HOCHSCHULE MITTWEIDA
University of Applied Sciences

MASTERARBEIT

Herr BSc.
Alex Kemloh Kouyem

Design und Implementierung vertraulicher Transaktionen für ERC20-Token. Lösungen zum Schutz der Privatsphäre für tokenisierte Assets auf der Blockchain Ethereum

Mittweida, Dezember 2023

Fakultät **Angewandte Computer- und Biowissenschaften**

MASTERARBEIT

Design und Implementierung vertraulicher Transaktionen für ERC20-Token. Lösungen zum Schutz der Privatsphäre für tokenisierte Assets auf der Blockchain Ethereum

Autor:
Alex Kemloh Kouyem

Studiengang:
Blockchain & Distributed Ledger Technologies

Seminargruppe:
BC21w1-M

Erstprüfer:
Prof. Dr.-Ing. Andreas Ittner

Zweitprüferin:
Dipl.Volkswirt Mario Oettler

Einreichung:
Mittweida, 08.12.2023

Verteidigung/Bewertung:
Mittweida, 2024

Faculty of **Applied Computer Sciences and Biosciences**

MASTER THESIS

Design and implementation of confidential transactions for ERC20 tokens: privacy protection solutions for tokenized assets on the Ethereum blockchain

Author:
Alex Kemloh Kouyem

Course of Study:
Applied Computer Science

Seminar Group:
BC21w1-M

First Examiner:
Prof. Dr.-Ing. Andreas Ittner

Second Examiner:
Dipl.Volkswirt Mario Oettler

Submission:
Mittweida, 08.12.2023

Defense/Evaluation:
Mittweida, 2024

Bibliografische Beschreibung

Kemloh Kouyem, Alex:
Design und Implementierung vertraulicher Transaktionen für ERC20-Token. Lösungen zum Schutz der Privatsphäre für tokenisierte Assets auf der Blockchain Ethereum. – 2023. – 39 S.
Mittweida, Hochschule Mittweida – University of Applied Sciences, Fakultät Angewandte Computer- und Biowissenschaften, Masterarbeit, 2024.

Referat

Diese Arbeit präsentiert ein Protokoll für vertrauliche Transaktionen auf Ethereum, das auf einer kontenbasierten Struktur und Paillier-Verschlüsselung basiert. Die Integration von Non-Interactive Zero-Knowledge Range Proofs (NIZKRP) verbessert die Sicherheit. Die Implementierung und Tests auf Ethereum zeigen vergleichbare Transaktionskosten (Sicherheitsparameter 40) im Vergleich zu Protokollen mit Bulletproofs. Bei einem Sicherheitsparameter von 128 (NIZKRP-Empfehlung) ist das Protokoll jedoch nicht anwendbar. Die Arbeit betont die Effizienz und Wettbewerbsfähigkeit, hebt jedoch die Herausforderung bei höheren Sicherheitsparametern hervor. Das Protokoll bildet eine solide Grundlage, erfordert jedoch weitere Optimierungen für breitere Anwendbarkeit.

Abstract

This work introduces a protocol for confidential transactions on Ethereum, leveraging an account-based structure and Paillier encryption for ensuring confidentiality of account balances. The integration of Non-Interactive Zero-Knowledge Range Proofs (NIZKRP) enhances security. Implementation and tests on Ethereum reveal comparable transaction costs (security parameter 40) to protocols utilizing Bulletproofs. However, the protocol is inapplicable with a security parameter of 128 (NIZKRP recommendation). While emphasizing efficiency and competitiveness, the work underscores the challenge at higher security parameters. The protocol establishes a robust foundation, necessitating further optimizations for broader applicability.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Danksagung

Ich möchte mich herzlich bei meiner Familie, meinen Eltern, Geschwistern und Freunden bedanken, die mich während des gesamten Prozesses der Erstellung meiner Masterarbeit unterstützt und ermutigt haben. Ihr uneingeschränktes Vertrauen und eure Unterstützung haben einen großen Beitrag zu meiner Motivation und meinem Erfolg geleistet.

Ein besonderer Dank gilt auch meinem Betreuer, Dipl.Volkswirt Mario Oettler, für seine engagierte Betreuung, wertvolle Ratschläge und die konstruktive Kritik, die meine Arbeit maßgeblich verbessert hat. Sein Fachwissen und seine Unterstützung haben mir wertvolle Einblicke in das Forschungsfeld ermöglicht.

Vielen Dank an alle, die dazu beigetragen haben, dass diese Masterarbeit zu einem erfolgreichen Abschluss geführt hat. Eure Unterstützung hat diesen Meilenstein in meinem akademischen Werdegang erst möglich gemacht.

1 Einleitung

In den letzten Jahren hat die zunehmende Verbreitung der Blockchain-Technologie zu einer verstärkten Tokenisierung von Vermögenswerten geführt, insbesondere durch die Einführung von ERC-20-Token auf der Ethereum-Blockchain. Dieser Standard hat die Entwicklung von Kryptowährungen und dezentralen Anwendungen (DApps) erleichtert und die Interoperabilität sowie den Handel zwischen verschiedenen Token ermöglicht.

Die Flexibilität des ERC-20-Standards ermöglicht es, ihn an spezifische Anforderungen anzupassen, solange grundlegende Funktionen beibehalten werden. Während ERC-20-Token die Verbreitung von Kryptowährungen und DeFi-Anwendungen vorangetrieben haben, sind damit auch Herausforderungen im Bereich der Privatsphäre entstanden.

Eine der zentralen Herausforderungen bei tokenisierten Vermögenswerten, insbesondere ERC-20-Token, ist der Mangel an Privatsphäre bei Transaktionen. Da alle Transaktionen auf der Ethereum-Blockchain öffentlich sichtbar sind, ergeben sich Datenschutzbedenken bezüglich Transaktionsbetrag, Absender- und Empfängeradressen.

Die Zielsetzung dieser Arbeit besteht darin, ein durchdachtes Design für vertrauliche Transaktionen für tokenisierte Vermögenswerten zu entwickeln und umzusetzen. Hierbei liegt der Fokus auf der Berücksichtigung von Sicherheitsaspekten und der Lösung potenzieller Herausforderungen. Die erfolgreiche Umsetzung verspricht nicht nur ein vertieftes Verständnis der technologischen Implementierung, sondern auch eine Steigerung der praktischen Anwendbarkeit und Akzeptanz in der Blockchain-Gemeinschaft.

Die Forschungsfragen konzentrieren sich auf die Auswahl geeigneter Technologien für vertrauliche Transaktionen für tokenisierte Vermögenswerten, die Minimierung von Sicherheitsrisiken und die erfolgreiche Integration des Designs in die bestehende Infrastruktur. Die Hypothese postuliert, dass die Einführung vertraulicher Transaktionen einen wesentlichen Beitrag zur Stärkung der Privatsphäre in der Blockchain-Welt leisten kann. In den folgenden Abschnitten werden das Design und die Implementierung vertraulicher ERC-20-Transaktionen näher beleuchtet.

Die Struktur dieser Arbeit ist wie folgt gestaltet: Im zweiten Kapitel wird eine Einführung in die Blockchain gegeben, wobei der Schwerpunkt auf dem ERC-20-Token-Standard liegt. Das darauf folgende dritte Kapitel beschäftigt sich mit einer detaillierten Darstellung von Technologien im Bereich vertraulicher Transaktionen. Anschließend werden im vierten Kapitel einige bereits existierende Lösungen für vertrauliche Transaktionen analysiert. Das Design des im Rahmen dieser Arbeit entwickelten Protokolls wird im fünften Kapitel präsentiert, gefolgt von der tatsächlichen Implementierung im sechsten Kapitel. Im siebten Kapitel erfolgt die Evaluation des Protokolls und es werden Ausblicke für mögliche zukünftige Optimierungen gegeben. Abschließend fasst das achte Kapitel die Erkenntnisse dieser Arbeit zusammen.

2 Blockchain: Grundlagen

In diesem Kapitel werden die fundamentalen Prinzipien der Blockchain-Technologie vermittelt, die als Grundlage für das Verständnis dieser Arbeit dienen. Wir beginnen mit einer allgemeinen Einführung und untersuchen anschließend die vielfältigen Kategorien von Blockchains. Darüber hinaus richten wir unseren Fokus auf die Ethereum-Blockchain. Zum Abschluss werfen wir einen detaillierten Blick auf den ERC-20-Token-Standard, der in dieser Arbeit eine entscheidende Rolle spielt.

2.1 Übersicht

Die Blockchain ist eine dezentrale digitale Datenbank oder ein verteiltes Ledger, das Transaktionen und Daten in Blöcken speichert. Diese Blöcke sind miteinander verknüpft und bilden eine chronologische Kette, daher der Name `Blockchain`. Die Blockchain-Technologie wurde 2028 [1] erstmals im Zusammenhang mit der Kryptowährung Bitcoin eingeführt, hat jedoch seitdem Anwendungsfelder in verschiedenen Bereichen gefunden.

Hier sind die drei grundlegenden Merkmale und Konzepte der Blockchain:

1. Dezentralisierung: Die Blockchain wird von einem Netzwerk von Computern oder Knoten betrieben, anstatt von einer zentralen Behörde oder Institution. Dies bedeutet, dass keine einzelne Partei die volle Kontrolle über die Datenbank hat.
2. Transparenz: Die Informationen in der Blockchain sind für alle Teilnehmer des Netzwerks sichtbar und können in Echtzeit überprüft werden. Dies erhöht das Vertrauen, da Transaktionen öffentlich und nachvollziehbar sind.
3. Unveränderlichkeit: Einmal in die Blockchain geschriebene Daten können nicht rückgängig gemacht oder geändert werden. Dies macht die Blockchain zu einer sicheren Möglichkeit, Transaktionen und Aufzeichnungen zu speichern.

