Institut für Informationssicherheit

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Private Function Evaluation mit Oblivious RAM

Jonas Geiselhart

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. Ralf Küsters
Betreuer/in:	Sebastian Hasler, M.Sc.
Beginn am:	11. April 2022
Beendet am:	11. Oktober 2022

Kurzfassung

Diese Arbeit untersucht inwiefern Oblivious Random Access Machines zur Private Function Evaluation genutzt werden können.
Dazu werden zuerst Grundlagen zur Secure Computation vorgestellt und insbesondere der Stand der Forschung im Bereich Oblivious RAM zum Einsatz als Secure-Computation-Datenzugriffsstruktur betrachtet. Danach werden aktuelle bestehende Konstruktionen zur privaten Evaluation von Funktionen vorgestellt und untersucht. Hierbei werden besonders Effizienz, Sicherheit, Speicherbedarf, Kommunikationsaufwand, und Ausdrucksstärke der Gatter betrachtet.
Es wird eine eigene PFE-Konstruktion vorgestellt, die auf Integer Werten rechnet und Oblivious RAM zur Adressierung der Variablen nutzt. Dies ermöglicht eine Implementation von Funktionsauswertungen als Programm basierend auf CPU-Step-Circuits, die ähnlich wie Garbeled RAM-Konstruktionen [16, 34, 35], Kontrollfluss und somit auch die Berechnung komplexerer arithmetischer Funktionen zulässt.
Schlussendlich wird eine Implementation dieses PFE-Schemas vorgestellt und mit den zuvor betrachteten Protokollen verglichen. Das Schema berechnet arithmetische Funktionen in linearer Zeit respektive der Schrittgröße und polylogaritmischer Zeit respektive der Variablenanzahl. Allerdings entsteht ein hoher konstanter Aufwand pro Auswertungsschritt, der die Praktikabilität deutlich einschränkt.

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

CTH Circuit Topology Hiding.

FHE Fully Homomorphic Encryption.

GRAM Garbeled RAM Program.

HE Homomorphe Verschlüsselung.

MAC Message Authentification Code.

MPC Secure Multi-Party-Computation.

OEP Oblivious Extended Permutation.

ORAM Oblivious Random Access Machine (-Model).

PFE Private Function Evaluation.

PGE Private Gate Evaluation.

RAM Random Access Machine.

SFE Secure Function Evaluation.

SH Structure Hiding.

SHE Somewhat Homomorphic Encryption.

SS Secret Sharing Schemes.

UC Universelle Schaltungen.

ZKP Zero-Knowlegde Proof.

1 Einleitung

In der immer vernetzteren Welt haben viele Technologien in den letzten Jahrzehnten Einzug gehalten, die verlangen, dass sensitive Daten online gegenüber dritten Parteien preisgegeben werden. In Cloud Computing, Machine Learing, oder genereller Datenanalyse erfordert eine Auswertung der Daten aktuell fast ausschließlich, dass Daten in unverschlüsselter Form vorliegen. Das kann problematisch sein, falls man dem Dienstleister gegenüber, dem man die unverschlüsselten Daten vorlegen muss, nicht das entsprechende Vertrauen entgegenbringen will. Auch kann eine verschlüsselte Auswertung sensible Daten gegenüber Schwachstellen in der IT-Sicherheitsinfrastruktur anderer Parteien schützen.

Secure Multi-Party-Computation (MPC) zielt darauf ab dieses Problem zu lösen, indem es ermöglicht wird sicher und effizient auf verschlüsselten Daten zu rechnen. Während dieser Forschungszweig schon über 30 Jahre alt ist, begründet von *Yao* mit seinen verschlüsselten Schaltungen aus [43], erhielt die Arbeit durch Entwicklung effizienter *homomorpher Verschlüsselungs*-Algorithmen (HE) in den letzten Jahre große Aufmerksamkeit. Man erhofft sich damit Privacy-Probleme besonders in den Bereichen Privacy-Preserving Machine Learning, E-Voting, E-Health, und FinTech zu lösen.

Ein häufiges Problem ist dabei, dass die zu berechnenden Funktionen im allgemeinen MPC-Setting allen Teilnehmern bekannt sein müssen. Die effiziente Auswertung von im Allgemeinen nicht bekannten Funktionen wird durch den Oberbegriff Private Function Evaluation (PFE) beschrieben. Hierbei hält eine Partei P_0 eine Funktion f, die nur ihr bekannt ist, und Parteien $P_1 \ldots P_n$ die Argumente x der Funktion. Ziel ist es gemeinsam die Funktion auszuwerten und entweder gegenüber P_0 oder $P_1 \ldots P_n$ das ermittelte Ergebnis $f(x)$ offenzulegen.

Wenn beispielsweise ein Nutzer ein Bild über ein soziales Netzwerk versenden will, aber sichergestellt werden soll, dass dieses Bild nicht gegen die Richtlinien der Plattform verstößt, kann mittels PFE über einen Privacy-Preserving Machine Learning Algorithmus die Wahrscheinlichkeit dafür berechnet werden und ggfs. weitere Prüfungen folgen. Damit würde der Anbieter der Plattform seiner Schutzpflicht gegenüber der Nutzer nachkommen, aber die versendeten Daten könnten vertraulich bleiben.

Eine weitere Anwendung ergibt sich, falls sich jemand auf einen Kredit bewirbt, ist es im Interesse der Allgemeinheit vorher die Kreditwürdigkeit einzuschätzen. Bisher werden dafür immer sämtliche Finanzdaten von Bankinstituten eingesammelt und daraus eine Kreditbewertung berechnet. Die Bewertungsfunktion ist hierbei natürlich ein Geschäftsgeheimnis des Kreditbewertungsinstitutes. Dieses Verfahren ist umstritten, da sensible Daten gegenüber dritten Parteien offengelegt werden. Stattdessen könnten Banken und Kreditbewertungsinstitute gemeinsam mittels PFE den Kreditwert berechnen und nur den Banken offenlegen, bei denen die Person Kunde ist.

Eine Realisierung von PFE teilt sich vorangehenden Arbeiten typischerweise in zwei Funktionalitäten. Einmal der Verdeckung der Beziehung des In- bzw. Outputs einzelner Operatoren und zum Anderen die Evaluation einer (beliebigen) Operation aus verschiedenen verfügbaren Operatoren.

Die Verdeckung von sogenannten Variablenzugriffsmustern werden in Schaltungsparadigmen in der Regel von Permutationsnetzen realisiert. Oblivious Random Access Machine (-Model) (ORAM) wurde ursprünglich zur Obfiszierung von Softwarezugriffen im Kontext von Software-Protection von *Goldreich* [17] entwickelt. Hierbei steht RAM für Random Access Machine und beschreibt eine Möglichkeit in einem Client Server Setting einen Speicher auf einer wesentlich größeren Maschine zu haben, sodass der Speicher „blind“gegenüber möglichen Zugriffsmustern ist, man spricht in diesem Fall von *oblivious Memory*. Diese Technik kann nun modifiziert und verwendet werden um Variablenbeziehungen in PFE zu verstecken.

Im Folgenden werden zuerst in Kapitel 2 alle nötigen Grundlagen und mathematischen Primitive erklärt, die später als Referenzpunkte für eine eigene PFE Konstruktion gesehen werden. Darauf aufbauend werden in Kapitel 3 bestehende Protokolle verwandter und vorangegangener wissenschaftlicher Arbeiten betrachtet und eine eigene Konstruktion zur sicheren Evaluation von privaten Funktionen entwickelt. Diese Konstruktion wird so ausformuliert sein, dass sie in Kapitel 4 in ein MPC-Framework integriert werden kann.

2 Grundlagen

PFE baut auf einer Reihe grundlegender Primitive auf, die verschlüsselte Auswertungen möglich machen sollen. In diesem Kapitel gebe ich einen kurzen Überblick über diese kryptographischen Protokolle und Algorithmen.

2.1 Multi-Party-Computation

MPC ist die Technik zur sicheren Berechnung von Funktionen auf verschlüsselten Daten. In dem grundlegenden Setting gibt es N Parteien, die jeweils einen Secret Share $(sk_i)_{1 \leq i < N}$ besitzen. Die Verschlüsselung von Klartexten ist i.A. mit öffentlich bekannten Informationen möglich, zur Entschlüsselung hingegen müssen mehrere Parteien die resultierende Chiffre mit ihrer Secret Share verrechnen und die Ergebnisse sogenannte Decryption Shares müssen kombiniert werden.

2.1.1 Homomorphe Verschlüsselung

Homomorphe Verschlüsselung (HE) ist eine Klasse von Verschlüsselungsschemata die homomorphe Abbildungen darstellen.

Definition 2.1.1 (Homomorphe Verschlüsselung)
Eine Verschlüsselungsschema $\mathcal{E} = (Gen, Enc, Dec)$ ist homomorph respektive einer Operation $\circ$, falls für die Verschlüsselung $Enc : A \to B$ gilt:

$$\exists * : B \to B : Enc(m \circ m') = Enc(m) * Enc(m')$$

Kogos et al. [30] haben einen ausführlichen Überblick über aktuellen Entwicklungen in dem Feld gegeben. Sie haben das Feld in die folgenden drei Kategorien unterteilt und diese betrachtet:

Partially Homomorphic Encryption Bezeichnet ein Verschlüsselungsschema, wie Pallier [38] oder ElGamal [13], die nur bezüglich einer Basisoperation (Addition oder Multiplikation) homomorph sind.
Aufgrund der Einfachheit und niedrigen Komplexität grundlegender Operationen sind sie derzeit in MPC die weitverbreitetsten Algorithmen. Es kann trotzdem durch weitere arithmetische Bausteine eine Erweiterung der Funktionalität erfolgen, weitere Details werden in Kapitel 4 näher erläutert.

Somewhat Homomorphic Encryption Bezeichnet ein Verschlüsselungsschema, in dem die Operationen je nach Anwendungsfall bedingt korrekt berechnet werden. Es würde zum Beispiel ein additiv homomorphes Schema mit genau einer möglichen Multiplikation unter diese Kategorie fallen. Dieses Schema ist besonders für begrenzte Aufgaben, wie z.B. Vorberechnungen in Kapitel 4, interessant. Allerdings im Allgemeinen eher weniger praktisch, da nur bestimmte Funktionen gut

berechnet werden können.

Fully Homomorphic Encryption Ein FHE System implementiert sowohl Addition und Multiplikation von Chiffretexten, ohne eine Entschlüsselung zwischen diesen Operationen vorzunehmen. Es ist einfach zu erkennen, dass Addition und Multiplikation im boolschen Falle, also über $\mathbb{Z}_2$, eine XOR und eine AND Operationen darstellen und FHE somit mit der Ausdrucksstärke von boolschen Schaltungen gleichzusetzen ist. Voll homomorphe Verschlüsselungsschemata basieren entweder auf Bootstrapping Techniken, die dazu führen, dass SHE korrekt auf allen Funktionen ausgeführt werden oder auf komplett neuen Konzepten. Zum aktuellen Zeitpunkt ist die benötigte Bitgröße, sowie die Ver- und Entschlüsselung, zu aufwändig um sie effektiv zu Nutzen.

2.1.2 Secret Sharing

Secret Sharing Schemes (SS) sind eine Grundlage MPC zu realisieren. Sie dienen dazu einen geheimen Wert, das *Secret*, zu verschlüsseln, sodass mehrere Parteien damit rechnen können, diese aber ihre *Secret Shares* kombinieren müssen um den eigentlichen Wert offenzulegen.

Shamir Secret Sharing Shamir Secret Sharing ist ein threshold-basiertes Schema um Werte basierend auf einem Threshold t zu rekonstruieren. Es wurde in *Shamir* [39] vorgestellt. Falls es zur Entschlüsselung angewendet wird, müssen t Decryption Shares eines Wertes von n Parteien kombiniert werden um den entschlüsselten Wert offenzulegen. Dabei basiert das Verfahren auf einem Interpolationsproblem. Es existiert also ein Polynom p in einem endlichen Körper $\mathbb{F}$, sodass der gewünschte Wert $q = p(0)$ ist. Jede Partei P_i kennt den Wert des Polynoms an der Stelle i. Genau dann, wenn nun dass Polynom so gewählt wurde, dass es von Grad $t-1$ ist lässt sich der Wert mittels kombinierter Interpolation von t Decryption Shares ermitteln, ansonsten nicht.
Eine Betrachtung von *Harn et Lin* [21] hat gezeigt, dass bei einem asynchronen Öffnen der Shares Fehler erkannt werden können, falls ein Minimum von t Parteien ehrlich agiert. Weiterhin benötigt es in diesem Setting mehr als $c+t-1$ ehrliche Parteien, wobei c hier die Anzahl der kompromittierten Shares bezeichnet, um die Partei zu ermitteln die den Fehler eingebaut hat. Falls eine solche Partei gefunden wird, kann eine neue Berechnung ohne die Secret Shares dieser Partei durchgeführt werden. Shamir Secret Sharing liefert uns also *Accountablility* für $c+t-1$ ehrliche Parteien und *Error Detection* / *Guranteed Output Delivery* für t ehrliche Parteien. Diese Garantien können noch verbessert werden indem man MACs nutzt. Diese können verifizieren, dass ein Secret Share aus eine vertrauten Quelle kommt und somit nicht modifiziert wurde.
Shamir Secret Sharing ist homomorph gegenüber Addition und ein Multiplikation mit allgemein bekannten Konstanten ist möglich. Es gibt einige erweiterte Verfahren, die auf Shamir Secret Sharing aufbauen und Sicherheitsgarantien, sowie praktische Eigenschaften verbessern.

Additives Secret Sharing Beim additiven Secret Sharing wird ein Geheimnis $q \in \mathbb{Z}_N$ in verschiedene Shares $(q_i)_{1 \leq i < n}$ aufgeteilt und an n Parteien weitergegeben. Diese Shares sind so konstruiert, dass $\sum_{i=0}^{n} q_i = q \ mod \ N$ gilt. Es ist einfach zu sehen, dass wieder Addition und Multiplikation mit allgemein bekannten Konstanten möglich ist. Es kann ein One-time MAC genutzt werden um diese Shares zu authentifizieren. Damit ist also *Error Detection* gesichert. Hierbei sind alle Shares notwendig um ein Secret zu rekonstruieren. Ist dies nicht der Fall, ist es nicht möglich ein valides Ergebnis zu erhalten.

2.2 Circuit Evaluation

Eine gegebene Funktion f kann in einen entsprechenden Schaltkreis C_f umgewandelt werden, welcher ausgewertet wird. Im Allgemeinen ist das aber nur für eine Untermenge der berechenbaren Funktionen möglich bei welchen auch nur eine begrenzte Eingabe möglich ist. Das Circuit Evaluation Paradigma beschreibt die Vorgehensweise Funktionen umzuwandeln und als (überwiegend logische) Schaltungen zu evaluieren. Das ist hier besonders interessant, da die meisten Funktionen in PFE bis jetzt als Schaltungen beschrieben und untersucht werden. Die ersten Methoden zur Evaluation von geheimen Schaltungen wurden von *Yao* [43] beschrieben.
Eine Schaltung $C_{\mathcal{F}}$ realisiert eine Funktion $\mathcal{F}$. Die Topologie einer Schaltung bezeichnet die Verschaltung einzelner Gatter ohne Betrachtung von deren semantischer Funktionsweise. Es wird immer eine Anzahl von Verschaltungen (Wires) und eine Anzahl von Gattern (Gates) betrachtet. In den beschriebenen Circuits in dieser Arbeit sind Wires aufgeteilt in Input-Wires, also Verschaltungen die ihren Wert aus Eingabedaten erhalten bzw. aus einem Gatter als Eingabekanal resultieren. Alle Wires die nicht zu den Input-Wires zählen sind Output Wires, diese sind nur Ergebniskanäle von Gattern. Nun kann man diese Gatter, falls keine zyklischen Verschaltungen definiert sindd, topologisch sortieren und die Verschaltungen in dieser Ordnung auswerten.

2.2.1 Wertebereiche

Ein Schaltkreis kann in verschiedenen Wertebereichen realisiert werden. Es ist allgemein bekannt, dass ein Wertebereich mit nur zwei Werten zur Auswertung der Gatter notwendig ist um die mögliche Ausdrucksstärke der Schaltungen zu maximieren. Daraus lässt sich allerdings noch nicht schließen, dass mögliche Implementierungen mit zusätzlichen Einschränkungen und unterschiedlichen Gatterarten nicht prinzipiell unterschiedliche Ausdrucksstärken haben können, sondern dass eine Einschränkung der Symbole keine Einschränkung der Berechenbarkeit zur Folge hat.

Boolsche Schaltkreise Die häufigste Realisierung von Schaltungen sind boolschen Schaltkreise. In dieser, auch als digitale oder binäre Schaltung bezeichneten, Schaltkreisfamilie kann jeder Kanal potenziell die Werte 0 oder 1 annehmen und so über Gatter verschiedene Werte propagiert werden. Für einen solchen Schaltkreis können unterschiedliche Gatterarten definiert werden. Es genügt hierbei entweder AND und OR, oder $NAND$ Gatter zu definieren um jede boolsche Formel zu berechnen. Bei der Berechnung von normalen Zahlen aus einer Menge $\mathbb{Z}_n$ leiden boolsche Schaltkreise unter einem hohen Blow-Up Faktor, der schon bei relativ simplen Rechnungen viele Gatter erfordert.

Arithmetische Schaltkreise Da binäre Schaltungen schlecht skalieren, falls sie zur Berechnung von Integer-Arithmetik genutzt werden, benötigt es als Alternative arithmetische Schaltungen. Die Gatter nehmen hier als Eingabe Integer Werte an und führen arithmetische Funktionen (i.d.R. nur Multiplikation und Addition) durch, die Ausgabe ist dementsprechend wieder ein ganzzahliger Wert. Es ist hierbei offensichtlich, dass in der Theorie Multiplikation und Addition ausreichen um boolsche AND und OR Gatter zu simulieren, daher genügen diese beiden Funktionalitäten um eine äquivalente Ausdrucksstärke zu boolschen Schaltungen zuzusichern.
Vorgriff: Für PFE bedeutet dies zum einen, dass die PGE-Methodik abgeändert werden muss und

nun eine homomorphe Funktionalität integriert werden muss. In Kapitel 3 wird genauer erklärt wie solche Schaltungen realisiert werden.

2.2.2 Evaluation von Funktionen

Im Folgenden werden zwei unterschiedliche Primitive erklärt um eine triviale PFE-Evaluation zu ermöglichen. Hierbei lässt sich jede PFE-Brechnung zu einer SFE-Berechnung reduzieren, indem man einen Universal Circuit berechnet und eine Partei den zu der berechnenden Funktion korrespondierenden Schaltkreis C_f verschlüsselt beisteuert. Hierfür nutzt die in der Literatur gängige triviale Basiskonstruktion *Yao's Garbeled Circuits* um eine *Universal Circuit Konstruktion* auszuwerten. Daher wird in diesem Unterkapitel ein kurzer Überblick über beide Techniken gegeben.

Universal Circuits Universelle Schaltungen (UC) bezeichnen Schaltungen, die neben Inputwerten auch Beschreibungen von Schaltungen als Eingabe nehmen und dann die von von der Beschreibung realisierte Schaltung auf dem entsprechenden Input realisieren. Hierbei ist die Anzahl der Gatter der Eingabeschaltung limitierend. Programme zur PFE in Circuit Spezifikation werden über Universal Circuits realisiert. Die umfangreichste Analyse von UC im PFE Kontext stammt von *Lipmaa et al.* [31]. Sie betrachten zuerst die binäre Valiant UC Konstruktion, die auf universellen Gattern und Permutation Networks aufbauend mit eigenen Verbesserungen eine Komplexität von $O(nlog(n))$ im Vergleich zu den Gattern der realisierten Funktion erreicht. Dafür müssen sie allerdings einen hohen konstanten Mehraufwand miteinbeziehen, der kleinere Schaltungen wenig realisierbar macht. *Kiss et Schneider* [29] haben gezeigt, dass $\Omega(nlog(n))$ eine untere Grenze für Optimierungen durch UC darstellt.

Yaos Garbeled Circuits *Yao* [43] veröffentlichte erstmals eine Möglichkeit die Zwischenergebnisse bei der Auswertung auf verschlüsselten Daten zu verstecken. Er veröffentlichte den Algorithmus damals nicht konkret selbst, sondern skizzierte nur das Konzept einer solchen Konstruktion. Es ist hier nur als unvollständiger Vorgriff auf PFE, wie in Kapitel 3, erklärt anzusehen. Dennoch stellen Yaos Garbeled Circuits ein gültiges SFE-Schema dar. Die Funktion wurde durch eine kryptographische Verschleierung der Ergebnisse eines Gatters versteckt. Diese Lösung war zu dem damaligen Zeitpunkt nur theoretisch interessant und versteckte nicht inhärent die Topologie der Schaltung (dennoch ist es möglich über Kodierung der Schaltung und Auswertung von universellen Schaltungen).
Es wird das (binäre) Input eines Gatters durch zufällig gewählte Schlüssel ersetzt und auch der Output Wert durch einen zufällig gewählten verschlüsselten Schlüssel ersetzt, der nur durch Entschlüsselung mithilfe der Input-Schlüssel offengelegt werden kann. Snsonsten schlägt die Entschlüsselung fehl. Nun kann eine zweite Partei diese Schaltung auswerten, sofern sie für ihren Input über das Input zu Input-Schlüssel Mapping verfügt. Dieses kann die erste Partei der zweiten Partei mittels *Oblivious Transfer* für genau ein mögliches Input mitteilen, ohne die anderen möglichen Mappings zu verraten.

2.3 Oblivious RAM

Oblivious RAM bezeichnet eine kyptographische Datenstruktur, die dazu genutzt werden kann, einzelne Zugriffe und Zugriffsmuster zu verstecken. ORAM steht hierbei für Oblivious Random Access Machine, es wird also das Problem des Versteckens der Zugriffspfadstruktur reduziert auf die Simulation einer Random Access Machine in einem "oblivous" Setting. Diese Konstruktion wurde zuerst von *Goldreich et Ostrovsky* [18] vorgestellt. Ursprünglich wurde dieses Primitiv zum Schutz gegen ungewollte Vervielfältigung von Software entworfen. Damals war der Bereich der PFE eher theoretisch fundiert und eine praktische Anwendung noch nicht absehbar. Heute jedoch findet PFE realistisch Anwendung und ORAM ist deutlich effizienter geworden (wie im folgenden Kapitel ausgeführt wird). Bei einer Betrachtung beider Problemstellungen, stellt sich natürlicherweise die Frage inwiefern man ORAM nutzen kann, um Teilprobleme in PFE zu lösen. Das ist die Motivation für meine Betrachtung von Oblivious Random Access Machine (-Model).

Ein ORAM wird in der Regel durch eine Client Server Struktur modelliert. Es wird von einem leistungsstarken Server mit viel Speicher und einem weniger leistungsstarken Client ausgegangen. ORAM wird generell als ein statisches Array modelliert, dass indexbasierte wahlfreie Blockadressierung zulässt. Hierbei wird der verschlüsselte Index für jeden Zugriff auf einen realen Index gemappt. Die Daten werden von einem Server gespeichert, der sie verwaltend organisiert. Es werden normalerweise Funktionen zum verschlüsselten indexbasiertem Abspeichern und Abrufen von Daten angeboten.
Eine **Random Access Machine** ist definiert durch *Goldreich et Ostrovsky* [18] als ein Tupel *(CPU,MEM)*, wobei *CPU* und *MEM* als interaktive Turingmaschinen modelliert werden, deren *Write-Only* Band dem *Read-Only* Band der jeweils anderen Maschine entspricht und vice versa. Die Eingabe ist ein Tupel (s, y), wobei s die Eingabe für *CPU* und y die Eingabe für *MEM* darstellt. Wir bezeichnen den Output von *CPU* als $CPU(s, y)$ und den Output von *MEM* als $MEM(y)$.
In dieser Arbeit ist *CPU* allgemeiner als Client bezeichnet und *MEM* allgemeiner als Server, da diese Bezeichnungen für die später betrachteten Fälle zutreffender sind.
Es ist eine solche RAM **oblivious**, falls das Zugriffsmuster in dem Fall von zwei unterschiedlichen Eingaben und einer gleichen Anzahl an Ausführungsschritten nicht voneinander unterscheidbar ist. Wir werden später sehen, dass diese Definition einer RAM noch erweitert werden muss. Es wird eine $probabilistische$ RAM definiert, die über ein Orakelband verfügt, das in konstanter Zeit zufällige Werte ausgeben kann. Das kann in späteren Realisierungen über einen Pseudo-Random-Number-Generator geschehen. Für den Fall einer probabilistischen RAM wird bei der Definition von *oblivious* nicht das Zugriffsmuster sondern die probabilistische Verteilung der möglichen Zugriffsmuster betrachtet.
Wir können die zwei wichtigsten Eigenschaften, die für einen ORAM gelten müssen wie folgt formalisieren:

1. *Oblivious* Ein ORAM ist Oblivious, falls bei Eingabe einer Größe n, eines Programmes Π_0 mit Zugriffsverhaltens D und einer Eingabe x_0, jede Ausführung *computationally indistinguishable* zu allen anderen Programmen Π_1 mit möglichen Eingaben x_1 und äquivalentem Zugriffsverhaltens D ist.

2. *Correctness* Ein ORAM muss auf jeden Lesezugriff mit Index i mit dem zuletzt in Index i gespeicherten Wert antworten. Es muss also $\text{READ}(i) = y$ genau dann gelten wenn der letzte Aufruf zum Speichern eines Elementes mit Index i $\text{WRITE}(i, y)$ ist.

zu (1): Diese Definition ist angelehnt an die Definition aus *Keller et Yanai* [28]. Sie hat allerdings den besonderen Unterschied, dass ORAM hier nicht als Zugriffsmechanismus, sondern Aufruf eines Subprotokolls notiert wird. Das erleichtert im späteren Verlauf die Betrachtung der Programme und ermöglicht eine Unterscheidung zwischen normal gespeicherten Variablen und im ORAM gespeicherten Variablen.
zu (2): Hier unterscheidet sich wieder die Definition in dieser Arbeit zu der aus [28], indem nur Subprotokollaufrufe, aber nicht das gesamte Protokoll betrachtet werden. Insbesondere ist das Verhalten beim Zugriff auf einen nicht vorher beschriebenen Index i nicht definiert.

Es ist zuerst zu Bemerken, dass jede Konstruktion eines ORAMs basierend auf zwei Parteien auch für mehrere Parteien angewendet werden kann, da mehre Parteien mittels MPC ein Programm simulieren können, dass so agiert als würde eine Partei handeln. Im Grundprinzip ist das aber nur für die Client-Seite sinnvoll, da die allgemeinen zusätzlichen Kommunikationskosten für eine serverseitige Aufteilung zu hoch sind. Es wird daher immer angenommen, dass genau eine Partei die serverseitige Rolle übernimmt. Alternativ schlagen *Lu et Ostrovsky* [33] eine ORAM Struktur mit zwei Server-Rollen ohne kommunikativen Overhead vor, solche Konstruktionen sind möglich, werden aber im Folgenden nicht betrachtet.

