Universität des Saarlandes
MI Fakultät für Mathematik und Informatik
Department of Computer Science

Bachelorthesis

Implementing Private Set Intersection

submitted by

Robin Gärtner

on August 18, 2021

Reviewers

Prof. Dr. Nico Döttling

Prof. Dr. Nils Ole Tippenhauer

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Saarbrücken, August 18, 2021,

(Robin Gärtner)

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, August 18, 2021,

(Robin Gärtner)

Abstract

Diese Arbeit ist eine Analyse, des in [1] vorgestellten Protokolls. In dem Paper wurde ein verbessertes Protokoll vorgestellt, das es beliebig vielen Parteien ermöglicht, die Schnittmenge ihrer Eingabemengen sicher zu berechnen. Ein großer Teil dieser Berechnung ist die Ermittlung der Größe der Schnittmenge. Die effiziente Lösung dieses Problems ist der Fokus des Papers.
Um das Protokoll zu analysieren, habe ich die relevanten Teil-Protokolle wie im Paper beschrieben in Java implementiert, und die Funktionalitäten der weniger interessanten, auch schon in anderen Papern beschriebenen, Protokolle abgebildet. Zusätzlich habe ich für die grundlegenden Funktionalitäten auch eine additiv homomorphe Verschlüsselung in Java implementiert und diese so erweitert, dass sie auch Multiplikation ermöglicht.
Die Analyse zeigt, dass das vorgestellte Protokoll wie beschrieben funktioniert und auch effizient ist.
Dadurch, dass nun gezeigt ist, dass das Protokoll nützlich ist, kann nun Forschung im Bereich der secure computation weiter auf dem Protokoll aufbauen, mit der Gewissheit, dass es keine grundlegenden Fehler enthält.

Contents

Abbreviations

Abkürzung	**Bedeutung**
MPCT	**M**ulty **P**arty **C**ardinality **T**esting
secDT	**Sec**ure **D**egree **T**est
OLS	**O**blivious **L**inear **S**ystem solver
SUR	**S**ecure **U**nary **R**epresentation
Weitere Abkürzungen hier eintragen	

Chapter 1

Einführung

Secure Multiparty Computation ein großes Forschungsfeld in der Kryptographie, in dem zwar schon in den 1980er Jahren die Forschung begann, das aber seit den 2000er Jahren mehr Aufmerksamkeit bekommt. [2] In diesem Forschungsfeld werden Methoden erforscht, mit denen gemeinsam Funktionen von Eingabedaten berechnet werden können, ohne dass dabei die anderen teilnehmenden Parteien die Eingabedaten erhalten.
Nachdem die Forschungsergebnisse in den 80ern eher theoretisch waren, sind die neueren Forschungen eher praktisch. Denn durch mehrere Effekte wurden die Berechnungen erst in breiteren Anwendungsbereichen sinnvoll nutzbar. Einerseits wurden die berechnenden Computer stärker, beispielsweise hat sich die CPU Geschwindigkeit in PCs ungefähr verdoppelt. Das allein kann aber noch nicht die mehr als 60.000 fache Beschleunigung der Berechnungsgeschwindigkeit erklären. [2]

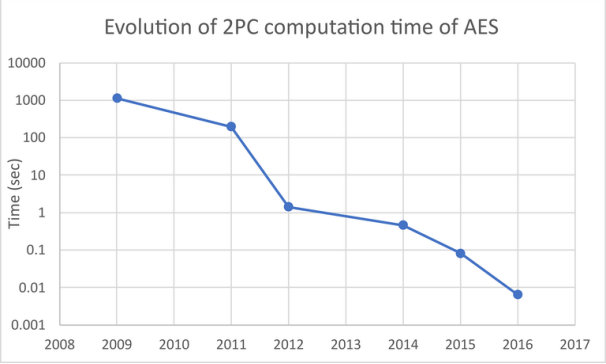

Figure 1.1: Die Veränderung der Berechnungzeit für secure computation [2]

Durch die Grafik 1.1 wird die sich immer weiter verbessernde Geschwindigkeit der secure computation Protokolle deutlich. Zur erleichterten Lesbarkeit zeigt die Grafik zwar nur die Berechnungszeiten für zwei Parteien, dennoch wird deutlich, dass sich

durch die Forschung in diesem Bereich die Effizienz von secure computation Protokollen stark gesteigert hat. Der größte Teil des Tempogewinns liegt also an den neuen oder verbesserten Protokollen, die entwickelt wurden.
Eine Funktion, die in vielen Bereichen interessant sein kann, ist die Schnittmenge der Eingabemengen der teilnehmenden Parteien. 2021 wurde das Paper [1] veröffentlicht. In diesem Paper wurde ein neues Protokoll vorgestellt, das genau diese Funktion erfüllt. Durch eine Analyse des Protokolls, kann nun festgestellt werden, ob dieses Protokoll funktioniert und effizient ist.

1.1 Anwendungsbeispiel

Der Browser Tor, den Menschen benutzen können, um in das sogenannte Darknet zu gelangen legt einen sehr großen Wert auf die Sicherheit der Nutzer. Das macht die Analyse bestimmter Statistiken sehr schwierig. Deshalb gibt es auch vom Anbieter selbst keine genauen Angaben über beispielsweise die Nutzerzahl, sondern nur Schätzungen. Um genauere Schätzungen über die Nutzerzahl zu ermöglichen, könnte man nun Daten von mehreren "Knoten" des Tor Browsers kombinieren. In diesem Fall möchte man gerne herausfinden, wie viele Überschneidungen es in den Daten der "Knoten" gibt, um Mehrfachzählungen zu vermeiden. Da viele Nutzer des Tor Browsers aber einen sehr großen Wert auf Sicherheit legen, möchte man natürlich tunlichst vermeiden, Daten von mehreren "Knoten" miteinander zu kombinieren, weil so möglicherweise Informationen gewonnen werden können, die geheim sein sollten.
Um zu garantieren, dass die Daten beispielsweise nur zur Ermittlung der Größe der Schnittmenge genutzt werden, können spezielle Protokolle genutzt werden, die die Daten nur verschlüsselt benutzen und nur ganz bestimmte Analysen auf den verschlüsselten Daten erlauben. So können Statistiken über den Tor Browser erstellt werden, ohne dass die Gefahr besteht, dass Informationen der Nutzer zugänglich werden.