Die Blockchain bietet eine große Sicherheit. Durch die Verwendung von fortschrittlichen kryptographischen Techniken wie Hashfunktion und digitale Signatur werden die Integrität und Sicherheit der Daten gewährleistet. Zudem verwenden Blockchains verschiedene Konsensmachanismen, um sicherzustellen, dass alle Knoten im Netzwerk zu einer Einigung über den Zustand der Blockchain gelangen. Bitcoin verwendet beispielsweise Proof-of-Work (PoW), während Ethereum zu Proof-of-Stake (PoS) übergegangen ist.

Blockchain kann weit über Kryptowährungen hinaus verwendet werden. Sie findet Anwendung in Bereichen wie Finanzwesen, Lieferkettenmanagement, Gesundheitswesen, Abstimmungen, Urheberrechtsverwaltung und mehr. Einige Blockchains, wie Ethereum, ermöglichen die Ausführung von Smart Contracts. Das sind selbstausführende Verträge, die automatisch ausgeführt werden, wenn vordefinierte Bedingungen erfüllt sind.

Insgesamt ist die Blockchain-Technologie eine innovative Methode zur Speicherung und Verwaltung von Daten, die das Potenzial hat, viele Branchen zu transformieren und neue Möglichkeiten für Transparenz, Sicherheit und Effizienz zu bieten.

2.2 Arte von Blockchains

Es gibt verschiedene Arten von Blockchains, die sich in ihrer Struktur, ihrem Zweck und ihren Zugriffsrechten unterscheiden. Hier sind einige der wichtigsten Arten von Blockchains:

1. Öffentliche Blockchains: Öffentliche Blockchains sind für jeden zugänglich und transparent. Jeder kann Transaktionen einsehen und validieren. Das bekannteste Beispiel ist die Bitcoin-Blockchain. Ethereum ist ein weiteres Beispiel für eine öffentliche Blockchain, die auch Smart Contracts unterstützt.
2. Private Blockchains: Private Blockchains sind auf bestimmte Benutzer oder Organisationen beschränkt. Sie bieten mehr Kontrolle über die Teilnehmer und die Zugriffsrechte. Diese Art von Blockchain wird oft in Unternehmen und Konsortien verwendet, um interne Prozesse zu optimieren. Hyperledger Fabric ist ein Beispiel für eine private Blockchain-Plattform.
3. Konsortiumsblockchains: Konsortiumsblockchains liegen zwischen öffentlichen und privaten Blockchains. Hier arbeiten mehrere Organisationen zusammen, um die Blockchain gemeinsam zu verwalten. Diese Art von Blockchain bietet einige der Vorteile einer öffentlichen Blockchain, während gleichzeitig die Teilnehmer begrenzt sind.
4. Hybride Blockchains: Hybride Blockchains kombinieren Elemente von öffentlichen und privaten Blockchains. Ein Teil der Blockchain kann öffentlich zugänglich sein, während ein anderer Teil privat ist. Diese Art von Blockchain kann in Fällen nützlich sein, in denen Transparenz und Privatsphäre gleichzeitig erforderlich sind.
5. Ethereum-ähnliche Blockchains: Ethereum hat eine Vielzahl von Blockchain-Projekten inspiriert, die ähnliche Funktionen und Smart Contract-Fähigkeiten bieten. Diese werden oft als `Ethereum-ähnliche` oder `Ethereum-kompatible` Blockchains bezeichnet.
6. Blockchain-Plattformen: Einige Unternehmen bieten Blockchain-Plattformen an, die es anderen ermöglichen, ihre eigenen Blockchains zu erstellen und zu verwalten. Beispiele hierfür sind Binance Smart Chain und Polkadot.
7. Interoperable Blockchains: Diese Art von Blockchains arbeiten daran, die Interoperabilität zwischen verschiedenen Blockchains zu ermöglichen. Sie sollen den Datenaustausch und die Kommunikation zwischen Blockchains erleichtern.
8. Sidechains: Sidechains sind eigenständige Blockchains, die an eine Hauptblockchain, wie Bitcoin oder Ethereum, angeschlossen sind. Sie sollen die Skalierbarkeit und die Entwicklung neuer Funktionen ermöglichen, ohne die Hauptblockchain zu überlasten.

2.3 Die Blockchain Ethereum

Die Ethereum-Blockchain ist eine der bekanntesten und am häufigsten verwendeten Blockchain-Plattformen. Sie wurde entwickelt, um weit mehr als nur Kryptowährungen zu unterstützen und hat die Art und Weise, wie Entwickler dezentrale Anwendungen (DApps) erstellen, revolutioniert. Ethereum wurde 2015 von Vitalik Buterin und einem Team von Entwicklern ins Leben gerufen[2]. Es wurde entwickelt, um die Idee der Blockchain weit über Bitcoin hinaus voranzutreiben, indem es die Möglichkeit bietet, intelligente Verträge (Smart Contracts) auszuführen.

Ethereum ist besonders bekannt für seine Unterstützung von Smart Contracts. Diese sind selbstausführende Verträge, die auf der Blockchain ausgeführt werden können, sobald vordefinierte Bedingungen erfüllt sind. Smart Contracts eröffnen Anwendungsfällen jenseits von Kryptowährungen, wie beispielsweise dezentralen Finanzdienstleistungen (DeFi), NFTs (Non-Fungible Tokens), und mehr.

Die native Kryptowährung von Ethereum wird Ether (ETH) genannt und wird für Transaktionsgebühren und zur Ausführung von Smart Contracts verwendet. Es ist auch eine der führenden Kryptowährungen nach Marktkapitalisierung.

Ähnlich wie Bitcoin basiert auch Ethereum auf einer dezentralen Struktur, bei der Transaktionen von einem Netzwerk von Knoten validiert werden. Dies macht die Ethereum-Blockchain widerstandsfähig gegen Zensur und Angriffe.

Ethereum arbeitet an einem Upgrade namens Ethereum 2.0 (Eth2), das die Skalierbarkeit und die Energieeffizienz der Plattform verbessern soll. Eth2 führt die sogenannte Proof-of-Stake (PoS)-Konsensmechanismus ein, um Transaktionen zu validieren und neue Blöcke hinzuzufügen.

Ethereum bietet eine Vielzahl an Token Standards, wobei ERC-20 der am häufigsten verwendete Token-Standard ist. Er definiert die Regeln, wie neue Token erstellt und Transaktionen zwischen Token-Haltern durchgeführt werden können. Dies hat die Entwicklung von Tausenden von verschiedenen Token auf Ethereum ermöglicht.

Ethereum ist die Heimat einer wachsenden Anzahl von dezentralen Anwendungen, die auf Smart Contracts basieren. Diese reichen von DeFi-Protokollen und digitalen Kunstwerken bis hin zu Spielen und Identitätslösungen.

Ethereum verfügt über eine große und aktive Entwicklergemeinschaft, die laufend an der Verbesserung und Erweiterung der Plattform arbeitet. Dies hat zu zahlreichen Upgrades und Verbesserungen geführt.

Ethereum hat die Blockchain-Landschaft maßgeblich geprägt und ist heute eine der führenden Plattformen für die Entwicklung von dezentralen Anwendungen und Smart Contracts. Die Ethereum-Community treibt ständig die Innovation voran, um die Plattform weiter zu verbessern.

2.4 ERC20-Token-Standard

ERC-20 wurde 2015 eingeführt und ist ein technischer Standard für fungible Token auf der Ethereum-Blockchain[3]. Fungibilität bedeutet, dass ERC20-Token untereinander austauschbar sind. Das bedeutet, dass ein ERC-20-Token im Wert und in der Funktion identisch mit einem anderen Token des gleichen Typs ist. Zum Beispiel ist ein einzelner ERC-20-Token A im Wert von 10 USD genauso viel wert wie ein anderer Token A im Wert von 10 USD.

ERC steht für Ethereum Request for Comments, und die Zahl 20 ist die eindeutige Identifikationsnummer, die diesem Token-Standard zugeordnet ist. Der ERC-20-Standard definiert die Regeln und Funktionen, die ein Token auf Ethereum haben sollte, um eine reibungslose Interoperabilität zwischen verschiedenen Anwendungen und Plattformen zu gewährleisten (siehe Abb 2.1).

ERC ist eine Reihe von Standards und Vorschlägen, die von der Ethereum-Community entwickelt wurden, um die Interoperabilität und Kompatibilität von Token und Smart Contracts auf der Ethereum-Blockchain sicherzustellen. Diese Standards dienen als Richtlinien und Spezifikationen für Entwickler, die auf Ethereum basierende Anwendungen erstellen[4].

```
interface IERC20 {
    event Transfer(address indexed _from, address indexed _to, uint256 _value);
    event Approval(address indexed _owner, address indexed _spender, uint256 _value);

    function name() external view returns (string memory);
    function symbol() external view returns (string memory);
    function decimals() external view returns (uint8);
    function totalSupply() external view returns (uint256);
    function balanceOf(address _owner) external view returns (uint256 balance);
    function transfer(address _to, uint256 _value) external returns (bool success);
    function transferFrom(address _from, address _to, uint256 _value) external returns (bool success);
    function approve(address _spender, uint256 _value) external returns (bool success);
    function allowance(address _owner, address _spender) external view returns (uint256 remaining);
}
```

Abbildung 2.1: ERC20 Schnittstelle
Quelle: Eigene Bilder

Jedes ERC-20-Token hat einen Namen, ein Symbol und eine Anzahl von Dezimalstellen, die die Genauigkeit der Token angeben. Ein Token kann zum Beispiel den Namen `Confidential Transactions` tragen, das Symbol `CT` haben und 18 Dezimalstellen zur Darstellung von Teilungen darstellen.
Der Standard ermöglicht es, Token von einer Ethereum-Adresse zur anderen zu übertragen. Die Transaktionen sind standardisiert und folgen einem einheitlichen Muster. Es ist auch eine Funktion definiert, mit der Benutzer ihre Token-Bilanz abfragen können.
Der Standard definiert zudem den Gesamtvorrat an Token, der jemals erstellt und in Umlauf gebracht werden kann. Einfacher ausgedrückt handelt es sich um die Gesamtzahl der vorhandenen Token. Der Wert des Gesamtvorrates ist eine wichtige Metrik und wird in der Regel bei der Erstellung des Token-Smart-Contract festgelet und kann nicht nachträglich geändert werden.
Der Standard ermöglicht es, Ereignisse zu definieren, die es DApps und Benutzern ermöglichen, auf wichtige Token-Transaktionen zu reagieren.

