoll das Sharing $[d_{\mathbb{Z}}]_+$, die Verschlüsselungen der Shares, den RSA Modulus $N$, den Threshold $t$ und einen Sicherheitsparameter $l$. Nach der Ausführung kennt jede Partei für jede Partei $P_j$ eine additive Share von $f(j)$ und eine ElGamal-Verschlüsselung dieser Share.

Jede Partei $P_i$ wählt $t$ Koeffizienten $a_{i,j}$ und setzt ihren Anteil des Polynoms $f$ auf

$$f_i(X) = [d_{\mathbb{Z}}]_{+,i} + \sum_{j=1}^{t} a_{i,j} \cdot X^j.$$

Die Partei $P_i$ erstellt dann für jede Partei $P_j$ ein additives Sharing von $f_i(j)$. Dazu wählt sie zufällige $[f_i(j)]_{+,k}$ in den ganzen Zahlen, sodass $\sum_{k=1}^{n} [f_i(j)]_{+,k} = f_i(j)$. Dann sendet $P_i$ jeder Partei $P_k$ die Share $[f_i(j)]_{+,k}$ zu. Die Parteien entschlüsseln $c_{f_i(j)}^{-1} \cdot \prod_{k=1}^{n} c_{f_{i,j,k}} = \mathsf{Enc}_{\mathsf{pk}_{EG}}\left(\sum_{k=1}^{n} [f_i(j)]_{+,k} - f_i(j)\right)$ und prüfen, ob die Differenz 0 ist, also ob $P_i$ die Werte korrekt geteilt hat. Mit den erhaltenen Werten kann jede Partei $P_k$ nun $[f(j)]_{+,k}$, ihre Share von $f(j)$, als die Summe der $[f_i(j)]_{+,k}$ berechnen, da $f(X)$ als $\sum_{i=1}^{N} (f_i(X))$ konstruiert wird.

Die Verschlüsselungen in den Zeilen 12 und 25 können von jeder Partei mithilfe der Homomorphie aus bekannten Ciphertexten berechnet werden:

$$c_{f_i(j)} = \mathsf{Enc}_{\mathsf{pk}_{EG}}(f_i(j)) = \mathsf{Enc}_{\mathsf{pk}_{EG}}\left([d_{\mathbb{Z}}]_{+,i} + \sum_{k=1}^{t} a_{i,k} \cdot j^k\right) = c_{[d_{\mathbb{Z}}]_{+,i}} \cdot \prod_{k=1}^{t} \left(c_{a_{i,k}}^{j^k}\right)$$

$$c_{f(j),k} = \mathsf{Enc}_{\mathsf{pk}_{EG}}\left([f(j)]_{+,k}\right) = \mathsf{Enc}_{\mathsf{pk}_{EG}}\left(\sum_{i=1}^{n} [f_i(j)]_{+,k}\right) = \prod_{i=1}^{n} c_{f_{i,j,k}}.$$

**Lemma 3.4.5**
*Die vom Protokoll POLY-SHARE ausgegebenen Sharings der $f(j)$ gehören zu einem Polynom $f$ von Grad $t$ mit $f(0) = d$, wobei $d$ der Wert des Sharings aus der Eingabe ist.*

**Beweis** Das Polynom $f$ wird als $f(X) = \sum_{i=1}^{n} f_i(X)$ konstruiert. Damit ist der Grad von $f$ gleich dem Grad der $f_i$, welcher $t$ ist. Zudem ist $f(0) = \sum_{i=1}^{n} f_i(0) = \sum_{i=1}^{n} [d_{\mathbb{Z}}]_{+,i} = d$. □

**Protokoll 3.13** POLY-SHARE

$([f(1)]_+, [f(2)]_+, \ldots, [f(n)]_+) \leftarrow \text{POLY-SHARE}([d_{\mathbb{Z}}]_+, N, l, t)$

1 : **for** $i = 1, \ldots, n; j = 1, \ldots, t$ **do**
2 : $P_i$ wählt $a_{i,j}$ gleichverteilt zufällig aus $\mathbb{Z}_{2^{2\log(N)+l}}$
3 : $c_{a_{i,j}} = \mathsf{Enc}_{\mathsf{pk}_{EG}}(a_{i,j})$
4 : BROADCAST $\left(i, c_{a_{i,j}}\right)$
5 : $P_i$ beweist allen Parteien Klartext-Kenntnis zu $c_{a_{i,j}}$ mittels ZK-ENC
6 : $P_i$ beweist allen Parteien mit ZK-BOUND, dass $a_{i,j} < 2^{2\log(N)+l}$
7 : **endfor**
8 : **for** $i = 1, \ldots, n$ **do**
9 : $f_i(X) = [d_{\mathbb{Z}}]_{+,i} + \sum_{j=1}^{t} a_{i,j} \cdot X^j$
10 : **endfor**
11 : **for** $i = 1, \ldots, n; j = 1, \ldots, n$ **do**
12 : $c_{f_i(j)} = \mathsf{Enc}_{\mathsf{pk}_{EG}}(f_i(j))$
13 : **for** $k = 1, \ldots, n$ **do**
14 : SEND $(i, k, [f_i(j)]_{+,k})$
15 : $c_{f_{i,j,k}} = \mathsf{Enc}_{\mathsf{pk}_{EG}}([f_i(j)]_{+,k})$
16 : BROADCAST $\left(i, c_{f_{i,j,k}}\right)$
17 : $P_i$ beweist jeweils die Kenntnis des Klartexts mit ZK-ENC
18 : $P_i$ beweist mit ZK-BOUND, dass $[f_i(j)]_{+,k} < \max\left(\ell_d, (\log^t(l))(2\log(N) + l)(\log(t))\right) + l$
19 : **endfor**
20 : **if** EG-DEC $\left(c_{f_i(j)}^{-1} \cdot \prod_{k=1}^{n} c_{f_{i,j,k}}\right) \neq g^0$ **then**
21 : **abort**
22 : **endif**
23 : **for** $j = 1, \ldots, n; k = 1, \ldots, n$ **then**
24 : $[f(j)]_{+,k} = \sum_{i=1}^{n} [f_i(j)]_{+,k}$
25 : $c_{f(j),k} = \mathsf{Enc}_{\mathsf{pk}_{EG}}([f(j)]_{+,k})$
26 : **endfor**
27 : **endfor**

**Lemma 3.4.6**
*Die Privacy der Eingabe- und Ausgabewerte bleibt durch das Protokoll POLY-SHARE gewahrt.*

**Beweis** Die Entschlüsselung $\mathsf{Dec}_{\mathsf{sk}_{EG}}\left(c_{f_i(j)}^{-1} \cdot \prod_{k=1}^{n} c_{f_{i,j,k}}\right)$ gibt keine neuen Informationen preis, wenn das Protokoll nicht abbricht, da sie dann immer 1 ergibt. Die versendeten $[f_i(j)]_{+,k}$ sind jeweils zufällige, von der Partei $P_i$ gewählte ganze Zahlen, deren Summe $f_i(j)$ ist. Da Partei $P_i$ die Share $[f_i(j)]_{+,i}$ selbst hält, können die anderen Parteien daraus keine Informationen ableiten. Ansonsten werden nur sichere Verschlüsselungen veröffentlicht. □

#### Verteilung der Shares

Mit dem Subprotokoll ASSIGN-SHARE wird der Partei $j$ ihre Share $[d]_{\mathbb{Z},j} \equiv f(j) \mod N\phi(N)$ des Shamir Sharings zugeteilt. Dabei ist $f$ das Polynom aus vorigem Abschnitt.

Das Protokoll erhält als Eingabe die additiven Sharings der Threshold Shares, die den einzelnen Parteien zugewiesen werden sollen, das Sharing von $\phi(N)$ und jeweils eine ElGamal-Verschlüsselung zu allen Shares, das RSA Modulus $N$ und einen Sicherheitsparameter $l$. Nach der Ausführung kennt jede Partei $P_i$ eine Threshold Share $[d]_{\mathbb{Z},i} \equiv f(i) \mod \phi(N)N$ und für jede Partei $P_j$ eine ElGamal-Verschlüsselung $c_{[d]_{\mathbb{Z},j}}$ der Share $[d]_{\mathbb{Z},j}$.

Jede Partei $P_j$ soll nach der Ausführung des Protokolls ihre Share $[d]_{\mathbb{Z},j} \equiv f(j) \mod N\phi(N)$ kennen. Da $N\phi(N)$ nur additiv geteilt vorliegt, können die Parteien keine Modulo-Reduktion durchführen. Um sicherzustellen, dass keine zusätzlichen Informationen preisgegeben werden, addieren sie große zufällige Vielfache von $N\phi(N)$ auf ihre Shares. Die Bitlänge $\ell_r$ der Zufallswerte $\left[r_j\right]_{+,i}$ ist $\ell_r = \max\left(\ell_d; (\log^t(l))(2\log(N)+l)(\log(t))\right) + \log(n) - 2\log(N) + 2l$. Dann sendet jeweils jede Partei $P_i$ die maskierte Share $\left[\sigma_j\right]_{+,i}$ an Partei $P_j$, welche dann ihre Share $[d]_{\mathbb{Z},j} = \sum_{i=1}^{n} \left[\sigma_j\right]_{+,i} \equiv f(j) \mod N\phi(N)$ berechnet. Die Parteien entschlüsseln die $c_{\sigma_{i,j}}$ für Partei $P_j$, damit $P_j$ prüfen kann, ob sie die korrekten Shares erhalten hat. Dazu senden die Parteien ihren Anteil der Entschlüsselung an $P_j$, statt ihn wie in der Entschlüsselung für alle Parteien zu broadcasten, und beweisen $P_j$, dass sie den richtigen Wert gesendet haben. So erhält die Partei $P_j$ die $g^{\sigma_{i,j}}$ und kann sie mit den erhaltenen Nachrichten abgleichen.

Die Verschlüsselungen können die Parteien jeweils mittels Homomorphie aus bekannten Ciphertexten berechnen.