1.2 Zielsetzung

Das Ziel der Arbeit ist es, die Effizienz der im Paper [1] vorgestellten neuen Protokolle zu testen. Einerseits, um zu demonstrieren, dass die vorgestellten Protokolle und Funktionalitäten korrekt funktionieren. Andererseits, um die neuen Funktionalitäten und ?Verschnellerungen? allen zugänglich zu machen.
Um die Funktionalitäten wie im Paper [1] beschrieben zu implementieren, werden auch Implementierungen von anderen Veröffentlichungen benötigt ([3] Beispielsweise für das secureRank Teilprotokoll). Da zu diesem Zeitpunkt noch keine derartigen

Implementierungen veröffentlicht sind, kann auch nicht auf diese Zurückgegriffen werden. Um trotzdem zu demonstrieren, dass das im [1] vorgestellte Protokoll funktioniert und effizient ist, habe ich die Unterprotokolle, die auf solche externen Paper zurückgreifen auf unsichere Weise implementiert. Dadurch funktioniert das Protokoll, und die unsicheren Unterprotokolle können ohne viel Zeitaufwand durch sichere Implementierungen ersetzt werden, wenn die Funktionalitäten der anderen Veröffentlichungen implementiert sind.

Chapter 2

Grundlagen

2.1 Homomorphe Verschlüsselungen

Homomorphe Verschlüsselung ist eine Form der Verschlüsselung, die bestimmte Berechnungen auf verschlüsselten Daten erlaubt, um dann ein verschlüsseltes Ergebnis zurückzugeben. Wenn dieses dann wieder entschlüsselt wird, ist das Ergebnis das gleiche, wie wenn diese Arbeitsschritte auf dem unverschlüsselten Anfangswert ausgeführt worden wären. [4]
In dieser Arbeit wird im speziellen additiv homomorphe Verschlüsselung benutzt. Diese erlaubt das Addieren von zwei verschlüsselten Zahlen und das Multiplizieren einer verschlüsselten mit einer unverschlüsselten Zahl.
Diese Funktionalitäten sind für das Protokoll wichtig, denn in jedem Unterprotokoll wird mit verschlüsselten Daten gerechnet. Und nur so können wir trotz einer sicheren Verschlüsselung mit den Daten rechnen, ohne den Inhalt der Verschlüsselung zu kennen.

2.2 Secure Computation

Secure computation ist ein Teil der Kryptographie. Das Ziel der secure computation ist es, den Teilnehmern des Protokolls zu erlauben eine Funktion ihrer geheimen Eingabewerte zu berechnen, ohne dass etwas anderes als das gewollte Ergebnis der Berechnung öffentlich wird. [5] Die grundlegendste Form der secure computation erlaubt nur die Teilnahme von Zwei unabhängigen Parteien, viele Protokolle erlauben jedoch sichere Berechnungen zwischen mehreren teilnehmenden Parteien. Die wichtigsten Eigenschaften der Protokolle sind Privacy und Correctness. Privacy bedeutet, dass niemand Informationen über die Eingaben der anderen erhält (außer das ist Teil des Rückgabewerts). Correctness bedeutet, dass jeder Teilnehmer sicher sein kann, dass das Ergebnis korrekt ist. [5]

Es gibt mehrere mathematische Grundlagen, auf denen secure computation aufbauen kann. Eine Grundlage ist beispielsweise Shamir´s Secret Sharing. Secure computation hat viele mögliche Einsatzbereiche. Bei Einigen steht die Privacy und bei Anderen die Correctness im Vordergrund [5]. In vielen Bereichen können dann Protokolle, bei denen man vorher den anderen Parteien vertrauen musste, durch Protokolle ersetzt werden, die kein Vertrauen in die anderen Parteien erfordern, und das ist hat viele Vorteile, auch wenn dafür die Kosten der Berechnung steigen.

Chapter 3

Multiparty Cardinality Testing

3.1 Neues am Paper

Diese Arbeit basiert auf dem Protokoll, das in [1] vorgestellt wurde. Es gab schon vorher einige Veröffentlichungen zum Thema Private Set Intersections, auf denen das Paper aufbauen kann. Zuerst wurden Protokolle für zwei Parteien entworfen, dann auch für mehrere Parteien. Das Paper [6] ist die letzte Veröffentlichung, auf der das Protokoll aufbaut.
Die große Neuerung in der Veröffentlichung [6] ist, dass die Kommunikations-Komplexität nun vor allem von O(t) (also dem threshold, oder der Anzahl an erlaubten Abweichungen) und nur noch logarithmisch von O(n) (also der Größe der Eingabemengen) abhängt. [6] Das von Gosh vorgeschlagene Cardinality Testing ist jedoch noch nicht für mehrere Parteien optimiert. Daher ist das neue Protokoll zur Bestimmung der Schnittmengen-Größe die wichtigste Neuerung des Papers [1]

3.2 Abwandlungen zum Paper

3.2.1 Broadcasts

Das Protokoll wurde offensichtlich entworfen um auch mit mehreren teilnehmenden Parteien effizient zu sein. Das Paper [1] nutzt also in vielen Unterprotokollen Broadcast-Funktionen.

Beispiel: secRank [1]

```
1 Each party Pi broadcasts an encrypted uniformly chosen at
2 random unit upper and lower triangular Toeplitz matrices...
```

Beispiel: secMult [1]

```
6  Each party Pi computes [...] and broadcasts c(i).
```

Ich habe versucht, nah an die Spezifizierungen des Papers ran zu kommen, doch um die Implementierung des Protokolls zu vereinfachen und um das Testen der Implementierung zu erleichtern, habe ich das System etwas abgewandelt. In meiner Implementierung gibt es einen Koordinator. Dieser Koordinator erleichtert das testen enorm, denn durch ihn kann sicher gestellt werden, dass alle "verschickten" Informationen immer zum richtigen Zeitpunkt am richtigen Ziel ankommen.
Diese Änderung wird die Sicherheit des Protokolls nicht schwächen, weil alle Informationen, die per Broadcast verschickt werden, immer verschlüsselt sind. Auch wenn Daten entschlüsselt werden, sind sie immer für einen Angreifer, der Informationen über die Eingabemengen erhalten will, nutzlos. Ein gutes Beispiel dafür ist das Unterprotokoll secInv

Protocol 6 Secure Matrix Invert secInv

Setup: Each party has a secret key share sk_i for a public key pk of a TPKE $\mathsf{TPKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$.
Input: Party P_1 inputs $\mathsf{Enc}(\mathsf{pk}, \mathbf{M})$ where $\mathbf{M} \in \mathbb{F}^{t \times t}$ is a non-singular matrix.