Der ERC-20-Standard beinhaltet eine Funktion namens [allowance], die es ermöglicht, dass ein Ethereum-Konto (normalerweise ein Smart Contract) im Namen eines Token-Inhabers Token an eine andere Adresse (normalerweise ein weiterer Smart Contract) senden kann. Dieser Mechanismus wird häufig in DeFi und DApps auf der Ethereum-Blockchain verwendet. Hier sind die wichtige Merkmale und Aspekte dieser Funktion:

1. Approval: Ein Token-Inhaber, der beabsichtigt, einem anderen Ethereum-Konto die Erlaubnis zu erteilen, Token in seinem Namen zu übertragen, muss die *approve*-Funktion aufrufen. Diese Funktion verlangt zwei Hauptparameter:
    - spender: Die Adresse des Ethereum-Kontos, dem die Erlaubnis erteilt wird, Token zu übertragen
    - value: Die Anzahl der Token, die dem Spender erlaubt sind

    Zum Beispiel kann ein Token-Inhaber einem DeFi-Smart Contract die Erlaubnis erteilen, 1.000 Tokens in seinem Namen zu übertragen.
2. *TransferFrom()*: Nachdem die Erlaubnis erteilt wurde, kann der Smart Contract (der Spender) Token im Namen des Token-Inhabers an einen anderen Empfänger übertragen. Diese Funktion erfordert:
    - from: Die Adresse des Token-Inhabers

- to: Die Adresse des Empfängers
- value: Die Anzahl der zu übertragenden Token

Der Smart Contract, der Token im Namen des Token-Inhabers überträgt, muss die 'transferFrom'-Funktion verwenden und sicherstellen, dass die Anzahl der Token innerhalb der zuvor erteilten Erlaubnisgrenze (allowance) liegt.

3. Verwaltung der Erlaubnis: Der Token-Inhaber kann die Erlaubnis jederzeit widerrufen oder ändern, indem er die *approve*-Funktion erneut aufruft. Das bedeutet, dass der Inhaber die Menge an Token, die dem Smart Contract zur Verfügung steht, ändern kann.

Der Einsatz von *allowance* bietet die Möglichkeit, Token-Inhabern mehr Kontrolle über ihre Token zu geben und gleichzeitig DApps und Smart Contracts zu ermöglichen, im Namen des Token-Inhabers bestimmte Aktionen durchzuführen, wie zum Beispiel Kreditvergabe, Liquiditätsbereitstellung oder den Handel auf dezentralen Börsen.

Es ist wichtig zu beachten, dass der *approve*-Mechanismus einige Sicherheitsrisiken birgt, wenn er nicht sorgfältig implementiert wird. Smart Contracts und Entwickler sollten sicherstellen, dass die Token-Nutzung und die Verwaltung von 'allowance' sicher und vertrauenswürdig sind, um Missbrauch oder Verlust von Token zu verhindern.

3 Technologien im Bereich vertraulicher Transaktionen

In diesem Kapitel erfolgt eine detaillierte Analyse verschiedener Technologien im Bereich vertraulicher Transaktionen auf der Ethereum-Blockchain. Der Schwerpunkt liegt darauf, einen umfassenden Einblick in die bestehenden Ansätze und Lösungen zu geben, die darauf abzielen, Transaktionen auf der Ethereum-Plattform vertraulicher zu gestalten. Dabei werden sowohl etablierte als auch innovative Technologien berücksichtigt, um die Bandbreite der verfügbaren Ansätze und deren jeweilige Vor- und Nachteile zu verstehen. Diese Analyse bildet die Grundlage für das im Rahmen dieser Arbeit zu entwickelnde Protokoll

3.1 Stealth-Adressen

Stealth-Adressen repräsentieren eine Mechanismus, der entwickelt wurde, um die Privatsphäre der Empfänger von Kryptowährungstransaktionen zu wahren. Ihr Fundament beruht auf ERC-5564 und ermöglicht dem Absender die nicht-interaktive Generierung einer neuen Adresse für den Empfänger[5]. Dieser Prozess erzeugt den Anschein, als hätte der Absender mit einer zufälligen Kontoadresse interagiert. Die Stealth-Adressen selbst sind reguläre Ethereum-Adressen, die vom Absender auf nicht-interaktive Weise generiert werden. Dabei wird die Gewissheit angestrebt, dass ausschließlich der beabsichtigte Empfänger auf diese Adressen zugreifen kann. Diese Methode erlaubt dem Absender, die direkte Interaktion mit einer spezifischen Entität, wie zum Beispiel "xyz.eth", zu vermeiden, und ermöglicht stattdessen die Kommunikation über ein neu erstelltes Konto, das nicht mit der genannten Entität in Verbindung steht. In diesem Kontext bedeutet "nicht interaktiv", dass Empfänger keine neuen Adressen für verschiedene Absender etablieren müssen. Vielmehr ist es lediglich erforderlich, einmalig eine persönliche Stealth-Meta-Adresse zu generieren, die anschließend von anderen Benutzern verwendet werden kann, um daraus Stealth-Adressen abzuleiten.

In der Abbildung 3.1 wird den üblichen Ablauf von Transaktionen mit Stealth-Adressen dargestellt, bei dem Alice Vermögenswerte an eine stealth addresse von Bob überweist. Die folgenden Schritten werden im Anschluss erläutert.

1. **Generierung und Veröffentlichung der Stealth-Meta-Adresse durch Bob**
   Die Generierung der Stealth-Meta-Adresse erfordert zunächst von Bob die Erzeugung zweier privater Schlüssel: den sogenannten Äusgabenschlüssel"(Spending Key) und den Änzeigeschlüssel"(Viewing Key). Der Viewing Key dient hierbei dem Schutz der Privatsphäre von Bob, während der Spending Key dazu dient, den Zugriff auf Vermögenswerte zu sichern. Die Stealth-Meta-Adresse setzt sich aus der Kombination von Bobs öffentlichem Spending Key und Viewing Key zusammen. Diese Adresse wird von Bob veröffentlicht, um seine Fähigkeit zur Entgegennahme von Stealth-Transaktionen zu signalisieren.
2. **Generierung einer Stealth-Adresse durch Alice**
   Die Generierung einer Stealth-Adresse erfordert zunächst, dass Alice einen temporären Schlüssel (ephemeral key) generiert, der ausschließlich in ihrem Wissen verbleibt und nur für eine einmalige Anwendung konzipiert ist. Dieser ephemeral key wird im Anschluss mit

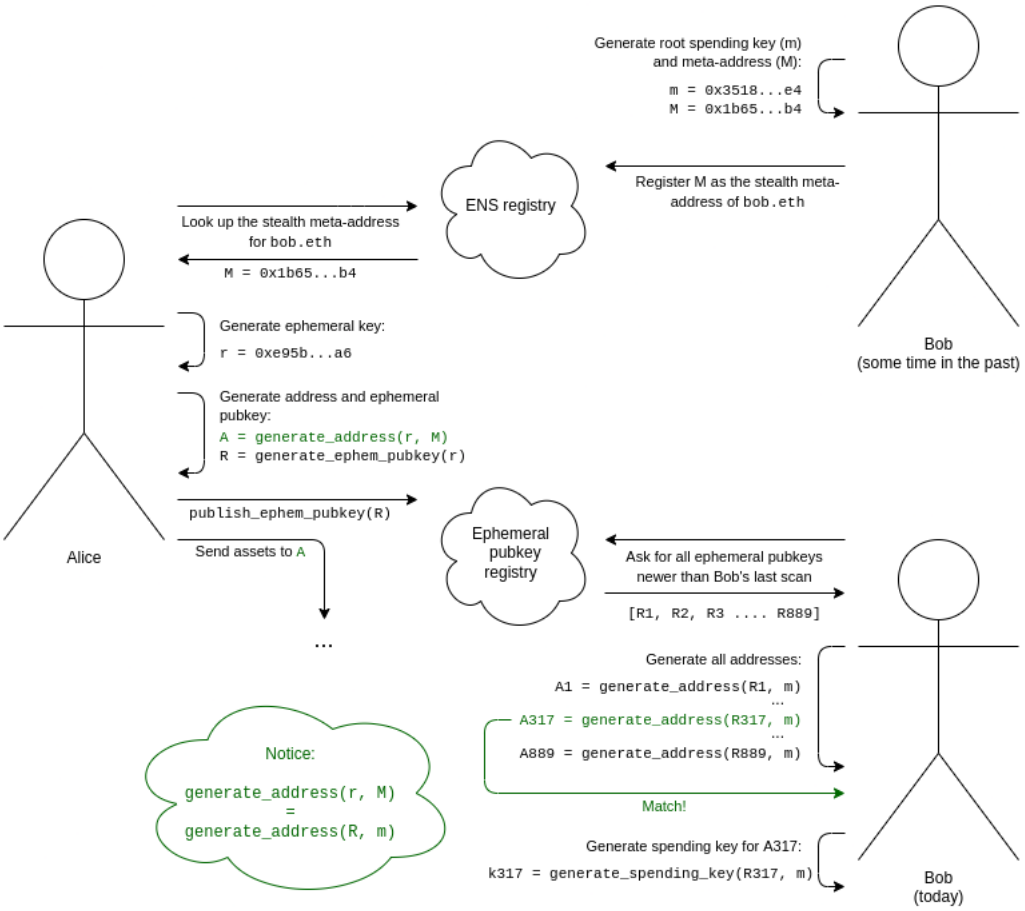

Abbildung 3.1: Stealth addresses workflow
Quelle: https://vitalik.ca/general/2023/01/20/stealth.html

Bobs Stealth-Meta-Adresse kombiniert, um die Stealth-Adresse zu konstruieren. Dabei ist es unabdingbar, dass Alice den öffentlichen Teil dieses ephemeral key öffentlich zugänglich macht.