Triviale Konstruktion Ein erster trivialer ORAM ergibt sich, indem beim Lesen eines Blockes der Server einem Client den gesamten Speicherinhalt schickt und beim Schreiben eines Blockes der Client alle Blöcke anfrägt, jeden verschlüsselten Block rerandomisiert, den entsprechend gewünschten Block überschreibt, und alle Speicherblöcke wieder zurücksendet.
Dieses Verfahren ist natürlich nicht effizient und schon für relativ kleine Speicher nicht praktikabel, da der Kommunikationsoverhead sehr hoch ist. Es ist jedoch simpel und kann somit als gutes erstes Gedankenexperiment verstanden werden um die Problemstellung in der ORAM-Entwicklung nachzuvollziehen. Es muss dem Server möglich sein Blöcke zu identifizieren, ohne deren (verschlüsselten) Index zu kennen. Falls diese Blöcke abgefragt werden, soll dieser Zugriff über eine Struktur uniform sein, also nicht vom Zustand oder dem Speicher der Struktur abhängen. Außerdem müssen die Datenstrukturen zur Speicherung neuer Daten über einen Rerandomisierungsalgorithmus verfügen und eine Speicherung in uniformer oder gleichverteilter Weise folgen.

Die Daten können über einen Identifier gespeichert werden, der aus einem Hash-Wert der aktuellen Position und einem Geheimnis des Clients entsteht. Dazu wird in den folgenden Betrachtungen entweder die Existenz allgemeiner One-Way-Funktionen angenommen oder aber auch speziell entwickelte Verfahren benutzt.
Es ist auch einfach zu sehen, dass bei der Speicherung eine Zufallskomponente (i.d.R. eine zufällige Permutation) benutzt werden muss, da sonst keine sublineare Zugriffe möglich wären. Es würde ohne Randomisiserung klar werden, dass die nicht mitgesendeten Daten nicht abgefragt werden, damit können Aussagen über das Zugriffmuster folgen, die nicht bekannt werden dürfen.

2.3.1 Squared Root ORAM

Die erste nichttriviale Konstruktion eines Oblivious Random Access Machine (-Model) lieferten *Goldreich et Ostrovsky* in [18]. Hierbei ist der amortisierte Aufwand für einen Zugriff in $\theta(\sqrt{n} \cdot log^2(n))$.
Zur Initialisierung eines Speichers der n Blöcke beinhaltet wird vom Server Platz für $n + 2\sqrt{n}$ Blöcke allokiert. Der Speicher wird zuerst mit möglichem Inhalt gefüllt und dann die ersten

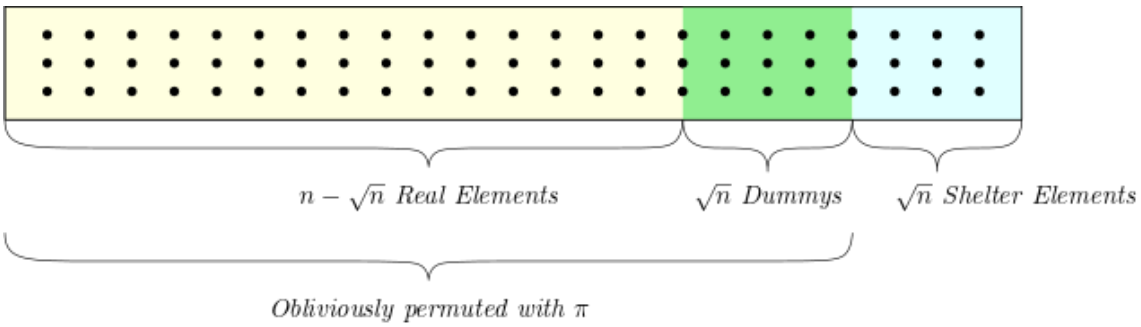


Abbildung 2.1: Speicheraufteilung eines Squared Root ORAMs

$n + \sqrt{n}$ Positionen, im Folgenden der Hauptspeicher genannt, mit Dummy Werten aufgefüllt. Diese aufgefüllten Daten werden dann eingehend mit einer Permuation π permutiert, diese Permutation wird später zyklisch wiederholt. Die letzten $\sqrt{n}$ Speicherpositonen werden als Shelter bezeichnet, er enthält die in dieser Epoche schon gelesenen bzw. geschriebenen Blöcke. Diese Aufteilung des Speichers ist in Abbildung 2.1 beschrieben.
Nun kann eine kombinierte *Read-Write-Operation* durchgeführt werden. Hierfür wird zuerst der gesamte Shelter gescannt und nach dem entsprechenden Block gesucht, befindet sich dieser Block im Shelter, wird er notiert und es wird fortgefahren, ansonsten wird ein Dummy Wert notiert und es wird fortgefahren. Als nächstes folgt ein Zugriff auf den Hauptspeicher, an der Position $\pi(i)$ für einen gesuchten Wert i, falls dieser nicht gefunden wurde. Ansonsten erfolgt ein Zugriff auf die Position $\pi(n + j)$, wobei j der Anzahl der gemachten Zugriffe seit dem Beginn der letzten Epoche entspricht. Damit wird also einer der zufälligen Dummy Werte notiert. Da der Server π nicht kennt, kann er nicht zwischen der Art der Zugriffe in diesem Fall unterscheiden, er weiß also nicht ob das Element im Shelter gefunden wurde. Es wird das abgefragte Element, egal ob es verändert wurde oder nicht, in den Shelter zurückgeschrieben.
Nach $\sqrt{n}$ Operationen ist der Shelter voll und die Dummy Werte Zugriffe verbraucht, also wird ein *Reshuffle* notwendig. Verschiedene Reshuffle Methoden werden später erklärt. Zur Erklärung dieser Methode reicht es diese Methode als eine Blackbox zu betrachten, die in $\theta(n \cdot log(n))$ Zugriffen eine neue Permutation realisiert. Dazu werden die Daten aus dem Shelter wieder in den Haupspeicher geschrieben. Nach der Permutation liegt der Speicher wieder in derselben Form vor wie nach der ersten Initialisierung. Die Epoche ist beendet.

Wir können sehen, dass dieses Vorgehen sicher ist, da alle Zugriffe entweder deterministisch oder uniform gleichverteilt geschehen. Der Scan des Shelters umfasst immer alle in diesem Shelter liegenden Daten. Der nachfolgende Zugriff auf den Speicher erfolgt immer auf ein Element so, dass falls π eine zufällige Einwegfunktion ist, nicht unterschieden werden kann ob das Element mit Index $n + j$ oder i gesucht wird. Wir können auch sehen, dass kein Element aus dem Hauptspeicher zweimal innerhalb einer Epoche abgefragt wird, da falls es sich um einen zweiten Zugriff auf Element i handelt, dieses Element nicht mehr im Haupspeicher gesucht wird und der Index von Dummy Elementen immer von der Schrittzahl der Epoche abhängt. Das Muster ist also oblivious.
Nun ergibt sich noch die Frage wie die Permutation realisiert werden sollte. Zuerst werden die Elemente mit Tags versehen, dieses Mapping bildet den Index auf einen (pseudo-)zufälligen Wert ab. Formal wird das Mapping $\tau : 1, 2, ..., n \rightarrow T_n$ beschrieben, wobei die Funktion groß genug sein muss, sodass die Kollisionswahrscheinlichkeit der Tags marginal ist. Diese Funktion sollte in konstanter Zeit berechenbar sein, was in der Praxis nicht schwer umzusetzen ist. Dann werden die

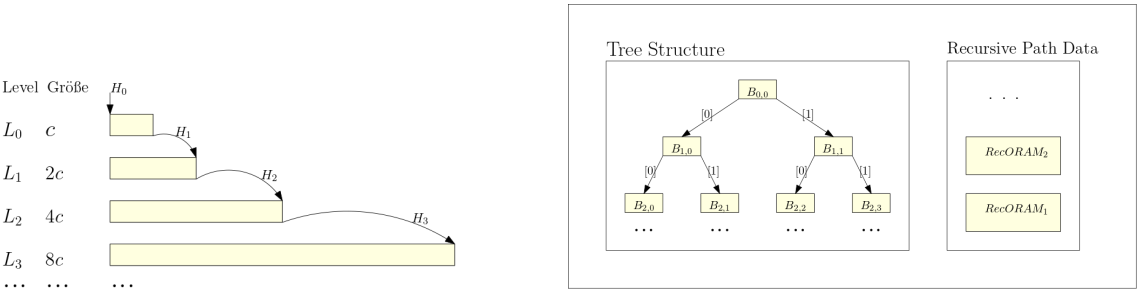
Tags durch *Oblivious Sorting* sortiert. *Goldreich et Ostrovsky* haben sich hierbei für Batchers Sorting Network entschieden. Diese Wahl ist im Allgemeinen austauschbar und andere Sorting Networks werden später noch diskutiert, daher ist hier nur noch zu erwähnen, dass dieser Algorithmus die Elemente nach Tags in $\theta(n \cdot log^p(n))$ sortiert. Nun wird auch offensichtlich, dass der Server verschlüsselte Elemente findet, indem er vom Client einen Tag erhält und in $\theta(log(n))$ mittels binärer Suche diesen Tag findet und das Element zurück gibt. Das automatische Zurückschreiben des Wertes in den Shelter ist trivial. Nun kann der amortisierte Aufwand eines Zugriffs berechnet werden. Hierbei wird der Kommunikationsaufwand zwischen Client und Server betrachtet, da dieser maßgebend für die Performance am Ende ist. Je Zugriff muss der Shelter gescannt werden ($\theta(\sqrt{n})$), die entsprechenden Tags berechnet und genau ein weiteres Element gesucht werden ($\theta(1)$). Alle $\sqrt{n}$ Zugriffe muss der Shelter in den Hauptspeicher zurückgeschrieben und dieser neu permutiert werden, diese Operationen können kombiniert werden ($\theta(n \cdot \log^2(n))$). Demnach sind die **amortisierten Zugriffskosten** $\theta(\sqrt{n} \cdot \log^2(n))$. Da das Sorting Network trivialerweise optimiert werden kann, ist es üblich diesen Algorithmus trozdem mit $\theta(\sqrt{n} \cdot \log(n))$ zu notieren.

Auch wenn dieser Algorithmus eine schlechtere Laufzeitkomplexität hat, als viele State-of-the-Art Algorithmen, ist er immer noch relevant, da der Overhead der Berechnungen im Allgemeinen relativ gering ist. Dies kann später bei einem gemeinsamen verschlüsselten Berechnen von ORAM-Zugriffen interessant werden.

2.3.2 Polylogaritmischer ORAM

Im Folgenden werden zwei maßgebende ORAM Konstruktionen aufgezeigt. Wie schon zuvor, handelt es sich bei diesen Konstruktionen nicht um State of the Art Konstruktionen, sondern eher um Grundlagen, die in einem Großteil der folgenden Forschung betrachtet und optimiert werden.

Hierachical Solution Die hierachische Lösung wurde auch von *Goldbach et. Ostrovsky* in [18] vorgestellt. Das Ziel ist, statt nur einem vergleichsweise großen Shelter, der periodisch erneuert wird, mehrere hierachisch strukturierte Speicher zu haben, deren Epochendauer invers zum *Reshuffling*-Aufwand ist, d.h. kleinere Speicher werden öfters permutiert, wohingegen große Speicher seltener permutiert werden. Interessant ist auch, dass bei dieser Methode kein Vorwissen über die Menge der gespeicherten Daten notwendig ist. Der Speicher erweitert sich dynamisch ohne zusätzlichen Overhead.
Der Speicher ist in *Level* unterteilt. Diese Aufteilung ist in Abbildung 2.2a dargestellt. Level L_0 ist von konstanter Größe, (simplifiziert) notiert als $|L_0| = 1$, alle weiteren Level sind durch die Größe $|L_i| = 2 \cdot |L_{i-1}|$ definiert. Dabei ist zu erwähnen, dass die Größe eines Speichers nicht die darin gespeicherten Elemente, sondern die Anzahl der Buckets notiert. Ein (Hash-) *Bucket* beinhaltet eine konstante Anzahl an Elementen (üblicherweise und auch in folgender Betrachtung ist diese Anzahl in $O(log(n))$). Jedes Level i ist also ein Hash-Tabelle, der beim Zugriff ein Element x mit Tag t über eine level-spezifische Hash-Funktion $H_i(t)$ auf einen Bucket abbildet. Es ist wichtig, dass diese Hash Funktion geheim ist und die Berechnung oblivious bleibt. Die Möglichkeit von Kollisionen besteht bei einem idealen *Random Mapping* mit einer invers-polynominellen Wahrscheinlichkeit, in diesem Fall wird ein Rehashing notwendig, was im schlechtesten Fall eine amortisierte Zugriffskomplexität von $O(n \cdot log^p(n))$ begründet.
Wenn nun ein Client ein Element x sucht, geht er wie folgt vor: Er berechnet einen Tag t, daraus Hash $H_0(t)$ und lässt sich den entsprechenden dazugehörenden Bucket schicken. Falls er nun das



(a) Struktur eines hierachischen ORAMs **(b)** Struktur eines Tree-based ORAMs

Abbildung 2.2: Weitere ORAM Strukturen

Element nicht gefunden hat, steigt er hierachisch ab und lässt sich jeweils von Level i den Bucket der $H_i(t)$ schicken, solange bis das Element x gefunden wurde. Dannach lässt sich der Client von jedem weiteren Level einen Dummy-Wert, der sich je nach Zugriffsnummer n unterscheidet schicken. Der Dummy-Wert-Hash wird $H_i(0, n)$ notiert.
Korrespondierend zu dem Squared Root ORAM muss auch hier nach einer Epoche der Speicher permutiert werden. Dabei nutzt der hierachischen Aufbau eine Permuation π und kombiniert sie mit einem "einsickern" der Elemente. Es ist also eine Epoche je nach Level unterschiedlich definiert. Allgemein wird die Epoche für Level i nach 2^i Zugriffen beendet (wir ignorieren hier und im Folgenden den konstanten Faktor den das erste Level mit sich bringt). Es wird eine Epoche beendet, indem alle Elemente aus Level i in Level $i + 1$ übernommen werden und beide Level obliviously permutiert werden. Die Elemente werden in Level i als invalide markiert und können als Dummy Werte benutzt oder überschrieben werden. Jedes nicht volle Level wird mit Dummy Werten aufgefüllt. Es erfolgt ein Rewrite gelesener Werte in Level 0. Nun beginnt eine neue Epoche für dieses Level.
Die Argumentation der Verschleierung des Zugriffsmusters ist ähnlich wie schon zuvor, im Fall von Square Root ORAM, und wird daher hier ausgelassen. Es sei nur erwähnt, dass hier für jedes Level argumentiert wird, dass jedes Element in einer Epoche nur einmal gelesen wird. Des weiteren werden mit demselben Prinzip wie die Kombination von Shelter und Hauptspeicher auch die Level obliviously kombiniert.
Es ist noch wichtige für die Betrachtung der Effizienz, dass eine Hashfunktion die Tags mit marginaler Kollisionswarscheinlichkeit auf die Buckets abbildet. Die Konstruktion und Berechnung dieser Funktion wird in [18] ausführlich erklärt, an dieser Stelle wird darauf nicht weiter darauf eingegangen. Auch das genutzte Sorting Network wird erst später in dieser Arbeit betrachtet und daher hier ausgelassen.
Nun lassen sich die amortisieren Kosten berechnen. Ein Bucket hat Größe m und ein Level-Speicher hat Größe b^i in Level i mit $b = 2$ für den binären Fall. Es wird zuerst der erste Speicher mit Aufwand bm gescannt, dann alle weiteren Level angefragt, was zu einem Aufwand von $(1 + \lceil log_2(n) \rceil) \cdot m$ akkumuliert wird. Das Rehashing der Elemente amortisiert nach Epoche hat einen Aufwand von $\frac{O(b^i m \cdot log_2(b^i m))}{b^{i-1}}$. Damit haben *Goldbach et. Ostrovsky* errechnet, dass ihre Konstruktion amortisiert eine Zugriffskomplexität von $O((log(n))^3)$ hat.

Tree Based ORAM Die Binärbaumstruktur von *Shi et al.* [40] markiert einen erneuten Meilenstein in der Entwicklung von ORAM. Sie reduziert den benötigten Overhead besonders im *MPC-Client* Fall enorm und bringt nützliche Worst-Case Effizienzgarantien. Es gelang ihnen, in den meisten

Fällen auf *Hashing* und *Oblivious Shuffling* Methoden zu verzichten und gegen ein über die Zeit verteiltes Einsickern der Elemente in eine Baumstruktur zu ersetzen. Da hier kein Rehashing benötigt wird, ist der Aufwand enorm gesteigert und die untere Grenze von Sorting Networks von $\Omega(n \cdot log(n))$ für eine erneute Permutation wird umgangen.
Die Konstruktion verbindet einzelne ORAMs mit einer deterministischen Zugriffsgarantie (also keine Verwässerung durch z.B. Rehashing) und einem nicht-kontinuierlichen Identifier Space durch eine Baumstruktur. Diese einzelnen ORAMs sind im Folgenden mit *Buckets* bezeichnet und unterstützen die Operationen *Read*, *Write*, und *Pop* (Lesen und Entfernen eines zufälligen Elements). *Shi et al.* bringen hierfür die triviale Konstruktion und den Square Root ORAM als hier schon betrachtete Beispiele vor.
Zum einfacheren Verständnis, wird hier zunächst eine ähnliche Struktur erklärt, die zwar einen bessere amortisierte Laufzeit hat, jedoch einen logarithmisch wachsenden Speicherbedarf hat, der unter Umständen komplexere arithmetische Operationen verlangt. Das ist für den eigentlichen ORAM-Anwendungsfall, sowie die abgeänderte MPC-Variante nicht erstrebenswert. Später wird diese Struktur modifiziert und der eigentliche ORAM dargelegt.
Jedes Element wird zum Write an der Wurzel in den Bucket eingefügt. Ein Bucket hat eine heuristische Kapazität von $log(n)$ Elementen. Falls nun dieser Bucket voll ist, kommt es zu einem Prozess den *Shi et al.* "Eviction" nennen. Hierbei sickern Elemente in die Buckets der zwei Kindknoten, dafür liest der Client jeden Wert aus dem zu leerenden Bucket und fügt ihn zufällig gleich verteilt entweder in den ersten oder den zweiten Kindknoten ein. Um hier Obliviousness zu garantieren wird in den jeweilig anderen Bucket ein zu ersetzenden Dummy Wert eingefügt. Vorerst nehmen wir an, dass der Client speichert, in welchen Bucket er die Werte einfügt und bildet daraus für jeden Index eine Zugriffssequenz $\{0,1\}^{\lceil log_2(n) \rceil}$. Dieses Einsickern findet rekursiv statt. Zum Lesen eines Wertes wird nun eine Leseanfrage auf jeden Bucket, der auf dem Pfad der Zugriffssequenz liegt durchgeführt und das entsprechende Element vom Client entnommen und im eigentlichen Bucket als invalide/Dummy-Element markiert. Es findet ein Rewrite in der Wurzel statt. Wird mehr als ein treffender Wert gefunden, ist nur der an der höchsten Ebene stehende Wert valide.
Hier kann man nun mit einem formalen Simulationsbeweis zeigen, dass solange die Buckets sicher implementiert sind und die Zugriffssequenz geheim ist, die gesamte Konstruktion *oblivious* ist.
Um nun den vorher angesprochenen clientseitigen Overhead, der durch die Speicherung der Zugriffssequenzen entsteht, zu eliminieren, werden die Zugriffspfade in einer rekursiven ORAM-Konstruktion gespeichert, wobei für jeden Index in einem Lookup-Tree, für jedes Level gespeichert wird, welchen der Kindknoten man auswählen muss. Es entsteht also zuerst ein zweiter ORAM der für den entsprechenden Index den Zugriffspfad speichert. Falls nötig setzt sich dieses Schema rekursiv fort. Abbildung 2.2b stellt hierbei die finale Struktur des ORAMs dar.
Nun können die **amortisierten Kosten** zusammengefasst werden. Für einen Zugriff werden jeweils $log_2(n)$ Anfragen an ORAMs der Größe $log(n)$ gestellt. Diese summieren sich, mit der Komplexität des Einsickerns für die triviale Konstruktion zu einer Komplexität von $\tilde{O}(log(n) \cdot log(n))$[1] und für die Wurzelkonstruktion zu $\tilde{O}(log(n) \cdot \sqrt{log(n)})$ auf. Der rekursive Lookup des Pfades enthält zudem einen Faktor von $log(n)$, was in beiden Fällen zu einer Worst-Case Komplexität von $\tilde{O}(log(n)^3)$ führt. Während die amortisierten Kosten für einen Lookup bei Square-Root ORAM etwas geringer sind, ist die eigentliche Größe der Buckets deutlich geringer und der Overhead fällt noch bei deutlich größeren n sehr ins Gewicht.

[1] $\tilde{O}$ zeigt hier für die in der Literatur gängige Unterschlagung von $log(log(n))$ Termen in O-Notation an

Zu dieser Konstruktion gibt es noch weiterführende Arbeiten wie z.B. von *Chung et Pass* [10], die diese Konstruktion verbessert haben, indem sie die Eviction-Prozeduren durch Flush Prozeduren ersetzen, die Einträge immer soweit wie möglich nach unten ziehen. Der Vorteil ist hierbei, dass die Anzahl der rekursiven Overflows reduziert wird und das Access Pattern, das zur Obfiszierung der eigentlichen Zugriffe verwendet wird, besser genutzt wird um zukünftigen Aufwand durch Kombination zu sparen. Zwei weitere maßgebliche verbesserte Tree-ORAM Strukturen werden im Folgenden noch vorgestellt.

Path ORAM Diese Konstruktion von *Stefanov et al.* [41] ist eine neue Erweiterung des Tree-based ORAMs, bei dem besonders die Eviction Prozedur von dem ursprünglichen Schema divergiert. Die Authoren beschränken sich hierbei nicht auf konstanten Client-Storage, sondern lassen einen statistisch wahrscheinlichen Overhead von $O(B \cdot log(N) + log^2(N))$ bei einer ORAM-Größe von N und Speicherblockgröße von B zu. Algorithmen mit wachsendem Client Speicher werdem in dieser Arbeit allgemein nicht genauer betrachtet, da sie i.A. mit frequenten komplexen Berechnungen im Client verbunden sind, die sie schlecht für Secure Computation skalierbar machen. Das ist bei Path ORAM nicht der Fall. Die hier angewendeten Operationen im Client-Speicher, dem sogenannten *Stash*, dienen hier besonders dazu die Eviction Prozeduren zu vereinfachen. Asymptotisch wird der ORAM mit einer Kommunikationskomplexität von $O(log^3(N))$ notiert.
Strukturell besteht diese Konstruktion serverseitig aus einem (nicht zwingend binären) Baum und wieder einem rekursiven ORAM zur Speicherung einer Position-Map (analog zum Tree-Based ORAM). Clientseitig werden in einem Stash die Overflow Blocks gespeichert, die aufgrund von Kollisionen in der Position-Map nicht in den Baum eingefügt werden können. Anfangs wird der Baum mit Dummy Werten gefüllt. Bei jedem Zugriff (Read o. Write) wird nun jeder Block mit einem Blattknoten assoziiert. Ist der Index des Blockes schon gespeichert, passiert das über eine Anfrage der Position Map an den rekursiven ORAM, der wieder Zugriffspfade speichert. Dann wird dieser Wert erneuert und ein neuer Blattknoten ermittelt. Alle Blöcke, die mit den Knoten auf dem alten Pfad assoziiert sind, werden eingelesen. Nun wird aus diesem Pfad der aktuelle Block durch einen Dummy Wert ersetzt und zurückgeschrieben. Nun kann der aktuelle Block mit potenziell neuen Daten an einen Knoten aus dem neuen Pfad zurückgeschrieben werden. Dabei wird immer der am weitesten von der Wurzel entfernte noch freie Knoten benutzt. Ist hier kein Knoten frei, lagert der Client diesen im Stash und schreibt die selben Werte (rerandomisiert) zurück.
Falls nun ein Block wieder heuristisch $O(log(n))$ Einträge beinhaltet und $O(log(n))$ Blöcke zur Verschleierung des Zugriffsmusters des Pfades gelesen werden müssen, ist zusätzlich mit der Adressierung der Blöcke über eine Position Map eine Gesamtkomplexität in der Kommunikation von $O(log^3(n))$ zu veranschlagen. Diese kann allerdings heuristisch mit der Größe der Blöcken verbessert werden. So ist ein $loglog(n)$ Faktor theoretisch besser, aber für kleine n ungeeignet. In der Praxis hat sich eine konstante Blockgröße von 4 Einträgen laut *Stefanov et al.* bewährt.
Ein klares Problem in der theoretischen Betrachtung ist die schwere Analyse der zusätzlich benötigten *Stash* Größe, die direkt von der Overflow Wahrscheinlichkeit abhängt. Da die allgemeine Betrachtung keine enge Garantie geben kann, muss hier die Wahl der Parameter des Baumes und der Blöcke klar eingeschränkt werden um den Stash möglichst gering zu halten. Hier entsteht wieder das Problem, dass die Stash Werte im MPC Fall verschlüsselt und in jedem Schritt alle möglichen Plätze ausgewertet werden müssen. Dadurch ist die loose Garantie der benötigten Plätze im Stash zusätzlich problematisch.

Circuit ORAM Ein weitere Verbesserung liefern *Wang et al.* [42]. Sie stellen fest, dass Path-ORAM aggressiv versucht die Blöcke zu den Blattknoten hin zu ziehen. Jedoch durch das ständige Lesen und

dem damit verbundenen Umsetzen der einzelnen Blöcke jedoch statistisch nicht vernachlässigbar viele freie Knoten auf tieferen Ebenen entstehen, die nicht gefüllt werden, bevor höhere Ebenen gefüllt werden. *Wang et al.* nutzen hierbei die Scans der Pfade um zu registrieren, wie viele freie Plätze für Blöcke noch unterhalb des ersten gefüllten Knotens auf dem Pfad von Wurzel zu Blatt existieren und füllen diese dann sukzessive unter Beachtung der Invariante, dass ein Block immer auf dem Pfad zwischen Wurzel und Position des Blattes liegt, auf. Diese Eviction Strategie lässt sich bildlich als nach "unten ziehen" der Blöcke in dem Baum beschreiben. Die Autoren zeigen, dass diese Konstruktion sehr nah an das Goldreich-Ostrovsky-Limit zur Effizienz von ORAM kommt. Die Effizienz liegt hier also wieder in $O(log^3(n))$ pro amortisiertem Zugriff. Zusätzlich haben sie ihr Schema so konstruiert, dass es für MPC optimiert ist und erzielen in diesem Anwendungsfall besonders gute praktische Ergebnisse.