1: Each party P_i samples a non-singular matrix $\mathbf{R}_i \leftarrow\!\!\$\, \mathbb{F}^{t \times t}$.
2: Set $\mathsf{Enc}(\mathsf{pk}, \mathbf{M}') := \mathsf{Enc}(\mathsf{pk}, \mathbf{M})$.
3: **for** i from 1 to N **do**
4: $\quad \mathsf{P}_i$ calculates $\mathsf{Enc}(\mathsf{pk}, \mathbf{M}') = \mathsf{Enc}(\mathsf{pk}, \mathbf{R}_i \mathbf{M}')$
5: $\quad \mathsf{P}_i$ broadcasts $\mathsf{Enc}(\mathsf{pk}, \mathbf{M}')$.
6: **end for**
7: All parties mutually decrypt the final $\mathsf{Enc}(\mathsf{pk}, \mathbf{M}')$. Then they compute its inverse to obtain $\mathsf{Enc}(\mathsf{pk}, \mathbf{N}') = \mathsf{Enc}(\mathsf{pk}, \mathbf{M}'^{-1} \prod_i \mathbf{R}_i^{-1})$.
8: **for** i from N to 1 **do**
9: $\quad \mathsf{P}_i$ computes $\mathsf{Enc}(\mathsf{pk}, \mathbf{N}') = \mathsf{Enc}(\mathsf{pk}, \mathbf{N}' \mathbf{R}_i^{-1})$.
10: $\quad \mathsf{P}_i$ broadcasts $\mathsf{Enc}(\mathsf{pk}, \mathbf{N}')$
11: **end for**
12: Finally, P_1 outputs $\mathsf{Enc}(\mathsf{pk}, \mathbf{M}^{-1}) = \mathsf{Enc}(\mathsf{pk}, \mathbf{N}')$.

Figure 3.1: Teil-Protokoll secInv
[1]

Hier wird zwar eine Matrix entschlüsselt (7), diese ist aber randomisiert(1-5), also nutzlos, außer, man besitzt eine der geheimen Matrizen(9), die ja nicht verschickt werden.
Das Protokoll ist also auf eine Art und Weise aufgebaut, dass kein passiver Zuhörer irgendwelche nützlichen Informationen aus den Nachrichten, die zwischen den Parteien verschickt werden, gewinnen kann. Also kann auch ein Angreifer, der alle Nachrichten zwischen den Teilnehmern des Protokolls erhält keine Informationen extrahieren. Also können wir auch sicher sein, dass, selbst wenn der Koordinator von einem Angreifer kontrolliert wird, die verschlüsselten Eingabedaten sicher sind.
Die neue Struktur, die einfacher zu implementieren und testen ist, macht das Protokoll

also nicht weniger sicher. Besonders deshalb, weil das Protokoll so konstruiert ist, dass selbst alle verschickten Nachrichten zusammen keine geheimen Daten preisgeben.

3.2.2 Koordinieren der Parteien

Die meisten der Unterprotokolle bestehen aus sich abwechselnden Teilen von Kommunikation zwischen den Parteien und Berechnungen der Parteien. Also habe ich die Unterprotokolle implementiert, indem die Berechnungen der Parteien in Abschnitte aufgeteilt sind, die dann von dem Koordinator aufgerufen werden können. Gut zu sehen ist das beispielsweise im Teil-Protokoll MPCT, wo der Koordinator nur die Koordinierung der Parteien übernimmt, indem die Parteien die richtigen Eingaben zum richtigen Zeitpunkt bekommen.

```
1  public boolean MPCT(List<BigInteger> inputAlphas, BigInteger setMod){
2          MPCTCounter++;
3
4          List<List<EncryptedNumber>> encPointsList = new LinkedList<>();
5          //line 1 already done in setup
6
7          //line 2
8          for (int i = 0; i < parties.size(); i++) {
9              encPointsList.add(parties.get(i).MPCTpart1(inputAlphas, setMod));
10         }
11         //line 3
12         return parties.get(0).MPCTpart2(encPointsList, inputAlphas, t);
13     }
```

Das ist die einfachste Möglichkeit, um zu erreichen, dass alle Parteien zum richtigen Zeitpunkt das richtige Berechnen, denn einige der Berechnungen hängen auch von den gesendeten Nachrichten der anderen Parteien ab.

3.3 andere Veröffentlichungen

Es gibt einige andere Veröffentlichungen in den letzten Jahren, die sich mit ähnlichen Problemen oder sogar dem gleichen Problem beschäftigen. Das Problem der Private Threshold Set Intersection lässt sich in zwei Teilprobleme aufteilen. Zum Einen in das Berechnen der Schnittmenge, worauf Gosh sich in [6] konzentriert, und das sogar noch erweitert wurde, um auch für mehr als 2 Parteien nutzbar zu sein.[1] Und zum anderen in der Ermittlung der Größe der Schnittmenge. Das ist der Fokus des Papers

[1]. Aber auch Badrinarayanan [7] beschäftigt sich mit der Ermittlung der Größe der Schnittmenge. Badrinarayanan nutzt jedoch andere kryptographische Annahmen und erhält die Ergebnisse mithilfe von anderen mathematischen Grundlagen. [1]
Es gibt also viele neue Entwicklungen in diesem Bereich, die für unterschiedliche Zwecke genutzt, oder kombiniert werden können, um die besten Ergebnisse zu erreichen. Das Protokoll aus [1] ist also eine relevante Neuerung und eine Analyse kann helfen, die Effizienz dieser Neuerung mit anderen zu vergleichen.