3. **Berechnung des privaten Schlüssel der Stealth-Adresse**
   Der Regenerierung der Stealth-Adresse durch Bob ist möglich, indem er den Ephemeral-Public-Key mit seinem Spending-Key kombiniert. Durch die Kombination des Ephemeral-Public-Key mit dem Spending-Key kann Bob den privaten Schlüssel der Stealth-Adresse generieren, der diesem Ephemeral-Public-Key entspricht.

Bob hat gegenwärtigen Zugriff auf die Vermögenswerte, wenngleich sein Konto ohne Guthaben ist. Infolgedessen ist es ihm nicht möglich, Vermögenswerte zu übertragen. Es bleibt jedoch die Option, ETH von seinem Hauptkonto auf das Stealth-Konto zu transferieren, doch dies würde zu einem Verlust seiner Privatsphäre führen. Eine Lösung, die die Wahrung von Bobs Privatsphäre sicherstellt, besteht in der Nutzung von ERC-4337-Token-Paymastern, welche den Prozess der Bezahlung der Transaktionskosten von den Endbenutzern entkoppeln[6].

Es ist von maßgeblicher Bedeutung zu unterstreichen, dass sich ERC-5564 und ERC-4337 zum Zeitpunkt der Abfassung dieser Arbeit nach wie vor in der Überprüfungsphase befinden. Folglich sind potenzielle Anpassungen und Änderungen am Protokoll keineswegs auszuschließen.

3.2 Ring Signature

Die Einführung der Ring-Signaturen datiert auf das Jahr 2001 und geht auf die bedeutenden Kryptographen Ron Rivest, Adi Shamir und Yael Tauman zurück [7]. Sie präsentierten das Konzept der Ring-Signaturen als eine innovative Methode zur anonymen Authentifizierung und Unterzeichnung von Nachrichten, ohne dabei die Identität des Unterzeichners zu enthüllen.

Das ursprüngliche Ziel von Ring-Signaturen bestand in der Gewährleistung der Privatsphäre und Anonymität von Benutzern in digitalen Signaturen. Dies erfolgt, indem ein Benutzer die Befugnis erhält, im Namen einer Gruppe von Teilnehmern, dem sogenannten "Ring", eine Nachricht zu signieren, ohne die eigene Identität preiszugeben. Diese Signaturtechnik bietet somit ein Höchstmaß an Datenschutz und Anonymität, da es für Dritte äußerst schwierig ist, den tatsächlichen Unterzeichner einer Nachricht innerhalb des Rings zu ermitteln.

Die Erzeugung einer Ring-Signatur erfolgt durch die sorgfältige Auswahl einer Gruppe von öffentlichen Schlüsseln, die zu einem Ring aggregiert werden. Die tatsächliche Signatur entsteht dann durch die geschickte Verwendung des privaten Schlüssels des Unterzeichners zusammen mit den ausgewählten öffentlichen Schlüsseln, wobei die tatsächliche Identität des Unterzeichners bewusst verschleiert wird.

Ring-Signaturen tragen entscheidend zur Intensivierung der Privatsphäre und Anonymität von Benutzern bei. Angesichts der Tatsache, dass verschiedene Schlüssel als potenzielle Unterzeichner fungieren, gestaltet sich die Ermittlung des tatsächlichen Unterzeichners einer Nachricht innerhalb des Rings äußerst kompliziert.

Trotz der gewährten Anonymität ermöglichen Ring-Signaturen die Überprüfung durch Dritte, um die Richtigkeit der Signatur zu gewährleisten. Dies ermöglicht es anderen Benutzern, die Gültigkeit von Transaktionen zu bestätigen, ohne Einsicht in die Identität des Absenders zu erhalten.

3.3 Homomorphe Verschlüsselung

Homomorphe Verschlüsselung ist eine bedeutende kryptografische Technik, die es ermöglicht, Berechnungen auf verschlüsselten Daten durchzuführen, ohne dass eine Entschlüsselung der Daten erforderlich ist. Mit anderen Worten, sie erlaubt die Ausführung von Operationen auf den verschlüsselten Daten selbst, wobei die Ergebnisse dieser Operationen ebenfalls verschlüsselt bleiben. Obwohl die Grundideen der homomorphen Verschlüsselung bereits in früheren Arbeiten diskutiert wurden, legte Craig Gentry im Jahr 2009 den Grundstein für eine umfassende Theorie und praktische Anwendung dieser Technologie[8]. Seine Arbeit hat dazu beigetragen, die Machbarkeit und Praktikabilität dieses Ansatzes zu demonstrieren und ein breites Forschungs- und Anwendungsgebiet eröffnet.

Homomorphe Verschlüsselungen weisen verschiedene Homomorphie-Eigenschaften auf, darunter:

1. Additive Homomorphie: Diese Eigenschaft ermöglicht die Entschlüsselung des Ergebnisses der Verschlüsselung der Summe zweier Nachrichten als die tatsächliche Summe der Entschlüsselungen der beiden Nachrichten.

2. Multiplikative Homomorphie: Bei der Multiplikativen Homomorphie führt die Verschlüsselung der Multiplikation zweier Nachrichten zu einem Ergebnis, dessen Entschlüsselung dem Produkt der Entschlüsselungen der beiden ursprünglichen Nachrichten entspricht.
3. Weitere Operationen: Je nach dem gewählten Verschlüsselungsschema können auch andere Operationen auf den verschlüsselten Daten durchgeführt werden, wie etwa Vergleiche.

Homomorphe Verschlüsselung kann in drei Hauptkategorien unterteilt werden:

1. Partielle homomorphe Verschlüsselung (PHE): Diese Art von Schema ermöglicht lediglich eine begrenzte Anzahl homomorpher Operationen, wie beispielsweise die Addition verschlüsselter Zahlen.
2. Vollständige homomorphe Verschlüsselung (FHE): FHE hingegen ermöglicht eine breite Palette von homomorphen Operationen, einschließlich Addition, Multiplikation und komplexerer Berechnungen. Allerdings sind FHE-Verschlüsselungen aufgrund ihrer intensiven Berechnungskosten in einigen Anwendungen weniger gebräuchlich.
3. Teilweise homomorphe Verschlüsselung: Dieser Typ bietet eine ausgewogene Lösung, die eine begrenzte Anzahl homomorpher Operationen zulässt, dabei aber Effizienz und reduzierte Berechnungskosten bietet. Insbesondere in Blockchain-Technologien hat das Interesse an teilweise homomorphen Verschlüsselungsverfahren in den letzten Jahren zugenommen, da sie eine effektive Balance zwischen Datenschutz und Rechenleistung bieten.

3.3.1 Pedersen Commitments

Ein Pedersen Commitment ist eine Form der Verpflichtung, die in der Kryptographie verwendet wird, um einen Wert zu verbergen, während gleichzeitig die Möglichkeit besteht, mathematisch zu beweisen, dass bestimmte Rechenoperationen korrekt durchgeführt wurden[9]. Im Folgenden wird das mathematische Konzept des Pedersen Commitments definiert.

1. Wahl der Gruppen und Parameter:

- Wähle eine große Primzahl p.
- Wähle zwei Generatoren g und h für eine zyklische Gruppe $\mathbb{G}$ modulo p.

2. Geheimer Wert und Blinding-Faktor:

- Der geheime Wert x ist die Zahl, deren Commitment erstellt werden soll.
- Der Blinding-Faktor r ist eine zufällige, geheime Zahl als Salt.

3. Berechnung des Commitment-Werts:
Der Commitment-Wert C wird wie folgt berechnet:

$$C = g^x \cdot h^r \mod p \tag{3.1}$$

4. Offenlegung und Verifikation:
Um den geheimen Wert x zu offenbaren, muss man x und r offenlegen. Die Verifikation erfolgt durch Überprüfung von

$$C \equiv g^x \cdot h^r \mod p \tag{3.2}$$

5. Homomorphe Eigenschaft:
Die homomorphe Eigenschaft ermöglicht es, Commitments für Summen oder Differenzen von Werten zu erstellen:
Für die Summe von Werten:

$$C(x + y) = C(x) \cdot C(y) \tag{3.3}$$

Für die Differenz von Werten:

$$C(x - y) = C(x) \cdot C(y)^{-1} \tag{3.4}$$

Hierbei repräsentiert $C(x)$ das Commitment für den Wert x, $C(y)$ das Commitment für den Wert y, und $C(y)^{-1}$ das multiplikative Inverse von $C(y)$ in Bezug auf die Commitment-Multiplikation.

Parameter:

- p - Große Primzahl.
- g, h - Generatoren der zyklischen Gruppe $\mathbb{G}$ modulo p.
- x - Der zu verbergende geheime Wert.
- r - Zufälliger Blinding-Faktor.
- C - Commitment-Wert.

Der Sicherheitsaspekt des Pedersen Commitments basiert auf der Annahme, dass das diskrete Logarithmusproblem schwer ist. Insbesondere wird angenommen, dass es schwierig ist, den Exponenten x zu berechnen, wenn $g^x \mod p$ und g bekannte Werte sind.