2.3.3 ORAM für MPC

Obwohl ORAM eigentlich als theoretische Basis für Softwareschutz konzipiert wurde, hat sich dieser Forschungsbereich besonders in Kombination mit MPC entwickelt. Die Forschung, die besonders in den letzten zehn Jahren entstanden ist, zielt darauf ab, mit einer praktisch anwendbaren homomorphen Verschlüsselung zusammen Anwendung zu finden. Daher ist eine Betrachtung der Konstruktionen in diesem Kapitel nicht vollständig ohne ein verteilt-verschlüsseltes Client-Modell zu betrachten.
Konzeptionell muss hierbei nur das Protokoll des Clients durch ein Protokoll mit gleicher Funktionalität, aber verteilt verschlüsselt, berechnet werden. Damit ist allerdings auch klar, dass diese Berechnungen des Clients auch zeitkritisch werden können. Insbesondere Berechnungen mit vielen Multiplikationen oder Vergleichen sind hier problematisch.
Detaillierter kann man sagen, dass jede Partei nun nicht mehr entweder Client oder Server ist, sondern dass jede Partei sowohl Server als auch einen Teil des Clients implementiert. Hierbei ist der ORAM aus einzelnen Teilen bei jeder Partei zusammengesetzt. Also falls Parteien $P_0, P_1, ..., P_n$ ein Anzahl an verschlüsselten Variablen a_i (für $1 \leq i \leq m$) in Form von Secret Sharings teilen, sodass $a_{i,j}$ das Secret Sharing von a_i und Partei P_j bezeichnet, so ist ein ORAM $O = O_1 + 0_2 + ... + O_n$ wobei für jeden Wert a_i im ORAM O die entsprechende Secret Share $a_{i,j}$ in O_j liegt. Der verschlüsselte Indexwert b für den Zugriff liegt wieder in Form eines Secret Sharings der Parteien vor. Hier berechnen die Parteien verschlüsselt aus b, den eigentlichen Index in der ORAM Struktur, sowie es jeweils oben beschrieben ist (bsp. bei Tree-Oram einen Bucket und einen zugehörigen Tag) und legen diese Indexinformation offen, damit kann nun jede Partei die eigenen Secret Shares ermitteln und so kann gemeinsam ein Wert aus dem ORAM gelesen oder geschrieben werden.
Bei den hier betrachteten Konstruktionen ist besonders die hierarchische Lösung negativ hervorzuheben, da hier sichere verschlüsselt Hashes berechnet werden und aus Sicherheitsgründen nur das Ergebnis bekannt werden darf. Im Squared Root ORAM muss sowohl die Berechnung des Tags, als auch das Oblivious Sorting, in verteilter Art und Weise berechnet werden. Der Overhead beim Tree Based ORAM ist hierbei abhängig vom unterliegenden Bucket-ORAM und es kommt kein weiterer Faktor dazu, da dass zusätzliche Mapping so konzipiert ist, dass es in einem zweitem möglicherweise linearem ORAM gespeichert ist. Es lässt sich also heuristisch sagen, dass die Performance des Tree Based ORAM besonders gut in Kombination mit MPC verwenden und die hierarchischen Lösung eher schlecht.

Optimierungen des Tree-based ORAMs für MPC *Gentry et al.* [15] betrachten den Tree-based

ORAM von *Shi et al.* [40] speziell im Kontext von *Secure Computation* und modifizieren den Algorithmus konkret an drei Punkten im Schema und betrachten inwiefern HE genutzt werden kann um einzelne Algorithmen für den ORAM Zugriff zu beschleunigen. Folgende Maßnahmen schlagen sie vor:

(i) Eine Reduktion der Höhe des Baumes, realisiert durch Verdoppelung der Anzahl der Elemente in einem Bucket, hilft den Speicherplatz zu reduzieren, da hier nicht mehr $\frac{n}{2}$ sondern $\frac{n}{m}$ Blätter gespeichert werden müssen (m ist hier die neue Größe der Buckets). Es müssen weniger Dummy Werte verrechnet werden, ohne dass die Overflow-Warscheinlichkeit der inneren Knoten steigt.

(ii) Einen höheren Branching Faktor hilft weiterhin die Größe des Baumes zu reduzieren. Speziell für einen neuen Faktor k reduziert sie die Größe um den konstanten Faktor $log(k)$. Allerdings wird dazu eine neue Eviction-Prozedur benötigt, die nicht nur eine binäre Auswahl unterstützt.

(iii) Eine neue Eviction Prozedur, wie schon in [10, 41, 42] vorgestellt, ist effizienter, da die Elemente bei linearen Scans direkt so weit wie möglich nach unten geschoben werden.

Weiterhin legen *Gentry et al.* dar wie homomorphe Verschlüsselung genutzt werden kann um einen MPC-Client zu simulieren. Dabei nutzen sie die geringe Tiefe der ORAM-Client Berechnungen aus [40] um mithilfe von SHE effiziente MPC-Client-Simulationsalgorithmen zu beschreiben. Sie betrachten nicht wie in dieser Arbeit betrachtet den wesentlich beschränkteren Fall der Berechnung von SS, sondern nehmen dennoch einen externen ORAM-Server an, der nicht zwangsweise von unterschiedlichen Parteien unterhalten wird. Das ist ein deutlicher Unterschied zu den Annahmen am Anfang dieses Abschnittes und dem Setting in Kapitel 4 daher sind diese Vorschläge in dieser Arbeit nicht weiter relevant. Der Grundgedanke, die geringe Circuit-Tiefe der Berechnungen zu nutzen, ist dennoch erwähnenswert.

Direkte Verknüpfung von Sharing und ORAM Um eine bessere Skalierbarkeit von ORAM im MPC Einsatz zuzusichern, stellen *Doerner et Shelat* [12] einen neuen ORAM, den sogenannten *FLORAM* vor. FLORAM steht für Function-secret-sharing Linear ORAM. Dabei ist die Laufzeitkomplexität linear, aber durch eine Verknüpfung zwischen Secret Sharing Algorithmus und ORAM sind praktische Anwendungen für selbst hohe ORAM Größen effizent. Sie nutzen Distributed Point Functions um ein Function-SS so zu definieren, dass der Evaluationsalgorithmus eine Maske aufdeckt, mit dem der Index des Elements aufgedeckt werden kann. Point Functions sind hierbei Funktionen, die nur an einer Stelle (Punkt) nicht zu Null sondern einem gegebenen Wert evaluieren. Um ein konkretes ORAM-Schema zu entwickeln, konstruieren *Doerner et Shelat* zunächst ein Oblivious Write-Only Memory Protokoll und ein Oblivious Read-Only Memory Protokoll.
Dabei nutzen beide Parteien zum Write-Only-Memory ihre alten Shares um mithilfe von privaten Schlüsseln den alten Wert in einem XOR Sharing durch einen neuen Wert zu ersetzten. Hierbei wurden die Schlüssel so generiert, dass sie an einem (für die Parteien unbekannten) Wert i ein Wert-Delta auf den alten Wert aufaddieren. Daher wird die Verschlüsselung mittels *Distributed Point Functions* benötigt, um zu verhindern, dass Werte an anderen Indizies verändert werden.
Im Read-Only Memory erhalten beide Parteien wieder Schlüssel, die aus einem pseudozufälligen Schlüssel und einem Index i so generiert werden, sodass genau der Wert der Share mit dem Index i durch XOR-ing entschlüsselt werden kann. Dabei kann nun für jeden Wert die Evaluation durchgeführt werden und man erhält ein Tupel (y_x, t_x) für $(x)_{1 \leq x \leq n}$ wobei jede Share W_x mit dem XOR von y_x verdeckt wurde und durch ein XOR mit t_i die Decryption-Share entsteht. Die Verschlüsselung ist nun so konstruiert, dass $W_x \oplus t_i$ nur für $x = i$ die Decryption Share entsteht und

der verschlüsselte Wert von allen anderen Parteien nicht verändert wird. Nun können beide Parteien kooperieren um über $PRF_k(i) \oplus v_a \oplus v_b$ den eigentlichen Wert offenzulegen (hier notieren v_a und v_b Decryption-Shares und $PFE_k(i)$ die genutzte pseudozufällige Funktion um den Schlüssel zu generieren).
Daraus kann nun ein kombinierter ORAM entstehen, wobei die *Read* und *Write* Konstruktionen semantisch getrennt sind und kombinierte *Read/Write* Zugriffe stattfinden. Es wird also immer ein Read in einem Read-Only ORAM und ein Write in einem Write Only ORAM stattinden. Zur Effizienzsteigerung werden Writes zusätzlich auf einem *Stash* gespeichert. Nach einer gewissen Epoche muss nun eine Refresh-Prozedur den Write Speicher mit Stash in den Read Speicher umschreiben (wieder eine Art von *Eviction* am Ende einer Epoche). Es ist einfach zu sehen, dass diese Konstruktion einen lokalen Aufwand von $O(n)$ hat, da bei jedem Read/Write alle Werte mit den Schlüsseln verrechnet werden. Des Weiteren hat ein Zugriff eine Komplexität von $O(\sqrt{n})$ verschlüsselten Operationen und Kommunikation. Hierbei ist die Anzahl der Runden konstant.

Diese Konstruktion zeigt besonders was möglich ist wenn MPC und ORAM kombiniert verwendet werden. Insbesondere die konstante Rundenzahl und der effiziente Overhead in verschlüsselten Operationen (XOR wird als billige Operation in homomorphen Verschlüsselungsverfahren betrachtet) macht das Protokoll interessant für den Anwendungsfall in PFE.

2.3.4 Weitere Arbeiten

Die bis hierhin vorgestellten ORAM-Konstruktionen bilden eine breite Grundlage, auf der verschiedene weitere Schemata entwickelt wurden, die einzelne Faktoren verbessern.
Es gibt eine Reihe von ORAM Konstruktionen, die amortisierte Zugriffskosten optimieren in $O(log(n))$ bzw. $O(log^2(n))$ liegen. Diese bereiten allerdings in der Praxis das Problem, dass ein nicht konstanter Client-Speicherplatz notwendig wird, was in MPC-Berechnungen oftmals mit einer sehr komplizierten Simulation eines Clients belegt ist, wodurch sie für PFE nicht interessant werden. (vgl. z. B. [4, 19, 20])
Auch mit der direkten Verbesserung der hier betrachteten ORAMs haben sich viele Arbeiten beschäftigt. Es wird argumentiert, dass bei den State-of-the-Art Konstruktionen nicht die Skalierung sondern der konstante Overhead selbst bei nominal kleinen Instanzen prohibitiv ist und eine Reduktion des Overheads diese Technik in der Praxis anwendbarer machen soll. Insbesondere rekursive Konstruktionen wie [40] können deutlich effizienter gemacht werden. Dazu ist besonders die Arbeit von *Zahur et al.* [44] im Falle des Squared-Root-ORAMs hervorzuheben. Auch für den schon sehr effektiven Tree-ORAM gibt es direkte Verbesserungsvorschläge wie z. B. von *Chung et Pass* [10].
Ein dritter Bereich zur Verbesserung und der für die Anwendung innerhalb einer PFE-Konstruktion, ist die Literatur zur Verbesserung der Effizienz eines MPC-Clients der über mehrere verschiedene Parteien geteilt wird. Dieser Teil der Forschung ist zeitlich gesehen eines der neueren Felder. Allerdings wurde schon jetzt die Effizienz einzelner ORAMs deutlich verbessert. (vgl. [14, 33])

2.4 Permutationen

Es werden eine Reihe von Primitiven benötigt um eine (pseudo-)zufällige effiziente Permutation zu realisieren, sodass Strukturen und Topologie von Programmen bzw. Schaltungen gegenüber Parteien, die die zu berechnenden Funktionen nicht kennen, zu verschleiern. Diese Algorithmen werden sowohl in gängiger PFE-, also auch ORAM-Literatur verwendet und sind weitestgehend austauschbar, sodass sie hier gesammelt betrachtet werden, um Redundanzen in den unterschiedlichen Kapiteln zu vermeiden.

2.4.1 Sorting Networks

Sorting Networks sortieren ein ungeordnetes Input $(x_i)_{1 \leq i \leq n}$. Dabei nutzt ein Sorting Network nur ein komperatives Element, was ggfs. die Positionen zweier Eingaben a und b tauscht, sodass eine Position $min(A, B)$ und eine andere Position $max(A, B)$ enthält. Des Weiteren sind Sorting Networks *data oblivious*, also die Schritte des Algorithmus sind, bis auf das Ergebnis des Komperators, deterministisch.
Aufgrund dieser Eigenschaften bieten sich Sorting Networks besonders gut für Aufgaben im PFE Bereich an. Es können Sorting Networks im Client-Server-Setting mit minimalem Client-Speicher realisiert werden (2 Elemente), aber es kann auch der Interaktionsrundenaufwand reduziert werden, indem die typischerweise hohe Parallelisierung ausgenutzt wird. Die Komperatorfunktion ist einfach zu implementieren und der deterministische Aufwand lässt in MPC, basierend auf einem Offline-Online Paradigma, sehr genau abschätzen und vorberechnen. Es ist aus Sicherheitsgründen immer notwendig, dass die Permutation der Elemente hier data oblivious geschieht, sodass keine Informationen aus der geheimen Permuation öffentlich werden, wie es bei normalen Sortieralgorithmen der Fall ist. In Abschnitt 2.3 wird angenommen, dass solche Permutationen und Sortierungen möglich sind, im Folgenden werden zwei einfache Varianten vorgestellt.

Batchers Sorting Network entwickelt von *Batcher* [5] umfasst zwei mögliche rekursive Implementierungen von Sorting Networks. Zum einen werden die Elemente je nach geradem oder ungeradem Index verglichen. Hierfür wird der Input in zwei Sequenzen $(a)_{1 \leq i \leq s}$ und $(b)_{1 \leq i \leq t}$ aufgeteilt und jeweils die korrespondierenden geraden bzw. ungeraden Indizes von a und b rekursiv sortiert. Hierbei werden wieder die korrespondierenden Indizes aus den Odd-Even-Sortern komperativ in eine richtige Sequenz zusammengeführt. Als zweite Möglichkeit schlägt *Batcher* einen "bitonic Sorter" vor. Beim bitonischen Sortieren, werden Elemente zuerst rekursiv in bitonische Sequenzen zerteilt und dann diese Eigenschaft zum Zusammensetzen ausgenutzt. Eine bitonische Sequenz besteht aus einer (im Wert) monoton aufsteigenden und einer monoton absteigenden Sequenz. In Batchers Algorithmus werden hierfür zwei eine bitonische Sequenzen mit insgesamt $2m$ Elementen in eine bitonische Sequenz transformiert, indem man jeweils Element i und $i + m$ für $i = 0..m$ miteinander vergleicht und das kleinere Bit an Position i in einen bitonischen Sortierer der Größe m gibt und das größere Bit an Position i in einen bitonischen Sortierer der Größe m gibt. Am Ende werden die Sequenzen beider Sortierer konkateniert und bilden wieder eine bitonische Sequenz. Die Rekursionsabbrüche sind in beiden Konstruktionen trivial. Das Ergebnis ist in beiden Fällen korrekt und der Datenfluss bis auf die Auswertung im Komperatorelement datenunabhängig. Diese Komperator Funktionalität kann clientseitig durchgeführt werden und ist effizient in MPC zu realisieren.

AKS Sorting Network entwickelt von *Ajtai, Komlós, und Szemerédi* [2] ist ein zweites Sorting

Network, dass sich für die Aufgabe des oblivious Shuffling eignet. Hierbei ist die Komplexität mit $O(nlog(n))$ besser als bei Batchers Sorting Networks, allerdings fällt der konstante Overhead gegenüber kleinen Größen deutlich mehr ins Gewicht und daher ist es nicht immer die beste Wahl. Das Sortierverfahren basiert wieder nur auf komperativen Elementen, ist ansonsten deterministisch und kann stark parallelisiert werden. Es besteht aus drei Phasen, $\varepsilon - halving$, $\varepsilon - nearsort$, und dem eigentlichen Algorithmus. Ziel ist es eine zu sortierende Sequenz möglichst früh in eine untere und eine obere Hälfte aufzuteilen. Dazu wird ein $\varepsilon - halver$ als Schaltungselement benutzt. Dieser teilt die Sequenz zwar, lässt aber einen geringen Fehler zu. Dadurch kann eine höhere Effizienz und eine frühere Parallelisierung ermöglicht werden, dafür müssen Fehler später in einer Postprocessing Phase ausgeglichen werden. Als zweites werden die Sequenzen von $\varepsilon - nearsort$ sortiert, hierbei ist auch ein begrenzter Fehler zulässig. Im eigentlichen Algorithmus wird eine Binärbaumstruktur konstruiert, wobei jeder Knoten hier das Intervall der darunterliegenden Knoten rekursiv abbildet. Dann werden logarithmisch oft, nacheinander die Knoten in den geraden bzw. ungeraden Leveln in einer bestimmten Sequenz sortiert. Es entsteht eine totale Ordnung. Die Ausführung dieses Sorting Networks ist oblivious, sofern der Client zwei Elemente obliviously vergleicht und diese wenn nötig vertauscht.

Switching Networks *Batcher* [5] betrachtet auch die Umwandlung von Sorting Networks in Switching Networks. Mit dieser Erweiterung können Sorting Networks beliebige Funktionen auf Sequenzen realisieren, indem vor dem Sortieren die Elemente auf eine neue Position (gegeben der Funktion) abgebildet werden. Das ist die gängige Weise in der Literatur zu ORAM und PFE eine zufällige Funktion der Elemente mittels Sorting Networks zu realisieren. *Batcher* betrachtet auch Methoden fehlerkorrigierende Switching Networks zu realisieren, diese wurden bis jetzt noch nicht in der Literatur zu ORAM betrachtet, da mit einem übermäßigen Overhead zu rechnen ist. In aktuellen Realisationen von zufälligen Permutationen wird ein Fehler mit Wahrscheinlichkeit $\frac{1}{poly(n)}$ in Kauf genommen. Fehlererkennungs und Korrekturmaßnahmen, die über die Methode von *Batcher* hinausgehen, werden in Abschnitt 2.3 und Kapitel 3 diskutiert.
Eine ähnliche Notation wird von *Mohassel et Sadeghian* [36] vorgeschlagen. Hierbei ist ein Switching Network eine Menge an verbundenen Switches, wobei jeder Switch zwei Eingaben bekommt und zwei Ausgaben ausgibt, die je nach Selector Bit geändert werden. Es ist festzustellen, dass ein Komperator, wie im vorigen Abschnitt zu Sorting Networks definiert, einen solchen Switch realisiert.

2.4.2 Permutation Networks

Ein Switching Network ist ein Permutation Network, falls das Mapping π eine Permutation realisiert, also die Werte nur vertauscht werden. Eine Permutation stellt ein bijektives Mapping dar, es dürfen keine Werte wegfallen. Nach [36] ist einfach zu sehen, dass ein Permutation Network, das mindestens $O(n \cdot log(n))$ Switches benötigt um beliebige Permutationen darzustellen.
Das ist ersichtlich, da es $n!$ Permutationen gibt und mit jeder Permutation maximal zwei Elemente verändert werden können. Betrachtet man für eine beliebige Eingabe die möglichen Ergebnisse hintereinander ausgeführter Switches, stellt man fest, dass $log(n!)$ Schritte benötigt werden um jede Permutation zu erreichen. Es ist bekannt, dass $log(n!) \in \theta(n \cdot log(n))$ gilt.
Damit ist eine untere Schranke gegeben, die später auch den Einsatz von ORAM zur Ersetzung von Permutationsnetzen begründet, da sich hier die Kosten über viele Zugriffe amortisieren können.

Extended Permutations wie in [36] definiert, erweitern die Notation der Permutationen, um Circuits

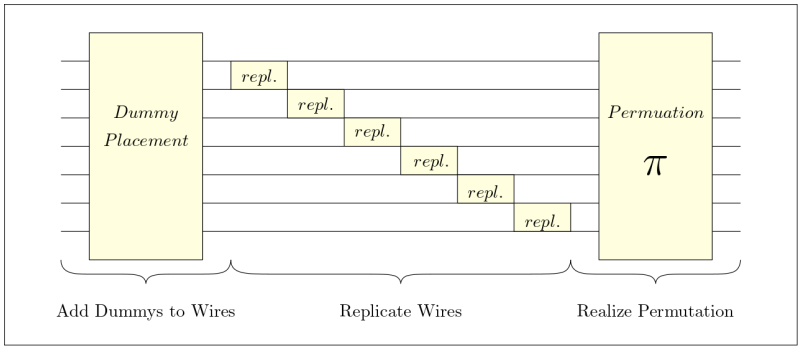


Abbildung 2.3: Oblivious Evaluation eines Swiching Networks zur Realisierung einer OEP

und Programme genereller abzubilden. Sie erlauben hierbei nicht nur, dass Projektionen bijektive Abbildungen sind, die einen Wert auf einen anderen Wert abbilden, sondern es darf auch ein Wert auf mehrere Werte abgebildet werden. Das Mapping ist immer noch injektiv, d.h. $\pi^{-1}(y)$ ist eindeutig.

Die Evaluation einer Extended Permutation kann nach [36] in drei Bestandteile aufgeteilt werden. Zum einen werden Dummy Werte hinzugefügt, die als Eingabe gelten, aber vorher nicht dabei gewesen sind. In einer zweiten Phase werden diese Dummy Werte durch die Replikation schon vorhandener Werte verändert und danach in einem normalen Permuationsnetzwerk wie eine normale Permutation behandelt. Das ist allerdings in der Praxis nur notwendig, falls Circuits berechnet werden sollen. Im generellen ORAM-Fall, kann hier einfach ein weiteres Write-Back stattfinden.

Abbildung 2.3 stellt dar wie nun in drei Schritten mittels zwei Switching Networks, eines zum Dummy Placement und eines zur Realisation einer Permutation π eine Erweiterte Permutation data-oblivious berechnet werden kann. Dazu werden die Netzwerke über eine Replikationskomponente verbunden, die nach dem Dummy Placement eine Wire beliebig oft auf die nachfolgenden Elemente replizieren kann. Diese Darstellung ist ein Vorgriff auf Kapitel 3, dort wird diese Notation in verschiedenen Konstruktionen verwendet (vgl. z. B. [7, 37]).

3 Private Function Evaluation

In diesem Kapitel ist das Ziel die Literatur bezüglich PFE zu sichten und ein sicheres State-of-the-Art PFE-Protokoll zur Evaluation von arithmetischen Programmen zu entwickeln, dass ORAM nutzt um die Topologie des Programmes zu verstecken. Dabei sollen zu Vorbereitung besonders eine Reihe von Vorüberlegungen dargelegt werden. Es gilt zu ermitteln welche Attribute ein ORAM erfüllen muss um sinnvoll eingesetzt zu werden. Es soll evaluiert werden, inwiefern eine Auswertung von einem arithmetischem Gatter kosteneffizient implementiert werden kann. Es soll ein Modell mittels CPU-Step-Circuits und Operation Evaluation Functionality entworfen werden um entsprechende Programme zu evaluieren.

3.1 Eigenschaften

Im Folgenden werden Angreifermodelle und korrespondierende Sicherheitseigenschaften, sowie allgemeinere angestrebte Eigenschaften einer idealen PFE-Funktionalität in unterschiedlichen Settings vorgestellt. Die Protokolle, die in diesem Kapitel vorgestellt werden, werden besonders unter diesen vorgestellten Eigenschaften betrachtet und verglichen. Das soll besondere Techniken und Konstruktionsweisen hervorheben, die später genutzt werden können um eine eigene PFE-Konstruktion nach den gegebenen Anforderungen zu modellieren.

Angreifermodelle

Angreifermodell für Passive Sicherheit Im passiven Angreifermodell kann ein Adversary t von n Parteien kontrollieren (wobei t hier je nach Modell zwischen 1 und $n-1$ gewählt werden kann, je nach gewählter Härte der gewünschten Garantien), sowie die gesamte (verschlüsselte) Kommunikation über das Netzwerk. Es ist dem Adversary hier nicht erlaubt von dem Protokoll abzuweichen, allerdings ist sein Ziel die gelernten Informationen in einer Weise zu kombinieren um zusätzliche Informationen, die nicht im Pre-Agreement des Protokolls gelistet sind und die auch nicht während des Protokolls öffentlich gemacht werden sollen oder ihm zugetragen werden sollen, zu erlangen.
Speziell im allgemeinen PFE-Fall kann man feststellen, dass Parteien einen unterschiedlichen Stellwert haben.

Angreifermodell für Aktive Sicherheit Ein aktiver Angreifer erhält mehr Fähigkeiten als ein passiver. Er kann wie auch im passiven Modell t von n Parteien kontrollieren, wobei entweder P_0 und $P_1..P_n$ nicht vollständig vom Adversary kontrolliert werden. Er enthält ein vordefiniertes Grundwissen, das Wissen kontrollierter Parteien, und ein Transkript der gesamten Netzwerkkommunikation. Nun kann er vom Protokoll abweichen und Nachrichten durch andere ersetzten. Es ist einem aktiven Angreifer nicht erlaubt zu sehr von seinen ursprünglichen (nicht manipulierten)

Eingaben abzuweichen, da es sonst trivial ist Informationen offenzulegen (siehe unterer Abschnitt). Hierbei bedeutet “nicht zu sehr abweichen”, dass die Eingaben entweder auf die ehrlichen Werte beschränkt sind (keine Abweichung) oder Funktionen/Eingaben aus einer vordefinierten Menge stammen, bei denen keine Abänderungen eingeführt wurden.

Sicherheitseigenschaften

Im Folgenden werden eine Reihe von Sicherheitseigenschaften formal definiert, anhand deren später die Protokolle verglichen werden können und mit denen die Stärke der Sicherheit gemessen werden kann.
In einer Private Function Evaluation sind Eingabegröße, Ausgabegröße und die Anzahl der Gates bekannt. Hierbei ist allerdings nicht zwischen verschiedenen Gattertypen unterscheidbar. Korrespondierend sollten in einem Oblivious Programm die Anzahl der Schritte nicht versteckt gehalten werden. Für den Fall, dass Schrittanzahl etwas über die Eingabe oder Evaluation der Funktion aussagt, kann die Partei, die eine Funktion beisteuert ein Padding hinzufügen um so Schritte geheim zu halten. Wir gehen hier immer davon aus, dass das eingegebene Programm wahrheitsgetreu ist. Es kann immer ein Programm so entwickelt werden, dass einzelne Teile des Inputs offengelegt werden (bsp. $f(x) = x$) oder die Berechnung nicht vorher bestimmten Parametern entspricht. Es ist jedoch erstrebenswert, dass das verschlüsselte Programm genau so berechnet wird wie es verschlüsselt vorliegt. Hier muss durch das Protokoll sichergestellt sein, dass die Berechnung korrekt ist.

Privacy
In der Definition von **Sicherheit gegenüber eines passiven Angreifers** ist das Ziel, ein Protokoll so zu definieren, dass ein Angreifer mit dem Wissen das den korrupten Parteien zugeteilt wird nichts ermitteln kann, was nicht durch eine Turingmaschine mit dem Input der korrupten Parteien und einem Transkript der Konversation der einzelnen Parteien ausrechnen kann. Diese Notation kann formalisiert werden, wie von *Katz et Malka* in [24] vorgestellt, und über ein Simulationsarguments umgewandelt werden. Dabei ist ein Protokoll sicher, falls es eine Turingmaschine gibt, die in polynomieller Zeit (respektive des Inputs, sowie einem Sicherheitsparameter) und mit dem Wissen der entsprechenden Partei als Eingabe genau die VIEW simulieren kann. Das Wissen der Partei beinhaltet alle Informationen die diese Partei in die Berechnung einbringt sowie ein Transkript der Konversation der Parteien.
Formaler kann man Sicherheit wie in Definition 3.1.1 definieren. Dabei handelt es sich um einen standardmäßigen Sicherheitsbeweis mittels Simulation der Informationen.

Definition 3.1.1 (Privacy gegenüber passiven Angreifern)
Ein Protokoll ist sicher, falls für jede Partei P_i $(0 \leq i \leq n)$ eine probabilistische polynomiell in der Zeit gebundene Turingmaschine S, einen SIMULATOR, gibt, der als Eingabe die Eingabe einer Partei P_i erhält und deren Ausgabemengenverteilung gleich (≙computationally indistinguishable) einer VIEW ist. Diese VIEW enthält abhängig von Protokoll, für jede Partei P_i als Eingabe den Sicherheitsparameter, die Eingabe einer Partei P_i, ein Transkript der Kommuikation der Protokollausführung, und ein Zufallsorakel.