Chapter 4

Funktion der Software

4.1 Verschlüsselungen

4.1.1 Die Wahl der Verschlüsselung

Im Paper [1] wird kein Verschlüsselungsverfahren genannt. Jedes additiv homomorphe Verschlüsselungsverfahren kann theoretisch genutzt werden. Es werden jedoch einige Vorschläge gemacht. Einer dieser Vorschläge ist das Paillier-Kryptosystem. Das Damgard-Jurik System, für das ich mich entschieden habe, ist eine Verallgemeinerung des Paillier-Kryptosystems. Unter anderem ist die Damgard-Jurik Verschlüsselung besser für Berechnungen mit mehreren Beteiligten geeignet, denn es gibt auch eine Threshold-Variante dieses Verschlüsselungs-Systems. [8] Ich habe mich bei der Verschlüsselung für das Damgard-Jurik Cryptosystem entschieden, weil es gut zu den Anforderungen, die wir haben, passt. Es ist speziell für den Anwendungsfall mit mehreren Parteien entworfen und ist deshalb auch in unserem Fall effizient.

4.1.2 Damgard-Jurik Verschlüsselung

Indem wir Shamirs Secret Sharing benutzen, können wir einen public Key und eine beliebige Anzahl an Private Key-Shares erstellen. Shamirs Secret Sharing basiert auf Polynominterpolation und kann beispielsweise auch genutzt werden, um dem System später noch neue Parteien hinzuzufügen.[9]
Durch die Nutzung der Damgard-Jurik Verschlüsselung kann jeder, der den public Key besitzt, Daten verschlüsseln. Entschlüsseln von Daten ist jedoch komplexer. Um Daten zu entschlüsseln, müssen genügend Parteien mit Private Key-Shares die Daten teilweise

entschlüsseln. Danach müssen die partiellen Entschlüsselungen kombiniert werden. Hat man aber weniger als die benötigte Anzahl an partiellen Entschlüsselungen, werden keine Informationen sichtbar. [8]
Einer der Hauptgründe, wegen dem ich mich für die Damgard-Jurik-Verschlüsselung entschieden habe ist, das man zum Entschlüsseln von Daten den Private Key nicht wiederherstellen muss. Dadurch kann der Schlüssel nicht missbraucht werden, um auch andere Dinge zu entschlüsseln. Dadurch müssen die Parteien keiner anderen Partei trauen und die Parteien können entscheiden, was entschlüsselt werden soll und was nicht. [8]
Die Verschlüsselung erfüllt natürlich auch die andere große Anforderung, die wir haben. Die Verschlüsselung ist additiv homomorph, was wir für die Funktionalität des Protokolls brauchen. [8]

4.1.3 Übertragung der Python Implementierung

Für die Implementierung der Verschlüsselung habe ich mich an der Python Implementierung [10] orientiert. Da die Sicherheit der Verschlüsselung auf mathematischen Annahmen basiert [11], muss mit großen Zahlen (mehr als 32 Bits) gerechnet werden, um sicher gegen Brute-Force-Angriffe zu sein. Daher entstehen bei der Übertragung der Python Implementierung nach Java einige Probleme. Da in Python Ganzzahlen beliebig groß werden können, aber in Java Ganzzahlen auf 32 Bit begrenzt sind, muss ich auf BigInteger zurückgreifen, um verschlüsselte Zahlen korrekt darstellen zu können. Ein ähnliches Problem ist beispielsweise das Erstellen von zufälligen Primzahlen von bestimmter Länge. In Python stellt das kein Problem dar, aber in Java ist das erstellen von beliebig langen Primzahlen komplizierter. Trotz dieser Probleme habe ich eine funktionierende Implementierung der Damgard-Jurik Verschlüsslung geschaffen, die ich dann nutzen kann, um die Implementierung der komplexeren Teil-Protokolle zu testen.

4.2 Architektur

4.3 Ring Realisierung

Da in vielen der Unterprotokollen (OLS, ODT, secMult, secRank, secInv, SUR) Berechnungen über endlichen Körpern stattfinden, um Korrektheit und Sicherheit zu gewährleisten, benötigen wir eine robuste und effiziente Möglichkeit, mit endlichen Körpern zu arbeiten. Da wir unter anderem lineare Gleichungssysteme über endlichen Körpern lösen müssen, und es nur wenige Java-Bibliotheken gibt, die diese Funktionalität anbieten, ist die

Open-Source Bibliothek JLinalg [12] gut für unseren Fall geeignet. Die Bibliothek bietet aber standardmäßig nur eingeschränkte Funktionalitäten für beliebige Körper an. Das Lösen von linearen Gleichungssystemen ist eines der Dinge, die aber nur in den Rationalen Zahlen oder dem Körper {0;1} möglich sind. Daher war es augenscheinlich am einfachsten, die Bibliothek zu erweitern, indem ich eine neue Klasse erstellt habe, wodurch die Bibliothek jetzt auch andere Funktionalitäten über beliebig großen Körpern berechnen kann. Unter anderem kann so auch das lineare Gleichungssystem gelöst werden.

```
1         private FModular(BigInteger val)
2         {
3             value = val.mod(modulus);
4         }
5
6         public static FModularFactory FACTORY(BigInteger i) {
7             modulus = i;
8             return FACTORY;
9         }
10        @Override
11        public FModular add(FModular val)
12        {
13            return new FModular(value.add(val.value).mod(modulus));
14        }
15
16
17        @Override
18        public FModular subtract(FModular val)
19        {
20            return new FModular((value.subtract(val.value)).mod(modulus));
21        }
22
23        @Override
24        public FModular multiply(FModular val)
25        {
26            return new FModular(value.multiply(val.value).mod(modulus));
27        }
28
29        @Override
30        public FModular divide(FModular val)
31        {
32            return new FModular(value.multiply(val.value.modInverse(modulus)).mod(modulus));
33        }
```

Listing 4.1: Ausschnitt aus Teil-Protokoll secDT

4.4 sichere Matrix Berechnungen

Das grundlegendste Teil-Protokoll, das im Paper [1] vorgestellt wird, ist das secMult Protokoll, das genutzt werden kann, um verschlüsselte Matrizen, also Matrizen mit

verschlüsselten Einträgen miteinander zu multiplizieren. Das ist ein wichtiger Bestandteil des Protokolls, denn diese Funktion wird in vielen Teil-Protokollen genutzt.

```
1         All parties mutually compute Enc(pk, N) = Enc(pk, XUML) via three invocations of FOMM.
```