3.3.2 ElGamal-Kryptosystem

Das ElGamal-Kryptosystem ist ein asymmetrisches Verschlüsselungsverfahren, benannt nach seinem Erfinder Taher ElGamal, das 1985 vorgestellt wurde[10]. Es basiert auf dem Schwierigkeitsgrad des diskreten Logarithmusproblems in zyklischen Gruppen. Das Kryptosystem bietet sowohl Verschlüsselung als auch digitale Signaturen. Im Folgenden wird das mathematische Konstrukt des ElGamal-Kryptosystems vorgestellt:

1. Schlüsselgenerierung
Während der Schlüsselgenerierung wird eine zyklische Gruppe $\mathbb{G}$ ausgewählt, und ein Primzahlgenerator g wird gewählt, um diese Gruppe zu erzeugen. Jeder Benutzer wählt einen privaten Schlüssel x zufällig und berechnet den entsprechenden
öffentlichen Schlüssel h.

$$h = g^x \mod p \tag{3.5}$$

Wobei p die Gruppenordnung ist.

2. Verschlüsselung:
Um eine Nachricht m an einen Benutzer mit dem öffentlichen Schlüssel h zu senden, wird ein zufälliger Wert y gewählt, und der gemeinsame Schlüssel K wird berechnet.

$$K = h^y \mod p \tag{3.6}$$

Der Geheimtext ist $c = (c_1, c_2)$.

$$\begin{aligned} c_1 &= g^y \mod p \\ c_2 &= m \cdot K \mod p \end{aligned} \tag{3.7}$$

3. Entschlüsselung:
Der Empfänger verwendet seinen privaten Schlüssel x, um K zu berechnen.

$$K = c_1^x \mod p \tag{3.8}$$

Durch Berechnung der Inversen von K wird die ursprüngliche Nachricht m wiederhergestellt.

$$m = c_2 \cdot K^{-1} \mod p \tag{3.9}$$

4. Digitale Signatur:
Während der Signaturerstellung wählt der Signierende einen zufälligen Wert k und berechnet r.

$$r = g^k \mod p \tag{3.10}$$

Die Signatur s wird als

$$k^{-1} \cdot (H(m) - x \cdot r) \mod (p-1) \tag{3.11}$$

berechnet, wobei $H(m)$ eine Hashfunktion auf die Nachricht m ist.

5. Signaturverifikation:
Der Verifizierende überprüft die Gültigkeit der Signatur durch Vergleich von

$$\begin{aligned} v_1 &= (h^r \cdot r^s) \mod p \\ v_2 &= g^{H(m)} \mod p \end{aligned} \tag{3.12}$$

Die Signatur wird als gültig betrachtet, wenn $v_1 \equiv v_2 \pmod p$.

6. Homomorphe Eigenschaften:
ElGamal bietet multiplikative Homomorphie. Angenommern, $E(m_1)$ und $E(m_2)$ repräsentieren zwei Ciphertexts, die die Nachrichten m_1 bwz. m_2 verschlüsseln, dann ist die Gleichung 3.13 gültig

$$E(m_1)\dot{E}(m_2) = E(m_1 \dot{m}_2) \tag{3.13}$$

3.3.3 Paillier-Kryptosystem

Das Paillier-Kryptosystem, eingeführt von Pascal Paillier im Jahr 1999, gehört zu den asymmetrischen kryptografischen Verfahren[11]. Die Sicherheit des Paillier-Kryptosystems basiert auf der Schwierigkeit des Factoring-Problems, das darauf abzielt, die Primfaktoren eines zusammengesetzten Zahlenprodukts zu berechnen. Im Folgenden wird das mathematische Konzept des Paillier-Kryptosystem näher erläutert.

1. Schlüsselgenerierung:

- Wähle zwei große Primzahlen p und q
- Berechne den öffentlichen Modulus $n = p \cdot q$
- Wähle einen öffentlichen Exponenten g aus der Gruppe $\mathbb{Z}_{n^2}^*$
- Berechne den privaten Exponenten λ

$$\lambda = lcm(p-1, q-1) \tag{3.14}$$

  Wobei lcm das kleinste gemeinsame Vielfaches ist (Englisch: Least Common Multiple).
- Der öffentliche Schlüssel ist (n, g) und der private Schlüssel ist λ.

2. Verschlüsselung:
Um eine Nachricht m zu verschlüsseln, wird ein zufälliges r aus der Menge $\mathbb{Z}_{n^2}^*$ gewählt. Der Geheimtext c wird wie folgt gebildet:

$$c = (g^m \cdot r^n) \mod n^2$$

3. Entschlüsselung:
Die Entschlüsselung erfolgt durch Anwendung des privaten Exponenten λ. Die ursprüngliche Nachricht m wird gemäß 3.3.3 wiederhegestellt.

$$m = \left(\frac{(c^\lambda \mod n^2) - 1}{n} \right) \cdot \mu \mod n,$$

wobei μ eine vorberechnete Konstante ist. Die Berechnung der Konstanten μ erfolgt gemäß der Formel 3.15.

$$\mu = L\left(\left(g^\lambda \mod n^2\right)\right)^{-1} \mod n, \tag{3.15}$$

wobei $L(x) = \frac{x-1}{n}$ die Funktion zur Berechnung des Carmichael Lambda-Werts ist[12].

4. Homomorphe Eigenschaften:
Das Paillier-Kryptosystem bietet additive Homomorphie. Angenommen, $E(m_1)$ und $E(m_2)$ repräsentieren zwei Ciphertexts, die die Nachrichten m_1 bzw m_2 verschlüsseln,
dann ist die Gleichung 3.16 gültig.

$$E(m_1) \cdot E(m_2) \mod n^2 = E(m_1 + m_2) \mod n^2 \tag{3.16}$$

Die Potenzierung mit einer Nachricht k wird gemäß 3.17 erreicht.

$$E(m)^k \mod n^2 = E(k \cdot m) \mod n^2 \tag{3.17}$$

3.4 Zero Knowledge Proof

Zero-Knowledge Proofs (ZKPs) sind ein Konzept aus der Kryptographie, das 1985 ursprünglich von Shafi Goldwasser, Silvio Micali und Charles Rackoff entwickelt wurde[13]. Es ermöglicht einem Beweisender (Prover), einer anderen Partei (Verifier) zu zeigen, dass er eine bestimmte Aussage kennt, ohne diese Aussage tatsächlich offenzulegen. In ursprünglichen ZKP-Protokollen erfolgte der Beweis interaktiv, wobei Prover und Verifier miteinander kommunizieren. Später wurden nicht-interaktive ZKP entwickelt, bei denen der Beweis in einer Nachricht übermittelt wird und kann von jedem verifizieren werden (auch von Smart Contracts)[14].

Zero-Knowledge-Proofs basieren auf verschiedenen mathematischen Konzepten, darunter modulare Arithmetik, Gruppentheorie und elliptische Kurven. Die spezifischen mathematischen Techniken variieren je nach Art des Zero-Knowledge-Proofs, aber sie sind alle darauf ausgerichtet, das Konzept der Geheimhaltung und Überzeugung sicherzustellen.

Laut[13] muss ein ZKP-Protokoll die drei folgenden Eigenschaften erfüllen:

1. Vollständigkeit: Wenn die Eingabe gültig ist, gibt das Zero-Knowledge-Proof immer „wahr“ zurück. Wenn also die zugrunde liegende Aussage wahr ist und der Beweiser und Verifizierer ehrlich handeln, kann der Beweis akzeptiert werden.
2. Korrektheit: Wenn die Eingabe ungültig ist, ist es theoretisch unmöglich, das Zero-Knowledge-Proof dazu zu bringen „wahr“ zurückzugeben. Daher kann ein lügender Beweiser einen ehrlichen Verifizierer nicht dazu bringen, zu glauben, dass eine un-gültige Aussage gültig ist (außer mit einem winzigen Wahrscheinlichkeitsspielraum).
3. Zero-Knowledge (Null-Wissen über die Aussage): Der Verifizierer erfährt nichts über eine Aussage, adgesehen von ihrer Gültigkeit oder Falschheit. Diese Anforderung hindert den Verifizierer auch daran, die ursprüngliche Eingabe oder Aussage aus dem Beweis abzuleiten.

Wir unterscheiden zwei wichtige nicht-interaktiven ZKPs im Bereich Blockchain

3.4.1 Succinct Non-Interactive Argument Of Knowledge(SNARKs)

SNARKs wurden erstmals 2012 von Alessandro Chiesa et al. eingeführt [15]. Hier sind die Eigenschaften von SNARKs

1. **Succinct:** befasst sich mit der Größe des erzeugten Nachweises. Dies spielt für Zero-Knowledge Proofs aus zwei Gründen eine sehr wichtige Rolle: Speicherplatz und Verifikationszeit auf verteilten Ledgern, da Speicherplatz sehr teuer sein kann. Daher möchten wir die Größe des Nachweises so klein wie möglich halten und ihn schnell überprüfen. Succinctness ist vorteilhaft, wenn das Medium, das wir für die Speicherung des Nachweises verwenden möchten, sehr teuer ist und schnelle Verifikationen erforderlich sind. Daher ist der Nachweis sehr klein und leicht zu überprüfen.
2. **Non-Interactive:** erfordert keine interaktive oder wiederholte Kommunikation zwischen den beiden Parteien. Tatsächlich gibt es nur eine einzige Runde, sodass ein Beweiser den Nachweis erstellen kann und der Überprüfer im Namen des Nachweises die Gültigkeit der Aussage überprüfen kann, jedoch ohne jegliche Interaktion. Diese Eigenschaft macht zk-SNARKs für die Blockchain nützlich.

3. **Argument:** bedeutet, dass nur wahre Aussagen gültige Nachweise haben, aber falsche Aussagen keinen Nachweis haben. Daher ist es rechnerisch nicht möglich, unlogische Nachweise zu erstellen, aber die korrekten Nachweise können effizient generiert werden.
4. **Of Knowledge:** ermöglicht es, dass eine bestimmte Aussage nur dann nachgewiesen werden kann, wenn der Beweiser den Zeugen (private Eingabe) kennt.