Dagegen kann ein **Angreifer nach dem aktiven Sicherheitsmodell** Kommunikation und Daten manipulieren. Diese erweiterten Fähigkeiten erfordern eine erweiterte formale Definition. Intuitiv muss ein Algorithmus dann auch sicher sein und keine Informationen offenlegen, die dazu führen, dass ein Angreifer beliebige Informationen über die Funktion f (ggfs. auch den Circuit C_f), Zwischenprodukte der Berechnung, oder ggfs. das Ergebnis erfährt. Eine formellere Version ist mittels des REAL/IDEAL-Framework in [24] genauer dargestellt. Hier wird diese Notation informeller dargestellt. Für ein PFE-Protokoll π mit Eingabe x und C_f kann eine reale und eine ideale Komponente definiert werden. Die reale Komponente defineren *Katz et Malka* mittels $\text{REAL}^{\pi}_{\mathcal{A}(z)}(1^{\kappa}, x, C_f) = (\text{VIEW}^{\pi}_{\mathcal{A}(z)}(1^{\kappa}, x, C_f), \text{OUT}^{\pi}_{\mathcal{A}(z)}(1^{\kappa}, x, C_f))$, wobei 1^{κ} der Sicherheitsparameter darstellt, VIEW, wie oben definiert ist und OUT die Ausgabe einer nicht korrupten Partei (VIEW und OUT erhalten dieselben parameter wie REAL). Dagegen würde eine ideale Ausführung eines PFE-Protokolls, die Eingaben C_f an P_0 und x an P_1 und 1^{κ} und Hilfsparameter z abbilden (bsp. ein Pre-Agreement wie in vielen Protokollen gegeben), bei dieser senden Parteien ihre Eingaben an eine vertraute Partei, die dann das richtige Ergebnis (hierbei ist das Ergebnis, das aus einer ehrlichen Auswertung resultiert gemeint) berechnet. Hierbei kann der Adversary die Eingabe alle korrupten Parteien kontrollieren. Die ideale PFE-Funktionalität wird formell $\text{IDEAL}^{PFE}_{A(z)}(1^{\kappa}, x, C_f) = (\text{VIEW}^{PFE}_{\mathcal{A}(z)}(1^{\kappa}, x, C_f), \text{OUT}^{PFE}_{\mathcal{A}(z)}(1^{\kappa}, x, C_f))$ notiert.

Definition 3.1.2 (Privacy gegenüber aktiven Angreifern)
(korrespondierend zu Definition 2 aus [24])
Π *ist sicher gegenüber aktiven Angreifern, falls für jede probabilistische polynomiell in der Zeit gebundene Turingmaschine $\mathcal{A}$, eine korrespondierende gleich definierte Turingmaschine $\mathcal{S}$ existiert, für die folgende Gleichheit (es hält wieder computationally indistinguishabitity anstatt Äquivalenz) gilt:*

$$\{IDEAL^{PFE}_{\mathcal{S}(z)}(1^{\kappa}, x, C_f)\}_{k\in\mathrm{N}, x\in\{0,1\}^l, C_f\in C, x\in\{0,1\}^*} = \{REAL^{\Pi}_{\mathcal{A}(z)}(1^{\kappa}, x, C_f)\}_{k\in\mathrm{N}, x\in\{0,1\}^l, C_f\in C, z\in\{0,1\}^*}$$

Im Gegensatz zu üblichen Definitionen aus Secure Computation Bereichen wird hier eine parallele Ausführung der Funktionalität nicht betrachtet. Die Frage inwiefern ein PFE-Protokoll unter solchen Sicherheitsbedingungen standhält, ist in den meisten Fällen eine zusätzliche Betrachtung, die im Allgemeinen nicht beachtet wird. Literatur die speziell wiederverwendbare Protkolle betrachtet, nutzt hierbei meistens das Universally Composable Security Model von *Canetti* [9].
Verdeckte Sicherheit oder *Covert Security* erlaubt es einer kompromittierten Partei, wie bei aktiver Sicherheit, vom Protokoll abzuweichen. Ein Protokoll gilt dennoch als sicher, solange der Adversary mit einer nicht-vernachlässigbaren (non-neglibile) Wahrscheinlichkeit erkannt wird. In diesem Fall kann die Ausführung abgebrochen werden.
Diese Sicherheit wird im Normalfall erweitert definiert als *publicly verifiable covert security*, wobei eine Partei P_i einen Nachweis erhält, mit der sie gegenüber dritten Parteien nachweisen kann, dass die andere an der Protokollausführung beteiligte Partei vom Protokoll abgewichen ist.
Verdeckte öffentlich verifizierbare Sicherheit im Kontext von PFE ist ein neues und dementsprechend eher seltener betrachtetes Konzept. *Hong et al.* [23] haben dazu verschiedene Möglichkeiten im Kontext von Garbeled Circuits entwickelt, wobei die Partei, die die Funktion hält, einige Gatterverbindungen offenlegen muss. Die verschiedenen Techniken basieren je nach CTH-Funktionalität auf signierten OT-Extension Verfahren oder pseudozufälligen Seeds die als Zeugen beim Öffnen von Gattern fungieren. Hier müssen Nachrichten immer unterschrieben werden um in einem Transkript später öffentlich die Manipulation zu belegen (hierbei sollte auch gesichert sein, dass ein Beleg nur

möglich ist, falls eine Partei manipuliert hat).
Insbesondere unter Betrachtung der Anforderungen, die für Parteien gelten, explizit, dass die Eingaben und die Funktion korrekt sind, ist Covert Security ein praktischer Trade-Off zwischen Effizienz und Sicherheit.

Correctness Ein Programm soll Ergebnisse liefern, die korrekt sind. Also soll eine Funktion f auf Input x das Ergebnis $f(x)$ zurückgeben. Die Eigenschaft sollte abhängig vom Angreifermodell untersucht werden und jeweils gelten.
In gängiger PFE-Literatur wird diese Anforderung oft sehr relaxiert betrachtet, da eine Partei P_0 im Allgemeinen das Ergebnis durch eine veränderte Funktion f manipulieren kann und die anderen Parteien das Ergebnis invalidieren können, indem sie falsche Daten beisteuern.
Dennoch sollte die Eigenschaft für alle Angreifermodelle gelten, da je nach Anwendung, *non-reputability* (äquivalent zu Correctness) eine wichtige Eigenschaft sein kann. (Beispielsweise wenn eine Partei die Ergebnisse mehrerer unterschiedlicher geheimer Funktionsauswertungen vergleichen will oder ein wiederverwendbares Schema eingesetzt wird und man so eine Sicherheit in der Funktion erhält.)

Accountability Falls eine Partei von Protokoll abweicht und so Fehler in die Auswertung einbaut, sollten diese auf die Partei zurückgeführt werden. Das hat den Vorteil, dass hier die Parteien verantwortlich gemacht werden können und aus zukünftigen Berechnungen ausgeschlossen werden.

Guaranteed Output Delivery Sichert einer Partei zu, dass sie definitiv einen Output erhält, falls genügend Parteien nicht kompromittiert sind. Dieses Attribut kann relaxiert werden und beispielsweise die Atomarität des Outputs sicher, also entweder erhalten alle die korrekte Ausgabe oder keiner.

Reusability Eine Eigenschaft des Protokolls, die eine mehrfache Online-Ausführung der Funktion zusichert, ohne dabei andere Eigenschaften (insbesondere Privacy) zu verlieren. Im Allgemeinen kann nicht davon ausgegangen werden, dass Daten, die in einer Offline-Phase generiert werden, wiederverwendbar sind. Insbesondere muss um Reusability zu zeigen (im nebenläufigen oder auch sequentiellen Setting), gezeigt werden, dass alle anderen Attribute sicher gegenüber Mehrfachausführung sind. Das wird im Generellen nicht von PFE-Konstruktionen verlangt. Dennoch kann es hier zu massiven Verbesserungen bei mehrfacher Funktionsauswertung führen, daher entstehen besonders zuletzt immer wieder Versuche, Konstruktionen zu entwickeln, die wiederverwendbar sind [8, 32].

Effizienzkriterien

Ein PFE-Protokoll hat neben den oben genannten binären Eigenschaften noch andere Eigenschaften um die Güte der Konstruktion zu bewerten. Diese Eigenschaften sind je nach Setting (insbesondere von Hardware, Software, Netzwerk, und zu berechnende Funktion abhängig) unterschiedlich wichtig und stellen nur in ähnlichen Fällen eine Metrik zum Vergleich dar. Dennoch sind sie wichtig um die Praktikabilität einer Konstruktion in Anwendungsfällen zu ermitteln.

Asymp. Kommunikationsaufwand In der aktuellen Literatur zu SFE zeigt sich immer wieder, dass nicht die Berechnung der einzelnen Funktionen nicht durch die lokale Berechnungen, sondern die Kommunikationzeit zwischen den einzelnen Parteien dominiert wird. Daher ist es üblich diese Kommunikation, anstatt von asymptotischer Programmkomplexität zu betrachten. *Bingöl et al.* haben verschiedene PFE-Konstruktionen betrachtet und die Kommunikationskosten analysiert (vgl. [8] Table 2). Sie haben für relativ geringe Schlüssellängen ermittelt, dass der

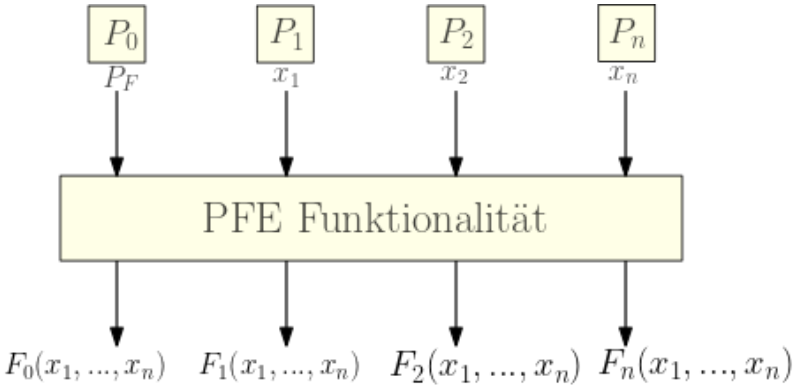
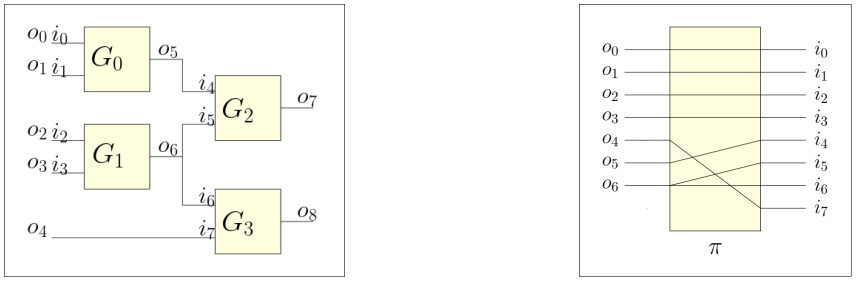


Abbildung 3.1: Aufgabe einer PFE-Funktionalität

Kommunikationsaufwand in reellen Anwendungsfällen schnell prohibitiv sein kann. Dabei ist diese Aufstellung besonders im boolschen Falle interessant, da dort ein deutlicher Gatter Blow-Up in der Dezimalwertigkeit und Genauigkeit der einzelnen zu berechnenden Zahlen stattfindet

Asymp. Rundenkomplexität *Beaver et al.* [6] haben schon 1990 erkannt, dass der Kommnikationsaufwand nicht allein maßgebend ist. Sie betrachten SFE mit besonderem Schwerpunkt auf die Anzahl der Runden in Interaktion zwischen einzelnen Parteien. Sie kommen zu dem Ergebnis, dass ein SFE Setting mit konstanter Rundenzahl möglich ist und damit die Praktikabilität um Größenordnungen gesteigert werden kann. In aktuellen PFE-Konstruktionen sind mittlerweile auch eine konstante Anzahl an Runden möglich, sofern boolsche Schaltungen betrachtet werden. Hier ist oft die Bandbreite (also der gesamte Kommunikationsaufwand) relevanter.

Gatter-Ausdrucksstärke Da Effizienz in PFE meistens in der Anzahl der Gatter gemessen wird, ist es relevant die Anzahl der Gatter möglichst niedrig zu halten.
Boolsche Schaltungen haben in der Regel einen sehr großen Overhead. Dazu haben *Lipmaa et al.* die größen einfacher Boolscher Funktionalitäten analysiert. Sie erreichen jeweils für die Schaltungen folgende Gatteranzahlen 32-Bit Adder: 375, 64 Bit Adder 759, 32-Bit Multiplier 12374, und für AES 33616 Gatter (vgl. [31] Table 1). Diese Größen werden im Allgemeinen für kompliziertere Funktionen auf höher bit-stelligen Zahlen schnell (je nach Protokollunterschiedlich) unpraktikabel, da der Kommunkationsaufwand zu hoch wird.
Arithmetische Schaltungen dagegen sind oft unabhängig von der Bit-Stelligkeit der Zahlen (nach oben begrenzt durch mögliche Kryptosystemparameter) und es können je nach Funktionalität auch komplexere Funktionen in einem Gatter realisiert werden. Dazu können nun heuristisch Gattertypen definiert werden, die je nach Funktion grob- oder feingranulare Operatoren auswerten (die Granularität ist gegeben der Anwendbarkeit in gegebenen Funktionen, bsp. Addition und Multiplikation sind universell und daher grobranular, ein Less-Then Gatter ist deutlich weniger universell einsetzbar und daher feingranular). Wie schon in Kapitel 2 erwähnt reichen allerdings Multiplikation und Addition aus um jede boolsche Funktion darzustellen. Alle weiteren Funktionen können heuristisch zur Optimierung genutzt werden.



(a) Beispiel einer Schaltungstopologie **(b)** Korrespondierendes Mapping π

Abbildung 3.2: Schaltungstopologie und zugehöriges Mapping

3.2 PFE-Setting

Es wird das allgemeine PFE-Setting betrachtet, in dem eine Partei P_0 eine Funktionsauswertung einer Funktion F codiert und verschlüsselt besitzt. Andere mögliche Parteien besitzen den Input hierfür, falls P_0 auch Eingabewerte besitzt können diese fest in die Funktion als konstante Werte eingetragen werden. Dieses Setting wird in 3.1 skizziert. Insbesondere für die späteren Betrachtung von kompromittierten Parteien ist es wichtig, dass P_0 getrennt ist. Hierbei ist diese Auswertung entweder als Schaltkreis C_F oder als Programm P_F codiert. Es darf über einen Schaltkreis die Anzahl der Gatter, die Anzahl der Verschaltungen (Input- und Outputwires) sowie mögliche Gattertypen bekannt sein. Die Topologie und Zwischenergebnisse dürfen während der Auswertung einzelnen Parteien, die sie nicht ggfs. schon vorher wissen, nicht offengelegt werden. Ein Programm P_F sollte durch eine Reihe von Basisoperationen definiert sein. Es können Variablen angelegt, geschrieben, und gelesen werden. Es darf die Schrittzahl, also wie viele Basisoperationen durchgeführt werden, bekannt werden. Die Anzahl der Variablen, einzelne Befehle oder verschiedene Zwischenergebnisse dürfen nicht bekannt werden, müssen also so obfisziert werden, sodass einzelne Parteien keinen (wahrscheinlicheren) Wert ermitteln könnten, das wird später noch formalisiert. Alle im Weiteren erwähnten und nicht näher spezifizierten Kryptosysteme müssen *CPA-sicher* sein. Es ist einfach zu sehen, dass diese Bedingung notwendig ist.

PFE-Framework

Mohassel et Sadeghian [36] beschreiben ein Framework, in dem PFE-Protokolle aus einzelnen Bausteinen zusammengesetzt werden können. Die bestehenden Arbeiten werden unter dieser semantischen Aufteilung betrachtet. Dadurch wird es ermöglicht einzelne Konstruktionen wesentlich besser aufgeteilt zu analysieren und somit kann zwischen verschiedenen Methodiken besser unterschieden werden. Es werden die folgenden zwei Funktionalitäten beschrieben. Die Verbindung dieser Funktionalitäten zu einer PFE-Funktionalität ist intuitiv und es wird daher nur auf [36] verwiesen

Circuit Topology Hiding (CTH) Die CTH-Funktionalität beschreibt ein Mapping π, sodass für eine Circuit Evaluation die Werte in einer versteckten Weise auf Schaltungen abgebildet werden. *Mohassel et Sadeghian* beschreiben diese Abbildung als *on-demand mapping*. Hierbei bringt eine

Partei dieses Mapping und einzelne Auswertungsschritte und andere Parteien bringen verschlüsselte Eingaben und Zwischenergebnisse ein. Ziel ist es, dass die erstere Partei das verschlüsselte Ergebnis der Berechnung erhält (Man bemerke, dass es trivial ist die Ergebnisse für eine beliebige der beiden Parteien beliebig zugänglich zu machen, siehe hierfür Abschnitt 3.3). Abbildung 3.2 zeigt wie eine beispielhafte Schaltung in eine Mapping übertragen werden kann. Diese Darstellung gemäß des Frameworks ist in der Literatur gängig [7, 8, 32, 37].

Private Gate Evaluation (PGE) Eine PGE-Funktionalität bezeichnet eine Funktionalität zur verschlüsselten Auswertung eines Gatters. Als Eingabe wird dieser Algorithmus mit zwei (oder beliebig aber bestimmt vielen) Werten gestartet und daraus entsteht als Ausgabe eine beliebige, aber bestimmte Anzahl an neuen Werten. Diese Werte werden durch eine CTH-Funktionalität geliefert und wieder entgegengenommen. Auch wenn bei Programmen nicht von Gattern gesprochen wird, ist die Definition von *Mohassel et Sadeghian* immer noch valide und kann auf Programme übertragen werden. Hierbei muss der Gattertyp genau dann geheim bleiben, falls es mehr als einen möglichen Gattertyp gibt.

3.3 Bestehende Protokolle

In diesem Kapitel werden Protokolle aus der Literatur betrachtet und die resultierenden Konstruktionen vergleichend dargestellt. Ich habe verschiedene Arbeiten ausgewählt, die neue Konzepte realisieren, besonders gut zu dem Konstruktionsziel passen, ein theoretisches Paradigma besonders gut anwenden, oder besonders effizient sind.
Ein erstes triviales PFE-Protokoll besteht rein aus einer Secure Multi-Party-Computation Komponente. Es ist dabei möglich das Problem PFE auf SFE zu reduzieren, indem Universal Circuits berechnet werden und eine Partei die Konfiguration der Schaltung als obfiszierten Input hält. Diese triviale Berechnung ist möglich, aber nicht effizient, und verfehlt einige der gegebenen Eigenschaften, sodass diese Methode schnell verworfen wurde und in der Literatur bessere Konstruktionen entwickelt wurden.

3.3.1 Secure Circuits - Abadi et Feigenbaum [1]

Abadi et Feigenbaum stellen hier ein erstes Protokoll vor um die Informationen der Funktion zu verstecken. Hierzu ähnelt ihr Schema konzeptionell stark dem ursprünglichen Schema von *Yao* [43]. Dieses Schema realisiert boolsche Schaltungen, was die Auswertung mittels diesem Protokoll ineffizient macht.
Es können zwei Arten von Gattern ausgewertet werden, einmal NOT-Gatter und einmal AND-Gatter. Die Auswertung basiert auf dem *Quadratic residue problem (QRP)* in Z_n^* mit $n = p * q$, wobei p und q prim, etwa gleichgroß, und kongruent 3 mod 4 sind. Wir betrachten ein u das ein quadratischer Rest modulo k ist und definieren $r(u, k)$ als 1 falls es ein a gibt sodass $a^2 \equiv u \ mod \ k$. Es ist hier einfach ein Bit b in einer anderen Zahl y zu kodieren, man kann einfach y so wählen, dass $r(y, k) = b$ gilt.
Nun kann die Partei P_1, die den Input hält, diesen kodieren indem sie k und pro Input Bit b_x eine entsprechende Zahl y_x ausrechnet. Die Partei P_0 kann mit diesen Zahlen dann die Schaltung wie folgt auswerten: Für ein NOT Gatter kann sie $-y$ errechnen, da -1 ein quadratischer Nicht-Rest darstellt, wird das Bit invertiert. Für ein AND Gatter maskiert P_0 y indem sie $E(y) = y \cdot b^2 \cdot (-1)^c \ mod \ n$

mit $b \overset{\$}{\leftarrow} Z_n^*$ und $c \overset{\$}{\leftarrow} \{0,1\}$ errechnet und sendet diesen Wert. Diese Maskierung kann später von P_0 mit $F(y) = y + c \; mod \; 2$ entfernt werden. P_1 kann nun die die codierten Bits effizient mittels p und q errechnen und eine Reihe von Bitkombinationen je nach Input berechnen und diese als neue $(y_i)_{1 \leq i \leq 4}$ zurücksenden.
So wird jedes Gatter ausgewertet. Je nach vereinbartem Output sendet entweder P_0 die maskierten Output Bits und P_1 errechnet das codierte Bit und sendet das zurück, und P_0 demaskiert die Output Bits um die richtigen Bits zu erhalten oder P_0 sendet die Bits demaskiert und P_1 errechnet sich daraus das Ergebnis.
In diesem Algorithmus sind CTH und PGE nicht getrennt. Dieses Protokoll hat nur passive Sicherheit in der oben definierten Notation, da beide Parteien zu jeder Zeit das Ergebnis abändern können, indem sie ausgewählte (auch maskierte) Bits bei der Gatterauswertung invertieren oder die Zahl, die ein Bit im quadratischen (Nicht-)Rest kodiert, mit -1 multipliziert. Es ist hier anzumerken, dass *Abadi et Feigenbaum* trotzdem in ihrer Betrachtung anmerken, dass aktive Sicherheit gegeben ist, wobei sie argumentieren, dass die Sicherheitsanforderungen durch die Auswirkungen des geheimen Inputs relaxiert sind. Da hier von einem korrekten Input ausgegangen wird ist die Sicherheitsauswertung hier anders. Auch können Fehler nicht erkannt werden und falls das Ergebnis geteilt werden soll, ist es nicht möglich Fehler zuzuordnen oder sicherzustellen, dass die Ergebnisse beider Parteien gleich sind.

3.3.2 Constant-Round PFE - Katz et Malka [24]

Die Konstruktionen von *Katz et Malka* zeichnen sich besonders durch einen reduzierten Kommunikationsaufwand und eine deutliche Steigerung der Praktikabilität aus. Es werden zwei Konstruktionen entwickelt, wovon eine aktive Sicherheit zumindest teilweise gewährleistet.
Anmerkung: Die Notation für diese Erklärung wurde aus der ursprünglichen Literatur angepasst. Da diese Zusammenfassung nur lückenhaft und konzeptuell ist, sollte für eine tiefer gehende Erklärung [24] konsultiert werden. Es sind P_0 und P_1 vertauscht um die Konsistenz innerhalb des Kapitels zu gewährleisten.
In diesem Schema stellen die Autoren erstmals *Wire Soldering* als Möglichkeit eine CTH-Funktionalität zu realisieren dar. Hierbei werden die Gatter topologisch sortiert und nach Outgoing und Ingoing Wires unterschieden. Jedes Gate hat zwei Incoming und ein Outgoing Wire. Dann werden die Gates von der Partei P_0, die f und den korrespondierenden Circuit C_f hält verbunden. Die entsprechenden Verfahren, werden je nach erklärter Version unten genauer erklärt.
Ein Gatter wird mittels einer Methode ähnlich zu Yao's Garbeled Circuits evaluiert. Hierbei werden je Outgoing Wire von P_1 berechnete Schlüssel für beide Bits asymmetrisch verschlüsselt und an P_0 gesendet. Hier ist dieser Schlüssel $Enc(s_j^i)$ für j Nummer des Outgoing Wire und i den Bitwert. P_0 kann nun zwei Variablen erzeugen um diesen Schlüssel zu maskieren, also a_j, b_j sodass $Enc(a_j \cdot s_j^i + b_j)$ gilt. Das ist mit gängigen homomorphen Kryptosystemen effizient berechenbar. Nun kann ein Garbeled Gate von P_0 als Reihe von verschlüsselten Schlüsseln, die jeweils maskierte Schlüssel für die beide Eingaben und jeweils für beide Bitwerte beinhalten und den unmaskierten Ergebnisschlüssel des Gates jeweils für beide Bitwerte, dargestellt werden. Also wird ein Gate i definiert als $< Enc(a_i \cdot s_j^0 + b_i), Enc(a_i \cdot s_j^1 + b_i), Enc(a_i' \cdot s_k^0 + b_i'), Enc(a_i' \cdot s_k^1 + b_i'), Enc(s_{l+i}^0), Enc(s_{l+i}^1) >$, wenn die Eingangkanäle j, k und der Ausgangskanal $l + i$ ist. P_0 schickt diese definierten Gates an Partei P_1, welche nun aus diesen Schlüsseln das Garbeled Gate berechnet. Dabei sind die Ciffren jeweils die sequentiell angewandte symetrische Verschlüsselung (mit dem zu den Eingangsbits korrespondierenden Schlüssel) der Ausgangsschlüssel $s_{l+i}^{NAND(b,c)}$ wobei b, c die Eingangsbits

sind. Nun kann P_0 mit einer Reihe von Informationen von P_1 das Ergebnis berechnen. P_0 erhält die fertigen Gates, sowie von jedem Eingabekanal den entsprechenden Schlüssel korrespondierend zu dem geheimen Eingabebit von P_1. Falls nun P_1 auch die Schlüssel für die Ausgabekanäle und das entsprechende Mapping sendet, kann P_0 das Ergebnis berechnen, anderenfalls kann P_1 das Ergebnis empfangen, indem P_0 den Ergebnisschlüssel sendet und P_1 das Mapping prüft.

Semi-Honest PFE Eine Implementierung des Protokolls mit passiver Sicherheit basiert direkt auf dem Schema wie oben beschrieben. P_1 erstellt je Gate zwei Schlüssel und schickt diese zu P_0. P_0 maskiert dann diese Schlüssel und bildet verschlüsselte Gates, genau wie oben definiert, tauscht diese ggfs. und sendet sie wieder zurück, mit dieser Soldering Technik ist die Topologie versteckt. P_1 schickt die zu den Input Bits korrespondierenden Schlüssel (je nach Output Delivery auch das Mapping für die Outputschlüssel und -bits). Damit kann P_0 den Schaltkreis auswerten. *Katz et Malka* liefern in [24] einen Simulationsbeweis um passive Sicherheit zu zeigen.

Malicious Adversary PFE Die Nutzung von homomorpher Kryptographie um die symmetrischen Schlüssel zu maskieren erlaubt nun mittels dem Einsatz verschiedener Zero-Knowlegde Proofs (ZKP) eine aktive Sicherheit zu ermöglichen. Es müssen folgende Eigenschaften bewiesen werden: P_1 beweist, dass der Public Key wohlgeformt ist. Des Weiteren müssen alle symmetrisch verschlüsselten Ciffretexte zu verschiedenen Klartexten evaluieren und wohlgeformt sein. P_1 kann auch bei Erhalt der *Garbeled Circuits* betrügen und P_0 eine andere Konstruktion schicken, daher ist hier ein Korrektheitsbeweis der Entschlüsselungen der Gatter nötig (Dabei ist dieser Schritt nicht in allen homomorphen Verschlüsselungen effizient umsetzbar). Als letztes muss P_1 zeigen, dass das letzte Mapping, falls die Ausgabe an P_0 gehen soll, korrekt ist. Hierbei ist zu beweisen, dass jeder Chiffretext C_i^x wohlgeformt und eine Verschlüsselung von s_i^x ist. Bei allen ZKPs reicht Honest-Verifier Security, da keine Sicherheitsgarantien gegeben werden müssen, falls beide Parteien kompromittiert sind.
Neben der Wire Soldering Methode ist dieses Protokoll besonders wegen der Nutzung von homomorpher Verschlüsselung in Kombination mit ZKPs um aktive Sicherheit zu erreichen erwähnenswert.