Listing 4.2: Ausschnitt aus dem secRank Teil-Protokoll [1]

Interessanterweise lassen sich dadurch auch verschlüsselte Zahlen multiplizieren, indem Matrizen mit nur einem Eintrag erstellt werden und diese dann multipliziert werden. Dadurch, dass wir nun verschlüsselte Zahlen addieren können (durch die additiv homomorphe Verschlüsselung) und wir jetzt auch verschlüsselte Zahlen multiplizieren können, erreichen wir ähnliche Funktionalität, wie bei voll-homomorpher Verschlüsselung. Um Zahlen zu multiplizieren, brauchen wir zwar die Mithilfe der anderen Parteien, aber nicht das Vertrauen der anderen Parteien. Denn die anderen Parteien müssen zwar ihren Teil des Secret-Keys benutzen, um ihren Teil zur Matrix Multiplikation beizutragen, aber durch das gewählte Verschlüsselungsverfahren müssen sie weder ihren Secret-Key-Share verschicken, noch werden Informationen über ihren Secret-Key-Share bekannt.
Durch die nun praktisch voll-homomorphe Verschlüsselung bekommen wir viele mathematische Möglichkeiten, die wir in den anderen Teil-Protokollen nutzen können. Ein Beispiel für die entstandenen Möglichkeiten ist die Polynommultiplikation im Teil-Protokoll secDT [1].

4: All parties compute the polynomials $C_r^{(1)}(x) = \sum_{j=0}^{t} \mathbf{c}_{r,j}^{(1)} x^{t-j}$, and $C_r^{(2)}(x) = x^t + \sum_{j=1}^{t} \mathbf{c}_{r,j-1}^{(2)} x^{t-j}$, for $r \in \{v, w\}$, then compute

$$\mathsf{Enc}(\mathsf{pk}, z) = \mathsf{Enc}(\mathsf{pk}, C_v^{(1)}(x) \cdot C_w^{(2)}(x) - C_w^{(1)}(x) \cdot C_v^{(2)}(x))$$

Figure 4.1: Ausschnitt aus Teil-Protokoll secDT [1]

Um die verschlüsselten Polynome Cv(1) und Cw(2) miteinander zu multiplizieren, müssen wir verschlüsselte Zahlen miteinander multiplizieren. Wir können das Problem zwar teilweise umgehen, indem wir die verschlüsselten Polynome erst auswerten, wofür wir noch keine Multiplikation von verschlüsselten Zahlen nutzen müssen. In diesem Fall müssen wir aber danach die Ergebnisse der Auswertungen multiplizieren. In jedem Fall ist die Berechnung nur durch die erweiterte Funktionalität möglich.

Chapter 5

Analyse

5.1 Die Effizienz von MPCT

Das Protokoll MPCT, das die Größe der Schnittmenge der Eingabemengen sicher berechnet, ist bei meinen Tests auch bei insgesamt 100*99, also 9.900 Eingaben sehr schnell gewesen und hat auch bei rund 10.000 Eingaben schon nach weniger als einer Sekunde an secDT übergeben.

```
1 MPCT start: 2021-08-05 15:28:12.345
2 MPCT end: 2021-08-05 15:28:13.007
```

Listing 5.1: Ausschnitt von Rückgabe von Test MPCTTestBig. Dauer von MPCT ohne SDT

Das Protokoll MPCT benötigt also nur wenige zeitintensive Berechnungen, um die Eingabemengen so vorzubereiten, dass das nächste Teil-Protokoll secDT das Ergebnis berechnen kann. Die Berechnungsdauer von MPCT hängt also zu einem großen Teil von der Berechnungsdauer von secDT ab.

5.2 Die Effizienz von secDT

Das Protokoll secDT, das berechnet, ob der Grad der Eingabepolynome kleiner als ein gegebener Wert ist, hat in meiner Analyse länger gebraucht, als MPCT ohne secDT. Das liegt jedoch daran, dass secDT auf mehreren anderen Teilprotokollen basiert. Diese anderen Teilprotokolle, wie OLS sind teilweise sehr rechenintensiv. Vor allem, da sie nicht wie im Paper beschrieben implementiert werden konnten. Wenn man aber die

Berechnungszeit der anderen Teilprotokolle abzieht kann man einen besseren Überblick erhalten, wie effizient seccDT ist.

```
1  SDT start: 2021-08-05 15:28:13.031
2  getRank start: 2021-08-05 15:28:13.046
3  getRank end: 2021-08-05 15:28:24.265
4  getRank start: 2021-08-05 15:28:24.281
5  getRank end: 2021-08-05 15:28:36.097
6  getRank start: 2021-08-05 15:28:36.128
7  getRank end: 2021-08-05 15:28:47.155
8  getRank start: 2021-08-05 15:28:47.171
9  getRank end: 2021-08-05 15:28:59.148
10 OLS start: 2021-08-05 15:28:59.18
11 OLS end: 2021-08-05 15:29:10.59
12 OLS start: 2021-08-05 15:29:10.59
13 OLS end: 2021-08-05 15:29:22.251
14 SDT end: 2021-08-05 15:29:29.44
```

Listing 5.2: Ausschnitt von Rückgabe von Test MPCTTestBig. Dauer von SDT

1:16 min hat die die Berechnung von SDT insgesamt gedauert. Die Berechnung der vier Aufrufe von secRank hat rund 0:34 min gedauert und die Berechnung der zwei Aufrufe von OLS rund 0:25 min. Wenn man also die Berechnungszeit der beiden ineffizient implementierten Teilprotokolle abzieht, benötigt auch SDT nur rund 17 Sekunden um das Ergebnis der 42 verschlüsselten Eingaben zu berechnen. Diese 42 verschlüsselten Eingaben entsprechen ebenfalls den fast 10.000 Eingaben, die MPCT erhalten hat.