Die Nutzung von SNARKs erfordert jedoch eine vertrauenswürdige Einrichtung, um effektiv durchgeführt zu werden. Dieser Prozess beinhaltet die Erstellung eines Prüfschlüssels und eines Verifizierungsschlüssels unter Verwendung eines geheimen Zufallselements. Sollte diese geheime Zufälligkeit versehentlich oder absichtlich öffentlich gemacht werden, besteht die Gefahr, dass fehlerhafte Beweise generiert werden, die für den Verifizierer als gültig erscheinen könnten. Ein herausragendes Beispiel für die Anwendung von SNARKs in der Blockchain-Technologie ist Zcash. Zcash ist die erste Blockchain, die SNARKs verwendet, um die Vertraulichkeit von Transaktionsbeträgen zu wahren und gleichzeitig die Authentizität und Integrität der Transaktionen sicherzustellen[16].

3.4.2 Scalable Transparent Argument of Knowledge(STARKs)

STARKs wurden im Jahr 2018 von einem Forscherteam, bestehend aus Eli Ben-Sasson, Iddo Bentov, Yinon Horesh und Michael Riabzev, als eine Lösung für die Herausforderungen eingeführt, die bei SNARKs auftraten[17]. STARKs zeichnen sich insbesondere durch ihre Skalierbarkeit aus, da sie in der Lage sind, große Datenmengen äußerst effizient zu verarbeiten und gleichzeitig in polynomieller Zeit überprüft werden können. Dies bedeutet, dass der Rechenaufwand für die Erzeugung und Validierung von Beweisen in einem polynomialen Verhältnis zur Größe der zu überprüfenden Daten steht. Sie sind *transparent* in dem Sinne, dass die Beweise von jedem überprüft werden können, ohne dass geheime Informationen offenbart werden.

Eine bemerkenswerte Eigenschaft von STARKs ist die Fähigkeit zur *Batch-Verarbeitung*. Dies ermöglicht die gleichzeitige Überprüfung mehrerer Beweise in einem einzigen Schritt, was die Verarbeitungszeit erheblich reduziert und parallele Überprüfungen von Beweisen ermöglicht. In dezentralen Systemen wie Blockchains, in denen viele Transaktionen gleichzeitig überprüft werden müssen, ist diese Eigenschaft besonders wertvoll.

Zusätzlich zu ihrer Skalierbarkeit sind STARKs auch post-quantum-sicher, was bedeutet, dass sie auch in einer Welt mit leistungsstarken Quantencomputern ihre Wirksamkeit behalten. Es sei jedoch darauf hingewiesen, dass STARKs tendenziell größere Beweise erzeugen, obwohl sie keine vertrauenswürdige Einrichtung erfordern.

3.5 Zero Knowledge Range Proof (ZKRP)

Eine Range Proof ist ein kryptographisches Protokoll, das dazu dient, zu beweisen, dass eine geheime Zahl innerhalb eines bestimmten Bereichs liegt, ohne dabei die exakte Zahl preiszugeben. Die ersten Konstruktionen von ZKRP-Protokollen wurden vor Jahrzehnten vorgestellt, mit Schemata wie dem im Jahr 1995 von Damgård [18] vorgeschlagenen und im Jahr 1997 von Fujisaki und Okamoto [19]. Leider sind diese Vorschläge nicht effizient genug für die praktische Anwendung. Die erste praktische Konstruktion wurde von Boudot im Jahr 2001 vorgeschlagen [20].

3.5.1 Bulletproofs

Im Jahr 2017 stellten Benedikt Bünz et al.[21] einen innovativen Ansatz zur Entwicklung von nicht-interaktiven ZKRP vor, den sie als Bulletproofs bezeichneten. Bulletproofs zeichnen sich besonders durch ihre Fähigkeit aus, äußerst kompakte Beweise zu generieren, die schnell und effizient verifiziert werden können. Die Größe des Beweises skaliert logarithmisch mit der Größe des nachzuweisenden Bereich. Dies steht im Gegensatz zu früheren Systemen für Bereichsnachweise, bei denen die Größe des Nachweises linear mit der Größe des Bereichs skalierte. Bulletproofs weisen einen besonderen Vorteil auf, indem sie keine vertrauenswürdige Einrichtung erfordern und gleichzeitig vor Angriffen durch Quantencomputer geschützt sind. Daher stellen sie eine sinnvolle und ausgewogene Alternative zu anderen kryptografischen Beweissystemen wie SNARKs und STARKs dar.

Der Konstruktionsmechanismus von Bulletproofs basiert auf einer Vielzahl fortschrittlicher mathematischer Konzepte, die im Kontext von Kryptographie angewendet werden und die Rahmen dieser Arbeit überschreiten. Hier werden einige zentrale Aspekte detaillierter betrachtet:

1. Parametrisierung und Vorbereitung:
Die Auswahl einer geeigneten elliptischen Kurve und ihrer Generatoren ist ein grundlegender Schritt. Die Sicherheit und Effizienz von Bulletproofs hängen stark von der Wahl dieser kryptographischen Parameter ab.

2. Polynomial Commitments:
Bulletproofs verwenden Polynomial Commitments, um Verpflichtungen zu Polynomen zu erstellen. Ein Verpflichtungspolynom repräsentiert die geheimen Werte in kompakter Form. Dies ermöglicht es, mehrere Beweise in einem einzigen Bulletproof zu aggregieren.

3. Inner-Product Argument:
Das Inner-Product Argument ist eine Technik, um das Skalarprodukt von zwei Vektoren $\mathbf{a}$ und $\mathbf{b}$ zu beweisen, ohne die Vektoren selbst offenzulegen. Das Ziel ist es, den Beweis zu führen, dass das Skalarprodukt $\mathbf{c} = \langle \mathbf{a}, \mathbf{b} \rangle$ korrekt berechnet wurde. Die Fiat-Shamir Heuristik wird oft verwendet, um interaktive Protokolle in nicht-interaktive ZKPs zu überführen.

4. Homomorphe Verschlüsselung:
Bulletproofs integrieren Pedersen-Commitments als essenziellen Bestandteil ihres Konstruktionsmechanismus, um Verpflichtungen zu Polynomen zu schaffen. Durch die homomorphen Eigenschaften von Pedersen-Commitments wird es möglich, Berechnungen auf verschlüsselten Polynom- oder Vektorverpflichtungen auszuführen. Diese homomorphen Eigenschaften ermöglichen es, komplexe Berechnungen effizient durchzuführen, ohne die genauen Werte der verpflichteten Polynome oder Vektoren offenzulegen.

5. Beweis für lineare Beziehungen:
Bulletproofs ermöglichen den Beweis von linearen Beziehungen zwischen geheimen Werten, ohne die präzisen Werte offenzulegen. Diese Fähigkeit wird durch die geschickte Nutzung der Homomorphie von Pedersen-Commitments erreicht. Pedersen-Commitments erlauben es, Verpflichtungen zu geheimen Werten zu erstellen, wobei die Verpflichtungen additiv sind. In Bulletproofs werden spezifische Prüfwerte verwendet, um lineare Beziehungen zu überprüfen. Das bedeutet, dass der Beweis demonstriert, dass die geheimen Werte bestimmte lineare Eigenschaften erfüllen.

6. Effiziente Verifikation:
Ein wesentlicher Aspekt von Bulletproofs liegt in der effizienten Verifikation, die durch ein sorgfältiges Design und die Anwendung von fortgeschrittenen Techniken wie Batch-Verarbeitung erreicht wird. Die Effizienz der Verifikation ist von zentraler Bedeutung, insbesondere in Anwendungen, in denen eine große Anzahl von Beweisen überprüft werden muss, wie beispielsweise in Blockchain-Systemen. Durch das kluge Design von Bulletproofs wird ermöglicht, dass die Verifikation schnell und ressourcenschonend erfolgt. Die Verwendung von Batch-Verarbeitungstechniken erlaubt es, mehrere Beweise gleichzeitig zu überprüfen, was die Skalierbarkeit und Leistungsfähigkeit von Bulletproofs erheblich verbessert.

4 Verwandte Arbeiten

Im vorliegenden Kapitel werden unterschiedliche verwandte Arbeiten eingehend analysiert und detailliert beschrieben. Dies schließt sowohl bereits etablierte Produkte auf dem Markt als auch Prototypen und theoretische Überlegungen ein. Der Schwerpunkt liegt darauf, einen umfassenden Überblick über die aktuelle Forschung und Entwicklung im Bereich vertraulicher Transaktionen zu bieten. Dabei werden verschiedene Aspekte und Ansätze berücksichtigt, um die Vielfalt der existierenden Arbeiten zu verdeutlichen und relevante Erkenntnisse für die eigene Forschung und Entwicklung zu extrahieren.

4.1 Tornado Cash

Tornado Cash ist ein vollständig dezentralisiertes Protokoll, das auf der Ethereum-Blockchain basiert, mit dem Ziel unter Verwendung von Smart Contracts, die Privatsphäre von Kryptowährungstransaktionen zu schützen, insbesondere von Ether (ETH) und ERC-20-Token[22]. Es wurde 2019 von Roman Semenov et al. entwickelt und ermöglicht Benutzern, ihre Transaktionen zu anonymisieren und die Rückverfolgbarkeit ihrer Gelder zu erschweren.
Als nicht verwahrendes Protokoll behalten die Benutzer die Kontrolle über ihre Kryptowährungen, während sie Tornado Cash verwenden. Dies bedeutet, dass ihnen bei jeder Einzahlung der private Schlüssel zur Verfügung gestellt wird, der den Zugriff auf die hinterlegten Gelder ermöglicht. Dadurch haben die Benutzer die vollständige Kontrolle über ihre Vermögenswerte.