**Lemma 3.4.7**
*Nach Ausführung des Protokolls ASSIGN-SHARE kennt die Partei $P_j$ ein $[d]_{\mathbb{Z},j}$, sodass $[d]_{\mathbb{Z},j} \equiv f(j) \mod N\phi(N)$ gilt, wobei $f$ ein Polynom mit Grad $t$ und $f(0) \equiv d \mod N\phi(N)$ ist.*

**Beweis** Die Parteien berechnen zunächst für jede Partei $P_j$ ein additives Sharing von

$$\sigma_j = f(j) + s_j = f(j) + r_j N\phi(N) \equiv f(j) \mod N\phi(N),$$

wobei $r_j$ ein zufälliger Wert mit $\ell_r$ Bits ist und $f$ ein Polynom des Grads $t$ mit $f(0) = d$. Jede Partei $P_j$ erhält die Shares $\left[\sigma_j\right]_{+,i}$ von allen Parteien $P_i$ und berechnet damit seine Share von $d$ als

$$[d]_{\mathbb{Z},j} = \sum_{i=1}^{n} \left[\sigma_j\right]_{+,i} = \sigma_j \equiv f(j) \mod N\phi(N).$$

**Protokoll 3.14** ASSIGN-SHARE

$[d]_{\mathbb{Z}} \leftarrow \text{ASSIGN-SHARE}\left(([f(1)]_+, [f(2)]_+, \ldots, [f(n)]_+), [\phi(N)]_+, N, l\right)$

1 : **for** $i = 1, \ldots, n$ **do**
2 : $\quad [N\phi(N)]_{+,i} = N \cdot [\phi(N)]_{+,i}$
3 : $\quad c_{N\phi(N),i} = \mathsf{Enc}_{\mathsf{pk}_{EG}}\left([N\phi(N)]_{+,i}\right)$
4 : **endfor**
5 : **for** $i = 1, \ldots, n; j = 1, \ldots, n$ **do**
6 : $\quad P_i$ wählt $\left[r_j\right]_{+,i}$ gleichverteilt zufällig aus $\mathbb{Z}_{2^{\ell_r}}$
7 : $\quad c_{r_{i,j}} = \mathsf{Enc}_{\mathsf{pk}_{EG}}\left(\left[r_j\right]_{+,i}\right)$
8 : $\quad \text{BROADCAST}\left(i, c_{r_{i,j}}\right)$
9 : $\quad P_i$ beweist jeweils die Kenntnis des Klartexts mit ZK-ENC
10 : $\quad P_i$ beweist mit ZK-BOUND, dass $\left[r_j\right]_{+,i} < \ell_r$
11 : **endfor**
12 : **for** $j = 1, \ldots, n$ **do**
13 : $\quad \left[s_j\right]_+ \leftarrow \text{MULTIPLY}(\left[r_j\right]_+, [N\phi(N)]_+)$
14 : $\quad \left[\sigma_j\right]_{+,i} = \left[s_j\right]_{+,i} + [f(j)]_{+,i}$
15 : $\quad c_{\sigma_{i,j}} = \mathsf{Enc}_{\mathsf{pk}_{EG}}\left(\left[\sigma_j\right]_{+,i}\right)$
16 : **endfor**
17 : **for** $j = 1, \ldots, n; i = 1, \ldots, n$ **do**
18 : $\quad \text{SEND}\left(i, j, \left[\sigma_j\right]_{+,i}\right)$
19 : $\quad$ Die Parteien entschlüsseln $c_{\sigma_{i,j}}$ für Partei $P_j$
20 : $\quad P_j$ prüft, ob sie die korrekten Shares erhalten hat
21 : **endfor**
22 : **for** $j = 1, \ldots, n$ **do**
23 : $\quad [d]_{\mathbb{Z},j} = \sum_{i=1}^{n} \left[\sigma_j\right]_{+,i}$
24 : $\quad c_{[d]_{\mathbb{Z},j}} = \mathsf{Enc}_{\mathsf{pk}_{EG}}\left([d]_{\mathbb{Z},j}\right)$
25 : **endfor**

Durch die Entschlüsselung der Ciphertexte kann die Partei $P_j$ prüfen, ob sie die richtigen Werte von den anderen Parteien erhalten hat. Somit kennt die Partei $P_j$ am Ende des Protokolls ihre korrekte Share von $d$. □

Wenn der Adversary mehr als $t$ Parteien kontrolliert, kann er mit den Shares dieser Parteien das zugrundeliegende Polynom $f$ interpolieren und erhält damit den Schlüssel $f(0) + rN\phi(N) \equiv d \mod \phi(N)N$.

**Lemma 3.4.8**
*Die Privacy der Eingabe- und Ausgabewerte bleibt durch das Protokoll ASSIGN-SHARE gewahrt, wenn der Adversary höchstens t Parteien kontrolliert.*

**Beweis** Das Polynom $f$ ist als nahezu zufälliges Polynom in $\mathbb{Z}_{\phi(N)N}$ mit Grad $t$ konstruiert. Die Shares sind dann Punkte aus diesem Polynom addiert mit einem zufälligen Vielfachen von $N\phi(N)$. Wenn der Adversary höchstens $t$ Parteien kontrolliert, kann er das Polynom nicht interpolieren, da er für die Interpolation $t + 1$ Punkte braucht. Der fehlende Punkt kann beliebige Werte annehmen, folglich kann der Adversary so keine Information über $d \equiv f(0) \mod \phi(N)N$ erlangen. Ansonsten erhalten die Parteien nur sicher verschlüsselte Werte. Damit ist die Privacy der Eingabe und Ausgabe gewahrt, wenn die aufgerufenen Protokolle die Privacy wahren. □

## 3.5 Schwächen der Schlüsselerzeugung

In diesem Abschnitt wird auf Schwächen der beschriebenen Schlüsselerzeugung eingegangen.

### 3.5.1 Effizienz

Um die Wahrscheinlichkeit für eine erfolgreiche Erzeugung zu erhöhen, wird bei der RSA Modulus Erzeugung der Trial Division Test (Protokoll 3.5) durchgeführt. Hazay et al. [14] geben zu einer einfacheren Version des Protokolls für nur zwei Parteien an, dass der Trial Division Schritt der aufwändigste Teil der Berechnung ist. Mit ihrer experimentellen Implementierung benötigt die Trial Division einen erheblichen Zeitaufwand in Abhängigkeit von dem Bound $B$, so geben sie für $B = 3181$ eine erwartete Zeit von 5 Stunden und 23.3 Stunden für $B = 15973$ für die Generation eines RSA Modulus mit einer Länge von 2048 Bits an. Hazay et al. merken an, dass der größte Teil des Aufwands der Trial Division durch die ElGamal-Entschlüsselung entsteht. Die Anzahl der nötigen Entschlüsselungen in diesem Schritt entspricht der Anzahl der Parteien $n$, der Aufwand nimmt also mit der Anzahl der Parteien linear zu. Entsprechend hoch ist der Aufwand des hier beschriebenen Protokolls.

Um die Performanzprobleme zu verringern, schlagen Hazay et al. vor einen Teil der Trial Division lokal durchzuführen. Die Parteien führen dazu das Trial Division Protokoll auf den Kandidaten für einen deutlich kleineren Bound $B'$ aus, berechnen das Produkt für die Kandidaten und testen anschließend lokal, ob eine Primzahl $\alpha < B$ das Produkt $N$ der Primzahlkandidaten teilt. Mit dieser Optimierung gelingt es Hazay et al. für das Protokoll für zwei Parteien den Aufwand deutlich zu senken. Diese Modifikation lässt sich problemlos auch auf die Verallgemeinerung mit $n$ Parteien anwenden.

Insgesamt bleibt der Aufwand für das Protokoll aber hoch. Allerdings muss die Erzeugung eines Schlüssels nicht oft durchgeführt werden, folglich kann sich der Aufwand für einige Anwendungen lohnen.

### 3.5.2 Guaranteed Output Delivery

Wie in Abschnitt 3.4 erwähnt, kann der Adversary das Protokoll während der Ausführung abbrechen, wenn er sich weigert einen Wert beizutragen. Die Eigenschaft Guaranteed Ouput Delivery ist also verletzt. Wenn ein Protokoll Sicherheit gegen einen malicious Adversary, welcher bis zu $n - 1$ Parteien kontrolliert bieten soll, ist es nicht möglich, dass dieses auch Guaranteed Output Delivery bietet, wie aus [8] folgt.

Eine Partei, welche dem Protokoll nicht folgt, kann nach einem Abbruch identifiziert und ausgeschlossen werden. Da aber, wie in vorigem Abschnitt beschrieben, der Zeitbedarf des Protokolls hoch ist, kann die zusätzliche Verzögerung durch den Adversary problematisch werden. Hier hängt es von der Anwendung ab, ob der Adversary ein Interesse daran hat, die Erzeugung eines Schlüssels aufzuhalten.

# 4 Protokolle für Berechnungen in der Offline Phase

In diesem Kapitel wird beschrieben, wie die Vorberechnung von Zufallswerten in der Offline Phase durchgeführt werden kann. Dafür wird davon ausgegangen, dass die Parteien ein Threshold Paillier-Kryptosystem mit einem Schlüsselpaar wie im vorigen Kapitel beschrieben vorbereitet haben.

In Abschnitt 4.1 wird zunächst beschrieben, wie die Parteien Grundbausteine für die MPC Berechnungen aus dem Threshold Paillier-Kryptosystem aufbauen können. In den folgenden Abschnitten werden Protokolle zu Berechnungen mithilfe dieser Bausteine vorgestellt. In Abschnitt 4.9 werden einige Möglichkeiten für auf diesen Protokollen aufbauende Wertberechnungen beschrieben.

## 4.1 Multiparty Computation basierend auf dem Threshold Paillier-Kryptosystem

In diesem Abschnitt wird beschrieben, wie basierend auf dem Threshold Paillier-Kryptosystem MPC-Grundbausteine konstruiert werden können. Jede Partei kennt den Public Key $N$ und eine Share des Secret Keys wie in Kapitel 3 beschrieben. Die Berechnungen werden in dem Ring $\mathbb{Z}_N$ durchgeführt. $\ell = \lceil \log_2(N) \rceil$ bezeichnet die Bitanzahl von $N$. Die Gruppe der invertierbaren Elemente bezüglich der Multiplikation in $\mathbb{Z}_N$ wird im Folgenden $\mathbb{Z}_N^*$ genannt. Dabei wird ein Ciphertext des Werts $a \in \mathbb{Z}_N$ mit $[a]_N$ notiert. $[a]_B$ steht für eine Sammlung von Ciphertexten der Bits von a. Die Parteien warten jeweils, bis alle Nachrichten einer Anweisung empfangen wurden, bevor sie mit dem Protokoll fortfahren. Dies kann umgesetzt werden, indem alle Parteien eine Nachricht broadcasten sobald sie alle Nachrichten zu einer Anweisung erhalten haben.