Holz et al. [22] haben die Konstruktion ausführlich evaluiert. Sie haben das Protokoll in vier Phasen aufgeteilt und eine Offline/Online-Trennung durchgeführt. Durch diese und andere Optimierungen, wie Pipelining oder Parallelisierung, haben sie gemessen, dass je nach Setup, Kryptosystem, und gemessenem Parameter die optimierte Konstruktion um einige Größenordnungen ($\leq 10^4$) besser ist im Vergleich gegenüber den zu diesem Zeitpunkt effizientesten Universal Circuit Lösungen. Dadurch wurde diese Konstruktion zu damaligen Zeitpunkt sehr interessant und ein Großteil der folgenden Arbeit basiert darauf die hier angewandten Methodiken zu verbessern. Dieses PFE Schema ist das erste mit einer konstanten Rundenanzahl und einer linearen Komplexität in Kommunikation.

3.3.3 How to Hide Circuits in MPC - Mohassel et Sadeghian [36]

In der Arbeit von *Mohassel et Sadeghian*, die auch das Framework aus Abschnitt 3.2 umfasst, stellen die Autoren drei verschiedene Versionen eines PFE-Systems dar.
Um die CTH-Funktionalität zu realisieren betrachten die Autoren zuerst verschiedene Konzepte von Oblivious Extended Permutation (OEP).
Die Durchführung einer solchen Permutation, kann auch „blinden “genannt werden. Man sagt P_0 blinded die Werte von P_1, indem das zufällige Mapping angewandt wird. Hierbei ist es gängig entweder *Yaos Garbeled Circuits* oder homomorphe Verschlüsselung zu verwenden, wobei im ersteren Fall ein XOR mit geheimen zufälligen Eingabebits von P_0 stattfindet und bei dem zweiten

Fall ein zufälliger Wert homomorph aufaddiert wird. Es ist einfach zu sehen, dass beides ein zufälliges, aber eindeutiges replizierbares Mapping darstellt. P_1 erhält die Werte nachdem die zufällige aber bestimmte Permutation angewandt wurde. Bei der späteren Auswertung wird dieses Mapping genutzt um die Werte entlang der Gates zu propagieren.
Die Autoren schlagen hingegen auch ein neues Protokoll vor, das Switching Networks wie in Abschnitt 2.4.2 definiert nutzt. Zuerst werden die realen Inputs mit einer Reihe zufälliger Inputs (=Dummys) ergänzt und mittels einem Switching Netzwerk ersetzt. Dann wird für jede Position der Dummy Wert gegen einen realen Wert mit Vorrang zu ersetzt, um diesen dann erneut in einem Switching Network zu sortieren. Ein Switch kann hier mit dem XOR mit einem vorher zufällig berechneten Wert realisert werden, es ist keine Sortierung notwendig. Es kann Oblivious Transfer zur Implementierung dieser Switches genutzt werden.
Arithmetische Gatter werden realisiert, indem P_1 beide Input Shares, sowie das Produkt beider Shares verschlüsselt an P_0 schickt. Dieser rechnet auf diese Shares homomorph seine Shares auf und sendet das verschlüsselte Ergebnis zurück. Damit ist die PGE-Funktionalität komplett.

Das zweite vorgeschlagene Protokoll realisiert die Auswertung von boolschen Schaltkreisen.
Hierbei ist die CTH-Funktionalität darauf ausgelegt pro Wire die Position von zwei Möglichen Schlüsseln zu verdecken. Dafür werden zwei Switching Networks genutzt, die jeweils die Schlüssel aller Gatter für eines der beide Bits (korrespondierend zu der Yao-Gatterauswertungsmethodik) mittels einer OEP permutieren. Ein drittes Switching Network (kein vollwertiges Netzwerk) schaltet dann gegeben eines zufälligen Selektor-Bits die Schlüssel um. Damit wird die zufällige Vertauschung der Schlüssel in Yaos Protokoll gewährleistet. Die Schlüssel sind geblinded und werden erst zur Evaluation des Ergebnisses wieder entblinded.
Ein Gatter wird mithilfe eines (leicht adaptierten) Protokolls von Yao [43] realisiert, wobei an dieser Stelle nicht weiter darauf eingegangen wird, da eine erneute Betrachtung redundant wäre.

Insbesondere wegen der Auftrennung von CTH und PGE im Falle der arithmetischen Schaltkreise ist diese Konstruktion in dieser Arbeit besonders interessant. Auch ist eine genauere Betrachtung der OEPs zur Anwendung im allgemeineren Fall für Switching Networks mittels Oblivious Transfer im MPC Kontext auch für spätere Konstruktionen von PFE mittels ORAM relevant.

3.3.4 Actively Secure PFE - Mohassel et al. [37]

Dieses Schema basiert größtenteils auf dem vorherigen, wurde allerdings um verschiedene Funktionalitäten zur Sicherung von aktiver Sicherheit ergänzt.
Das neue Protokoll ist wesentlich strikter in eine Offline und eine Online Funktionalität aufgeteilt. Hierbei wird deutlich mehr aus der Online in die Offline-Funktionalität ausgelagert, was effektiv eine Effizienzsteigerung zur Folge hat, falls man die Online Phasen separat betrachtet. Dennoch sind beide Phase separat linear realisiert, die Unterscheidung findet sich besonders im Kommunikationsaufwand, der bei der Online Phase reduziert wurde.
In der **Offline Phase** werden zufällige Vektoren generiert die als One-Time-Pad Schlüssel genutzt werden.
Es werden zu jedem Input Wire i zwei Zahlen r_i, s_i und zu jedem Output Wire j zwei Zahlen l_j und t_j generiert. Dann werden die Input Wires über eine Permutation π mit den Output wires über das geöffnete und bekannte Tupel $([r_i] - [l_{\pi(i)}], ([s_i] - [t_{\pi(i)}]) + [r_i] - [l_{\pi(i)}] \cdot [K])$ für ein zufällig bestimmtes geheimes K verbunden. π ist hier das Mapping der Output Wires auf Input Wires, wobei Eingabedaten auch als Output Wires modelliert werden. Bei $[r_i] - [l_{\pi(i)}] \cdot [K]$ handelt es

sich um einen One-Time-MAC mit Schlüssel K. Die Macs werden erst am Ende der Online Phase geöffnet, daher ist der Schlüssel für die Evaluation noch sicher. Partei P_0 teilt für jedes Gatter die Funktionalität (Addition oder Multiplikation) mittels eines verschlüsselten Bits mit. In der Online Phase werten die Parteien (hier nur Partei P_1) den Schaltkreis aus). Dafür ist eine Vorbereitung der Input Werte notwendig. Die Parteien addieren auf ihren Input $[x_i]$ den vorbereiteten Wert $[l_i]$ auf, wobei die Eingabe auf dem i-ten Kabel erfolgt. Dann evaluieren sie gemeinsam die einzelnen Gatter wie folgt: Zuerst werden für zwei Bits, die den Eingangskanal darstellen, jeweils das vorberechnete $x_{\pi(i)}$ aufaddiert und bei dem zweiten Kanal noch zusätzlich $s_{i_j} + (x_{\pi(i)} + r_{i_j}) * K$ berechnet und an Partei P_1 geschickt. Hier wird deutlich, dass die vorberechneten Werte eine Maskierung der eigentlichen Werte darstellen, die in diesem Schritt eliminiert wird. P_1 überprüft die korrekte Anwendung dieser Maske und brechen ggfs. ab. Ansonsten kann das Gatter evaluiert werden mit der Funktion $[z]_{j+n} = (1 - [G_i]) \cdot [y_{i_0}] \cdot [y_{i_1}] + [G_i] \cdot [y_{i_0}] \cdot [y_{i_1}]$. Dannach muss P_1 wieder die Ausgabe als Eingabe des nächsten Gatters vorbereiten Indem $[u_j] = [z_j] + [l_j]$ und $[v_j] = [t_j] + u_j \cdot [K] = [t_j + (z_j + l_j) \cdot K]$ ausgerechnet und offengelegt werden. Hier wird deutlich, dass die Online-Phase besonders gut für eine generelle MPC Anwendungen mit einer Partei P_1, die verteilt verschlüsselt über mehrere Clients realisiert ist, optimiert ist.
Die Realisierung der CTH über Extended Permutations kann mit ZKPs aktiv sicher implementiert werden. Dafür haben *Mohassel et al.* einen Proof of Extended Permutation entwickelt.
Die Konstruktion ist für meine Betrachtunge wegen dem geringen Overhead im Allgemeinen MPC Fall und der aktiven Sicherheit besonders interessant.

3.3.5 2-Party PFE - Bingöl et al. [7]

Die von *Bingöl et al.* entwickelte Konstruktion, basierend auf Half-Gates ist aktueller Stand der Technik für boolsche Schaltungsevaluationen.
Das Schema nutzt eine optimierte Version von [36] zur CTH-Funktionalität, also ein OEP-Protokoll, dass durch 1-out-of-4 Oblivious Transfer realisiert wird. Hier ist es besonders anzumerken, dass durch die Konstruktion der Half-Gates in der Offline-Phase der Kommunikationsaufwand bei der Auswertung der Garbeled Gates minimiert wird.
Die Funktionalität eines Gatters wird über die Funktionaliät f zweier Halbgatter definiert. Die folgende Notation ist direkt aus [7] übernommen. Hier steht σ_x^y für die Inputs auf Eingabekanal $x \in \{a, b\}$ mit Wert $y \in \{0, 1\}$ und w_z^y für die Ausgabe des Gatters z. R ist ein global gesetzter zufälliger Bitstring ungleich Null, der eine Relation zwischen Null und Eins darstellt. Es ist $w^1 = w^0 \oplus R$. Der Bitstring startet mit least-significant-bit 1 um eine Gleichheit der Schlüssel für 0 und 1 auszuschließen.
Ein Garbler-Gate wird definiert über die Funktionalität $f_G(v_a, p_b) = (\alpha_1 \oplus v_a)(\alpha_2 \oplus p_b) \oplus \alpha_3$ und ein Evaluator Gate wird definiert als$f_E(v_a, v_b \oplus p_b) = (\alpha_1 \oplus v_a)(p_b \oplus v_B) \oplus \alpha_3$. In der Vorbereitung rechnet P_0 jeweils

$$T_{GC} = H(\sigma_a^0) \oplus H(\sigma_a^1) \oplus (p_b \oplus \alpha_2)R \qquad T_{EC} = H(\sigma_b^0) \oplus H(\sigma_b^1) \oplus \sigma_a^{\alpha_2} R \qquad \psi = w_{GC}^0 \oplus w_{EC}^0 \oplus w_C^0$$

mit $w_{GC}^0 = H(\sigma_a^{p_a}) \oplus f_G(v_a, p_b)R$ und $w_{EC}^0 = H(\sigma_b^{p_b})$ aus und sendet alle drei Werte zu P_1. T_{GC} und T_{EC} stellen die halben Gatter dar.
Nun kann man sehen, dass eine Partei P_1 von einem Output w_x zu einem Output w_{1-x} nur dann kommt wenn sie R kennt, was sie während der Ausführung nicht hat. Die Eingabe σ erhält P_1 durch Oblivious Transfer. Eine Auswertung der gegebenen Gatter kann nun mittels der CTH-Funktionalität einfach konstruiert werden.

Dieses Protokoll fokussiert sich besonders darauf den Kommunikationsaufwand zu reduzieren und die Anzahl der Ver-/Entschlüsselungen zu minimieren, daher ist es eines der effizientesten Protokolle mit passiven Sicherheitsgarantien und nicht wiederverwendbarer Offline-Phase.

3.3.6 Re-Executable PFE - Biçer et al. [8]

Biçer et al. stellen hier ein wiederverwendbares PFE-System mit einer konstanten Rundenzahl und linearer Komplexität vor. Es zeichnet sich durch einen besonders niedrigen Kommunikationsaufwand (Reduktion um mehr als 90%) gegenüber der zuvor betrachteten Protokollen aus.
Die Wiederverwendbarkeit wird durch zwei unterschiedliche Protokolle gewährleistet. Bei der initialen Ausführung generiert P_1 eine Reihe von von Generatoren $\mathcal{P} = (p_i)_{1 \leq i \leq n}$ für einen Circuit C_f mit n Gattern. P_0 blinded und verbindet standartmäßig die Generatoren (Garbling) in ein Tupel $\mathcal{Q} = (q_i)_{1 \leq i \leq n} = (t_i \cdot p_{\pi_f^{-1}})_{1 \leq i \leq n}$, wobei t_i hierbei die Elemente zum Blinden und π die OEP darstellen. Nun kann man $(\mathcal{P}, \mathcal{Q})$ als Template zur Funktionsauswertung behalten. Da in dieser Evaluation nur ein Gattertyp betrachtet wird, ist diese Information öffentlich und P_1 kann den garbeled Circuit aus den garbeled Gates selber, ohne zusätzliche Information generieren. Es wird sowohl $\mathcal{P}$, als auch $\mathcal{Q}$ mit zufälligen Gruppenelementen $\alpha_\cdot$ zur Evaluation multipliziert um $\mathcal{W}_i^j = \alpha_j \cdot p_i$ und $\mathcal{V}_i^j = \alpha_j \cdot q_i$ mit $1 \leq i \leq n$ und $j \leq \{0, 1\}$. Dazu werden klassische Garbling-Techniken wie schon in den zuvor betrachteten Konstruktionen verwendet. P_0 kann mit diesem garbeled Circuit und den verschlüsselten Eingaben das Ergebnis berechnen und ggfs. mittels Oblivious Transfer P_1 mitteilen. Hier wird eine Dual-Key Ciffre mit $Enc_{k_1,k_2}(m) = [H(k_1, k_2, gateID)]_l \oplus m$ (die Notation $[\cdot]_l$ bezeichnet in diesem Falle nur die ersten l Bits) verwendet und die resultierenden Chiffretexte als Garbeled Gate zusammenzufassen.
Zur Wiederverwendung werden neue Gruppenelemente α_i gesucht und die Werte abhängig von α neu gewählt.
Diese Konstruktion ist besonders effizient in der Online Phase. Sie ist linear und benötigt 3 Runden. Der Kommunikationsaufwand ist erheblich reduziert und P_1 kann in Kombination von mehreren Parteien effizient und sicher ausgeführt werden. Nachteilig kann man besonders hervorheben, dass nur boolsche Gatter unterstützt werden, sonst wäre auch eine Auswertung in konstanter Rundenzahl nicht möglich. Das Protokoll ist nur gegen das passive Angreiffermodell sicher.

3.3.7 Making PFE Safer, Faster, and Simpler - Liu et al. [32]

Liu et al. stellen hier zwei Konstruktionen vor. Beide sind *reuseable* und linear in der Gatteranzahl. Ein Protokoll stellt eine aktiv sichere PFE-Ausführung, allerdings mit großem Overhead durch Zero-Knowlegde Proofs, basierend auf einem multiplikativ homomorphen Kryptosystem (z.B. El-Gamal [13]) vor. Das andere Protokoll ist nur sicher in der *PVC*-Definition (*publicly verifiable covert secure*), dafür allerdings ist es deutlich praktischer in der Ausführung.
Betrachtung des aktiv sicheren Protokoll Dieses Protokoll teilt sich in eine wiederverwendbare Offline- und eine Online-Phase. Es hält P_0 die Schaltung C_f und Input x_0 (die in meiner Definition auch fest eingestetzte Werte der Funktion sein können), und P_1 den Input x_1. Als Pre-Agreement Parameter haben sich beide auf eine zyklische Gruppe $\mathcal{G} =< g >$ von Ordnung p geeinigt. In der Offline Phase wird P_0 mittels einer Extended Permutation die Wires verbinden. Danach wird P_1 wird die dazugehörigen Gates garbeln, sodass P_0 den Schaltkreis auswerten kann. Beide Parteien kontrollieren ihre verschlüsselten Berechnungen mittels ZKPs gegenseitig.

Detaillierter wählt P_1 zuerst Elemente $G = (g_i)_{1 \leq i \leq n+\theta-m}$ (für n Input Wires, θ Gates, und m Output Wires) aus $\mathcal{G}$ und sendet sie zu P_0, diese stellen die Wires zum verbinden dar. Nun wendet P_0 die EP π_f auf G an und verschlüsselt die permutierte Liste (Mittels ElGamal-Verschlüsselung). Es entsteht eine neue Liste $G' = (Enc(g_{\pi(i)})^{(t_{\pi(i)})})_{1 \leq i \leq n+\theta-m}$, wobei $(t_i)_{1 \leq i \leq n+\theta-m}$ Blinding Werte sind. Diese sendet P_0 zurück und hilft bei der Entschlüsselung der einzelnen Werte, sodass P_1 letztendlich $(g_{\pi(i)}^{t_{\pi(i)}})_{1 \leq i \leq n+\theta-m}$ erhält, ohne daraus Eigenschaften von π zu erhalten. Alle bis hier berechneten Werte können als Template gespeichert werden. Nun kann P_1 wie im Algorithmus von *Biçer et al.* in [8] Garbeled Gates definieren und diese zurückschicken. Mittels dieser Garbeled Gates, π_f, $(t_i)_{1 \leq i \leq n+\theta-m}$, G, und den Input Wire Labels von P_1 definiert als g^{α_i} für $i \in \{0, 1\}$ (analog zur Beschreibung von [8]) kann nun einen Evaluationsalgorithmus starten (wie im vorherigen Abschnitt beschrieben).
Es müssen die Berechnung der verschlüsselten Extended Permuation, das Blinding (mittels Proof of Knowledge) und die korrekte Wahl der α_is durch Zero-Knowledge Beweise abgesichert sein, sodass das Protokoll aktiv sicher ist.
Betrachtung des PVC sicheren Protokolls Um verdeckte Sicherheit zu erhalten, wird die Online Phase des aktiv sicheren Protokolls abgeändert. Parteien P_0 und P_1 erstellen hier nicht nur eine sondern λ verschiedene Auswertungen, wobei alle Auswertungen auf einem der Gegenpartei unbekannten Seed beruhen. Die Parteien committen sich gegenüber der anderen Partei auf diese Seeds. Nun können $\lambda - 1$ Auswertungen geöffnet werden, wobei die Ergebnisse nicht entschlüsselt werden, sodass Privacy nicht verletzt wird. Es kann nun die Gegenpartei kontrollieren, dass alle Schritte einzeln richtig durchgeführt wurden, wobei die Auswertung so geöffnet wird, dass die einzelnen Schritte nicht zusammengesetzt werden können. Erkennt eine Partei hier eine Manipulation, kann sie diese zusammen mit einem Transkript der signierten Kommunikation (insbesondere des Commitments) veröffentlichen. Auf die Techniken zur Öffnung der Auswertung wird hier nicht weiter eingegangen, da sie für diese Arbeit nicht wichtig sind.
Trozdem ist das Protokoll interessant zu betrachten, um eine Alternative zur aktiven Sicherheit mittels ZKPs (was i.A. sehr Runden und kommunikationsintensiv ist) zu illustrieren, falls die Sicherheitsbedingungen in der Praxis relaxierter betrachtet werden.

3.3.8 Garbeled ORAM - Lu et Ostrovsky [35]

Die Bezeichnung von garbeled RAM-Programmen wurde erstmals von *Lu et Ostrovsky* [35] benutzt. Ziel dieser Arbeit ist es einen ORAM so zu entwickeln, dass eine arithmetische Version von Yao's Garbeled Circuits (siehe Abschnitt 2.2.2) ausführbar sind. Dadurch soll es möglich werden, anstatt jede Funktion $f(\cdot)$ zur PFE-Evaluation in eine Circuit-Form umzuwandeln und diese auszuführen, einzelne Garbeled RAM Programe zu kompilieren, die einen einfachen Garbeled Circuit implementieren und diesen über eine universelle Programmausführung mit ORAM-Speichermethoden zu ergänzen und so auch komplexere Kontrollflussstrukturen (Bedingungen, Rekursion, Funktionen ...) über einen einfachen Programmcounter zu realisieren.
Das Konstrukt von *Lu et Ostrovsky* ist dabei linear in den Zeitschritten eines äquivalenten Programm, dass nicht oblivious geschrieben ist und polylogarithmisch in der Größe des Programms.
In dieser Technik werden Speicherpositionen mit Encodings versehen, die deterministisch, aber abhängig vom Programmschritt und eines geheim bestimmten vorberechneten Zufallswert, sind. Diese Encodings werden nur geheim als *Garbeled Locations* vom ORAM ausgewertet. Eine Programmevaluation geschieht in zwei Hauptschritten, davor werden die Zustandsdaten des Programms (z.B. Programm Counter) und Inputs initialisiert (in Circuit Form kompiliert). Dann

wird für jeden Befehl zuerst eine Read/Write-Abfrage der Daten an der entsprechend im Befehl codierten Garbeled Location durchgeführt (1) und diese Daten auf einem mit dem Befehl codierten Garbeled Instruction Circuit ausgewertet und eine neue Read/Write Anfrage mit Position (und ggfs. Daten) erlangt (2). Die Auswertung der einzelnen Daten erfolgt hier wie ursprünglich in [43] vorgestellt.

Dieses Schema hat den Vorteil, dass der Kommunikationsoverhead minimal ist. Es werden insgesamt also zwei Nachrichten benötigt um jeden möglichen Wert zu berechnen. Auch unterstützt diese Konstruktion mit dem Speicherzugriff über ORAM anstatt auf Basis von Permutationen eine generellere Programmstruktur, was PFE ausdrucksstärker und praktischer macht. Dennoch sieht diese Konstruktion eine Auswertung mittels Yao's Garbeled Circuits, anstatt von arithmetischen Gattern vor, was einen Gatter Blow-Up in den Funktionen erzeugt und insbesondere bei einem (poly-)logarithmischem Overhead für Read-/Write Zugriffe die Auswertung von großen Funktionen (in Gatteranzahl und Wertebereich) schnell ineffizient werden lässt. Dieses Schema stellt trotzdem ein solides Modell für die Berechnung von Privaten Funktionen mittels ORAM dar.

Weiterführende Literatur zu Garbeled ORAM

Als Erweiterung dieser Arbeit wurden in dieser Funktion die Pseudo-Random-Function basierte Auswertung von *Gentry et al.* [16] durch andere Primitive ersetzt, sodass die Konstruktion simpler und effizienter wird. Hierbei wird speziell an der Auswertung der boolschen Befehlsauswertungen gearbeitet. Auch sie betrachten keine arithmetischen Programme. Eine zweite direkte Erweiterung liefern *Lu et Ostrovsky* selbst in [34]. Dabei gelang es ihnen ein Garbeled RAM Schema zu konstruieren, dass durch das Minimum der Ausführungszeit oder des polynomiellen Speicherbedarfs begrenzt ist. Dafür verwenden sie wieder pseudo-zufällige Funktionen und einen Algorithmus, der die Ausführung der Programms in einer Liste als Aufzählung von Schlüsseln gespeichert, die je nach Nutzung "revoked" werden können. Dabei wird der Schlüssel bei mehrfacher Nutzung durch einen anderen Schlüssel aus einer assoziierten Menge ersetzt. Dieser Zugriff wird als Oblivious Read auf eine verschlüsselte Position charakterisiert.

Es exisistiert bereits eine aktiv sichere Implementation dieses Garbeled RAM Schemas. Auch wenn diese Ideen einen theoretischen Vorschub in der Fähigkeit private Funktionen effizient auszuwerten aufweisen, ist die praktische Resonanz und der Effizienzgewinn niedrig, sodass nach diesen zwei Erweiterungen wenig Literatur diese Schemata aufgreifen und weiterführen.

3.3.9 Multiparty Computation for RAM - Keller et Yanai [28]

Eine eher praktische Betrachtung der Schemata liefern *Keller et Yanai* [28]. Hierbei implementieren sie ein allgemeines Protokoll zur Berechnung von Programmen, das wie schon in der Arbeit von *Lu et Ostrovsky* beschrieben oblivious ist. Dabei wird ein *CPU-Step-Circuit* definiert, der nach jedem Schritt einen kombinierten Read/Write-Aufruf durchführt. Die Autoren haben hier besonderes Augenmerk auf Rundenkomplexität und Speicherverbrauch gelegt. Es ist ihnen gelungen eine im CPU-Schritt konstante Rundenkomplexität und linearen Speicherverbrauch zu erreichen, sie betonen aber gleichzeitig, dass die Beschränkungen durch die Verwendung von ORAM in der Anzahl der interaktiven Runden sowie des Speicherverbrauchs immer noch schwerwiegend sind. Hier ist die Verbesserung des Protokolls klar nach unten durch die Effizienz des ORAMs beschränkt.
Es wird ein Schritt über das sogennante BMR-SPDZ Garbling Scheme realisiert. An dieser Stelle

ist darauf hinzuweisen, dass das Garbling Scheme hier nicht dem Zweck dient die Topologie des Programmes zu verstecken, sondern eine Multi-Party-Computation realisiert, bei der jede Partei sowohl Garbler als auch Evaluator ist. Dabei werden immer noch aktiv sichere SPDZ-Shares verwendet um verschlüsselte Schlüssel zur Evaluation einzelner Gatter zu benutzen. Die Evaluation einzelner Gatter passiert aber durch eine BMR-Funktionalität. Diese Technik erlaubt es die einzelnen Schritte vorzubereiten, wodurch die Rundenzahl maßgeblich reduziert wird.
Zur Realisierung der CPU-Step Funktionalität werden drei verschiedene Methoden dargestellt. Eine Möglichkeit besteht in der Nutzung von "Embedded Subcircuits", wobei ein CPU-Step-Circuit um zwei Subprotokoll-Aufrufe erweitert wird, bei denen ein gelesener Wert überprüft und ein neues authentifiziertes zu schreibendes Bit (in Form einer Share) erzeugt. Eine weitere Möglichkeit bietet das schon in vorherigen Konstruktionen betrachtete Wire Soldering. Auch hier wird die Wire Soldering Methode nicht versteckt durchgeführt, sondern kann von allen Parteien berechnet werden, da die Funktion zur Errechnung des nächsten Indexwertes auf Basis der verschlüsselten Befehls bekannt ist. Es unterscheiden sich dennoch die einzelnen Schlüssel, da der Wert je Secret Sharing unterschiedlich ist. Hier ist also auch zusätzliche Kommunikation notwendig, welche aber gegenüber dem Resharing in der Subcircuit-Methode deutlich reduziert ist.
Die dritte Methode kombiniert Vorteile der beiden Vorgänger. Es wird eine Konversion zwischen den Share-Values der Wires und den Keys in einem Garbling Scheme eingeführt. Die Parteien erhalten die Permutation-Bits der Positionen eines ORAMs abhängig vom Zeitschritt in der Garbling Prozedur. Danach können sie aus dem Ergebnis des letzten CPU-Schritts nicht nur den zu speichernden Wert, sondern auch einen Schlüssel mit dem sie den Wert des Wires aufdecken und dorthin in diesem Schritt schreiben. Zum Lesen der Schlüssel kann jede Partei den resultierenden Schlüssel für den richtigen Wert (0 oder 1) veröffentlichen. Daraufhin können die anderen Parteien überprüfen, dass der Schlüssel richtig veröffentlicht wurde und damit dann den permutierten Read Index i offenlegen.

Keller et Yanai stellen die erste konkrete Implementierung eines Garbeled RAMs vor, die aktiv sicher ist und auf boolschen Schaltungen basiert. Dabei ist zu sehen, dass der BMR-SPDZ-Garbling in Kombination mit Circuit-ORAM in WAN am besten skaliert und eine rein auf SPDZ-Shares und Path-ORAM basierende Konstruktion hingegen im lokalen Netzwerk effizienter ist.