Tornado Cash bietet Pools an, in denen Benutzer eine festgelegte Menge an ERC-20-Token einzahlen und abheben können, wie beispielsweise 1 ETH, 0,1 wBTC oder 100 DAI. Allerdings erlaubt Tornado Cash aus Sicherheitsgründen keine willkürlichen Beträge. Wenn ein Benutzer beispielsweise 100 ETH einzahlt und alle anderen nur 1 ETH einzahlen, könnte dies dazu führen, dass Ihre Transaktionen leicht zurückverfolgt werden können. Tornado Cash Nova ermöglicht jedoch Transaktionen mit unterschiedlichen Beträgen, wobei darauf geachtet werden sollte, dass der Transaktionswert nicht signifikant von der durchschnittlichen Menge abweicht.

Angenommen, Alice besitzt 1 ETH und möchte ihre Transaktionen anonymisieren, um zu verhindern, dass jemand sehen kann, wohin sie diese ETH sendet. Sie muss dann folgendes tun:

Schritt 1: Einzahlung

1. Alice erzeugt zwei zufällige Werte: das Geheimnis (r) und den Nullifikator (k).
2. Diese beiden Werte (r und k) werden von Alice gehasht, um den Einzahlungsbeleg $C = H1(r, k)$ zu erstellen, wobei $H1$ die Pedersen-Hash-Funktion ist. Dieser Einzahlungsbeleg C wird für spätere Auszahlungen verwendet.
3. Alice überträgt $C' = H2(C)$ und k an den Tornado Cash-Vertrag, wo sie öffentlich gespeichert werden, wobei $H2$ die MIMC-Hash-Funktion ist. Gleichzeitig wird 1 ETH in einen Depositenvertrag eingezahlt.

Während des Einzahlungszeitraums werden die ETH von Alice mit Einzahlungen anderer Benutzer vermischt, die ebenfalls ETH anonymisieren möchten. Diese Vermischung erschwert die Rückverfolgung der ursprünglichen Quelle der ETH.

Tornado Cash verwendet einen in der Blockchain verankerten Merkle-Baum, um die Hashes der Einzahlungsbelege(C') zu speichern. Die zur Generierung des Merkle-Baums verwendeten Hash-Funktionen sind MIMC-Hashes (Merkle-Damgård Iterated Multiplication and Circuits), die als die erste ZK-freundliche Hash-Funktion auf der ASIACRYPT 2016 eingeführt wurden [23].

Schritt 2: Auszahlung

1. Für die Auszahlung muss Alice nachweisen, dass sie im Besitz von r und k ist.
2. Sie erstellt einen sogenannten ZK-SNARK-Beweis, der belegt, dass das Geheimnis und der Nullifikator die Bedingung $C = H1(r, k)$ erfüllen. Dieser Beweis wird an den Tornado Cash-Vertrag übermittelt. Für die Beweiserstellung werden öffentliche Eingaben wie die Merkle-Wurzel, der Nullifikator-Hash und die Auszahlungsadresse verwendet, während private Eingaben das Geheimnis, der Nullifikator und die Einzahlungsbestätigung (C) einschließen.
3. Der Vertrag führt eine Überprüfung des eingereichten Beweises in der Blockchain durch und achtet auf mögliche doppelte Ausgaben.
4. Bei erfolgreicher Überprüfung erfolgt die Überweisung der Mittel an die angegebene Auszahlungsadresse.

Der Nullifikator k hat die wichtige Funktion, Double-Spend-Angriffe zu verhindern. Ohne Nullifikator könnte ein böswilliger Benutzer einmal einzahlen und dann mehrmals abheben, indem er vorgibt, das Geheimnis zu kennen. Der Nullifikator wird nach einer erfolgreichen Auszahlung aus der Liste der Nullifikatoren entfernt.

Zusammengefasst funktioniert Tornado Cash, indem es die Einzahlungen der Benutzer miteinander vermischt, verzögerte Auszahlungen ermöglicht und Zero-Knowledge-Proofs verwendet, um die Privatsphäre zu schützen. Dies erschwert es Dritten erheblich, die Transaktionen und Geldflüsse zu verfolgen. Tornado Cash hat jedoch auch seine Einschränkungen, und Benutzer sollten sich bewusst sein, dass ihre Anonymität nicht absolut ist und von verschiedenen Faktoren abhängt.

4.2 AZTEC-Protokoll

Das AZTEC-Protokoll (Anonymous Zero-Knowledge Transactions with Efficient Communication) definiert eine Reihe von Zero-Knowledge-Beweisen, die ein vertrauliches Transaktionsprotokoll für den Einsatz in Blockchain-Systemen mit Turing-vollständigen Berechnungen, wie Ethereum, ermöglichen[24]. Durch die Kombination des Pedersen-Commitment, homomorpher Arithmetik und Bulletproofs ermöglicht das Protokoll eine effiziente Überprüfung von vertraulichen Transaktionen.

Im Zentrum des Protokolls steht die AZTEC-Note, die als verschlüsselte Darstellung eines abstrakten Werts definiert ist. Eine AZTEC-Note besteht aus einem Tupel von elliptischen Kurvenverpflichtungen sowie drei Skalaren: einem Viewing Key, einem Spending Key und einer Nachricht (verschlüsseltes Guthaben). Die Kenntnis des Viewing Keys ermöglicht die Entschlüsselung der Note und damit die Offenlegung der Nachricht. Zusätzlich kann der Viewing Key verwendet werden, um gültige Zero-Knowledge Proofs (ZKP) zu erstellen, die dann mit dem Spending Key signiert werden. Alle AZTEC-Notes werden in einem speziellen Register, dem Note-Register, gespeichert.

Ein weiterer wichtiger Bestandteil des Protokolls ist der Join-Split-Transaktionstyp, der für die Durchführung vertraulicher Transaktionen verwendet wird. Bei einer Join-Split-Transaktion werden mindestens eine Note aus dem Note-Register entnommen, kombiniert und in mindestens eine Ausgabe-Note aufgeteilt. Die Eingabe-Notes werden aus dem Register entfernt und durch die neu erstellten Ausgabe-Notes ersetzt. Dieser Mechanismus ermöglicht die Vertraulichkeit der Transaktionen und stellt sicher, dass die privaten Informationen der Benutzer geschützt bleiben.

4.3 Zama

Zama ist ein quelloffenes Kryptografietool, das die Erstellung von Anwendungen mit vollständig homomorpher Verschlüsselung ermöglicht. Das fhEVM (fully homomorphic Encryption Virtual Machine) von Zama ermöglicht die Ausführung vertraulicher Smart Contracts auf verschlüsselten Daten, wodurch sowohl Vertraulichkeit als auch Zusammensetzbarkeit gewährleistet werden[25].

Zamas fhEVM löst das traditionelle Dilemma in der Blockchain, bei dem man sich zwischen der Offenlegung von Anwendungs- und Benutzerdaten in der Kette und dem Verlust der Zusammensetzbarkeit von Verträgen außerhalb der Kette entscheiden musste. Dank des Durchbruchs in der homomorphen Verschlüsselung ermöglicht das fhEVM von Zama die Ausführung vertraulicher Smart Contracts auf verschlüsselten Daten und gewährleistet so sowohl Vertraulichkeit als auch Zusammensetzbarkeit.

Zama stellt eine JavaScript-Bibliothek bereit, die die Interaktion mit Blockchains unter Verwendung der fortschrittlichen TFHE-Technologie (Fully Homomorphic Encryption over the Torus) ermöglicht. Diese Bibliothek integriert die TFHE-Verschlüsselungsfunktionen nahtlos in Web3-Anwendungen und ermöglicht somit sichere und private Interaktionen mit Smart Contracts.

Die TFHE Solidity-Bibliothek von Zama ist ein leistungsstarkes Werkzeug, das Entwicklern die Manipulation verschlüsselter Daten in Smart Contracts ermöglicht. Mit dieser Bibliothek können Entwickler Berechnungen über verschlüsselte Daten durchführen, einschließlich Addition, Multiplikation, Vergleich und mehr, und dabei die Vertraulichkeit der zugrunde liegenden Informationen bewahren.

4.4 Zether

Zether ist ein dezentrales Protokoll und eine Plattform für vertrauliche, dezentrale und skalierbare Zahlungen auf Basis von Ethereum[26]. Zether verwendet eine Kontobasierte Architektur, wobei jedes Konto aus einem ElGamal-Ciphertext besteht , der das Guthaben des Kontos unter Verwendung seines eigenen öffentlichen Schlüssels verschlüsselt.

Zether schlägt Σ-Bullets vor, eine Verbesserung von Bulletproofs. Um die Praktikabilität des Designs zu zeigen, wurde Zether als Ethereum-Smart Contract implementiert, und es ergab sich, dass eine vertrauliche Zether-Transaktion etwa 0,014 ETH oder ungefähr 1,51 USD kostet (Stand Anfang Februar 2019).

Um Geld zu übermitteln, veröffentlicht Alice eine geordnete Liste von öffentlichen Schlüsseln, die sowohl sie selbst als auch den Empfänger sowie andere willkürlich ausgewählte Parteien einschließt. Hierbei begleitet sie die Liste mit entsprechenden ElGamal-Ciphertexts, die unter den zugehörigen

öffentlichen Schlüsseln die Beträge verschlüsseln. Dies ermöglicht es Alice, die Kontostände dieser verschiedenen Konten anzupassen. Der Smart Contract wendet diese Anpassungen unter Verwendung der homomorphen Eigenschaft der ElGamal-Verschlüsselung an.
Schließlich veröffentlicht Alice einen Σ-Bulletproof, der mehrere Aussagen beinhaltet. Erstens bestätigt er, dass Alice im Besitz ihres eigenen geheimen Schlüssels ist. Zweitens zeigt er, dass Alice über ausreichende Mittel verfügt, um ihre Abzüge zu decken. Drittens belegt er, dass Alice Mittel nur von ihrem eigenen Konto abgezogen und diese ausschließlich Bob gutgeschrieben hat – und zwar genau in der Höhe, die sie abgebucht hat, nicht weniger. Selbstverständlich weist der Σ-Bulletproof auch nach, dass Alice die Kontostände außer ihren eigenen und denen von Bob nicht verändert hat.