Das Threshold $t$ des Threshold Paillier-Kryptosystems wird in der Schlüsselerzeugung auf $\lceil n/2 \rceil - 1$ gesetzt. Der Adversary darf also höchstens $t$ Parteien kontrollieren, da eine größere Menge von Parteien Ciphertexte entschlüsseln kann. Dies stellt einen Kompromiss zwischen dem Sicherstellen der Privacy gegen einen Adversary, der viele Parteien kontrolliert, und Guaranteed Output Delivery dar. Wenn der Adversary mehr als $t$ Parteien kontrolliert, bleibt den Parteien bei einer Weigerung des Adversary bei der Entschlüsselung seinen Anteil beizutragen keine Wahl als die Ausführung des Protokolls abzubrechen. Wenn der Adversary höchstens $t$ Parteien kontrolliert, kann die ehrliche Mehrheit der Parteien Werte ohne die Kooperation der von dem Adversary kontrollierten Parteien entschlüsseln. Der Schlüssel ist das einzige geteilte Geheimnis. Die Ciphertexte, auf denen die Berechnung durchgeführt wird, sind allen Parteien bekannt. Folglich können die Parteien das Protokoll ohne die Kooperation des Adversary zu Ende führen und daher garantieren die Protokolle Guaranteed Output Delivery. Für die folgenden Protokolle wird Privacy unter der Annahme, dass der Adversary höchstens $t$ Parteien kontrolliert, untersucht.

Mit dem Threshold Paillier-Kryptosystem können folgende MPC Grundbausteine konstruiert werden.

#### Teilen eines Werts

Die Partei $P_i$ muss eine Möglichkeit haben, einen geheimen lokalen Wert $a$ an die anderen Parteien zu verteilen, ohne diesen den Parteien zu verraten. Dazu verschlüsselt die Partei den Wert $a$ mithilfe des Public Keys $N$, broadcastet den Ciphertext an alle Parteien und beweist allen Parteien die Kenntnis des Klartexts mit ZK-ENC. Dies wird in dieser Arbeit mit

$$[a]_N \leftarrow \text{SHARE}(i, a)$$

notiert.

#### Offenlegen eines Werts

Eine Menge von mindestens $t$ Parteien kann einen Wert $a$ aus $[a]_N$ offenlegen, indem sie das Entschlüsselungsprotokoll PAIL-DECRYPT aus Kapitel 3 ausführt. Dies wird im Folgenden mit

$$a \leftarrow \text{REVEAL}([a]_N)$$

dargestellt.

#### Addition

Mithilfe der Homorphieeigenschaft von Paillier kann jede Partei die Summe $c \in \mathbb{Z}_N$ zweier Werte $a \in \mathbb{Z}_N$ und $b \in \mathbb{Z}_N$ auf Ciphertexten $[a]_N$ und $[b]_N$ ohne Interaktion berechnen. Dies wird innerhalb dieser Arbeit geschrieben als

$$[c]_N \leftarrow [a]_N + [b]_N \,.$$

#### Multiplikation mit einem bekannten Wert

Ebenso kann jede Partei lokal den Wert $a \in \mathbb{Z}_N$ eines Ciphertexts mit einem ihr bekannten Wert $b$ mittels additiver Homomorphie multiplizieren. Ergebnis ist ein Ciphertext $[c]_N$ mit $c = ab \mod N \in \mathbb{Z}_N$. Dieser Schritt wird im Folgenden mit

$$[c]_N \leftarrow [a]_N \, b$$

notiert.

#### Multiplikation von geheimen Werten

Die Multiplikation zweier Werte $a$ und $b$, die als Ciphertexte $[a]_N$ und $[b]_N$ gegeben sind, können die Parteien wie von Schoenmakers und Veeningen [17] beschrieben berechnen. Die Ausgabe des Protokolls ist ein Ciphertext $[c]_N$ des Werts $c = ab \mod N \in \mathbb{Z}_N$.

Jede Partei $P_i$ wählt dazu einen zufälligen Wert $d_i \in \mathbb{Z}_N$ und sendet Ciphertexte von $d_i$ und $d_i b$ mittels SHARE an die anderen Parteien. Die Parteien berechnen dann jeweils $[a + d_1 + d_2 + \cdots + d_n]_N$ und entschlüsseln diesen Wert gemeinsam, um $a + d_1 + d_2 + \cdots + d_n$ zu erhalten. Jede Partei kann dann den Ciphertext $[b(a + d_1 + d_2 \cdots + d_n)]_N$ und daraus zusammen mit den Ciphertexten $[d_i b]_N$ einen Ciphertext $[xy]_N$ berechnen. Die Notation dafür ist im Folgenden

$$[c]_N \leftarrow \text{MULT}([a]_N, [b]_N).$$

Dieses Protokoll kann mithilfe von ZK Proofs gegen malicious Adversaries abgesichert werden.

Die Sicherheit der folgenden Protokolle ist von der Sicherheit des zugrundeliegenden Threshold Kryptosystems und des Multiplikationsprotokolls abhängig.

## 4.2 Zufällige Werte

Ein Ciphertext eines gleichverteilt zufälligen Werts aus $\mathbb{Z}_N$ kann wie von Damgard et al. [10] beschrieben berechnet werden. Dieses Protokoll wird im Folgenden RAN genannt.

Jede Partei $P_i$ wählt mit gleichverteilter Wahrscheinlichkeit einen zufälligen Wert $r_i$ aus dem Körper $\mathbb{Z}_N$ und broadcastet einen Ciphertext von diesem Wert. Der generierte geheime Wert ist dann die Summe aller von den Parteien gewählten Werten.

Wenn eine Partei keinen Ciphertext beiträgt, wird die Summe mit den vorhandenen Ciphertexten berechnet.

**Lemma 4.2.1**
*Das Protokoll RAN gibt einen Ciphertext eines gleichverteilt zufälligen Werts aus.*

**Beweis** Solange mindestens eine Partei einen gleichverteilt zufälligen Wert aus $\mathbb{Z}_N$ beiträgt, ist der ausgegebene Wert, die Summe der Werte der Parteien, ebenfalls gleichverteilt zufällig in $\mathbb{Z}_N$. □

**Protokoll 4.1** RAN

$[r]_N \leftarrow$ RAN ()

1 : **for** $i = 1, \ldots, n$ **do**
2 : $P_i$ wählt $r_j$ gleichverteilt zufällig aus $\mathbb{Z}_N$
3 : $[r_i]_N \leftarrow \text{SHARE}(i, r_i)$
4 : **endfor**
5 : $[r]_N \leftarrow \sum_{j=1}^{n} [r_j]_N$

**Lemma 4.2.2**
*Die Privacy des ausgegebenen Werts bleibt durch das Protokoll RAN gewahrt.*

**Beweis** Es wird kein Wert veröffentlicht. Folglich ist die Privacy gewährleistet solange mindestens eine Partei einen zufälligen Wert beiträgt und das Protokoll SHARE den Parteien keine Information außer den jeweiligen Ciphertexten der Werte verrät. □

### 4.2.1 Zufällige invertierbare Werte

Angelehnt an das Invertierungs-Protokoll von Bar-Ilan und Beaver [1] beschreiben Damgard et al. [10] die Generierung des Ciphertexts eines zufälligen invertierbaren Elements aus $\mathbb{Z}_N$ mit seinem Inversen.

Die Parteien erzeugen zwei zufällige geheime Werte mithilfe des Protokolls RAN und legen das Produkt der Werte offen. Das Produkt $c$ ist genau dann in $\mathbb{Z}_N^*$, wenn beide Zufallswerte in $\mathbb{Z}_N^*$ sind. Das Protokoll RAN$^*$ bricht ab, wenn $c \notin \mathbb{Z}_N^*$ ist. Dies ist der Fall, wenn $a \notin \mathbb{Z}_N^*$ oder $b \notin \mathbb{Z}_N^*$. Das Protokoll RAN generiert gleichverteilt zufällige Werte, davon sind $\phi(N)$ invertierbar. Daher ist seine Ausgabe mit der Wahrscheinlichkeit $1 - \frac{\phi(N)}{N}$ nicht invertierbar. Für große $N$ ist diese Wahrscheinlichkeit vernachlässigbar klein. Folglich ist auch die Wahrscheinlichkeit eines Abbruchs gering. Um mit einer hinreichend großen Wahrscheinlichkeit ein Element aus $\mathbb{Z}_N^*$ zu erhalten, kann das Protokoll mehrmals parallel ausgeführt werden. Die Ausgabe ist die der ersten Ausführung, die nicht abbricht. Der Wert $c^{-1}$ kann mit dem veröffentlichten $c$ von jeder Partei lokal berechnet werden. Daraus können die Parteien das Inverse von $a$ berechnen.

**Lemma 4.2.3**
*Das Protokoll RAN$^*$ gibt Ciphertexte von a und $a^{-1}$ aus $\mathbb{Z}_N^*$ aus oder bricht ab.*

**Beweis** Das Protokoll bricht nur dann nicht ab, wenn $a$ und $b$ in $\mathbb{Z}_N^*$ sind. Wenn dies gilt, ist

$$c^{-1}b = (ab)^{-1}b = b^{-1}a^{-1}b = a^{-1}.$$

Folglich ist die Ausgabe des Protokolls je ein Ciphertext eines Wertes $a$ aus $\mathbb{Z}_N^*$ und seines Inversen $a^{-1}$. □

**Lemma 4.2.4**
*Die Privacy der ausgegebenen Werte wird durch das Protokoll RAN$^*$ sichergestellt.*

**Protokoll 4.2** RAN$^*$

$([a]_N, [a^{-1}]_N) \leftarrow \text{RAN}^*\ ()$

1: $[a]_N \leftarrow \text{RAN}\ ()$
2: $[b]_N \leftarrow \text{RAN}\ ()$
3: $[c]_N \leftarrow \text{MULT}([a]_N, [b]_N)$
4: $c \leftarrow \text{REVEAL}([c]_N)$
5: **if** $c \notin \mathbb{Z}_N^*$ **then**
6: **abort**
7: **endif**
8: $[a^{-1}]_N = c^{-1}\,[b]_N$

**Beweis** $a$ und $b$ werden mit RAN gleichverteilt zufällig in $\mathbb{Z}_N$ generiert. Somit ist auch ihr Produkt $c$ gleichverteilt zufällig in $\mathbb{Z}_N$. Wenn das Protokoll nicht abbricht, ist $c$ und damit auch $a$ aus $\mathbb{Z}_N^*$ und das Veröffentlichen von $c$ lässt keine Rückschlüsse auf $a$ zu. Damit ist die Privacy des Werts $a$ gewahrt. □

## 4.3 Präfixprodukte

Basierend auf Bar-Ilan und Beaver [1] beschreiben Damgard et. al [10], wie die Multiplikation von $m$ Werten, gegeben als Ciphertexte $[a_1]_N, [a_2]_N, \ldots, [a_m]_N \in \mathbb{Z}_N^*$, durchgeführt werden kann. Das in diesem Abschnitt vorgestellte Protokoll dient der Berechnung aller Präfixprodukte.