5.3 Die Berechnungen in beiden Protokollen

Im Folgenden werden die Berechnungen in secDT und MPCT zusammen betrachtet. Die Berechnungen in anderen Teilprotokollen werden nicht analysiert, weil sie nicht wie im Paper [1] beschrieben implementiert wurden.

Testname	**Parteien**	**Mengengröße**	**threshold**	**decrypt**	**encrypt**	**Berechnunge**
MPCTTest	2	5	2	8	86	378
MPCTTest10Numbers	2	10	2	8	86	378
MPCTTest4	2	5	4	8	126	618
MPCTTestBigThreshold	2	11	10	8	246	1722
MPCTTest10Parties	10	5	2	40	790	12058
MPCTTestBig	40	99	10	160	11646	670826

Table 5.1: Tabelle der Testergebnisse

5.3.1 Erklärung der Tabelle

HIER TEXT EINFÜGEN

5.3.2 Analyse der Testergebnisse

In der Tabelle 5.1 kann man gut die Auswirkungen der unterschiedlichen Eingabeparameter auf die Berechnungszeit vergleichen. Da die Anzahl der Aufrufe der Teilprotokolle sich nicht ändert und die Anzahl der Entschlüsselungen auch immer gleich bleibt, ist die Anzahl der Verschlüsselungen eine gute Abschätzung der Kosten für die beiden Teilprotokolle MPCT und secDT.
Wie an Test MPCTTest10Numbers zu sehen, hat die Größe der Eingabemengen keine Auswirkungen auf die Anzahl der Verschlüsselungen oder Entschlüsselungen in den beiden getesteten Protokollen. Auch die Anzahl der verschlüsselten Berechnungen verändert sich nicht, wenn die Eingabemengen größer werden. Die Berechnungszeit auf meinem Gerät liegt bei beiden Protokollen (MPCTTest und MPCTTest10Numbers) zwischen 50 und 100 Millisekunden. Die Berechnungen hängen also nicht bedeutend von der Größe der Eingabemengen ab. Die Protokolle wurden entworfen, um diese Eigenschaft zu erfüllen. [1] Und diese Eigenschaft wird auch von meinen Tests bestätigt.
Im Gegensatz dazu hat die Veränderung der Anzahl beteiligten Parteien eine Auswirkung auf die Kosten der Protokolle. Wie zu sehen bei Test MPCTTest10Parties hat die Vergrößerung der Anzahl der teilnehmenden Parteien auch die Anzahl an Verschlüsselungen deutlich erhöht. Die Anzahl der Verschlüsselungen und die der verschlüsselten Berechnungen steigt mit zusätzlichen Teilnehmern deutlich an. Die Steigerung ist sogar größer als die Steigerung, die bei einer Veränderung des thresholds auftritt.
Das ist erkennbar, wenn man die beiden Tests MPCTTest10Parties und MPCTTestBigTreshold vergleicht. Um die erlaubten Abweichungen im Test MPCTTestBigTreshold auf neun zu erhöhen musste ich auch die Mengengröße auf 10 erhöhen, da das Protokoll nur sinnvoll für die Berechnung ist, wenn die erlaubten Abweichungen geringer sind, als die Mengengröße. Andernfalls wäre der Rückgabewert immer true.
Wie im vorigen Abschnitt zu sehen verändert die Mengengröße jedoch nicht die Anzahl der Verschlüsselungen. Dadurch wird das Ergebnis also nicht verfälscht. Wenn man nun also die Anzahl der Verschlüsselungen der beiden Tests vergleicht, sieht man, dass eine Veränderung des threshold von Zwei auf Zehn eine geringere Steigerung nach sich zieht, als die Steigerung der Teilnehmeranzahl von Zwei auf Zehn.
Auch die Anzahl an kostenintensiven Entschlüsselungen steigt nur, wenn sich die Anzahl der Parteien erhöht. Die Veränderung der Teilnehmeranzahl hat also die größten Auswirkungen auf die Berechnungskosten des Protokolls. Ein ähnliches Ergebnis ist auch in der Berechnungszeit der Protokolle zu beobachten.
Die Berechnungszeit der beiden Tests MPCTTest10Parties und MPCTTestBigThreshold ist jedoch ähnlich, da die Unterprotokolle OLS und secRank im Test MPCTBigThreshold länger für ihre Berechnungen brauchen.
Die Veränderung der Größe des Körpers über dem die Berechnungen stattfinden hatte

in meinen Tests wiederum keine messbaren Auswirkungen. Das ist auch erwartbar, denn die einzigen Berechnungen, bei denen der Körper relevant ist, sind Berechnungen mit BigIntegers. Und in diesen Berechnungen wird nur der Modulus verändert. Die Veränderungen sollten also kaum Auswirkungen auf die Berechnungszeit haben.

5.4 Analyseergebnisse

5.4.1 Einordnung der Ergebnisse

Bei den durchgeführten Tests wurden alle Berechnungen vom Test ausgeführt. Wenn das Protokoll im Einsatz ist, werden diese Berechnungen jedoch auf die unterschiedlichen teilnehmenden Parteien aufgeteilt. Die Analyse im vorangegangenen Kapitel bezieht die Berechnungen von allen teilnehmenden Parteien (und dem Koordinator) mit ein. Betrachtet werden also die Gesamtkosten des Protokolls, jedoch nicht die Berechnungen pro teilnehmender Partei.
Wenn man davon ausgeht, dass die Berechnungen gleichmäßig auf alle Parteien aufgeteilt werden, haben wir bei dem Test MPCTTestBigThreshold 861 Berechnungen pro Partei. Bei dem Test MPCT10Parties bekommen wir einen Wert von 1205 Berechnungen pro Partei. Die Veränderung der Teilnehmeranzahl hat also Große Auswirkungen auf die Anzahl der Gesamtberechnungen, beeinflusst die Berechnungen pro Partei jedoch nur rund doppelt so stark, wie der threshold.
Die Berechnungen werden jedoch nicht ganz gleichmäßig auf die Parteien aufgeteilt, denn beide Teilprotokolle, vor allem das Teilprotokoll SDT, ist asymmetrisch. Der erste Schritt von SDT wird nur von einer einzelnen Partei berechnet.

```
1 P1 sets [...] It homomorphically generates an encrypted linear system[...]
```

Listing 5.3: Ausschnitt des Teilprotokolls SDT [1]

Daher muss eine Partei mehr Berechnungen als die anderen ausführen, was den vorangegangenen Durchschnittswert ungenau werden lässt.