3.3.10 Vergleich

In diesem Kapitel wurden elf verschiedene Konstruktionen aus acht Arbeiten betrachtet, die alle jeweils eine neue Komponente in das Felder der privaten Funktionsauswertung gebracht haben.
Dabei haben *Abadi et Feigenbaum* als erstes ein Schema vorgestellt, was nicht die Berechnung durch Universelle Circuits bestimmter Größe als Funktion vorsieht. Ihr Schema ist linear in der Anzahl der Runden und dementsprechend auch in der Komplexität. Das Protokoll ist nur passiv sicher und die Berechnung von CTH und PGE nicht getrennt, was es schwer macht, die Effizienz durch Änderung einzelner Komponenten oder in der Sicherheitsnotation zu verstärken. Den nächsten großen Schritt stellt die Konstruktion von *Katz et Malka* dar, da sie als CTH Methode zum ersten mal verschlüsselte Permuationen einführt und linear in Komplexität und konstant in der Anzahl der Interaktionsrunden ist. Dadurch werden PFE Konstruktionen um mehrere Größenordnung praktikabler.
Mohassel et Sadeghian erweitern den Forschungszweig hierbei erneut massiv durch die erste Realisierung von arithmetischen Schaltungen, die ersten nach aktuellen Maßgaben aktiv sicheren Konstruktionen (mit zusätzlich ersten formalen Definitionen der Sicherheit), und besonders ihrem Framework zur Entwicklung und Definition von PFE-Konstruktionen. Auch mit Betrachtung der folgenden Entwicklungen, sind die Konstruktionen aus [36, 37] durchaus vergleichbar in

Kommunikation und Effizienz.
Daraufhin entstanden viele weitere Konstruktionen die großteils jeweils darauf abzielen einzelne Eigenschaften zu optimieren. Die Definitionen und Entwicklungen von *Mohassel et Sadeghian* sind hier zum Standard in der Literatur geworden.
Eine nächste Erweiterung skizzieren *Lu et Ostrovsky* mit ihrer Entwicklung von sogenannten *GRAM-Programmen*. Hier werden nicht mehr einzelne Schaltungen, sondern auch Funktionen, die einen komplexen Programmfluss erfordern, ausgewertet. Dabei bleiben die Autoren eher unkonkrete bezüglich der exakten Realisierung, skizzieren aber eine solche Funktionalität umfassend. Hierbei ist klar, dass eine solche Erweiterung der Funktionalität nicht ohne zusätzlichen Aufwand realisierbar ist.
An diese Forschung wird in dieser Arbeit weitergehend angeknüpft. Es soll festgestellt werden, inwiefern ein ORAM nutzbar gemacht werden kann um PFE in einem arithmetischen Setting zu realisieren und ob ein solches Programm praktikabel sein kann. Die Betrachtung der Forschung in diesem Bereich dient dazu sinnvolle Maßgaben und Eigenschaften zu finden, die ein solches GRAM-Programm haben sollte.

Tabelle 3.1: Vergleich der Eigenschaften der betrachteten PFE-Konstruktionen

Konstruktion		CTH	PGE	Circuit Art	Sicherheit	Zusätze	Sonstiges
Abadi et Feigenbaum		Universal Circuits/QRP	QRP	boolesch	Passiv[1]		
Katz et Malka		Wire Soldering/EP	HE	boolesch	Passiv [2]	ZKPs	linear/runden-konstant
Mohassel et Sadeghian	1	OEP via OSN/OT	HE	arithmetisch	Passiv		Wenig Kommunikation
	2	XOR Blinding	Yao based	boolesch	Passiv		Nur 2PC mögl.
Mohassel, Sadeghian, & Smart		OEP via obl. Shuffling	HE	arithmetisch	Aktiv	ZKPs	
Bingöl, Bicer, et al.	1	OEP via OT	Haf-Gates	boolesch	Passiv		
	2	OEP	Yao based	boolesch	Passiv		Wiederverwendbarkeit
Liu, Wang, et al.	1	OEP	Yao based	boolesch	Aktiv	ZKPs	Wiederverwendbarkeit
	2	OEP	Yao based	boolesch	PVC	Gate opening	Wiederverwendbarkeit
Lu et Ostrovsky		ORAM	Yao based	boolesch	Passiv		
Keller et Yanai		ORAM[3]	BMR-SPDZ	boolesch	Aktiv		
Eigene Konstruktion		ORAM	HE	arithmetisch	Aktiv[4]		

[1] Eigene Definition abweichend
[2] Aktiv Sicher gegen einen Client-Adversary
[3] Realisierung mittels Sub-Circuits, Wire Solderung, und kombinierter Methoden
[4] Unter den Bedingungen wie in Abschnitt 3.4.6 genauer beschrieben

3.4 Entwicklung

Im diesem Kapitel wird ein eigenes PFE-Schema entwickelt. Dafür wird zunächst definiert welche Ziele diese Konstruktion erzielen soll und welche Eigenschaften sie besitzen soll. Danach wird ein valides PFE-Protokoll vorgestellt und dessen Eigenschaften betrachtet. Hierfür muss zuerst definiert werden was eine PFE-Funktionalität darstellt, da das bis hier hin nicht allgemeingültig gemacht wurde. Es war bisher nicht sinnvoll eine formalere Definition anzubringen, da diese in der Literatur eher heterogen beschrieben wird. Allerdings wird im Folgenden eine klare Definition benötigt, an die sich die Zielsetzung richten kann.

Definition eines PFE-Systems

Diese Definition greift nun die bisherigen Beschreibungen eines PFE-Systems auf und formalisiert diese nochmals um so später eine genauere Betrachtung der Algorithmen zu ermöglichen.
Ein Protokoll Π realisiert eine PFE-Funktionalität, wenn Parteien $P_0, ..., P_n$ so existieren, sodass P_0 ein Programm eingibt, dass eine Funktion $\mathcal{F}$ berechnet und Parteien $P_1, ..., P_n$ Eingaben $x_1, ..., x_n$ hinzugeben und dieses Protokoll dann für Partei P_i die Ausgabe $\mathcal{F}_i(x_1, ..., x_n)$ berechnet. Die Formatierung der Eingaben und die Formatierung der Ausgaben bekannt. Die Befehle dürfen auf eine im Pre-Agreement festgelegte Menge an Operatorentypen zugreifen. Des Weiteren wird ein Sicherheitsparameter κ festgelegt. Es ist P_0 nicht erlaubt eine triviale oder manipulierte Funktion $\tilde{\mathcal{F}}$ einzugeben, da es sonst einfach zu sehen ist, dass kein sicheres PFE-System existieren könnte (in der Praxis muss diese Eigenschaft außerhalb des Protokolls zugesichert sein). Es muss in einem Pre-Agreement kommuniziert werden, inwiefern die Parteien die Anzahl der Evaluationsschritte ggfs. begrenzen wollen, sodass in der Laufzeit eines Programmes nicht zwingendermaßen Informationen über die Wertigkeit der Eingabe verraten kann (z. B. kann eine konstante Laufzeitangabe durch P_0 erfolgen, sodass nach dieser Zeit getestet wird ob das Programm terminiert ist, andernfalls wird abgebrochen, mehr dazu später in diesem Kapitel).

3.4.1 Zielsetzung

Das zu entwickelnde PFE-Schema soll eine Funktion $\mathcal{F}$ in Assembler-Style Notation erhalten um diese durch das Programm $\mathbb{P}_{\mathcal{F}}$ zu berechnen. Dabei soll ein ORAM zur Realisierung einer Structure Hiding-Funktionalität genutzt werden um einzelne Befehle, hier als atomare Berechnungen definiert, zu verbinden. Dabei können einzelne Variablen, hier Werte die in den Blöcken der RAM abgespeichert werden, beliebig oft nacheinander wiederverwendet werden. Es sollte auch später möglich sein einen Kontrollfluss in die Befehlsstruktur einzubauen.
Hierzu wird im folgenden Abschnitt ein *State-Machine-Modell* vorgeschlagen, das zur privaten Evaluation genutzt werden kann. Unter diesen Vorraussetzungen kann man schon folgende Beobachtungen im Vergleich zu gängigen Methoden zur Private Function Evaluation mittels Schaltungsbeschreibungen treffen:

- Da *Obliviousness* garantiert werden soll, müssen in jedem Evaluationsschritt Werte abgerufen und ggfs. ein Ergebnis geschrieben werden. Damit muss die ORAM mindestens linear in der Anzahl der Befehle sein, solange keine weiteren Einschränkungen erhoben werden. Im Gegensatz zu Konstruktionen aus der Literatur können wir also keine konstante Komplexität

pro Operation erreichen, da die Kosten pro Variablenzugriff mit $O(log^p(n))$ begrenzt sind (n ist die Anzahl der Variablen und p konstant) und eine mindestens lineare Anzahl an Lesezugriffen in der Anzahl der reellen Zeitschritte gegeben sein muss.

- Da ein ORAM als Structure Hiding Komponente genutzt werden soll, müssen bei Betrachtung aller State-Of-The-Art Konstruktionen mindestens eine in der Schrittzahl lineare Anzahl an Runden der Interaktion stattfinden, da der Client immer eine zufällige Permutation oder Tagging Funktion realisieren muss und das Ergebnis danach entschlüsseln muss um zu sehen, ob ein Ergebnis richtig ist.
- Da hier Input Daten zur Auswahl der folgenden Befehle notwendig sein können, kann der Programmpfad (die Sequenz der Schritte der Auswertung) nicht vorher definiert sein, sondern muss in Kooperation zwischen den berechnenden Parteien ermittelt werden. Das impliziert keinen inhärenten Grenzen im Bezug auf Kommunikation oder Komplexität, ist allerdings eine wichtige initiale Beobachtung.

Es wird deutlich, dass die Auswertung einer Instruktion teurer in der Anzahl der verschlüsselten Operationen werden muss, als bei Schaltungsauswertungen.
Hier kann man nun feststellen, dass eine Kosten-Nutzen-Abwägung zwischen der Evaluation eines Operator und der Ausdrucksstärke des Operatoren viel wichtiger ist. Ein Operator könnte hierbei auch komplexere Funktionen als nur Addition und Multiplikation abbilden, damit würde ein in der Zeit linearer Mehraufwand anfallen, allerdings könnte man, falls die Operatoren frequent genutzt werden die Anzahl der gesamten Programmschritte erheblich reduzieren und so deutlich mehr Effizienz sparen. Durch die gegebenen Beschränkungen in Komplexität und Rundenanzahl wird eine solche Abwägung deutlich interessanter.
Daher schlage ich eine Auswertung auf Basis eines universellen “Gatters” mittels einer Operatoren Suite vor. Details dazu finden sich im nächsten Abschnitt.
Das entwickelte PFE Protokoll soll aktiv sicher sein, sofern der genutzte ORAM sicher ist und der ORAM-Client als MPC-Protokoll unter den verschiedenen Parteien verteilt ist.

Algorithmus 3.1 Setup

procedure SETUP PROGRAM($(t_i, x_i)_{1 \le i \le m}$, $[pc_0]$)
  for all $i \in \{1 \le i \le m\}$ **do**
    WRITE($ORAM_b, t_i, x_i$)
    $pc \leftarrow [pc_0]$
  end for
end procedure

procedure SETUP DATA($(x_j)_{1 \le j \le p}$)
  for all $j \in \{1 \le j \le p\}$ **do**
    $t_i \leftarrow$ RECIEVEPOSITION(i)
    WRITE($ORAM_b, t_i, x_i$)
  end for
end procedure

Algorithmus 3.2 Programm Evaluation

procedure EVALUATEPROGRAMM
  $t \leftarrow 0$
  while $\neg$ ABORT(pc, t) **do** // Differs depending on Function
    $x \leftarrow$ READ$(pc, ORAM_B)$
    $< i_1, i_2, o_1, opcode, pc_{off} > \leftarrow$ DECODE(x)
    $v_1 \leftarrow$READ$(i_1, ORAM_A)$
    $v_2 \leftarrow$READ$(i_2, ORAM_A)$
    $res, off \leftarrow$EVALUATEINSTRUCTION$(v_1, v_2, opcode)$
    $out \leftarrow o_1 + off$ // Possible use of Arrays
    WRITE$(out, res, ORAM_A)$
    $t \leftarrow t + 1$
    $pc \leftarrow pc + 1 + pc_{off} \cdot res \cdot off$
  end while
  $results \leftarrow$RETRIVEPLAINTEXT
  return $results$
end procedure

3.4.2 Modell der Realisierung

Nun kann ein Protokoll Π entwickelt werden, eine PFE Funktionalität wie vorgesehen realisiert. Dafür wird ein Zustand, eine Schritt-/Transition Funktion, ein Anfangs- und Endzustand, ein Programmspeicher und ein Datenspeicher benötigt. Auf jede dieser Entitäten wird hier im Folgenden eingegangen.
Das Protokoll Π soll auf Eingabe einer Menge von Tupeln $P_{\mathcal{F}} = \{([t_i], x_i) | 1 \leq i \leq m, x_i \in B\}$ die Befehle x_i aus der Menge der möglichen Befehle B in einer ORAM $ORAM_B$ an Stelle i abspeichern. Gleichzeitig soll auf Eingabe der Menge der Daten in Form einer Liste $(x_j)_{1 \leq j \leq p}$ jeweils von einer Partei i in einer ORAM $ORAM_A$ abspeichern. Die Formatierung der Daten und das dementsprechende Pre-Agreement wird davor ermittelt. Außerdem wird ein verschlüsselter Verweis pc auf die aktuelle Position im Programm auf einen vorher bestimmten Wert initialisiert. Dieser kann je nach Verschlüsselungsschema entweder gemeinsam berechnet oder von P_0 als Trusted Dealer implementiert werden. Allgemein gilt, dass jede Partei für die von ihr beigesteuerte Daten als Trusted Dealer betrachtet werden kann. Der Zustand des Protokolls kann nun durch $Z = (ORAM_A, ORAM_B, pc)$ beschrieben werden. Jeder Zustandsübergang kann nun durch die Ausführung eines Algorithmus wie Algorithmus 3.2 definiert werden. Es können Endzustände über die Menge der Zustände, bei denen das Prädikat ABORT(PC,T), definiert werden. Es existiert genau ein Anfangszustand, bei dem $pc = 0$ gilt und $ORAM_A$/$ORAM_B$ genau so wie im Pre-Agreement ausgemacht und mit den Daten der einzelnen Parteien gefüllt sind. Nun kann eine State Machine implementiert werden, die jeweils Zustandsübergänge zwischen Befehlen realisiert um so eine Evaluation durchzuführen. Hierfür kann ein spezieller pc Wert in Betracht gezogen werden, auf diesen immer wieder geprüft wird. Gleichzeitig kann ein Ergebnis nur in Kooperation mit anderen Parteien entschlüsselt werden. Dabei teilt jede Partei die Decryption-Shares der Werte die für andere Parteien bestimmt sind mit diesen und empfängt die eigenen.

Algorithmus 3.3 Instruction Evaluation

procedure EVALUATEINSTRUCTION($v_1, v_2, opcode$)
$off \leftarrow opcode_0 \cdot v_2 + opcode_1$
$res \leftarrow \sum_{i=2}^{n} opcode_i \cdot op_i(v_1, v_2)$
end procedure

Realisierung eines atomaren Befehls

Zur Auswertung eines Befehls werden eine Reihe von atomaren Operationen auf zwei selektierten Variablen ausgeführt und das Ergebnis in einer dritten Variable im ORAM gespeichert. Diese Variablen werden durch Register in einem Array dargestellt. Zusätzlich muss noch ein Offset für den Programm Counter berechnet werden, sodass bedingte Sprünge möglich werden.
Hierbei muss zuerst ein Befehl aus dem ORAM decodiert werden. Dieser beinhaltet verschlüsselt: Eingaberegister, Zielregister, und Opcode. Ein Opcode legt fest welche Operation durchgeführt werden soll. Dann sollen die Variablen aus den Eingaberegistern geladen, die atomare Operation verschlüsselt durchgeführt, der Programm Counter angepasst und das Ergebnis in die Ausgaberegister geschrieben werden. Die Auswertung des Befehls ist in Algorithmus 3.3 beschrieben. Danach ist pc_{off} der mögliche Offset um einen bedingten Branch zu realisieren und $opcode_i$ bezeichnet den Wert aus $opcode$, der sich auf die entsprechende vorher definierte Operation aus dem Pre-Agreement bezieht (bsp. Addition oder Multiplikation).

3.4.3 Änderungen durch Einführung des Kontrollflusses

Es ist einfach zu sehen, dass eine Partei P_0 ein Programm $P_{\mathcal{F}}$ so abändern kann, dass ein äquivalentes Programm $\hat{P}_{\mathcal{F}}$ entsteht, welches sich nicht in Ein- und Ausgaben, aber in der Länge der Programmausführung unterscheidet. Allgemein ist es bei Programmauswertungen mit Kontrollfluss immer möglich über die Laufzeit einen Rückschluss auf die Eingabe, insbesondere wenn es sich um Laufvariablen von Schleifen handelt.
Ich nehme in meiner späteren Analyse, wie schon bei dem meisten anderen PFE-Konstruktionen an, dass die P_0 das Programm $P_{\mathcal{F}}$ nicht manipuliert, sondern wahrheitsgetreu, beisteuert. Diese Voraussetzung muss aus den in der Literatur gängigen Gründen gelten, da es sonst trivial wäre bestimmte Eingaben offenzulegen. In der Realität kann so etwas z.B. durch eine Trusted Party die nicht an dem Austausch beteiligt ist zugesichert werden. Trotzdem ist es wichtig hier noch einmal zu betonen, dass diese triviale Möglichkeit Eingaben (auch nur teilweise) offenzulegen nun nicht mehr nur im Ergebnis der Partei P_0, sondern auch in der Auswertung selbst enthalten ist.

Möglichkeit von Einführung von komplexeren Datenstrukturen

Nun ergibt sich ein zweites Problem. Die Anzahl der möglichen Variablen, die gespeichert werden müssen sind möglicherweise unabhängig von der Anzahl der Schritte. Konkret heißt dass, da es im Allgemeinen nicht bekannt ist, wie viele Evaluationsschritte ein Programm benötigt, muss die Anzahl der Variablen dynamisch angepasst werden.
Falls nun keine indexbasierten Datenstrukturen genutzt werden, ist es relativ einfach zu sehen, dass die Anzahl der Daten im $ORAM_B$ durch die Anzahl der Befehle im $ORAM_A$ zuzüglich der

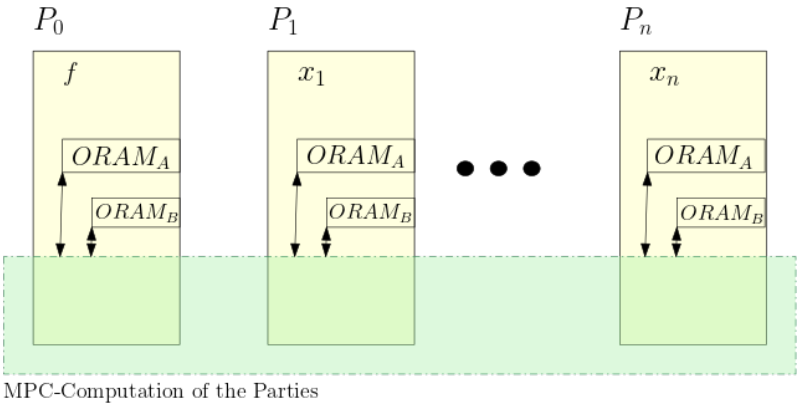


Abbildung 3.3: Darstellung der ORAM-Struktur

Eingaben und der zweier Startelemente (Dummy Placeholder & konstanter Wert) begrenzt ist. Dies schränkt allerdings wieder die möglichen Funktionen ein, da nun nur Funktionen mit konstanter Speicherplatzbeschränkung genutzt werden können.
Eine allgemeine Lösung wäre es eine obere Grenze der Anzahl der Variablen, also die Größe des ORAMs, zuerst durch eine mögliche obere Begrenzung an Werten m zu schätzen und dann nach m Schritten die Größe des $ORAM_A$s, zu verdoppeln. Das ist entweder möglich durch ein komplettes Rewrite oder eine dynamische Erweiterung. Die Kosten eines Rewrites sind linear in der Variablenanzahl, da ein solches Rewrite in der ersten Initialisierung über eine *Batch Initialization* Prozedur erfolgen kann, die ausnutzt, dass hier weder bereits Daten gespeichert wurden, noch zwischen den Write-Befehlen Daten gelesen werden. Also würde sich amortisiert auf die Anzahl der Zugriffe ein kleiner konstanter Overhead ergeben. Eine dynamische Erweiterung kann erfolgen, falls der ORAM dies unterstützt. Insbesondere Konstruktionen auf Basis von Baumstrukturen, wie der schon betrachtete ORAM von *Shi et al.* [40], bieten sich hier an. Hierbei können neue Knoten an die Blätter des Baumes angehängt werden. Diese Änderungen betreffen allerdings die Read-Write-Performance, da die Größe der Buckets ja zumindest in der Größenordnung logarithmisch gewählt werden sollte. Dennoch ist der Mehraufwand der Kosten auch hier vernachlässigbar.

3.4.4 Betrachtung des ORAMs

Es ist zuerst festzustellen, dass eine ORAM-Implementation nun beliebig eingesetzt werden kann. Die Abbildung 3.3 stellt dar, wie nun die Berechnung einzelne Shares aus dem ORAM entnommen werden können. In jedem Lese-Zugriff auf bekannte ORAM-Konstruktionen in einem MPC-Client, muss ein logarithmischer Aufwand betrieben werden um diesen durchzuführen. Es muss immer eine pseudozufällige Permutation berechnet werden um einen möglichen Zugriffspfad zu ermitteln und danach muss aus diesen Elementen, die in dem möglichen Zugriffspfad sind, das Richtige ausgewählt werden. Diese Operationen werden im Allgemeinen als teuer betrachtet , da die komplette Evaluation hier für alle möglichen Fälle voll verschlüsselt ausgeführt werden muss. Daher ist es sinnvoll die Anzahl der Auswertungsschritte auf ein Minimum zu reduzieren.

3.4.5 Betrachtung der Effizienz

Im Folgenden betrachte ich die Effizienz eines Programms mit m Befehlen und n Zeitschritten. Hierbei liefern die i Parteien akkumuliert p Input Daten und erhalten q Ergebnisse. Der Algorithmus 3.1 kann als Offline Phase vor der eigentlichen Auswertung durchgeführt werden. Da es am Anfang möglich ist die Daten kummulativ in den ORAM zu schreiben, liegt die Setup Phase in $\theta(m+p)$ mit $\theta(i)$ Kommunikationsrunden.
In der Onlinephase muss nun zu jeder einzelnen Befehlsauswertung ein Befehl aus dem ORAM gelesen werden. Es ist trotzdem ein Write-Back notwendig, um Obliviousness zu garantieren. Deshalb liegt das Auslesen von einem konstant großen $ORAM_B$ in $O(n \cdot log^s(m))$ mit einem konstanten Faktor s. Die Rundenanzahl lässt sich hierbei auch nicht reduzieren, da pc ja erst nach einer Befehlsevaluation sicher gelesen werden kann.
Bezogen auf einzelne Instruktionen: Dort erfolgen zwei Reads und ein Write in einer $ORAM_A$, sodass dieser bei jeder Instruktion um genau ein Element wächst, dieser Aufwand lässt sich mit $O(n \cdot log^s(n))$ abschätzen. Hierbei ist zu notieren, dass wie schon im Abschnitt darüber dargestellt, dieses n auf m reduziert werden kann, falls P_0 hierbei eine Funktion mit konstantem Speicherplatz berechnet. Des Weiteren werden für eine Befehlsauswertung von a Befehlen $2 + a + \sum_{b=0}^{a} c(op_b)$ Multiplikationen benötigt, wobei $c(op_b)$ die Kosten der Operationen von op_b bezeichnet. Hier können die Runden durch Parallelisierung auf eine konstante Anzahl verbessert werden, insgesamt wird also eine lineare Anzahl an Runden notwendig. Es wird also ein Aufwand von $O(n \cdot (log^s(m) + 2 + a + \sum_{b=0}^{a} c(op_b))$ benötigt.
Es ergibt sich ein Gesamtaufwand von $O(n \cdot (log^s(m) + 1 + a + \sum c(op) + n \cdot log^s(n))$ was sich zu $O(n \cdot log^s(m \cdot n))$ reduzieren lässt. Im Falle von im Speicherplatz konstanten Funktionen ergibt sich weiter durch abschätzen des im Logarithmus enthaltenen n zu einem m die Komplexität $O(n \cdot log^s(m))$.
Auch wenn diese Komplexität durchaus skalierbar ist, darf hier nicht vergessen werden, dass dieses Schema in den Interaktionsrunden genauso skaliert. Diese Skalierung wirkt sich im Allgemeinen sehr schlecht auf die Laufzeit aus. Des Weiteren bringt diese Konstruktion einen hohen konstanten Faktor mit sich, der die Praktikabilität deutlich einschränkt.

Warum eine teurere Befehlsevaluation sinnvoll sein kann

Zuerst ist hier festzustellen, dass die Anzahl der Kosten zusätzlicher Operationen durch eine Multiplikation begrenzt wird um eine angemessene Ausdrucksstärke zur Auswertung arithmetischer Funktionen zu erhalten. Falls nun Kontrollflussstrukturen realisiert werden sollen, müssen möglicherweise noch andere kompliziertere Bedingungen realisiert werden. In anderen Anwendungsfällen (bsp. Machine Learning) sind einzelne Evaluationsschritte standardmäßig möglich und eine explizite Berechnung würde auch hier wenig Informationen über die zu berechnende Funktion offenbaren. Ein Beispiel sind die Berechnung von Aktivierungsfunktionen, Pooling Verfahren oder Convolution Anwendungen in verschiedenen Deep Learning Verfahren. In solchen Fällen könnte ein funktionsbeitragende Partei P_0 sich entscheiden bestimmte Teile explizit zu berechnen oder ggfs. auf schon vorher etablierte PFE-Operationssuites zurückzugreifen. Im Gegensatz zur vollen PFE Auswertung mittels minimaler Operatoren, können hierfür in einer Offline Phase vorberechnete Masken z.B. zur Zerlegung in einzelne Bits für kompliziertere Operationen genutzt werden. So könnten (Un-)Gleichheitstests, Modulo Operationen, oder auch arithmetische Operationen, wie z.B. Wurzelfunktionen, realisiert werden.

Würde dies nicht möglich sein, müsste (bei regelmäßiger Nutzung dieser Operatoren) eine relativ hohe Anzahl an Befehlen hinzukommen, die dementsprechend wieder einen zusätzlichen erhöhten Speicherbedarf und mehr *Read-Write-Zugriffe* benötigen. Dabei würde das Protokoll deutlich schlechter skalieren.