Die Parteien multiplizieren die Eingabe $a_i$ zunächst mit zufälligen Werten $b_{i-1}$ und $b_i^{-1}$ aus $\mathbb{Z}_N^*$. Die Produkte $b_{i-1}a_ib_i^{-1}$ legen sie offen und berechnen damit jeweils lokal die Produkte $b_0 \prod_{i=1}^{k}(a_i)b_k^{-1}$. Daraus kann das k-te Präfixprodukt durch die Multiplikation mit $b_0^{-1}$ und $b_k$ gewonnen werden.

**Lemma 4.3.1**
*Das Protokoll MULT* berechnet die Präfixprodukte $\prod_{k=1}^{i} a_k$ der Eingaben $a_1, a_2, \ldots, a_m$ für $i \in \{1, \ldots, m\}$.*

**Beweis** Es gilt

$$d_{1,i} = \prod_{k=1}^{i} d_i = \prod_{k=1}^{i}(b_{i-1}a_ib_i^{-1}) = b_0b_i^{-1}\prod_{k=1}^{i} a_i$$
$$\Rightarrow \prod_{k=1}^{i} a_i = d_{1,i}b_0^{-1}b_i.$$

**Protokoll 4.3** MULT*

$\left([a_1]_N, \ldots, [a_1a_2\cdots a_m \mod p]_N\right) \leftarrow \text{MULT}^*([a_1]_N, \ldots, [a_m]_N)$

1 : **for** $i = 0, \ldots, m$ **do**
2 : $\left([b_i]_N, \left[b_i^{-1}\right]_N\right) \leftarrow \text{RAN}^*()$
3 : **endfor**
4 : **for** $i = 1, \ldots, m$ **do**
5 : $[d_i]_N \leftarrow \text{MULT}\left([b_{i-1}]_N, [a_i]_N, \left[b_i^{-1}\right]_N\right)$
6 : $d_i \leftarrow \text{REVEAL}([d_i]_N)$
7 : **endfor**
8 : **for** $i = 1, \ldots, m$ **do**
9 : $\left[d_{1,i}\right]_N = \prod_{k=1}^{i} d_i$
10 : $\left[c_{1,i}\right]_N \leftarrow \text{MULT}(\left[b_0^{-1}\right]_N, [b_i]_N)$
11 : $[a_1 \cdots a_i \mod p]_N = d_{1,i}\left[c_{1,i}\right]_N$
12 : **endfor**

Die Ausgabe des Protokolls sind folglich die korrekten Präfixprodukte. □

**Lemma 4.3.2**
*Das Protokoll MULT* wahr die Privacy der Eingabe- und Ausgabewerte, wenn $a_i \in \mathbb{Z}_N^*$ ist.*

**Beweis** Für alle $i$ muss $a_i \in \mathbb{Z}_N^*$ gelten, da sonst bei $d_i \in \mathbb{Z}_N$ und $d_{i-1} \neq \mathbb{Z}_N^*$ Rückschlüsse auf $a_i$ gemacht werden können. Die Privacy der Eingaben und Ausgaben bleibt gewahrt, da die zufälligen Werte $b_i$ und ihre Inversen geheim bleiben. Folglich lassen die veröffentlichten $d_i$ keine Rückschlüsse auf die $a_i$ zu. □

Für einen Effizienzgewinn können die Schritte in den Schleifen parallel ausgeführt werden.

## 4.4 Kleiner Funktion auf Ciphertexten von Bits

In diesem Abschnitt wird das Protokoll BIT-LT von Damgard et al. [10] vorgestellt. Es berechnet die Funktion $<^?$: $\mathbb{Z}_N \times \mathbb{Z}_N \to \{0,1\}, (x <^? y) = 1 \iff x < y$. Das Protokoll BIT-LT nimmt Ciphertexte der Bits $[a]_B$ und $[b]_B$ der Eingabewerte $a$ und $b$ als Eingabe und gibt einen Ciphertext von $(x <^? y) \in \{0,1\} \subsetneq \mathbb{Z}_N$ aus. Das BIT-LT Protokoll interpretiert die eingegebenen Zahlen als vorzeichenlos.

Die Parteien berechnen zunächst die höchstwertige Stelle an der sich $a$ und $b$ unterscheiden. Wenn $b$ an dieser Stelle 1 ist, gibt das Protokoll einen Ciphertext von dem Wert 1 aus sonst einen von 0.

**Lemma 4.4.1**
*Das Protokoll BIT-LT gibt einen Ciphertext von* 1 *aus, wenn a kleiner als b ist, sonst von* 0*.*

**Protokoll 4.4** BIT-LT

$[h]_N \leftarrow \text{BIT-LT}([a]_B, [b]_B)$

1 : **for** $i = 0, \ldots, \ell - 1$ **do**
2 : $\quad [e_i]_N \leftarrow \text{XOR}([a_i]_N, [b_i]_N)$
3 : **endfor**
4 : $([f_{\ell-1}]_N, \ldots, [f_0]_N) = \text{PRE}_\vee([e_{\ell-1}]_N, \ldots, [e_0]_N)$
5 : $[g_{\ell-1}]_N \leftarrow [f_{\ell-1}]_N$
6 : **for** $i = 0, \ldots, \ell - 2$ **do**
7 : $\quad [g_i]_N \leftarrow [f_i]_N - [f_{i+1}]_N$
8 : **endfor**
9 : **for** $i = 0, \ldots, \ell - 1$ **do**
10 : $\quad [h_i]_N \leftarrow \text{MULT}([g_i]_N, [b_i]_N)$
11 : **endfor**
12 : $[h]_N \leftarrow \sum_{i=0}^{\ell-1} [h_i]_N$

**Beweis** Das Protokoll BIT-LT berechnet zunächst das bitweise XOR von $a$ und $b$. Wenn $a \neq b$ ist, gibt es einen größten Index $i_0$ mit $a_{i_0} \neq b_{i_0}$ und folglich ist $\text{XOR}(a_{i_0}, b_{i_0}) = 1$. Dann ist das berechnete $f_{i_0} = 1$ und es gilt $\forall i > i_0 : f_i = 0$ und $\forall i < i_0 : f_i = 1$. Damit ist $g_{i_0} = f_{i_0} - f_{i_0+1} = 1$ das einzige $g_i$ ungleich 0. Nur wenn $a$ kleiner als $b$ ist, ist $b_{i_0} = 1$ und $h = b_{i_0} g_{i_0} = 1$. Ist $a$ größer als $b$, dann ist $h = 0$. Im Fall $a = b$ sind alle $e_i$ und damit alle $f_i$ und $g_i$ gleich 0 und $h$ ist wie gewünscht 0. Das Protokoll ist folglich korrekt. □

**Lemma 4.4.2**
*Die Privacy der Eingabe- und Ausgabewerte wird durch das Protokoll BIT-LT gewahrt.*

**Beweis** Die Privacy der Werte bleibt durch dieses Protokoll gewahrt, da kein Wert veröffentlicht wird und ausschließlich sichere Subprotokolle aufgerufen werden. □

Die Anweisungen in den Schleifen können für eine Effizienzsteigerung parallel ausgeführt werden.

#### Exklusives Oder

Das für das BIT-LT Protokoll benötigte XOR kann wie in diesem Abschnitt beschrieben berechnet werden. Das XOR-Protokoll nimmt zwei Ciphertexte $[a]_N$ und $[b]_N$ mit $a, b \in \{0, 1\}$ als Eingabe und gibt $[c]_N$ mit $c \in \{0, 1\} \subset \mathbb{Z}_N$ aus.

**Lemma 4.4.3**
*Das Protokoll XOR gibt einen Ciphertext des Exklusiv-Oders der Eingabewerte a und b aus.*

**Beweis** Das XOR-Protokoll ist korrekt, da für $a$und $b$ aus $\{0, 1\}$ gilt

$$d = a - b = \begin{cases} 1 & \text{wenn } a = 1 \wedge b = 0 \\ -1 & \text{wenn } a = 0 \wedge b = 1 \\ 0 & \text{wenn } a = b \end{cases}$$

$$\Rightarrow \left(d^2 = 1\right) \iff (a \neq b) \iff ((a \text{ XOR } b) = 1)\,. \qquad \square$$

Die Privacy der Werte bleibt durch die Sicherheit der Subprotokolle gewahrt.

**Protokoll 4.5** XOR

$[c]_N \leftarrow \text{XOR}([a]_N, [b]_N)$

1 : $[d]_N \leftarrow [a]_N - [b]_N$
2 : $[c]_N \leftarrow \text{MULT}([d]_N, [d]_N)$

**Protokoll 4.6** Protokoll für symmetrische Funktionen

$[f(a_1, a_2, \ldots, a_m) \mod N]_N \leftarrow f([a_1]_N, [a_2]_N, \ldots, [a_m]_N)$

1 : $[a]_N \leftarrow 1 + \sum_{i=1}^{m} [a_i]_N$

2 : $([a]_N, \left[a^2 \mod p\right]_N, \ldots, \left[a^{m+1} \mod p\right]_N) \leftarrow \text{MULT}^*([a]_N, \ldots, [a]_N)$

3 : $[f(a) \mod p]_N \leftarrow \sum_{i=0}^{m} \alpha_i \left[a^i \mod p\right]_N$

#### Symmetrische Boolesche Funktionen

In diesem Abschnitt wird beschrieben, wie symmetrische Boolesche Funktionen auf $m$ Inputs $[a_1]_N, [a_2]_N, \ldots [a_m]_N$ mit $a_1, a_2, \ldots, a_m \in \{0, 1\}$ berechnet werden können. Dafür muss $\min(p, q) > m + 1$ für die Primfaktoren $p$ und $q$ von $N$ gelten.