5.4.2 Aussage der Ergebnisse

Die Berechnungszeit der Teilprotokolle MPCT und SDT hängt nicht von der Anzahl der Eingaben ab. Die Veränderung des thresholds und der Teilnehmeranzahl beeinflussen jedoch die Berechnungszeit. Sowohl die Anzahl der Gesamtberechnungen als auch die Berechnungen pro Partei werden von der Teilnehmeranzahl stärker beeinflusst, als

vom threshold. Das ist interessant, denn das vorstellende Paper [1] geht von einer Kommunikationskomplexität von O(N*t*t) aus. Sie steigt also mit dem threshold stärker an, als mit der Teilnehmeranzahl. Das ist jedoch kein Widerspruch, denn das Paper gibt mit O(N*t*t) die Menge der Kommunikation, nicht die der Berechnungen an.
Das zentrale Ziel des Papers [1] ist also erreicht, denn das MPCT Protokoll und das SDT Protokoll wurde entworfen, damit die Laufzeit der Protokolle nicht mehr von der Anzahl der Eingaben abhängt, was meine Tests bestätigen.

Chapter 6

Probleme der Software

6.1 Nicht alle Protokolle sicher implementiert

Da einige der Protokolle auch auf anderen Veröffentlichungen basieren, für die keine Implementierungen zu finden waren, hat die Zeit nicht ausgereicht, um alle Protokolle sicher zu implementieren. Deshalb musste ich einige Teil-Protokolle unsicher implementieren, um die Funktionalität des Protokolls zeigen zu können.

```
1  private List<EncryptedNumber> OLS (EncMatrix matrix, EncMatrix vector){
2
3          //decrypt the Matrix so the library can handle it
4          Matrix<FModular> fMatrix = new Matrix<>(decryptMatrix(matrix));
5
6          // decrypt the Vector so the library can handle it
7          Vector<FModular> fVector = new Vector<>(decryptMatrix(vector));
8
9          // calculate the solution
10         Vector<FModular> res = LinSysSolver.solve(fMatrix, fVector);
11
12         //encrypt the result of the libary computation
13         return encrypt(res);
14 }
```

Listing 6.1: Ausschnitt aus unsicherer Implementierung von OLS (vereinfacht)

Wie im vorangehenden Codeausschnitt 6.1 zu sehen, habe ich, um die Funktionalität von OLS zu simulieren, die verschlüsselten Matrizen in der Eingabe erst entschlüsselt, dann das Ergebnis berechnet, und am Ende das Ergebnis wieder verschlüsselt. Dadurch berechnet diese Implementierung zwar das richtige Ergebnis, auch wenn das Teilprotokoll im Paper [1] anders beschrieben ist. Durch diese Implementierung hat

das Protokoll eine längere Laufzeit, als das beschriebene Protokoll, weil es viele Zeit-intensive Entschlüsselungs-Operationen berechnen muss. Die Entschlüsselung könnte Eigenschaften der Eingabewerte offenbaren, was diese Implementierung unsicher macht. Die im Paper beschriebene Funktionsweise basiert jedoch auf dem Teilprotokoll SUR, und das wiederum auf der Veröffentlichung [3]. Daher wäre eine korrekte Implementierung zu aufwändig gewesen.
Wenn dann die relevanten Teile der anderen Paper implementiert sind, können diese dann die unsicheren Teil-Protokolle ersetzen und diese Implementierung damit sicher machen. Die Teil-Protokolle, bei denen eine sichere Implementierung möglich war, habe ich wie im Paper gezeigt sicher implementiert. Dazu zählen unter anderem die grundlegenden Protokolle, wie "secure Matrix Multiplikation", und die interessantesten Protokolle, wie "secure Cardinality Testing".
Da für die Protokolle, die auf anderen Papern basieren teilweise noch keine sicheren Implementierungen vorliegen, muss auch darauf vertraut werden, dass diese in den Papern vorgestellten Protokolle auch wie beschrieben funktionieren und sie effizient genug sind, um die Laufzeit des secDT Protokolls nicht so stark zu verändern, dass es ineffizient wird.

6.2 Bei großen Zahlen interferieren von Ring und Verschlüsselung

Bei der Implementierung der zu analysierenden Teilprotokolle habe ich bemerkt, dass sich die Verschlüsselten Zahlen in Zwei unterschiedlichen Modulo befinden. Einmal finden die Berechnungen über einem endlichen Körper statt. Gewählt wurde ein Körper dessen Anzahl an Elementen einer Primzahl entspricht. Um Berechnungen in diesem Körper anzustellen, werden Berechnungen modulo dieser Primzahl angestellt.
Gleichzeitig befinden sich die verschlüsselten Zahlen jedoch auch in einem anderen Körper: Dem der Verschlüsselung. Um beispielsweise Zwei verschlüsselte Zahlen zu addieren, werden auch Berechnungen modulo einer nicht-Primzahl angestellt.
Diese beiden unterschiedlichen Modulo können theoretisch zu Problemen führen, denn durch Berechnungen mit beiden Modulos kann es zu unerwarteten Ergebnissen kommen. Beispielsweise ist $(73 \% 7) \% 22 = 3$, aber $(73 \% 22) \% 7 = 0$. Und da es sich bei den Berechnungen teilweise um verschlüsselte Zahlen handelt, ist die richtige Reihenfolge der Anwendungen der Modulos nicht immer möglich. Die verschlüsselten Zahlen können nicht einfach Modulo der Größe des Körpers berechnet werden können, da sich die Zahlen durch die Verschlüsselung im Modulo der Körpers verändern.
Das Problem sollte jedoch nicht auftreten, wenn ein Modulo sehr viel größer ist als das