Ausblick auf mögliche Effizienzsteigerungen

Nun kann versucht werden dieses Schema noch durch eine Reihe von Verbesserungen effizienter zu gestalten. Diese Verbesserungen sind unter anderem oft heuristisch. Folgende mögliche Optimierungen haben nach erster Betrachtung das höchste Optimierungspotenzial:

- Es können die Indikatoren, die bestimmen, welche Operation selektiert wird, nur mit 0 und 1 gesetzt werden. Dann ist es wie schon in [35] erklärt möglich die Anzahl der gespeicherten Indikatoren halbiert, das spart Zeit, dadurch dass keine neue Randomisierung in der ORAM-Speicherung notwendig wird. Außerdem können die Kommunikationskosten so weiter minimiert werden. Im Gegensatz dazu steht, dass die Möglichkeit lineare Kombinationen einzelner Operatoren nicht mehr zur Verfügung steht.
- Es besteht die Möglichkeit sich auf genau ein ORAM-Read beschränken und den zweiten Wert aus dem vorherigen Befehl übertragen als Teil des Programmzustandes aufzufassen. Das führt allerdings dazu, dass es hierbei möglich ist Dummy Befehle in den Speicher einzufügen um ein Element zu tauschen. Ob das im realen Anwendungsfall eine Optimierung darstellt ist ungewiss.
- Wie schon beobachtet wird die größte Reduktion dadurch geschaffen, die Größe des ORAMs zu begrenzen. Inwiefern man hier die Anzahl der gespeicherten Variablen real dynamisch skalieren kann, also genau dann wenn der Speicher voll ist und nicht dann wenn das Programm ausreichend viele Zeitschritte gegangen hat, bleibt zu evaluieren. Damit hier die Privacy Eigenschaft nicht verletzt wird, muss allerdings hier eine größere Summe an möglichen Elementen aus dem ORAM übertragen werden. Daher muss diese Optimierung mit Vorsicht eingesetzt werden. Hierbei wäre sie deutlich besser, falls dieser Übertrag mit den zyklischen Rehashing einer Epoche oder ähnlichen Eviction Prozeduren gemeinsam geschieht, um so wenig bis keinen Overhead zu generieren.

Zusammenfassend lässt sich sagen, dass es zwar Möglichkeiten der Verbesserung gibt, aber diese das System nur um einen (u.U. sehr kleinen) linearen Faktor verbessern können.

3.4.6 Betrachtung der Sicherheitseigenschaften

Im Folgenden werden die wichtigsten Sicherheitseigenschaften wie schon am Anfang dieses Kapitels definiert betrachtet und erörtert warum das Evaluations-System privat und korrekt ist.
Es darf angenommen werden, dass ein aktiv sicheres Sharing vorliegt. Das heißt, dass jede Partei genügend Shares eines Secrets hält, sodass ohne diese Shares keine Entschlüsselung möglich ist. Ein solches Secret-Sharing muss die Möglichkeit bieten, die Shares zu rerandomisieren, da das zur sicheren Speicherung im ORAM notwendig ist (insbesondere für Rewrites). Des Weiteren ist es

leicht zu sehen, dass die Chiffre sicher gegenüber Choosen-Plaintext-Attacks (*CPA-Secure*) sein muss. Das ist wichtig, da sonst eine Partei P ihre Eingabe so wählen kann, sodass sie ggfs. in der weiteren Berechnung Informationen über die Eingaben anderer Parteien erhalten könnte.

Privacy

Zur Hilfe lassen sich wie in [3] korrespondierend zu Circuit Topology Hiding und Private Gate Evaluation zwei neue Eigenschaften definieren. Zuerst angelehnt an CTH die Eigenschaft *Structure Hiding*:

Definition 3.4.1 (Structure Hiding)
Bezeichnet die Eigenschaft eines Protokolls die Eingaben und Ausgaben eines verschlüsselten atomaren Befehls in einem verschlüsselten Programm, sowie deren Beziehungen zu anderen Befehlen, zu verstecken. Insbesondere darf ein polynomiell beschränkter Angreifer nicht in der Lage sein, mit einem nicht-vernachlässigbaren Vorteil zu raten, ob sich zwei Zugriffe auf denselben Wert beziehen.

Demnach muss eine Structure Hiding Funktionalität aus mindestens zwei Eingabevariablen und einer Ausgabevariable realisieren. Es ist zu notieren, dass SH nur die durchgeführten Variablenabrufe, nicht aber den Befehl selbst realisiert. Im Gegensatz zu CTH soll die hier konstruierte SH Komponente nicht zwingend ein on-demand Mapping realisieren, sondern deutlich genereller einen RAM-Zugriff, der als oblivious charakterisiert wird. Ein Angreifer hat einen Vorteil, falls er einen polynomiell in der Zeit begrenzten probabilistischen Algorithmus konstruieren kann, der erraten kann, ob die Werte zweier Zugriffe übereinstimmen, und hierbei deutlich von der Gleichverteilung über die Anzahl der Zugriffe abweichen kann. Falls in einem Angriff P_0 korrupt ist, darf der Angreifer trotzdem kein Wissen über f verwenden.

Als zweites wird korrespondierend zu PGE die Eigenschaft *Private Instruction Evaluation* definiert:

Definition 3.4.2 (Private Instruction Evaluation)
Bezeichnet die Eigenschaft einer Funktionalität einen Befehl, bestehend aus möglicherweise mehreren arithmetischen Operationen in einer Weise auszuführen, sodass später von keiner berechnenden Partei ermittelt werden kann welche der Operationen auf die Eingabe angewendet wurde um die dementsprechende Ausgabe zu erreichen. Insbesondere darf auch hier wieder ein polynomiell-zeit beschränkter Angreifern nicht mit einem nicht vernachlässigbaren Vorteil erraten können, welche Operation gewählt wurde.

Auch hier wieder hat ein Angreifer einen Vorteil, falls er in der Lage ist einen probabilistischen Algorithmus zu konstruieren, der für ein Transkript der Berechnungen der Funktionalität an der er, je nach Angreifermodell unterschiedlich, verschiedene Parteien korrumpiert haben könnte, errät welche korrespondierende Klartextoperation durchgeführt wurde. Hierbei darf die Erfolgsrate nicht nicht-vernachlässigbar von der Gleichverteilung über alle Operationen abweichen. Auch hier gilt, dass falls in einem Angriff P_0 korrupt ist, so darf der Angreifer trotzdem kein Wissen über f verwenden.

Theorem 3.4.1
Ein Protokoll bestehend aus den Algorithmen Algorithmus 3.1, Algorithmus 3.2, und Algorithmus 3.3 bildet ein sicheres Private Function Evaluation-Protokoll.

Beweis (informal) Das Protokoll ist genau dann privat, falls es die Structure Hiding und die Private Instruction Evaluation Komponenten (Algorithmen 3.2 & 3.3) sicher sind. Das ist trivial zu sehen, da der Kontrollfluss außerhalb dieser Komponenten nur in $\text{ABORT}(pc, t)$ divergiert, was laut Definition in einem Pre-Agreement sicher zu wählen ist.

Theorem 3.4.2
Die Structure Hiding Komponente in Algorithmus 3.2 ist sicher (privat), genau dann wenn der unterliegende ORAM sicher implementiert ist.

Beweis (informal) Die Structure Hiding Komponente ist sicher, da der ORAM nach Defintion sicher(oblivious) implementiert ist und somit keine Beziehungen der Variablen(insbesondere deren Werte) untereinander bekannt werden. Insbesondere ist der gesamte Kontrollfluss deterministisch. Alle anderen Berechnungen werden verschlüsselt mittels einer CPA-sicheren Chiffre durchgeführt und es werden keine Variablen entschlüsselt.

Theorem 3.4.3
Die Private Instruction Evaluation Komponente in Algorithmus 3.3 ist privat, genau dann wenn alle unterliegenden Operatoren sicher evaluiert werden.

Beweis(informal) Die Berechnung wird mit deterministischem Kontrollfluss und CPA-sicheren MPC-Komponenten durchgeführt. Es werden keine Werte entschlüsselt, damit ist die Private Instruction Evaluation Komponente sicher.

Correctness

Es gilt hier zu zeigen, dass das gegebene Protokoll die korrekten Daten errechnet. Hierfür ist es wichtig, dass ein aktiv sicheres Secret Sharing vorliegt, in diesem Fall ist die Berechnung genau dann korrekt, falls alle einzelnen Komponenten sicher sind.

Aktiv sichere Secret Sharings Ein aktiv sicheres Secret Sharing kann hier über verschiedene Möglichkeiten realisisert werden. Es können Zero-knowledge-Proofs für einzelne Berechnungen verlangt werden, die sowohl Addition, als auch Multiplikation absichern. Das würde zu hohen Einbußen in der Effizienz des Protokolls führen und wird daher in der Praxis nicht oft verwendet. Eine gängigere Variante ist die Implementation über kryptographisches Tagging. Hierbei werden zu jeder Share ein Tag verteilt, der auch dieselben homomorphen Eigenschaften hat wie die Ciffre. Mit jeder Wertberechung, werden zusätzlich auch die Tags in derselben Weise verrechnet, was dazu führt, dass Falls zwei Tags t_1 und t_2 für Klartexte korrespondierend zu den Chiffren x_1 und x_2 valide ist, so ist der Tag $op(t_1, t_2)$ zum Klartext der Chiffre $op(x_1, x_2)$ valide. Ein Beispiel einer solchen Anwendung von *Authenticated Shares* findet sich in der Beschreibung zu dem von *Keller et al.* entwickelten Protokoll [26]. Somit kann ein aktives Secret Sharing in der unterliegenden Chiffre ohne Beschränkung der Allgemeinheit angenommen werden.

Falls nun bei jeder Entschlüsselung der verschlüsselten Texte jede Partei die Tags überprüfen kann, so ist es leicht zu sehen, dass das entwickelte Protokoll eine aktiv sicheres MPC-Protokoll realisiert. Also garantiert dieses Protokoll Korrektheit, falls die darunterliegenden Operationen/ORAM-Zugriffe aktiv sicher berechnet werden.

Andere Eigenschaften

Es ist einfach zu sehen, dass wenn eine Eigenschaft für dieses PFE-Schema gilt, das diese Eigenschaft auch für den darunterliegenden MPC-Algorithmus gelten soll. Daher kann man jetzt schon argumentieren, dass *Guranteed Output Delivery* und *Fairness* nicht möglich sein können, da nicht von einem Honest-Majority Setting auszugehen ist. Das Honest-Majority Setting würde auch als Annahme im PFE-Kontext wenig Sinn ergeben, da die Parteien ja über die unterschiedlichen Auswirkungen ihrer Eingaben einen unterschiedlichen Einfluss auf die Berechnungen haben können. Hier ist es in der Literatur typischer eine Evaluation als Evaluation zwischen zwei Parteien, wobei eine davon verteilt ist, zur betrachten. In diesem Fall wäre genau die Hälfte der Parteien möglicherweise kompromittiert und es kann keine Honest-Majority angenommen werden.
Reusability wird in diesem Kontext wenig Sinn ergeben. Hierbei ist das einzige was Offline stattfinden könnte die intialisierung des ORAMs dieser ist allerdings nicht sicher, wenn derselbe ORAM in verschiedenen Auswertungen nacheinander betrachtet wird. Trivialerweise kann man z. B. über die Unterschiede der Shares, die aus dem Instruction-ORAM genommen werden Informationen erhalten, wann der Kontrollfluss zweier Ausführungen divergiert. Dazu müssten die Shares rerandomisiert werden. Auch könnten zwei Zugriffe auf denselben (verschlüsselten) Index des ORAMs ohne eine neue Permutation oder ein Rewrite erfolgen, was trivialerweise als unsicher anzusehen ist. Rerandomisierung und Einführung einer neuen Permutation sind insgesamt teurer als eine optimierte Intialisierung des ORAMs.

4 Implementation

In diesem Kapitel wird eine Implementation der in Kapitel 3 vorgestellten Konstruktion beachtet. Dazu beschreibe ich zuerst das genutzte Framework in Abschnitt 1, um dann in Abschnitt 2 ein mögliches Programm zur Evaluation von privaten Funktionen vorzustellen. Danach wird die praktische Effizienz an einer Reihe verschiedener Indikatoren betrachtet.

4.1 MP-SPDZ

Das MP-SPDZ Framework ist eine Implemtierung des SPDZ-Protokolls von Damgård et al. [11] von *Marcel Keller* [25]. Mittlerweile unterstützt das Framework eine Reihe von Protokollen zu den MPC-Varianten Secret-Sharing und Yao's Garbeled Circuits. Dabei unterscheiden sich die Protokolle besonders in den Sicherheitsvorraussetzung. Es kann entweder Honest- oder Dishonest-Majority ausgewählt werden. Des Weiteren werden die drei in der Arbeit betrachteten Angreifermodelle bzw. korrespondierende Sicherheitsdefinitionen *aktiv*, *passiv*, und *(publicly verifiable-) covert* unterschieden.
MP-SPDZ wird mittlerweile von einem Großteil der Autoren in den Forschungsbereichen zu PFE und ORAM angewendet und für gemessene Benchmarks genutzt.
Insbesondere die Implementierungen zu ORAM sind für die Benchmarking Anwendungen interessant. Hierfür liefern *Keller et Yanai* [28] eine Implementation diverser Datenstrukturen wobei eine Berechnung ähnlich wie bei der PFE-Konstruktion über eine *CPU-Step-Circuit* durchgeführt wird, der einer geöffneten Private Instruction Evaluation entspricht. Diese Entwicklung wird von *Keller et Scholl* [27] weitergeführt. Dabei werden beispielhaft einige Einsatzmöglichkeiten der verschiedenen verschlüsselten Datenstrukturen und einzelne Analysen zur Effizienz geliefert. Allerdings wurde noch kein Allgemeines PFE System betrachtet oder implementiert.

Grundlagen zur Realisierung der Berechnungen

MP-SPDZ implementiert eine Reihe von Primitiven zur Verbesserung der Effizienz. Die für eine Evaluation wichtigsten werden im Folgenden kurz erklärt.

Somewhat Homomorphic Encryption Das Framework ist in zwei strikt getrennte Phasen aufgeteilt. Es wird zuerst in einer Offline Phase das Programm lokal kompiliert. Dabei wird die Programmausführung einmal simuliert und die Menge der benötigten Masken zur Steigerung der Effizienz grundlegender Bit-Operationen gespeichert. Insbesondere werden sogenannte SPDZ-Tripel von zufälligen Zahlen zur schnellen Multiplikation und Bit-Zerlegungsmasken, also ein zufälliger verschlüsselter Wert in Bit und Dezimaldarstellung, generiert. Diese beschleunigen die Anwendung von komplexen arithmetischen Operationen.
Die Verschlüsselungen in MP-SPDZ sind i.A. homomorph gegenüber der Addition.

Zur Multiplikation zweier Zahlen $[x]$ und $[y]$ wird ein vorberechnetes Tripel $[a]$, $[b]$, und $[c] = [a \cdot b]$ verwendet. Dabei kann $[x - a]$ und $[y - b]$ geöffnet werden und $[x \cdot y] = [c] + (x - a) \cdot [b] + (y - b) \cdot [a]$ lokal berechnet werden (vgl. [25] Kapitel 2).
Weitere kompliziertere Masken sind in der Regel abhängig von dem ausgeführten Protokoll.

Secret Sharing Methoden Es werden hauptsächlich additive Sharings, Shamir Sharings, und replizierte Sharings genutzt. Dabei liegt ein besonderes Augenmerk auf effizientem Resharing zur Randomisierung der Chiffretexte, was in dem Algorithmus im ORAM oft genutzt wird.
In dem bei der Evaluation standardmäßig benutzten Protokoll MASCOT von *Keller et Orsini* [26] wird diese Variante durch eine effiziente aktiv sichere Evaluation mittels One-Time-MACs ergänzt. Hierbei existiert ein geheimer secret-shared Schlüssel δ, der auf jede Share aufmultipliziert wird und genauso homomorph ist wie der Verschlüsselungsalgorithmus. Nun kann beim Entschlüsseln eines Wertes der geheime Schlüssel δ aufgedeckt werden und dann muss für das Ergebnis x $x \cdot \delta$ gelten. Das verbessert die Effizienz deutlich. Auch ist dadurch klar, dass Shares möglichst wenig geöffnet werden sollten, sodass möglichst wenig Daten verarbeitet werden müssen.
Es wird auch deutlich, dass es hier im Allgemeinen keine Guranteed Output Delivery oder Accountability möglich wird, solange nicht Shamir Secret Sharing im Honest Majority Setting verwendet wird.

ORAM-Implementationen Es wird ein adaptiver ORAM verwendet, der je nach Größe im Typ divergiert, in den Benchmarks wird meistens die trivialen Konstruktion basierend auf Linear Scanning aus [17] verwendet. Für Zugriffe auf erweiterte Speicherstrukturen werden die zwei Implementationen aus [27] in Betracht gezogen. Das beinhaltet eine auf MPC-Computation optimierten Variante des Path-ORAMs aus [41]. Alternativ könnte auch auf ein Tree-ORAM aus [40] aufgebaut werden. Hierbei verwendet der Tree-ORAM entweder Squared-Root-ORAM oder einen trivialen ORAM als unterliegenden Bucket ORAM.
Zur Optimierung wird eine Batch Initialisierung mittels einer Reihe von Werten eingesetzt. Diese nutzt aus, dass der Befehls-ORAM am Anfang an einer Reihe von Werten, bei denen jeder unterschiedlich ist und ohne Read dazwischen befüllt wird, dadurch sind Optimierungen möglich. Im Gegensatz zu den Annahmen im Kapitel zur Entwicklung ist es hier aufgrund von Implementationsentscheidungen nicht möglich Tupel von verschiedenen verschlüsselten Werten zu speichern. Das wird mitigiert indem verschiedene Werte in eine verschlüsselte Zahl codiert werden. Dadurch wird die Domäne abhängig von der Größe der Chiffretexte ggfs. stark eingeschränkt, aber es vereinfacht die Messungen und macht die Ergebnisse deutlich besser interpretierbar.

Oblivious Compuation Innerhalb von MP-SPDZ wurden bereits eine Vielzahl von Datenstrukturen zur Oblivious Compuation realisiert. Die Mehrheit davon wurden von *Keller et Scholl* in [27] beschrieben und ausgewertet. Dort findet sich neben der Analyse der in der Implementation benutzten ORAM Methoden auch andere Datenstrukturen die oblivious sind. Hierfür wurden Dictionaries, (Sorted-)Arrays, und Priority-Queues ausgewählt. Bis auf den trivialen ORAM besitzen alle Datenstrukturen polylogarithmische Zugriffskomplexität. Mithilfe dieser Strukturen konnte die Ausführung von komplexen Algorithmen wie z. B. Dijkstra's Algorithmus oder dem Gale-Shapley Algorithmus (zur Lösung des Stable Marriage Problems) simpler realisiert werden. Es findet sich allerdings keine Betrachtung der Möglichkeiten inwiefern solche Datenstrukturen für private Evaluation weiter genutzt werden könnten, bis auf die triviale Idee ORAM zur Speicherung der Befehle zu nutzen.

4.2 Eigene Konstruktion

Im Folgenden wird kurz erklärt wie die Implementation des in Kapitel 3 entwickelten PFE-Schema aufgebaut ist, und welche Designentscheidungen zu berücksichtigen sind. Hierbei wird das Programm noch einmal mit normalen (nicht privaten/verschlüsselten) CPU-Step-Circuit verglichen.

Ausführung

Das Programm besteht aus vier Komponenten, die zur Ausführung benötigt werden. Zum einen eine Compiler Klasse, die MPC-Funktionalitäten realisiert. Dazu ein MPC-Programm, was das Compiler Programm mit Basisklassen aufruft. Diese MPC-Programm kann nun kompiliert werden um für mögliche Eingabe vorzuberechnen welche Daten zur SHE-Berechnung benötigt werden. Nun muss zur Ausführung das Programm in eine Form gebracht werden, sodass es automatisiert eingelesen werden kann. Dafür habe ich mich entschieden manuell ein Programm in einfacher Befehlssatz-Architektur in jeweils Instruction-Codes fester Länge umzuwandeln.
Aufgrund der Limitationen in der Interaktivität und Dateizugriffen in MP-SPDZ muss die Konfiguration, also das Pre-Agreement fest im Programm codiert sein, da sonst eine Vorberechnung der Werte nicht möglich wäre. Ein zweites Programm nimmt nun stattdessen ein Programm in Instruction-Set-Form entgegen und rechnet es in Instruction Codes um, die eine ausführende Partei P_0 automatisch einlesen kann.
Zur Vorbereitung der Evaluation verteilen alle Parteien ihre Eingaben in einem sicheren Secret Sharing Schemes und befüllen gemeinsam die ORAMs mit den entsprechenden Eingaben.

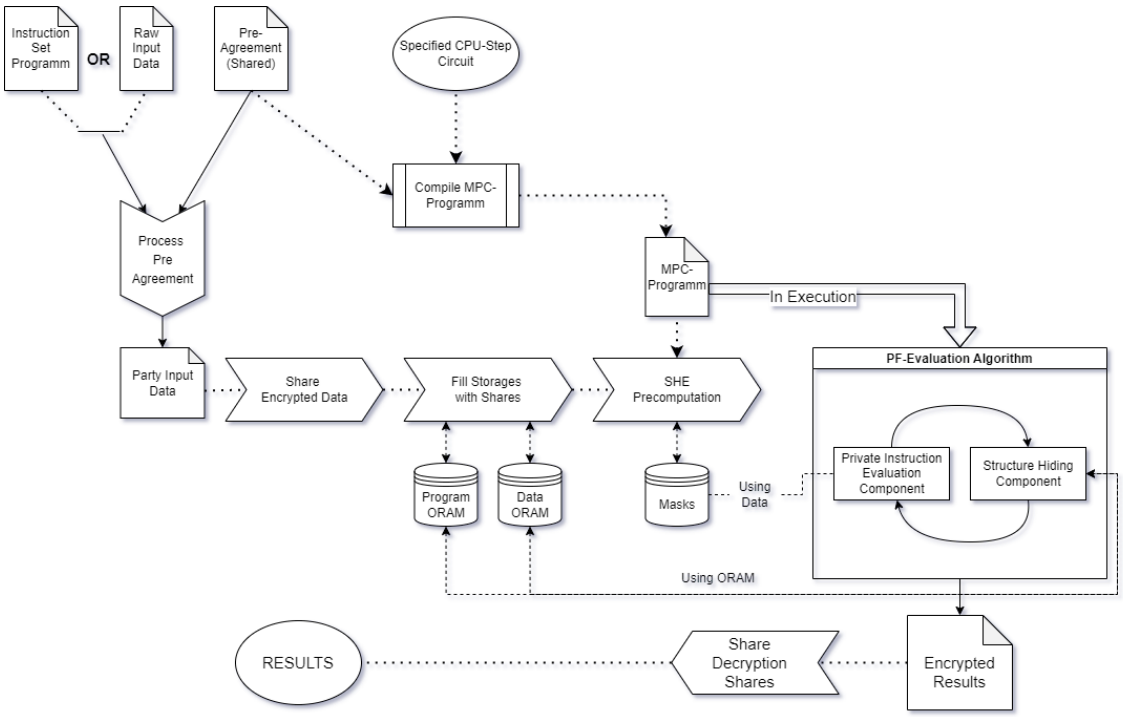


Abbildung 4.1: Ausführung der Implementation eines Private Function Evaluation Algorithmus

Danach kann eine gemeinsame Evaluation als reine MPC-Berechnung gestartet werden. Dabei kann man diese Berechnung als *Ausführung einer Virtuellen Maschine eines verschlüsselten Prozessors* betrachten. Eine genauere Auflistung der einzelnen Prozessschritte findet sich in Abbildung 4.1.

Variable Charakteristiken des Programmes Innerhalb einer Befehlssatzarchitektur gibt es verschiedene Charakteristika. Diese sind z.T. redundant mit den Betrachtungen in Kapitel 3, allerdings werden hier die Charakteristika hier von der Seite der Registermaschinen betrachtet. In der Realisation ist die Betrachtung der Vereinigung von PFE und ausführbaren Programmen mittels virtueller Maschinen dennoch nur vollständig, falls auch Optimierungen und Parameter zur normalen *CPU*-Ausführung betrachtet werden und untersucht wird, inwiefern diese auf kryptographische *CPU-Step-Circuits* zu übertragen sind.
Das wichtigste Charakteristikum ist der Typ des Befehlssatzes. In der Praxis gibt es viele verschiedene Paradigmen (z. B. Reduced Instruction Set Computing oder Complex Instruction Set Computing), allerdings wird bei den Betrachtungen aus Kapitel 3 klar, dass ein verschlüsseltes Instruction Set möglichst minimal sein muss.
Weitergehend ist auch die Anzahl der möglichen Operanden eine respektive der Effizienz kritische Variable. Hierbei ist die Entscheidung eine Befehlsauswertung mit 3 Lesezugriffen, bestehend aus zwei Operanden und einer Registeradresse zum Schreiben des Ergebnisses gefallen. Diese wird von der SH Komponente realisiert. Mögliche effizientere Alternativen wären eine Architektur mit zwei Variablen, wobei das Ergebnis aus der vorherigen Berechnung immer direkt in die nächste Evaluation übernommen wird. Hier ist das Problem, dass Befehle zum Wechseln der Operanden eingefügt werden müssen, was die Programmgröße und die Schrittzahl erhöht und somit wie schon in Kapitel 3 erörtert wahrscheinlich eine schlechtere Skalierbarkeit zur Folge hat. Eine experimentelle Bestätigung dieser Hypothese wird hier nicht vorgebracht. In der Theorie könnte man diesen Gedanken auch weiter fortsetzen und verallgemeinert ein Working-Set an Variablen außerhalb des ORAMs halten. Das würde die ORAM-Größen sowie Zugriffe um einen konstanten Faktor minimieren, allerdings auch einen erhöhten Aufwand in der einzelnen Befehlsauswertung mit sich bringen, da die Anzahl an benötigten Selektor-Operationen exponentiell steigt.
An die letzte Betrachtung kann man nun die Aufteilung des Speichers als Kriterium des Programmes an betrachten. Wie schon erläutert bringt das Halten eines kleinen Working Sets in Form von festgelegten Registern keinen theoretischen Effizienzgewinn mit sich. Natürlicherweise stellt sich hier die Frage inwiefern sich potenzielle Lokalität in der Berechnung ausnutzen lässt um z. B. über hierachische ORAM Strukturen die asymptotische Effizienz zu verbessern. Diese heuristischen Untersuchungen werden auch hier nicht weiter betrachtet.

Erfüllte (nicht-)funktionale Eigenschaften der Implementation Die Implementation bietet eine Reihe von Eigenschaften die eine sehr adaptive Ausführung erlauben. Hierbei ist zuerst die ORAM Abstraktion zu erwähnen. Es ist hierbei möglich jede aktiv sichere ORAM-Implementation einzubinden, solange Read- und Write-Funktionalitäten bereitgestellt werden. Des Weiteren sind *Instruction Evaluation* und *Structure Hiding* Komponenten weitestgehend getrennt und können beliebig ersetzt werden.
Es können verschiedene atomare Operationen basierend auf der Komplexität der auszuwertenden Funktion definiert werden, was eine dynamische Adaptivität an die Anforderungen der Evaluation ermöglicht.
Es werden die Variablen 0 und 1 in Registern 0/1 bereitgestellt, sodass P_0 ggfs. eigene Inputs oder Konstanten in die Funktion statisch eincodieren kann.
Diese Implementation stellt ein aktive sichere und korrekte Implementation dar.

Es konnten aber auch einige Eigenschaften nicht realisiert werden. Dazu gehört wie schon zuvor erwähnt Reusability, aber auch Accountability. Des Weiteren ist keine adaptive Bestimmung der Abbruchbedingung möglich, da MP-SPDZ eine compile-time deterministische Obergrenze der zu vorberechnenden Werte benötigt um das Programm zu kompilieren. Dadurch wird auch die Betrachtung von zyklischen Erweiterungen hinfällig. Auch der arraybasierte Zugriff ist nicht implementiert. Die ORAM Implementation bietet keine Möglichkeit Tupel von Zahlen abzuspeichern, was dazu führt, dass die Anzahl der Zugriffe, sowie die Größe des ORAMs deutlich größer ist als theoretisch möglich.