Symmetrische Funktionen sind Funktionen, bei welchen das Ergebnis unabhängig von der Reihenfolge der Eingabeparameter ist. Für den booleschen Fall bedeutet dies, dass die Funktionen als $f(x_1, \ldots, x_m) = \phi(1 + \sum_{i=1}^{m} x_i)$ dargestellt werden können, da sie nur von der Anzahl der Einsen in der Eingabe abhängen. Eine symmetrische Boolesche Funktion ist zum Beispiel das Oder über $\ell$ Eingabebits, da die Funktion 1 annimmt, sobald mindestens ein Bit 1 ist. Mithilfe der Lagrange Interpolation können Koeffizienten $\alpha_i$ des Polynoms $\phi : 1, 2, \ldots, m + 1, \phi(x) = \sum_{i=0}^{\ell} \alpha_i x^i$ berechnet werden. Die Anforderung $\left|\mathbb{Z}_N^*\right| > m + 1$ ergibt sich daraus, dass die Interpolation $m + 1$ paarweise verschiedene Stützstellen benötigt. Für ein Oder mit 2 Eingabebits zum Beispiel wird ein Polynom mit $\phi(1) = 0$ und $\phi(2) = \phi(3) = 1$ gesucht. Mit Lagrange Interpolation ergibt sich dann

$$\begin{aligned}\phi(x) &= 0 \cdot \left(\frac{x-2}{1-2}\right)\left(\frac{x-3}{1-3}\right) + 1 \cdot \left(\frac{x-1}{2-1}\right)\left(\frac{x-3}{2-3}\right) + 1 \cdot \left(\frac{x-1}{3-1}\right)\left(\frac{x-2}{3-2}\right) \\ &= -2^{-1}x^2 + 5 \cdot 2^{-1}x - 2\end{aligned}$$

und damit die Koeffizienten $\alpha_0 = -2$, $\alpha_1 = 5 \cdot 2^{-1}$ und $\alpha_2 = -2^{-1}$.

Die Addition der Bitanzahl mit 1 ist notwendig, da dann die Summe in Zeile 1 in $\mathbb{Z}_N^*$ ist, wenn $\min(p, q) > m + 1$ für die Primfaktoren $p$ und $q$ von $N$ gilt. Dann kann das MULT* Protokoll sicher angewandt werden. Privacy und Korrektheit des Protokolls sind damit gewahrt.

Die Koeffizienten $\alpha_i$ können für eine feste symmetrische Boolesche Funktion von jeder Partei vorberechnet werden.

#### Präfix-Oder

In diesem Abschnitt wird das $\text{PRE}_\vee$-Protokoll von Damgard et al. [10], welches auf der Technik von Chandra, Fortune und Lipton [5] basiert, beschrieben. Es berechnet auf der Eingabe von Werten $a_1, a_2, \ldots, a_\ell \in \{0, 1\}$ gegeben als $[a_1]_N, \ldots, [a_\ell]_N$ die Ciphertexte $[b_1]_N, \ldots, [b_\ell]_N$ mit $b_i = \vee_{j=1}^{i} a_j$.

Dazu werden die Bits $a_i$ in $\lambda$ Blöcke mit jeweils $\lambda$ Bits unterteilt. Im folgenden Protokoll werden daher die Bits $a_k$ mit $a_{i,j}$ bezeichnet, wobei $k = \lambda(i-1)+j$ für $i,j \in \{1,\ldots,\lambda\}$. Für ein $i$ bezeichnet $a_{i,1},\ldots,a_{i,\lambda}$ den $i$-ten Block von $a$. Entsprechend kann die Ausgabe aus den Blöcken zusammengesetzt werden.

Das Protokoll berechnet zunächst den ersten Block, der eine 1 enthält, und die Positionen der Einsen in diesem Block. In den ersten Block werden dann die Präfix-Oder-Werte über diesen Block gesetzt. Die Blöcke mit einem höheren Index werden mit Einsen aufgefüllt. Da Oder eine symmetrische Funktion ist, kann $\vee_{k=1}^{\lambda}$ wie in vorigem Abschnitt beschrieben berechnet werden.

**Lemma 4.4.4**
*Das Protokoll $PRE_\vee$ berechnet Ciphertexte des Oders über alle Präfixe der bitweisen Eingabe a.*

**Protokoll 4.7** $\mathrm{PRE}_\vee$

$([b_1]_N,\ldots,[b_\ell]_N) \leftarrow \mathrm{PRE}_\vee([a_1]_N,\ldots,[a_\ell]_N)$

1 : **for** $i = 1,\ldots,\lambda$ **do**
2 : $\quad [x_i]_N = \vee_{j=1}^{\lambda} [a_{i,j}]_N$
3 : **endfor**
4 : **for** $i = 1,\ldots,\lambda$ **do**
5 : $\quad [y_i]_N = \vee_{k=1}^{i} [x_k]_N$
6 : **endfor**
7 : $[f_1]_N = [x_1]_N$
8 : **for** $i = 2,\ldots,\lambda$ **do**
9 : $\quad [f_i]_N = [y_i]_N - [y_{i-1}]_N$
10 : **endfor**
11 : **for** $i = 1,\ldots,\lambda; j = 1,\ldots,\lambda$ **do**
12 : $\quad [g_{i,j}]_N = \mathrm{MULT}([f_i]_N, [a_{i,j}]_N)$
13 : **endfor**
14 : **for** $j = 1,\ldots,\lambda$ **do**
15 : $\quad [c_j]_N = \sum_{i=1}^{\lambda} [g_{i,j}]_N$
16 : **endfor**
17 : **for** $j = 1,\ldots,\lambda$ **do**
18 : $\quad [b_{.,j}]_N = \vee_{k=1}^{j} [c_k]_N$
19 : **endfor**
20 : **for** $i = 1,\ldots,\lambda; j = 1,\ldots,\lambda$ **do**
21 : $\quad [s_{i,j}]_N = \mathrm{MULT}([f_i]_N, [b_{.,j}]_N)$
22 : **endfor**
23 : **for** $i = 1,\ldots,\lambda; j = 1,\ldots,\lambda$ **do**
24 : $\quad [b_{i,j}]_N = [s_{i,j}]_N + [y_i]_N - [f_i]_N$
25 : **endfor**

**Beweis** Der Wert $x_i = \vee_{j=1}^{\lambda} \left[a_{i,j}\right]_N$ ist 1, wenn der $i$-te Block eine 1 enthält und $y_i = \vee_{k=1}^{i} [x_k]_N$ ist daher 1, wenn einer der ersten $i$ Blöcke eine 1 enthält. Sei $i_0$ das kleinste $i$ mit $y_i = 1$. $i_0$ ist also der Index des ersten Blocks der eine 1 enthält. Dann ist $f_{i_0} = 1$ und für $i \neq i_0$ ist $f_i = 0$. Damit ist

$$g_{i,j} = \begin{cases} a_{i,j} & \text{wenn } i = i_0 \\ 0 & \text{sonst} \end{cases}$$

und $c_j = a_{i_0,j}$, wenn ein $i_0$ existiert und 0 sonst. Der Wert $b_{.,j}$ ist das Präfix-Oder des $i_0$-ten Blocks, $s_{i,j}$ übernimmt das Präfix-Oder in dem $i_0$-ten Block und ist 0 an allen anderen Stellen. $b_{i,j}$ wird als Summe $s_{i,j} + y_i - f_i$ berechnet. Für $i < i_0$ sind wie gewünscht $f_i = s_{i,j} = y_i = 0$. Für $i > i_0$ sind $f_i = s_{i,j} = 0$ und $y_i = 1$. Daher ist $b_{i,j} = 1$ und für $i = i_0$ ist $f_i = y_i = 1$, also ist $b_{i_0,j} = s_{i_0,j}$ das Präfix-Oder im $i_0$-ten Block. Damit berechnet das Protokoll das Präfix-Oder der Eingaben. □

**Lemma 4.4.5**
*Das Protokoll $PRE_\vee$ wahrt die Privacy der Eingabe- und Ausgabewerte.*

**Beweis** Die Privacy der Werte bleibt gewahrt, da kein Wert veröffentlicht wird und alle aufgerufenen Protokolle die Privacy wahren. □

## 4.5 Generierung eines Zufallsbits

In diesem Abschnitt wird ein Protokoll für die Generierung eines zufälligen Bits beschrieben.

Dieses Protokoll verwendet die Funktion $f : \{0,1\}^n \mapsto \{0,1\}$,

$$f(a_1, \ldots, a_n) = \begin{cases} 0 & \text{wenn } \sum_{i=1}^{n} a_i \text{ gerade.} \\ 1 & \text{sonst.} \end{cases}$$

Diese Funktion ist eine symmetrische Boolesche Funktion, kann also mit dem Protokoll aus Abschnitt 4.4 berechnet werden.

**Lemma 4.5.1**
*Das Protokoll $RAN_2$ gibt gleichverteilt zufällig einen Ciphertext eines Werts $r \in \{0,1\} \subset \mathbb{Z}_N$ aus.*

**Beweis** Solange eine Partei $P_i$ den Ciphertext eines gleichverteilt zufälligen Bits beiträgt, gibt das Protokoll den Ciphertext eines gleichverteilt zufälligen Bits aus, da für feste Bits $r_j$ mit $j \in \{1, \ldots, n\} \setminus \{i\}$ der Wert $f(r_1, \ldots, r_n)$ genau dann gleichverteilt zufällig ist, wenn $r_i$ ein gleichverteilt zufälliges Bit ist. □

**Protokoll 4.8** $\mathrm{RAN}_2$

$[r]_N \leftarrow \mathrm{RAN}_2()$

1 : **for** $i = 1, \ldots, n$ **do**
2 : $P_i$ wählt $r_i$ gleichverteilt zufällig aus $\{0,1\} \subset \mathbb{Z}_N$
3 : $[r_i]_N \leftarrow \mathrm{SHARE}(i, r_i)$
4 : $P_i$ beweist den anderen Parteien mit ZK-BOUND, dass $r_i < 2$
5 : **endfor**
6 : $[r]_N \leftarrow f([r_1]_N, \ldots, [r_n]_N)$

Sollte eine Partei $P_i$ keinen Ciphertext eines Bits $r_i$ beitragen, wird dieser durch einen festen Ciphertext zu dem Wert 0, auf welchen die Parteien sich einigen, für weitere Berechnungen ersetzt.

**Lemma 4.5.2**
*Das Protokoll $RAN_2$ wahrt die Privacy des Ausgabewerts.*

**Beweis** Solange eine Partei $P_i$ ein gleichverteilt zufälliges Bit beiträgt, ist die Ausgabe ein Ciphertext zu einem gleichverteilt zufälligen Bit. Die Parteien teilen jeweils nur sichere Ciphertexte ihrer Bits. Damit ist die Privacy des Ausgabewerts gewahrt. □

## 4.6 Bitdarstellung eines zufälligen Werts

Das Protokoll SOLVED-BITS von Damgard et al. [10] generiert gleichverteilt zufällig ein $b \in \mathbb{Z}_N$ und gibt die Ciphertexte der Bits $[b]_B$ und einen Ciphertext $[b]_N$ aus.

Das Protokoll generiert Ciphertexte von zufälligen Bits und prüft dann, ob diese eine Zahl in $\mathbb{Z}_N$ darstellen. Wenn das nicht der Fall ist, bricht das Protokoll ab.