andere Modulo. In unserem Fall ist das Modulo der Verschlüsselung sehr viel größer, als das Modulo des Körpers. Die Größe der Eingaben ist also durch die Größe des Körpers nach oben beschränkt. Und da wir nur eine begrenzte Zahl an Berechnungen auf den verschlüsselten Abgaben ausführen, können die Werte nur begrenzt wachsen. Wenn nun das Modulo in der Verschlüsselung groß genug gewählt ist, kann sicher gegangen werden, dass das Modulo der Verschlüsselung innerhalb der Berechnungen der Protokolle nicht erreicht werden kann.
Als Beispiel dafür kann das folgende Beispiel betrachtet werden:
Der Körper hat die Größe 10, also sind die Eingaben immer kleiner als 10.
Gehen wir davon aus, dass 3 Multiplikationen ausgeführt werden.
Das Ergebnis der ersten Multiplikation muss kleiner als 100 sein.
Das Ergebnis der zweiten Multiplikation muss kleiner als 10.000 sein.
Das Ergebnis der dritten Multiplikation muss kleiner als 100.000.000 sein.
Wenn nun das Modulo der Verschlüsselung größer als 100.000.000 ist, ist es unmöglich, dass die Zahl durch die Verschlüsselung verändert wird, auch wenn bei den Zwischenergebnissen kein Modulo des Körpers angewandt wurde.
Wenn das Modulo aus der Verschlüsselung also groß genug gewählt wurde, ist es für die Berechnungen irrelevant und hat keine Auswirkungen auf die Korrektheit der Berechnungen, auch wenn durch die Verschlüsselung zwischenzeitliche Modulo-Berechnungen des Körpers unmöglich sind.
Dieses Problem, dass Verschlüsselung und der Körper interferieren können, ist auch bei meinen Tests aufgetreten. Der Test MPCTTestBig gab teilweise falsche Ergebnisse zurück. In diesem Fall war das Modulo der Verschlüsselung im Bereich von 32 Bits und das Modulo des Körpers ebenfalls im Bereich von 32 Bits. Nachdem das Modulo der Verschlüsselung auf rund 64 Bits erhöht wurde, trat dieser Fehler nicht wieder auf.

Chapter 7

Fazit

Das Ziel war es, zu zeigen, dass die Neuerungen aus dem Paper [1] funktionieren und effizient sind. Durch meine Implementierung wird deutlich, dass das vorgestellte Protokoll funktioniert.
Die Analyse der Teilprotokolle MPCT und secDT hat ergeben, dass sich die Laufzeit der Protokolle wie im Paper beschrieben verhält. Die Laufzeit hängt nun also nicht mehr von der Größe der Eingabemengen ab, sondern nur von der Anzahl der "erlaubten Abweichungen" oder dem "threshold", wie er im Paper genannt wird, und der Anzahl der teilnehmenden Parteien ab.
Damit kann je nach Aufgabe und Anwendungsbereich sehr viel effizienter als in früheren Protokollen festgestellt werden, ob die Schnittmenge größer ist, als ein gegebener wert. Das kann dann mit anderen Protokollen kombiniert werden, wie beispielsweise dem von Gosh und Simkin [6] entworfenen Protokoll, um noch komplexere Protokolle effizienter zu machen.
Leider konnte ich durch die Zeitbegrenzung dieser Arbeit keine komplette, sichere Implementierung des ganzen vorgestellten Protokolls liefern, da zu viele der Teil-Protokolle auf andere Implementierungen zurückgreifen, die es zu diesem Zeitpunkt noch nicht gibt. Das Protokoll ist so nicht direkt in Anwendungen nutzbar, die auf der Datensicherheit der Eingabemengen bestehen.
Dennoch habe ich gezeigt, dass das Protokoll funktioniert. Dadurch kann weitere Forschung auf dem Protokoll aufbauen und es erweitern oder verbessern.
Das Ziel dieser Arbeit ist also erreicht, auch wenn bis zu einer sicheren Implementierung des Protokolls noch einige Arbeit fehlt.

List of Figures

List of Tables

Bibliography

[1] S. P. Pedro Branco, Nico Döttling, "Multiparty cardinality testing for threshold private setintersection," 2021.

[2] D. Kogan, "A few lessons from the history of multiparty computation," https://theorydish.blog/2021/05/26/few-lessons-from-the-history-of-multiparty-computation, May 2021, stand: 17.7.2021.

[3] B. Schoenmakers and P. Tuyls, "Efficient binary conversion for paillier encrypted values," in *Advances in Cryptology - Eurocrypt 2006 (Proceedings 25th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Saint Petersburg, Russia, May 28-June 1, 2006)*, ser. Lecture Notes in Computer Science, S. Vaudenay, Ed. Germany: Springer, 2006, pp. 522–537.

[4] X. Yi, R. Paulet, and E. Bertino, "Homomorphic encryption. in: Homomorphic encryption and applications. springerbriefs in computer science," 2014.

[5] Y. Lindell, "Secure multiparty computation (mpc)," Cryptology ePrint Archive, Report 2020/300, 2020, https://ia.cr/2020/300.

[6] S. Ghosh and M. Simkin, "The communication complexity of threshold private set intersection," Cryptology ePrint Archive, Report 2019/175, 2019, https://eprint.iacr.org/2019/175.

[7] S. Badrinarayanan, P. Miao, S. Raghuraman, and P. Rindal, "Multi-party threshold private set intersection with sublinear communication," Cryptology ePrint Archive, Report 2020/600, 2020, https://eprint.iacr.org/2020/600.

[8] M. J. Ivan Damgard, "A generalization of paillier's public-key systemwith applications to electronic voting," 2004.

[9] A. Shamir, "How to share a secret," 1979.

[10] swansonk14, "Damgard-jurik," https://github.com/cryptovoting/damgard-jurik.

[11] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *Advances in Cryptology — EUROCRYPT '99*, J. Stern, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 223–238.

[12] "Jlinalg library," *http://jlinalg.sourceforge.net*.

[13] E. Kiltz, P. Mohassel, E. Weinreb, and M. K. Franklin, "Secure linear algebra using linearly recurrent sequences," in *Theory of Cryptography, 4th Theory of Cryptography Conference, TCC 2007, Amsterdam, The Netherlands, February 21-24, 2007, Proceedings*, ser. Lecture Notes in Computer Science, vol. 4392. Springer, 2007, pp. 291–310. [Online]. Available: https://iacr.org/archive/tcc2007/43920291/43920291.pdf