4.3 Evaluation

Die Evaluation wurde auf zwei über LAN verbundenen Rechnern [1] ausgeführt. Dabei wurden Ausführungszeit, vorberechnete SPDZ-Tripel, Interaktionsrunden, und Datenaustausch gemessen. Falls nicht anders angegeben wurde das Programm mit 25 Instruktionen (in Bitcode-Notation), einer Datenkapazität von 50 Variablen, 1000 Auswertungsschritte, Multiplikation/Addition/Branch if eqal, und MP-SPDZ mit SPDZ evaluiert.

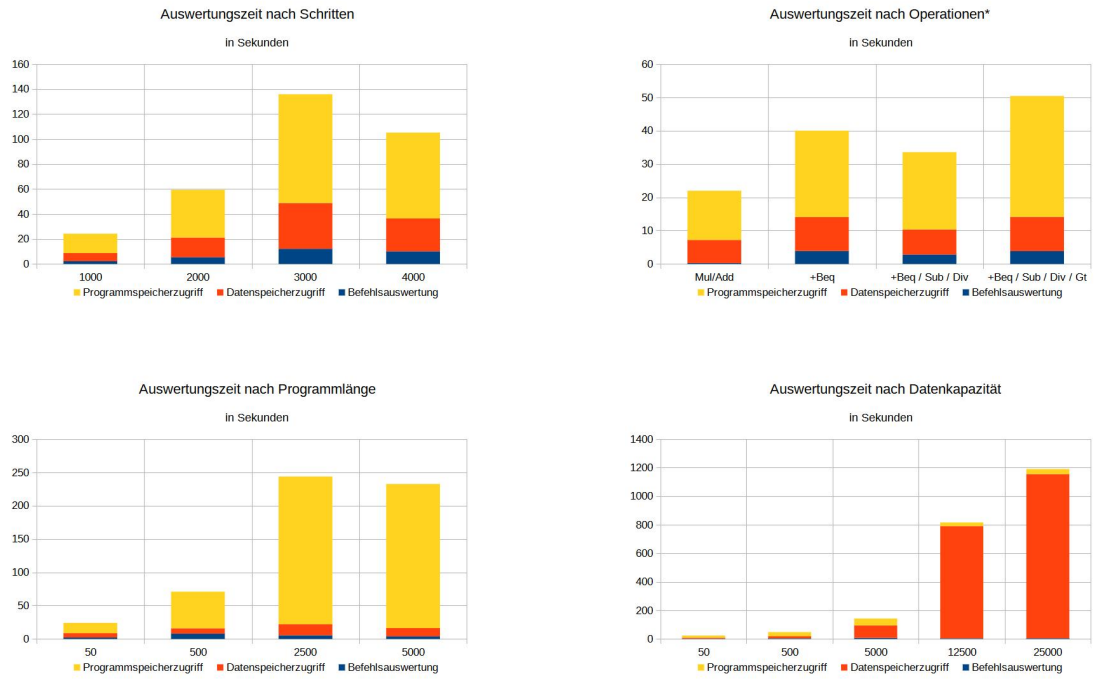


Abbildung 4.2: Ausführungsdauer des Protokolls

[1] Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz, 8GB RAM
* Gemessen mit 50 statt 25 Instruktionen

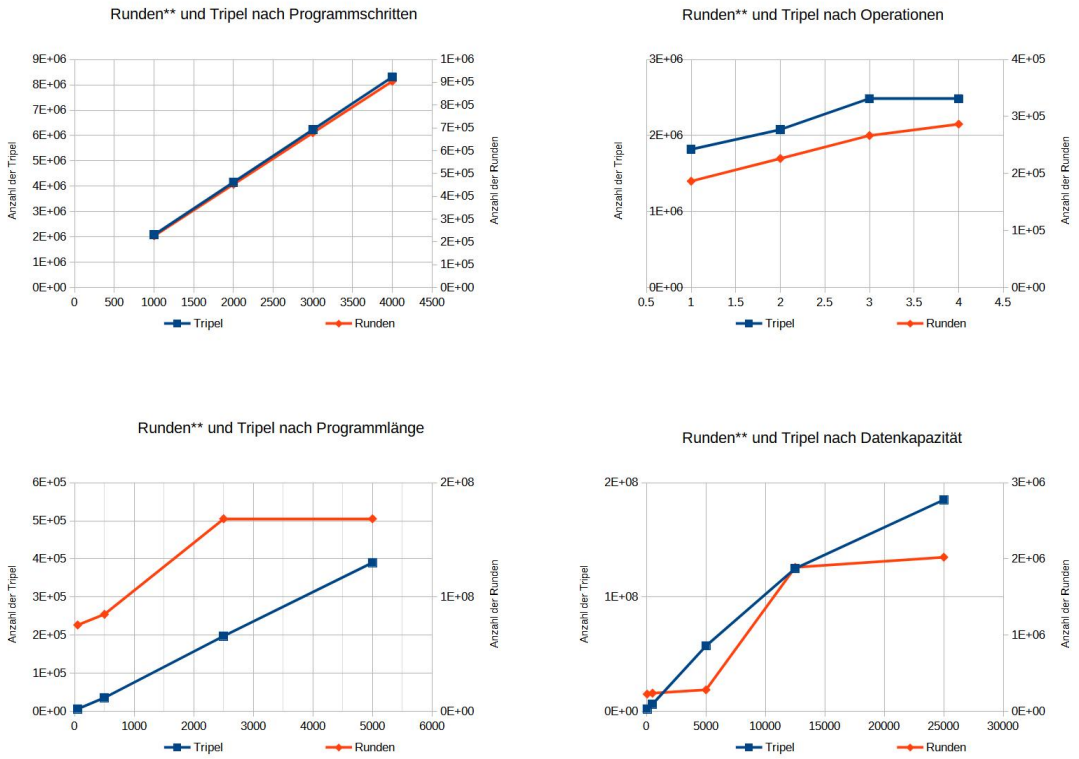


Abbildung 4.3: Vorberechnungen und Rundenkomplexität des Protokolls

Die Ausführungszeiten in Abbildung 4.2 sind hier weiter unterteilt in Aufwände zur Datenspeicherung (Zugriffe auf den Daten-ORAM), Programmspeicherung (Zugriffe auf den Programm-ORAM) und Befehlsauswertung. Es werden keine Vorberechnungs oder Kompilierzeiten betrachtet. Die Initialisierungen der ORAMs sowie Tests auf Abbruchbedingungen sind vernachlässigbar (« 1 sekunde). Die totale Zeit ist hier nur in der Größenordung interessant, da wie im nächsten Kapitel nochmals zusammengefasst einige konstante Verbesserungen im ORAM-Handling möglich sind. Es ist jedoch absehbar, dass die Entwicklung der Zeitkomplexität mit den zurzeit verfügbaren ORAM-Techniken nicht trivial verbessert werden kann, daher sind die zeitliche Aufteilungen der Berechnungen besonders in Relation beachtenswert. Die Evaluation bestätigt die theoretische Betrachtung, dass die ORAM Größe besonders kritisch zur Skalierung von PFE-Programmen ist und an effizienteren ORAM-Strukturen und einer größeren „Befehlsmächtigkeit “gearbeitet werden muss um die Praktikabilität zu erhöhen. Es ist also zu sehen, dass bei einem Versuch den Algrithmus zu skalieren immer die ORAM Zugriffe maßgeblich sind, hierbei ist anzumerken, dass MP-SPDZ die Wahl des ORAMs dynamisch adaptiv gestaltet, wodurch die Unregelmäßigkeiten in den Messungen zustande kommen, dadurch fällt es schwerer eine asymptotische Komplexität abzulesen.

Die Anzahl der vorberechneten Tripel, wie in Abbildung 4.3 dargestellt, operationalisiert den Vorberechnungsaufwand für Programme mit Offline-Phase, andere vorberechnete Masken (Bitzer-

** Rundenanzahl zt. approximativ gemessen

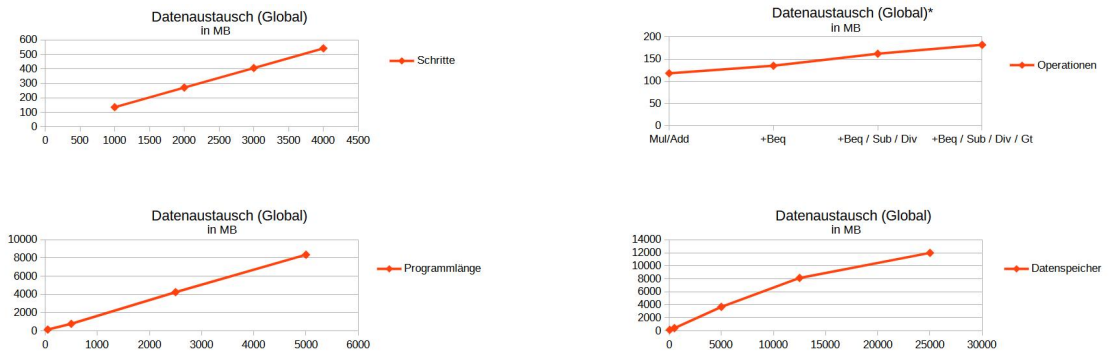


Abbildung 4.4: Datenaustausch des Protokolls

legung oder Inverse Masken) werden hier ignoriert, sind allerdings immer einfach aus dem Bedarf der Operationen zur Instruktionsevaluation abzuschätzen.

Die Anzahl der Runden stellt ein zuverlässiger Indikator für Zeitbedarf und Skalierbarkeit über andere Netzwerke (insbesondere WLAN und WAN) dar. Die Rundenanzahl wird z.T. heuristisch von der MP-SPDZ konfiguration beeinflusst und steht z.T. im Trade Off zur Bandbreite, die in dieser Evaluation global gemessen wurde. Man sieht in eine sehr gute Abbildung 4.4 asymptotische Entwicklung in allen vier Dimensionen, was wichtig ist, da die Bandbreite zunehmend in der Literatur als eine maßgebende Beschränkung von MPC, insbesondere bei PFE-Verfahren, diskutiert wird.

Insgesamt ist das Programm zwar durchführbar und reicht für simple Berechnungen, dennoch sind hohe statische Overheads und eine schlechte Skalierbarkeit des ORAMs problematisch, falls eine kontinuierliche und effiziente Anwendbarkeit gefordert ist. Die Evaluation bestätigt mit den gemessenen Werten eindeutig die theoretische Betrachtung aus Kapitel 3.

5 Ergebnisse und zukünftige Arbeiten

In dieser Arbeit wurden verschiedene ORAM-Strukturen und PFE-Konstruktionen, die dem Stand der Technik entsprechen, betrachtet und daraus ein eigenes PFE-Protokoll zur Auswertung komplexer arithmetischer Funktionen als Programme gebildet. In Abschnitt 4.3 wird deutlich, dass dieses Protokoll praktikabel und skalierbar, aber noch nicht effizient genug zum ständigen Einsatz in realen Anwendungen ist. Dabei ist klar die ORAM-Größe prohibitiv. Hier muss effektiv daran gearbeitet werden, die benötigten Grenzen klarer zu definieren und allgemein effizientere ORAMs zu finden. Dennoch zeigt dieses Protokoll, dass insbesondere aktive Sicherheit selbst mit einem simplen Schema zu erreichen ist.
Es wurde aber auch deutlich, dass der Preprocessing-Overhead in Form von genutzter Bandbreite und die Anzahl der Online-Kommunikationsrunden enorm ist. Eine Reduktion wird hierbei dringend benötigt. Außerdem wird klar, dass Addition und Multiplikation schnelle Basisoperationen sind, jedoch der ORAM-Overhead komplexere und zeitaufwändigere Operationssuites deutlich effizienter macht, wenn dadurch die Anzahl der Instruktionen reduziert werden kann.

Zukünfige Arbeiten

Es können die weiteren Optimierungs- und Erweiterungsmöglichkeiten in drei Obergruppen eingeteilt werden: (1) Verbesserung der unterliegenden kryptographischen Primitive, (2) Verbesserung der CPU-Step-Circuit Konstruktion, (3) Problemspezifische Verbesserungen. Im Folgenden wird jede Gruppe betrachtet und die schon in der Arbeit angesprochenen Möglichkeiten kurz zusammengefasst.

(1) **Verbesserung der unterliegenden kryptographischen Primitive**
Hierzu können allgemein verbesserte ORAMs, dynamisch anpassbare ORAMs, einen Möglichen Einsatz von Read-Only Instruction ORAM (z. B. FLOROM aus [12]) und eine bessere MPC Implementation zählen.
Zusätzlich sind aber auch andere Konstruktionen, die bspw. eine Trennung der Verschlüsselung von Befehlen und Daten beinhalten denkbar sein. Dabei wäre die Perspektive, dass man z. B. die geringe Circuit Tiefe der Instruction Evaluation Komponente nutzen könnte um SHE ohne Vorberechnungen zu nutzen. Dadurch würden zumindest die Anzahl der Kommunikationsrunden und die Limitiation in Bandbreite weniger kritisch werden. Auch wenn diese nur dadurch nicht komplett eliminiert werden können.
Es muss auch an möglichen weiteren Konstruktionen, die eine Reduktion der Online-Rundenzahl zum Ziel haben gearbeitet werden, da diese besonders für Evaluationen über *WAN* mit vielen Teilnehmern interessant sind.

(2) **Verbesserung der CPU-Step-Circuit Konstruktion**
Sie beschreiben hier besonders die schon in Abschnitt 4.2 beschriebenen heuristischen Design-Entscheidungen die in Befehlssatzarchitekturen getroffen werden. Hierbei ist eine erneute Betrachtung unter den Parametern der kryptographischen Primitive anstatt der normalen Parametern notwendig.

(3) **Problemspezifische Verbesserungen**
Die dritte Verbesserungsgruppe beschreibt hierbei allgemeine Änderungen am PFE-Schema. Diese können die genaueren Betrachtung dynamischer ORAMs mit heuristischen Parametern, der Auswahl geeigneter Abort-Bedingungen, oder der Eviction von kleineren in größere ORAM-Speicher zur Laufzeit sein. Auch die Verwendung verschiedener Operator-Suites zur Ausweitung der Befehlsmächtigkeit der einzelnen PFE-Instruktionen sollte sorgfältig durchdacht werden. Viele dieser Probleme können abhängig von dem zu optimierenden Indikator oder dem zu berechnenden Porgramm sein.

Literaturverzeichnis

[1] M. Abadi, J. Feigenbaum. „Secure circuit evaluation“. In: *Journal of Cryptology* 2.1 (1990), S. 1–12 (zitiert auf S. 37).

[2] M. Ajtai, J. Komlós, E. Szemerédi. „An 0(n Log n) Sorting Network“. In: *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*. STOC '83. New York, NY, USA: Association for Computing Machinery, 1983, S. 1–9. ISBN: 0897910990. DOI: 10.1145/800061.808726. URL: https://doi.org/10.1145/800061.808726 (zitiert auf S. 27).

[3] B. Applebaum, Y. Ishai, E. Kushilevitz. „How to Garble Arithmetic Circuits“. In: *Proceedings of the 2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*. FOCS '11. USA: IEEE Computer Society, 2011, S. 120–129. ISBN: 9780769545714. DOI: 10.1109/FOCS.2011.40. URL: https://doi.org/10.1109/FOCS.2011.40 (zitiert auf S. 55).

[4] G. Asharov, I. Komargodski, W.-K. Lin, K. Nayak, E. Peserico, E. Shi. „OptORAMa: Optimal Oblivious RAM“. In: *Advances in Cryptology–EUROCRYPT 2020* 12106 (), S. 403 (zitiert auf S. 26).

[5] K. E. Batcher. „Sorting Networks and Their Applications“. In: *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*. AFIPS '68 (Spring). Atlantic City, New Jersey: Association for Computing Machinery, 1968, S. 307–314. ISBN: 9781450378970. DOI: 10.1145/1468075.1468121. URL: https://doi.org/10.1145/1468075.1468121 (zitiert auf S. 27, 28).

[6] D. Beaver, S. Micali, P. Rogaway. „The round complexity of secure protocols“. In: *Proceedings of the twenty-second annual ACM symposium on Theory of computing*. 1990, S. 503–513 (zitiert auf S. 35).

[7] O. Biçer, M. A. Bingol, M. S. Kiraz, A. Levi. *Towards Practical PFE: An Efficient 2-Party Private Function Evaluation Protocol Based on Half Gates*. Cryptology ePrint Archive, Report 2017/415. https://ia.cr/2017/415. 2017 (zitiert auf S. 29, 37, 41).

[8] O. Biçer, M. A. Bingöl, M. S. Kiraz, A. Levi. „Highly Efficient and Re-Executable Private Function Evaluation With Linear Complexity“. In: *IEEE Transactions on Dependable and Secure Computing* 19.2 (2022), S. 835–847. DOI: 10.1109/TDSC.2020.3009496 (zitiert auf S. 34, 37, 42, 43).

[9] R. Canetti. „Universally composable security: a new paradigm for cryptographic protocols“. In: *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*. 2001, S. 136–145. DOI: 10.1109/SFCS.2001.959888 (zitiert auf S. 33).

[10] K.-M. Chung, R. Pass. *A Simple ORAM*. Cryptology ePrint Archive, Report 2013/243. https://ia.cr/2013/243. 2013 (zitiert auf S. 23, 25, 26).

[11] I. Damgård, V. Pastro, N. Smart, S. Zakarias. „Multiparty Computation from Somewhat Homomorphic Encryption“. In: *Advances in Cryptology – CRYPTO 2012*. Hrsg. von R. Safavi-Naini, R. Canetti. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, S. 643–662. ISBN: 978-3-642-32009-5 (zitiert auf S. 59).

[12] J. Doerner, abhi shelat. *Scaling ORAM for Secure Computation*. Cryptology ePrint Archive, Paper 2017/827. https://eprint.iacr.org/2017/827. 2017. DOI: 10.1145/3133956.3133967. URL: https://eprint.iacr.org/2017/827 (zitiert auf S. 25, 67).

[13] 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 (zitiert auf S. 13, 42).

[14] S. Faber, S. Jarecki, S. Kentros, B. Wei. „Three-Party ORAM for Secure Computation“. In: *Advances in Cryptology – ASIACRYPT 2015*. Hrsg. von T. Iwata, J. H. Cheon. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, S. 360–385. ISBN: 978-3-662-48797-6 (zitiert auf S. 26).

[15] C. Gentry, K. A. Goldman, S. Halevi, C. Julta, M. Raykova, D. Wichs. „Optimizing ORAM and Using It Efficiently for Secure Computation“. In: *Privacy Enhancing Technologies*. Hrsg. von E. De Cristofaro, M. Wright. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, S. 1–18. ISBN: 978-3-642-39077-7 (zitiert auf S. 24).

[16] C. Gentry, S. Halevi, S. Lu, R. Ostrovsky, M. Raykova, D. Wichs. „Garbled RAM Revisited“. In: *Advances in Cryptology – EUROCRYPT 2014*. Hrsg. von P. Q. Nguyen, E. Oswald. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, S. 405–422. ISBN: 978-3-642-55220-5 (zitiert auf S. 3, 44).

[17] O. Goldreich. „Towards a Theory of Software Protection and Simulation by Oblivious RAMs“. In: *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*. STOC '87. New York, New York, USA: Association for Computing Machinery, 1987, S. 182–194. ISBN: 0897912217. DOI: 10.1145/28395.28416. URL: https://doi.org/10.1145/28395.28416 (zitiert auf S. 12, 60).

[18] O. Goldreich, R. Ostrovsky. „Software Protection and Simulation on Oblivious RAMs“. In: *J. ACM* 43.3 (Mai 1996), S. 431–473. ISSN: 0004-5411. DOI: 10.1145/233551.233553. URL: https://doi.org/10.1145/233551.233553 (zitiert auf S. 17, 18, 20, 21).

[19] M. T. Goodrich, M. Mitzenmacher. „Privacy-preserving access of outsourced data via oblivious RAM simulation“. In: *International Colloquium on Automata, Languages, and Programming*. Springer. 2011, S. 576–587 (zitiert auf S. 26).

[20] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, R. Tamassia. „Privacy-Preserving Group Data Access via Stateless Oblivious RAM Simulation“. In: *Proceedings of the 2012 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, S. 157–167. DOI: 10.1137/1.9781611973099.14. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781611973099.14. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781611973099.14 (zitiert auf S. 26).

[21] L. Harn, C. Lin. „Detection and identification of cheaters in (t, n) secret sharing scheme“. In: *Designs, Codes and Cryptography* 52.1 (2009), S. 15–24 (zitiert auf S. 14).

[22] M. Holz, Á. Kiss, D. Rathee, T. Schneider. „Linear-Complexity Private Function Evaluation is Practical“. In: Sep. 2020, S. 401–420. ISBN: 978-3-030-59012-3. DOI: 10.1007/978-3-030-59013-0_20 (zitiert auf S. 39).

[23] C. Hong, J. Katz, V. Kolesnikov, W.-j. Lu, X. Wang. „Covert security with public verifiability: Faster, leaner, and simpler“. In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2019, S. 97–121 (zitiert auf S. 33).

[24] J. Katz, L. Malka. *Constant-Round Private Function Evaluation with Linear Complexity*. Cryptology ePrint Archive, Report 2010/528. https://ia.cr/2010/528. 2010 (zitiert auf S. 32, 33, 38, 39).

[25] M. Keller. „MP-SPDZ: A Versatile Framework for Multi-Party Computation“. In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 2020. DOI: 10.1145/3372297.3417872. URL: https://doi.org/10.1145/3372297.3417872 (zitiert auf S. 59, 60).

[26] M. Keller, E. Orsini, P. Scholl. „MASCOT: Faster Malicious Arithmetic Secure Computation with Oblivious Transfer“. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. Vienna, Austria: Association for Computing Machinery, 2016, S. 830–842. ISBN: 9781450341394. DOI: 10.1145/2976749.2978357. URL: https://doi.org/10.1145/2976749.2978357 (zitiert auf S. 56, 60).

[27] M. Keller, P. Scholl. *Efficient, Oblivious Data Structures for MPC*. Cryptology ePrint Archive, Paper 2014/137. https://eprint.iacr.org/2014/137. 2014. URL: https://eprint.iacr.org/2014/137 (zitiert auf S. 59, 60).

[28] M. Keller, A. Yanai. *Efficient Maliciously Secure Multiparty Computation for RAM*. Cryptology ePrint Archive, Paper 2017/981. https://eprint.iacr.org/2017/981. 2017. URL: https://eprint.iacr.org/2017/981 (zitiert auf S. 18, 44, 59).

[29] Á. Kiss, T. Schneider. „Valiant's universal circuit is practical“. In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2016, S. 699–728 (zitiert auf S. 16).

[30] K. G. Kogos, K. S. Filippova, A. V. Epishkina. „Fully homomorphic encryption schemes: The state of the art“. In: *2017 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)*. 2017, S. 463–466. DOI: 10.1109/EIConRus.2017.7910591 (zitiert auf S. 13).

[31] H. Lipmaa, P. Mohassel, S. Sadeghian. *Valiant's Universal Circuit: Improvements, Implementation, and Applications*. Cryptology ePrint Archive, Report 2016/017. https://ia.cr/2016/017. 2016 (zitiert auf S. 16, 35).

[32] Y. Liu, Q. Wang, S.-M. Yiu. *Making Private Function Evaluation Safer, Faster, and Simpler*. Cryptology ePrint Archive, Report 2021/1682. https://ia.cr/2021/1682. 2021 (zitiert auf S. 34, 37, 42).

[33] S. Lu, R. Ostrovsky. „Distributed Oblivious RAM for Secure Two-Party Computation“. In: *Proceedings of the 10th Theory of Cryptography Conference on Theory of Cryptography*. TCC'13. Tokyo, Japan: Springer-Verlag, 2013, S. 377–396. ISBN: 9783642365935. DOI: 10.1007/978-3-642-36594-2_22. URL: https://doi.org/10.1007/978-3-642-36594-2_22 (zitiert auf S. 18, 26).

[34] S. Lu, R. Ostrovsky. *Garbled RAM Revisited, Part II*. Cryptology ePrint Archive, Paper 2014/083. https://eprint.iacr.org/2014/083. 2014. URL: https://eprint.iacr.org/2014/083 (zitiert auf S. 3, 44).

[35] S. Lu, R. Ostrovsky. „How to Garble RAM Programs?“ In: *Advances in Cryptology – EUROCRYPT 2013*. Hrsg. von T. Johansson, P. Q. Nguyen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, S. 719–734. ISBN: 978-3-642-38348-9 (zitiert auf S. 3, 43, 54).

[36] P. Mohassel, S. Sadeghian. „How to Hide Circuits in MPC an Efficient Framework for Private Function Evaluation“. In: *Advances in Cryptology – EUROCRYPT 2013*. Hrsg. von T. Johansson, P. Q. Nguyen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, S. 557–574. ISBN: 978-3-642-38348-9 (zitiert auf S. 28, 29, 36, 39, 41, 45).

[37] P. Mohassel, S. Sadeghian, N. P. Smart. „Actively Secure Private Function Evaluation“. In: *Advances in Cryptology – ASIACRYPT 2014*. Hrsg. von P. Sarkar, T. Iwata. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, S. 486–505. ISBN: 978-3-662-45608-8 (zitiert auf S. 29, 37, 40, 45).

[38] P. Paillier. „Public-key cryptosystems based on composite degree residuosity classes“. In: *International conference on the theory and applications of cryptographic techniques*. Springer. 1999, S. 223–238 (zitiert auf S. 13).

[39] A. Shamir. „How to Share a Secret“. In: *Commun. ACM* 22.11 (Nov. 1979), S. 612–613. ISSN: 0001-0782. DOI: 10.1145/359168.359176. URL: https://doi.org/10.1145/359168.359176 (zitiert auf S. 14).

[40] E. Shi, T.-H. Chan, E. Stefanov, M. Li. „Oblivious RAM with O((log N) 3) Worst-Case Cost“. In: Bd. 2011. Dez. 2011, S. 197–214. ISBN: 978-3-642-25384-3. DOI: 10.1007/978-3-642-25385-0_11 (zitiert auf S. 21, 25, 26, 52, 60).

[41] E. Stefanov, M. van Dijk, E. Shi, T.-H. H. Chan, C. Fletcher, L. Ren, X. Yu, S. Devadas. *Path ORAM: An Extremely Simple Oblivious RAM Protocol*. 2012. DOI: 10.48550/ARXIV.1202.5150. URL: https://arxiv.org/abs/1202.5150 (zitiert auf S. 23, 25, 60).

[42] X. Wang, H. Chan, E. Shi. „Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound“. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. CCS '15. Denver, Colorado, USA: Association for Computing Machinery, 2015, S. 850–861. ISBN: 9781450338325. DOI: 10.1145/2810103.2813634. URL: https://doi.org/10.1145/2810103.2813634 (zitiert auf S. 23, 25).

[43] A. C.-C. Yao. „How to generate and exchange secrets“. In: *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*. 1986, S. 162–167. DOI: 10.1109/SFCS.1986.25 (zitiert auf S. 11, 15, 16, 37, 40, 44).

[44] S. Zahur, X. Wang, M. Raykova, A. Gascón, J. Doerner, D. Evans, J. Katz. „Revisiting Square-Root ORAM: Efficient Random Access in Multi-party Computation“. In: *2016 IEEE Symposium on Security and Privacy (SP)*. 2016, S. 218–234. DOI: 10.1109/SP.2016.21 (zitiert auf S. 26).

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