**Lemma 4.6.1**
*Das Protokoll SOLVED-BITS gibt Ciphertexte von Bits und einen Ciphertext des Werts eines gleichverteilt zufälligen Werts $b \in \mathbb{Z}_N$ aus.*

**Beweis** Die Bits $b_i$ sind gleichverteilt zufällig, wenn $\mathrm{RAN}_2$ gleichverteilt zufällige Bits ausgibt. Damit ist $b$ gleichverteilt zufällig in $\{0, 1, \ldots, 2^\ell - 1\}$ und wenn das Protokoll nicht abbricht, ist $b < p$, also ist $b \in \{0, 1, \ldots, p-1\}$ gleichverteilt zufällig. Da $[b]_N$ aus $[b]_B$ berechnet wird, ist $[b]_B$ eine Sammlung von Ciphertexten der Bits von $[b]_N$. □

**Lemma 4.6.2**
*Die Privacy des Ausgabewerts bleibt durch das Protokoll SOLVED-BITS gewahrt.*

**Beweis** Es wird nur $c = (b <^? p)$ veröffentlicht. Da $c = 1$ für alle gültigen Ausgaben gelten muss, wird so keine zusätzliche Information preisgegeben und die Privacy des Ausgabewerts ist gewahrt.

Das Protokoll SOLVED-BITS bricht ab, wenn $b \geq N$ ist. Bei einer ungünstigen Wahl von $N \in [2^{\ell-1}, 2^\ell]$ kann die Wahrscheinlichkeit dafür bei bis zu $\frac{1}{2}$ liegen. Um eine hinreichend große Wahrscheinlichkeit für eine erfolgreiche Ausführung zu erzielen, kann das Protokoll mehrmals parallel ausgeführt werden.

Für einen Effizienzgewinn können die Bits in Zeile 2 parallel generiert werden.

**Protokoll 4.9** SOLVED-BITS

$([b]_B, [b]_N) \leftarrow$ SOLVED-BITS ()

1 : **for** $i = 0, \ldots, \ell - 1$ **do**
2 : $\quad [b_i]_N \leftarrow \text{RAN}_2()$
3 : **endfor**
4 : $[b]_B = ([b_0]_N, [b_1]_N, \ldots, [b_{\ell-1}]_N)$
5 : $[c]_N \leftarrow$ BIT-LT$([b]_B, N)$
6 : $c \leftarrow$ REVEAL$([c]_N)$
7 : **if** $c = 0$ **then**
8 : $\quad$ **abort**
9 : **endif**
10 : $[b]_N = \sum_{i=0}^{\ell-1} 2^i [b_i]_N$

## 4.7 Summe auf Ciphertexten von Bits

In diesem Abschnitt wird das Protokoll BIT-ADD von Damgard et al. [10] vorgestellt. Es führt die Addition auf Ciphertexten von Bits $[a]_B = ([a_0]_N, [a_1]_N, \ldots, [a_{\ell-1}]_N)$ und $[b]_B = ([b_0]_N, [b_1]_N, \ldots, [b_{\ell-1}]_N)$ aus und gibt Ciphertexte der Bits der Summe $[d]_B = ([d_0]_N, [d_1]_N, \ldots, [d_\ell]_N)$ aus. Dabei ist zu beachten, dass keine Modulo-Reduktion bezüglich $N$ stattfindet. Der Wert der Ausgabe von BIT-ADD ist also nicht notwendigerweise in $\mathbb{Z}_N$.

Das Protokoll BIT-ADD nutzt das Subprotokoll CARRIES, um den Übertrag der Werte zu berechnen. Damit können die Ausgabebits direkt berechnet werden.

**Lemma 4.7.1**
*Das Protokoll BIT-ADD berechnet Ciphertexte der Bits der Summe der Eingabewerte a und b.*

**Beweis** Die einzelnen Bits werden als $d_i = a_i + b_i + c_i - 2c_{i+1}$ berechnet. Dabei bezeichnet $c_i$ für $i \in \{1, \ldots, \ell\}$ den Übertrag in die $i$-te Stelle, es gilt also $c_i = ((\sum_{k=0}^{i-1} 2^k (a_k + b_k)) >^? 2^i)$. Solange also das CARRIES Protokoll die $c_i$ korrekt berechnet, ist das BIT-ADD Protokoll korrekt. □

**Lemma 4.7.2**
*Das Protokoll BIT-ADD wahrt die Privacy der Eingabe- und Ausgabewerte.*

**Protokoll 4.10** BIT-ADD

$[d]_B \leftarrow$ BIT-ADD$([a]_B, [b]_B)$

1 : $([c_1]_N, [c_2]_N, \ldots, [c_\ell]_N) \leftarrow$ CARRIES$([a]_B, [b]_B)$
2 : $[d_0]_N = [a_0]_N + [b_0]_N - 2[c_1]_N$
3 : $[d_\ell]_N = [c_\ell]_N$
4 : **for** $i = 0, \ldots, \ell - 1$ **do**
5 : $\quad [d_i]_N = [a_i]_N + [b_i]_N + [c_i]_N - 2[c_{i+1}]_N$
6 : **endfor**
7 : $[d]_B = ([d_0]_N, [d_1]_N, \ldots, [d_\ell]_N)$

**Beweis** Es wird kein Wert veröffentlicht, daher garantiert das BIT-ADD Protokoll die Privacy der Eingabe- und Ausgabewerte, wenn das CARRIES Protokoll die Privacy der Werte wahrt. □

#### CARRIES

Das Subprotokoll CARRIES basiert auf dem Carry Set/Propagate/Kill Algorithmus. Es berechnet für $i \in \{0,\dots,\ell-1\}$ den Übertrag aus dem $i$-ten Bit.

Das Tupel $(1,0,0)$ repräsentiert dabei Set, $(0,1,0)$ Propagate und $(0,0,1)$ Kill. Der Carry-Propagation Operator $\circ$ ist für dieses Protokoll definiert als

$$\circ : \{(1,0,0),(0,1,0),(0,0,1)\} \to \{(1,0,0),(0,1,0),(0,0,1)\}$$

$$A \circ B = \begin{cases} (1,0,0) & \text{wenn } B = (1,0,0) \\ (0,0,1) & \text{wenn } B = (0,0,1) \\ A & \text{wenn } B = (0,1,0) \end{cases}.$$

Das Protokoll berechnet zunächst die Tupeldarstellung der Set-, Propagate- und Kill-Zeichen und führt dann die Propagierung der Werte durch. Der Wert $c_i$ ist 1, wenn an die $i$-te Stelle mittels des Operators ein Set-Zeichen propagiert wird.

**Lemma 4.7.3**
*Das Protokoll CARRIES berechnet für $i \in \{0,\dots,\ell-1\}$ jeweils die Übertragsbits aus der $i$-ten Stelle bei der Addition der Eingabewerte $a$ und $b$.*

**Beweis** Das Protokoll berechnet zunächst das Set-, Propagate-, oder Kill-Zeichen $e_i = (s_i, p_i, k_i)$ für jede Stelle $i \in \{0,\dots,\ell-1\}$. Das Set-Bit $s_i$ wird gesetzt, wenn $a_i \in \{0,1\}$ und $b_i \in \{0,1\}$ gleich 1 sind, also lässt es sich als $a_i \cdot b_i$ berechnen. Das Propagate-Bit $p_i$ wird gesetzt, wenn $a_i$ oder $b_i$ aber nicht $s_i$ gleich 1 sind. Dies wird mit $p_i = a_i + b_i - 2s_i$ umgesetzt. Das Kill-Bit $k_i$

**Protokoll 4.11** CARRIES

$[d]_B \leftarrow \text{CARRIES}([a]_B, [b]_B)$

1: **for** $i = 0,\dots,\ell-1$ **do**
2: $\quad [s_i]_N = \text{MULT}([a_i]_N, [b_i]_N)$
3: **endfor**
4: **for** $i = 0,\dots,\ell-1$ **do**
5: $\quad [p_i]_N = [a_i]_N + [b_i]_N - 2[s_i]_N$
6: $\quad [k_i]_N = 1 - [s_i]_N - [p_i]_N$
7: $\quad [e_i]_B = ([s_i]_N, [p_i]_N, [k_i]_N)$
8: **endfor**
9: $([f_0]_B,\dots,[f_{\ell-1}]_B) \leftarrow \text{PRE}_\circ([e_0]_B,\dots,[e_{\ell-1}]_B)$
10: **for** $i = 0,\dots,\ell-1$ **do**
11: $\quad ([s_i]_N, [p_i]_N, [k_i]_N) = [f_i]_B$
12: **endfor**
13: $[c]_B = ([s_0]_N, [s_1]_N,\dots,[s_{\ell-1}]_N)$

wird gesetzt, wenn weder $s_i$ noch $p_i$ gleich 1 ist. Daraus ergibt sich dann die Darstellung $(s_i, p_i, k_i)$ der Set-, Propagate- und Kill-Zeichen. In Zeile 9 wird die Übertragung der Zeichen durchgeführt. Danach ist das Set-Bit $s_i = 1$, wenn bei der Addition ein Übertrag aus der $i$-ten Stelle entsteht. Entsprechend ist die Ausgabe korrekt. □

**Lemma 4.7.4**
*Die Privacy der Eingabe- und Ausgabewerte bleibt durch das Protokoll CARRIES gewahrt.*

**Beweis** Da keine Werte veröffentlicht werden, folgt die Privacy des Protokolls CARRIES aus der Privacy der Subprotokolle. □

Die Schleifen lassen sich parallel ausführen, um so eine Steigerung der Effizienz des Protokolls zu erzielen.

### Carry Propagation

In diesem Abschnitt wird beschrieben, wie $\circ_{i=1}^{\ell} e_i$ berechnet werden kann, wobei $e_i \in \{(1,0,0),(0,1,0),(0,0,1)\}$. Das Protokoll nutzt die Präfix-Berechnung aus folgendem Abschnitt.

Das Protokoll berechnet zunächst das Propagate-Bit des Ergebnisses, welches nur gesetzt wird, wenn alle Eingaben das Zeichen Propagate darstellen. Dann berechnet das Protokoll das Kill-Bit, indem es über führende Propagate-Zeichen ein mögliches Kill-Zeichen propagiert. Das Set-Bit wird schließlich auf 1 gesetzt, wenn weder das Propagate-Bit noch das Kill-Bit 1 sind.

**Lemma 4.7.5**
*Das Carry Propagation Protokoll berechnet einen Ciphertext des Werts des $\circ$-Operators über die Eingabewerte.*

**Beweis** Es wird offensichtlich das Propagate-Bit $b$ wie gewünscht nur dann gleich 1, wenn alle $p_i$ der Eingabe 1 sind. Dies entspricht dem Fall, in dem alle Eingaben Propagate darstellen. Das Bit $b$ wird also korrekt gesetzt. Das Bit $c$ soll gesetzt werden, wenn für $i \in \{1,\dots,\ell\}$ der Wert $k_i$ gleich 1 ist und für alle $j \in \{i+1,\dots,\ell\}$ gilt $p_j = 1$ beziehungsweise $q_{i+1} = \wedge_{k=i+1}^{\ell} p_k = 1$. $p_i$ und

**Protokoll 4.12** Carry Propagation

$([a]_N, [b]_N, [c]_N) \leftarrow \circ_{i=1}^{\ell} ([s_i]_N, [p_i]_N, [k_i]_N)$

1: $[b]_N \leftarrow \wedge_{i=1}^{\ell} [p_i]_N$
2: $([q_\ell]_N, \dots, [q_1]_N) \leftarrow \mathrm{PRE}_\wedge([p_\ell]_N, \dots, [p_1]_N)$
3: $[c_\ell]_N = [k_\ell]_N$
4: **for** $i = 1, \dots, \ell - 1$ **do**
5: $\quad [c_i]_N \leftarrow [k_i]_N \wedge [q_{i+1}]_N$
6: **endfor**
7: $[c]_N = \sum_{i=1}^{\ell} [c_i]_N$
8: $[a]_N \leftarrow 1 - [b]_N - [c]_N$

$k_i$ sind niemals gleichzeitig 1, da das $i$-te Bit-Tupel nicht gleichzeitig ein Kill und ein Propagate darstellen kann. Daher kann maximal ein $c_i = k_i \wedge q_{i+1} = 1$ sein und so kann das Oder über alle $c_i$ als Summe berechnet werden. Es gilt für das Set-Bit $a = 1$ genau dann, wenn weder $b = 1$ noch $c = 1$. Es sind nie $b$ und $c$ gleich 1, daher kann dies als $a = 1 - b - c$ berechnet werden. Damit ist die Ausgabe korrekt. □

Die Privacy der Eingaben bleibt durch die Sicherheit der aufgerufenen Subprotokolle gewahrt. Für eine Verbesserung der Effizienz lassen sich die Schleifenschritte parallel ausführen.

#### Präfix-Berechnung für assoziative Binäroperatoren

In diesem Abschnitt wird beschrieben, wie für einen assoziativen Binäroperator $\circ$ auf Eingabe von $[a_1]_B, \ldots, [a_\ell]_B$ für $i = 1, \ldots, \ell$ die Ciphertexte der Bits der Werte $b_i = \circ_{j=1}^{i} a_j$ berechnet werden können. Die $[a_i]_B$ stehen dabei für Sammlungen von Ciphertexten von Bits $(\left[a_{i,1}\right]_N, \ldots, \left[a_{i,m}\right]_N)$. Die hierfür verwendete Technik stammt von Chandra, Fortune und Lipton [6].

Für eine Vereinfachung der Notation wird von $\ell = 2^k$ für ein $k$ ausgegangen. Für den Fall $\ell \leq k$ kann Zeile 2 so modifiziert werden, dass Werte über Intervalle der Größe kleiner $2^i$ an den Rändern entstehen.

In Zeile 2 werden die Zahlen jeweils in Intervalle der Länge $2^i$ für $i \in 1, \ldots, k$ geteilt, über welche dann der Operator $\circ$ angewandt wird. Die Ergebnisse $b_i$ können dann aus den entsprechenden Intervallwerten $b_{i,j}$ zusammengesetzt werden. Dabei setzt sich $b_i$ aus höchstens $k$ Werten $b_{i,j}$ und $a_i$ zusammen, indem jeweils iterativ der Wert, der den größten Teil des verbleibenden Intervalls abdeckt, hinzugerechnet wird.

Für $\ell = 16$ würden die Parteien zum Beispiel $b_{13}$ als $b_{1,8} \circ b_{9,12} \circ a_{13}$ berechnen.

Da die aufgerufenen Subprotokolle die Privacy wahren, bleibt die Privacy der Werte gewahrt. Die Schleifen können jeweils parallelisiert werden.

**Protokoll 4.13** $\text{PRE}_\circ$

$([b_1]_B, \ldots, [b_\ell]_B) \leftarrow \text{PRE}_\circ([a_1]_B, \ldots, [a_\ell]_B)$

1 : **for** $i = 1, \ldots, k; j = 0, \ldots, k - i$ **do**
2 : $\quad \left[b_{i,j}\right]_B \leftarrow \circ_{m=j2^i+1}^{j2^i+2^i} [a_m]_B$
3 : **endfor**
4 : **for** $i = 1, \ldots, \ell$ **do**
5 : $\quad [b_i]_B \leftarrow$ Berechne $b_i$ aus den $\left[b_{i,j}\right]_B$
6 : **endfor**

## 4.8 Zerlegung in Bits

Das in diesem Abschnitt beschriebene BITS-Protokoll dient dem Unterteilen eines Ciphertexts eines Werts in eine Sammlung von Ciphertexten der Bitdarstellung des Werts. Es berechnet also die Funktion $BITS : \mathbb{Z}_N \to (\mathbb{Z}_N)^\ell, a \mapsto (a_0, a_1, \ldots, a_{\ell-1})$ mit $a = \sum_{i=0}^{\ell-1} a_i 2^i$ und $a_0, a_1, \ldots, a_{\ell-1} \in \{0, 1\}$ auf Ciphertexten in $\mathbb{Z}_N$. Es stammt von Damgard et al. [10].

Das Protokoll generiert zunächst Ciphertexte der Bits eines zufälligen Werts $b$. Die Eingabe $a$ wird durch die Subtraktion von $b$ maskiert und der maskierte Wert wird veröffentlicht. Die Parteien zerlegen den maskierten Wert dann lokal in Bits und addieren darauf $b$ in der Bitdarstellung. Danach wird die Modulo Reduktion durchgeführt, welche bei der bitweisen Addition mit dem Protokoll BIT-LT fehlt.

**Lemma 4.8.1**
*Das Protokoll BITS gibt auf Eingabe eines Ciphertexts $[a]_N$ die Ciphertexte $[a]_B$ der Bits von $a$ aus.*

**Beweis** Es gilt: $c = a - b \mod N$ und $d = c + b = (a - b \mod N) + b = a + qN$ für $q \in \{0, 1\}$. Da $a \in \mathbb{Z}_N$, ist $q = 1 \iff N \leq d$. Die Modulo-Reduktion der Summe $d$ um $N$ wird mit

$$d + g = (a + qN + q(2^\ell - N)) \equiv a \mod 2^\ell$$

umgesetzt. Die Reduktion modulo $2^\ell$ wird durch das Entfernen der zwei höchstwertigen Bits in Zeile 12 erreicht. Folglich gibt das Protokoll die Bits von $a \in \mathbb{Z}_N$ aus. □

Die Berechnung der Bitdarstellung von $2^\ell - N$ kann lokal durchgeführt werden, da sowohl $\ell$ als auch $N$ allen Parteien bekannt sind.

**Lemma 4.8.2**
*Das Protokoll BITS wahrt die Privacy des Eingabe- und des Ausgabewerts.*

**Protokoll 4.14** BITS

$[a]_B \leftarrow$ BITS $([a]_N)$

1 : $([b_0]_N, [b_1]_N, \ldots, [b_{\ell-1}]_N, [b]_N) \leftarrow$ SOLVED-BITS ()
2 : $[a - b]_N \leftarrow [a]_N - [b]_N$
3 : $c \leftarrow$ REVEAL$([a - b]_N)$
4 : $[d]_B \leftarrow$ BIT-ADD$(c, [b]_B)$
5 : $[q]_N \leftarrow$ BIT-LT$(N - 1, [d]_B)$
6 : $(f_0, f_1, \ldots, f_{\ell-1}) =$ Bitdarstellung von $\left(2^\ell - N\right)$
7 : **for** $i = 0, \ldots, \ell - 1$ **do**
8 : $\quad [g_i]_N \leftarrow f_i\,[q]_N$
9 : **endfor**
10 : $[g]_B = ([g_0]_N, [g_1]_N, \ldots, [g_{\ell-1}]_N)$
11 : $[h]_B \leftarrow$ BIT-ADD$([d]_B, [g]_B)$
12 : $[a]_B = ([h_0]_N, [h_1]_N, \ldots, [h_{\ell-1}]_N)$

**Beweis** Da $b$ gleichverteilt zufällig, geheim und unabhängig von $a$ gewählt wird, ist auch $c$ gleichverteilt zufällig. Der Wert $b$ wird nicht veröffentlicht, also wird mit der Bekanntgabe von $c$ keine Information über $a$ preisgegeben. Solange nun die Subprotokolle die Privacy der Eingaben sicherstellen, gilt dies auch für das BITS-Protokoll. □

## 4.9 Werte für die Online Phase

Mit den beschriebenen Protokollen lassen sich nun geheime Werte mit bestimmten Eigenschaften vorberechnen. In diesem Abschnitt werden einige Möglichkeiten für solche Werte vorgestellt.

### 4.9.1 Zufällige Werte

Ein beliebiges zufälliges Element $r$ aus $\mathbb{Z}_N$ können die Parteien mit einem Aufruf des Protokolls RAN generieren.

### 4.9.2 Zufällige Bits

Ein zufälliges Bit kann mit $\text{RAN}_2$ erzeugt werden.

Ein zufälliges Element $r$ aus $\mathbb{Z}_N$ in Bitdarstellung können die Parteien durch das Protokoll SOLVED-BITS erstellen.

### 4.9.3 Zufälliges invertierbares Element

Mit $\text{RAN}^*$ kann ein zufälliges Element $r$ aus $\mathbb{Z}_N^*$ zusammen mit seinem Inversen $r^{-1}$ generiert werden. Zusätzlich kann die Bitdarstellung von $r$ mit dem Protokoll BITS berechnet werden.

### 4.9.4 Zufälliger Wert mit begrenzter Größe

Um einen Wert kleiner eines öffentlichen Werts $m \in \mathbb{Z}_N$ zu generieren, können die Parteien zunächst $\lceil \log_2(m) \rceil$ Bits für einen Wert $r$ mithilfe des Protokolls SOLVED-BITS erzeugen und dann mit dem Protokoll BIT-LT prüfen, ob $r < m$ ist. Dazu legen sie die Ausgabe des BIT-LT Aufrufs offen. Wenn diese Ausgabe 0 ist, wird die Generierung abgebrochen. Die Wahrscheinlichkeit für die Ausgabe 0 ist abhängig von dem Wert $m$ bis zu $1/2$. Um eine hinreichend hohe Wahrscheinlichkeit für eine erfolgreiche Ausführung zu erhalten, können die Parteien das Protokoll mehrfach parallel ausführen.

Zusätzlich können die Parteien mithilfe von BIT-LT prüfen ob der generierte Wert größer als ein $m'$ ist. Entsprechend wird ein Wert verworfen, wenn $m'$ größer oder gleich $r$ ist.

### 4.9.5 Präfixprodukt von zufälligen Werten

Aus einer Menge von zufällig generierten Werten $r_1, \ldots, r_m$ aus $\mathbb{Z}_N^*$ können mithilfe von MULT$^*$ die Präfixprodukte $\prod_{i=1}^{j} r_i$ für $j \in \{1, \ldots, m\}$ berechnet werden. Das BITS Protokoll erlaubt hier zusätzlich die Berechnung der Bitdarstellungen.

# 5 Zusammenfassung und Ausblick

In dieser Arbeit wurden wesentliche Schritte für die Offline Phase von auf Threshold Paillier basierender MPC behandelt. In diesem Kapitel werden die Ergebnisse der Arbeit abschließend zusammengefasst.

In Kapitel 3 wurde beschrieben, wie in einem Threshold Paillier-Kryptosystem verschlüsselt und entschlüsselt werden kann. Es wurde dargelegt, wie ein RSA Modulus mit geheimer Faktorisierung und daraus ein Paillier-Schlüsselpaar mit geteiltem Secret Key ohne einzelne vertrauenswürdige Partei generiert werden kann. Das Protokoll dafür ist korrekt und stellt die Privacy des generierten Schlüssels sicher. Anschließend wurden die Schwächen des Protokolls zur Schlüsselerzeugung betrachtet. Ein Adversary kann das Protokoll jederzeit zum Abbrechen bringen, Guaranteed Output Delivery ist also nicht gegeben. Zudem ist der Zeitaufwand hoch.

In Kapitel 4 wurde gezeigt, wie darauf aufbauend die Grundbausteine für auf Threshold Paillier basierender MPC konstruiert werden können. Zudem wurden Protokolle für die Generierung von Werten in der Offline Phase aus diesen Grundbausteinen vorgestellt. Es wurde gezeigt, wie mit diesen Protokollen Paillier-Ciphertexte für folgende geheime Werte vorberechnet werden können:

- Zufällige Werte aus $\mathbb{Z}_N$
- Zufällige invertierbare Werte, also Werte aus $\mathbb{Z}_N^*$
- Zufallsbits
- Zufällige Werte aus $\mathbb{Z}_N$ in Bitdarstellung
- Zufällige beschränkte Werte, also Werte kleiner als ein $m \in \mathbb{Z}_N$
- Präfixprodukte von zufälligen Werten aus $\mathbb{Z}_N$

Diese Protokolle bieten neben Korrektheit und Wahrung der Privacy auch Guaranteed Output Delivery.

Für das Beispiel der Wahl des Betriebsratsvorsitzenden aus der Einleitung liefern diese Protokolle Werkzeuge für die Offline Phase. Da die Generierung eines Schlüssels nicht oft durchgeführt wird, ist der Aufwand für diese hier akzeptabel. Die Generierung des Schlüssels abzubrechen ist nicht lukrativ, da ein Mitglied, das gegen das Protokoll verstößt, identifiziert und zum Beispiel durch den Ausschluss aus dem Betriebsrat bestraft werden kann.

## Ausblick

Die Effizienz der Protokolle ist bei MPC oft ein Problem, welches praktische Anwendungen hindert. Für die Schlüsselerzeugung wäre es interessant zu sehen, ob die Effizienz verbessert werden kann, wenn der Adversary höchstens eine Minderheit der Parteien kontrolliert.

Es bleibt abzuwarten, welcher Einsatz sich in der Praxis aus den vorgestellten Protokollen ergibt.

# Literaturverzeichnis


[1] J. Bar-Ilan, D. Beaver. „Non-cryptographic Fault-tolerant Computing in Constant Number of Rounds of Interaction“. In: *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing*. PODC '89. Edmonton, Alberta, Canada: ACM, 1989, S. 201–209. ISBN: 0-89791-326-4. DOI: 10.1145/72981.72995. URL: http://doi.acm.org/10.1145/72981.72995 (zitiert auf S. 37, 50, 51).

[2] M. Ben-Or, S. Goldwasser, A. Wigderson. „Completeness Theorems for Non-cryptographic Fault-tolerant Distributed Computation“. In: *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*. STOC '88. Chicago, Illinois, USA: ACM, 1988, S. 1–10. ISBN: 0-89791-264-0. DOI: 10.1145/62212.62213. URL: http://doi.acm.org/10.1145/62212.62213 (zitiert auf S. 11).

[3] P. Bogetoft, D. L. Christensen, I. Damgård, M. Geisler, T. Jakobsen, M. Krøigaard, J. D. Nielsen, J. B. Nielsen, K. Nielsen, J. Pagter, M. Schwartzbach, T. Toft. „Secure Multiparty Computation Goes Live“. In: *Financial Cryptography and Data Security*. Hrsg. von R. Dingledine, P. Golle. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, S. 325–343. ISBN: 978-3-642-03549-4. DOI: 10.1007/978-3-642-03549-4_20 (zitiert auf S. 11).

[4] D. Boneh, M. Franklin. „Efficient Generation of Shared RSA Keys“. In: *J. ACM* 48.4 (Juli 2001), S. 702–722. ISSN: 0004-5411. DOI: 10.1145/502090.502094. URL: http://doi.acm.org/10.1145/502090.502094 (zitiert auf S. 24, 27, 33, 35).

[5] A. K. Chandra, S. Fortune, R. Lipton. „Lower bounds for constant depth circuits for prefix problems“. In: *Automata, Languages and Programming*. Hrsg. von J. Diaz. Berlin, Heidelberg: Springer Berlin Heidelberg, 1983, S. 109–117. ISBN: 978-3-540-40038-7 (zitiert auf S. 54).

[6] A. K. Chandra, S. Fortune, R. Lipton. „Unbounded fan-in circuits and associative functions“. In: *Proceedings of the fifteenth annual ACM symposium on Theory of computing - STOC '83*. ACM Press, 1983. DOI: 10.1145/800061.808732 (zitiert auf S. 61).

[7] D. Chaum, T. P. Pedersen. „Wallet Databases with Observers“. In: *Advances in Cryptology — CRYPTO' 92*. Hrsg. von E. F. Brickell. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, S. 89–105. ISBN: 978-3-540-48071-6. DOI: 10.1007/3-540-48071-4_7 (zitiert auf S. 17).

[8] R. Cleve. „Limits on the security of coin flips when half the processors are faulty“. In: *Proceedings of the eighteenth annual ACM symposium on Theory of computing*. ACM. 1986, S. 364–369 (zitiert auf S. 46).

[9] R. Cramer, I. Damgård, J. B. Nielsen. „Multiparty Computation from Threshold Homomorphic Encryption“. In: *Advances in Cryptology — EUROCRYPT 2001*. Hrsg. von B. Pfitzmann. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, S. 280–300. ISBN: 978-3-540-44987-4. DOI: 10.1007/3-540-44987-6_18 (zitiert auf S. 18).

[10] I. Damgård, M. Fitzi, E. Kiltz, J. B. Nielsen, T. Toft. „Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation“. In: *Theory of Cryptography Conference*. Springer. 2006, S. 285–304. DOI: 10.1007/11681878_15 (zitiert auf S. 49–52, 54, 57, 58, 62).

[11] I. Damgård, M. Jurik. „A Generalisation, a Simpli.cation and Some Applications of Paillier's Probabilistic Public-Key System“. In: *Public Key Cryptography*. Hrsg. von K. Kim. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, S. 119–136. ISBN: 978-3-540-44586-9. DOI: 10.1007/3-540-44586-2_9 (zitiert auf S. 18, 21, 22).

[12] T. Elgamal. „A public key cryptosystem and a signature scheme based on discrete logarithms“. In: *IEEE Transactions on Information Theory* 31.4 (1985), S. 469–472. ISSN: 0018-9448. DOI: 10.1109/TIT.1985.1057074 (zitiert auf S. 15).

[13] O. Goldreich, S. Micali, A. Wigderson. „How to Play ANY Mental Game“. In: *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*. STOC '87. New York, New York, USA: ACM, 1987, S. 218–229. ISBN: 0-89791-221-7. DOI: 10.1145/28395.28420. URL: http://doi.acm.org/10.1145/28395.28420 (zitiert auf S. 11).

[14] C. Hazay, G. L. Mikkelsen, T. Rabin, T. Toft, A. A. Nicolosi. „Efficient RSA Key Generation and Threshold Paillier in the Two-Party Setting“. In: *Journal of Cryptology* 32.2 (2019), S. 265–323. ISSN: 1432-1378. DOI: 10.1007/s00145-017-9275-7. URL: https://doi.org/10.1007/s00145-017-9275-7 (zitiert auf S. 15, 17–19, 21, 24, 36, 45).

[15] P. Paillier. „Public-Key Cryptosystems Based on Composite Degree Residuosity Classes“. In: *Advances in Cryptology — EUROCRYPT '99*. Hrsg. von J. Stern. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, S. 223–238. ISBN: 978-3-540-48910-8. DOI: 10.1007/3-540-48910-X_16 (zitiert auf S. 15, 16).

[16] C. P. Schnorr. „Efficient signature generation by smart cards“. In: *Journal of Cryptology* 4.3 (1991), S. 161–174. ISSN: 1432-1378. DOI: 10.1007/BF00196725. URL: https://doi.org/10.1007/BF00196725 (zitiert auf S. 17, 18).

[17] B. Schoenmakers, M. Veeningen. „Universally Verifiable Multiparty Computation from Threshold Homomorphic Cryptosystems“. In: *Applied Cryptography and Network Security*. Hrsg. von T. Malkin, V. Kolesnikov, A. B. Lewko, M. Polychronakis. Cham: Springer International Publishing, 2015, S. 3–22. ISBN: 978-3-319-28166-7. DOI: 10.1007/978-3-319-28166-7_1 (zitiert auf S. 49).

[18] A. Shamir. „How to Share a Secret“. In: *Communications of the ACM* 22.11 (Nov. 1979), S. 612–613. ISSN: 0001-0782. DOI: 10.1145/359168.359176. URL: http://doi.acm.org/10.1145/359168.359176 (zitiert auf S. 16).

[19] A. C. Yao. „Protocols for secure computations“. In: *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*. 1982, S. 160–164. DOI: 10.1109/SFCS.1982.38 (zitiert auf S. 11).

Alle URLs wurden zuletzt am 6. 10. 2019 geprüft.

**Erklärung**

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

---

Ort, Datum, Unterschrift