Diese Anpassungs-Ciphertexts, undurchsichtig für Außenstehende, verbergen, wer wem Geld geschickt hat und wie viel dabei gesendet wurde.

5 Anforderungsanalyse und Design

5.1 Anforderungsanalyse

Im Rahmen dieser Studie werden die nachfolgenden Anforderungen formuliert, um die Vertraulichkeit von Transaktionen auf der Ethereum-Blockchain sicherzustellen. Diese Anforderungen sind entscheidend für die erfolgreiche Implementierung von Mechanismen zur Gewährleistung der Privatsphäre und Sicherheit bei Transaktionen auf dieser Plattform.

1. Funktionale Anforderungen:
   a) **Vertraulichkeit** und **Verschlüsselung**: Die Transaktionsdetails müssen durch Verschlüsselung geschützt werden, um sicherzustellen, dass nur autorisierte Parteien darauf zugreifen können, wodurch sowohl die Vertraulichkeit als auch die Datensicherheit gewährleistet sind.
   b) **ZKRPs**: Auswahl und Implementierung geeigneter ZKRPs sind erforderlich, um nachzuweisen, dass einen Benutzer über bestimmte Informationen verfügt, ohne diese preiszugeben, und somit die Privatsphäre der Benutzer zu wahren.
   c) **Privatspähre**: Gewährleistung der Privatsphäre der Benutzer durch den Einsatz von Zero-Knowledge Proofs, um persönliche Daten und Transaktionsdetails vertraulich zu behandeln.
2. Nicht-funktionale Anforderungen:
   a) **Skalierbarkeit**: Die Implementierung muss darauf ausgelegt sein, eine hohe Anzahl vertraulicher Transaktionen effizient zu verarbeiten, um die Skalierbarkeit sicherzustellen.
   b) **Gasverbrauch**: Es ist erforderlich, den Gasverbrauch zu minimieren, um Transaktionskosten zu optimieren und die Effizienz der vertraulichen Transaktionen zu gewährleisten. Dies ist besonders wichtig, um die Anwendbarkeit auf breiter Ebene sicherzustellen und die Akzeptanz in der Ethereum-Community zu fördern.

5.2 Design

Die Darstellung in Abbildung 5.1 veranschaulicht das im Rahmen dieser Masterarbeit konzipierte Protokoll auf einer abstrakten Ebene, während Abbildung 5.2 die Architektur der Smart Contracts in Form eines Klassendiagramms detailliert darstellt. Im weiteren Verlauf wird das Protokoll detailliert erläutert, um einen umfassenden Einblick in seine Funktionalitäten und Struktur zu gewähren.

Die Grundlage des Protokolls liegt in den Kontoständen, ähnlich wie bei ERC20, wobei eine Zuordnung von Adressen zu Token-Guthaben vorgenommen wird, um die Anzahl der Token für jeden Inhaber nachzuverfolgen. Jedoch erfolgt die Speicherung der Token-Guthaben in verschlüsselter Form. Hierbei kommt das Paillier-Verschlüsselungssystem zum Einsatz, um eine sichere und geschützte Aufbewahrung der Token-Guthaben zu gewährleisten.

Das Protokoll fokussiert ausschließlich auf die *transfer*-Funfktion und setzt dabei auf Stealth-Adressen, um die Privatsphäre des Empfängers zu wahren. Es ist jedoch zu beachten, dass die Identität des Senders weiterhin transparent bleibt.

Es ist wichtig zu betonen, dass Stealth-Adressen dazu dienen, die Rückverfolgbarkeit der Empfänger zu erschweren, indem für jede Transaktion eine eindeutige Adresse generiert wird. Auf diese Weise wird vermieden, dass Dritte leicht den Kontostand oder die Transaktionshistorie eines Empfängers nachverfolgen können. Dennoch bleibt die Identität des Senders durch den Mechanismus des Protokolls transparent. Es handelt sich um einen Kompromiss, der den Fokus auf die Privatsphäre der Empfänger legt, während die Transparenz bezüglich der sendenden Parteien beibehalten wird.

Im folgenden Pseudocode (Algorithmus 1) wird die Funktionsweise der Überweisungsfunktion veranschaulicht.

Algorithm 1 Transfer Function

function TRANSFER(to, valueToAdd, valueToSubtract, proof)
  Inputs:
    Receiver's address
    Value to add to receiver's balance (encrypted with the receiver paillier public key)
    Value to subtract from sender's balance (encrypted with the sender paillier public key)
    Zero-knowledge range proof
  Output:
    Success status (bool)
  Procedure:
    Check if sender's balance byte length is greater than zero. If not, exit with an error message ("Balance too low").
    Verify the zero-knowledge range proof. If the verification fails, exit with an error message ("The encrypted value is not in [0, $2^{256} - 1$]").
    Update the receiver's balance:
      If the receiver's balance byte length is currently zero, set it to valueToAdd.
      If the receiver's balance is not zero, add the encrypted value to the existing balance.
    Decrease the sender's balance by subtracting the valueToSubtract.
    Return true.

Vom Standpunkt der Benutzer aus sind die nachfolgenden Schritte auszuführen:

1. Bob erstellt zu einem vorherigen Zeitpunkt eine Stealth-Meta-Adresse (M) zusammen mit einem Paillier-Schlüsselpaar (skB, pkB). Anschließend übermittelt Bob sowohl M als auch pkB an Alice.
2. Alice hat nun die Aufgabe, die folgenden Schritte auszuführen:
   a) Generierung eines Paillier-Schlüsselpaars (skA, pkA).
   b) Erzeugung einer zufälligen ephemeren Zahl r.
   c) Erstellung einer Stealth-Adresse für Bob.
   d) Verschlüsselung des Betrags mit pkA (Ergebnis: x1).
   e) Verschlüsselung des Betrags mit pkB (Ergebnis: x2).
   f) Lesen und Entschlüsselung des aktuellen Kontostands.
   g) Erstellung einer Range-Proof (P), um sicherzustellen, dass die Differenz zwischen dem Kontostand und dem Betrag positiv ist (größer als 0).
   h) Übermittlung der Transaktion mit den Parametern A, pkB, pkA, x1, x2 und P.
   i) Generierung und Übermittlung des öffentlichen Schlüssels der ephemeren Zahl an Bob.
3. Nach Erhalt von R kann Bob die Stealth-Adresse (A) eigenständig generieren. Mit dieser Adresse ist es ihm möglich, den Kontostand von A zu lesen und mithilfe seines privaten Schlüssels skB zu entschlüsseln.

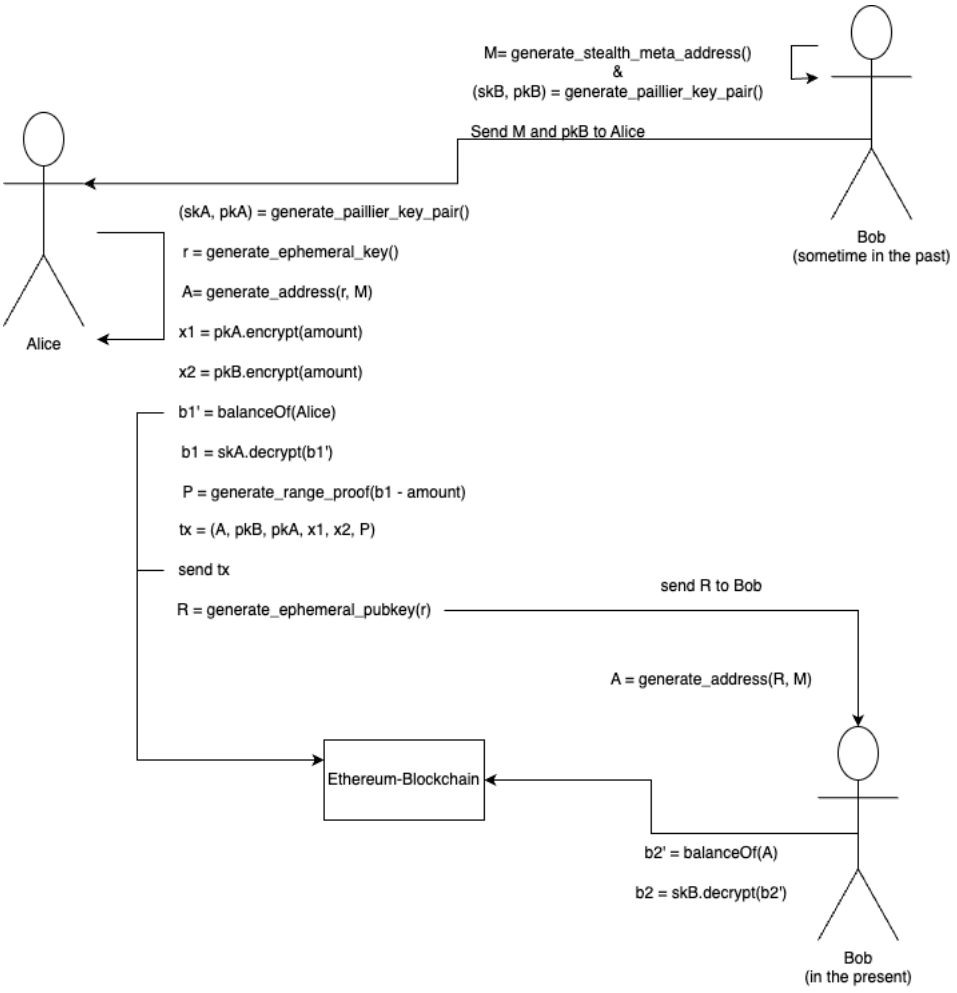

Abbildung 5.1: High level architecture
Quelle: Eigene Bilder

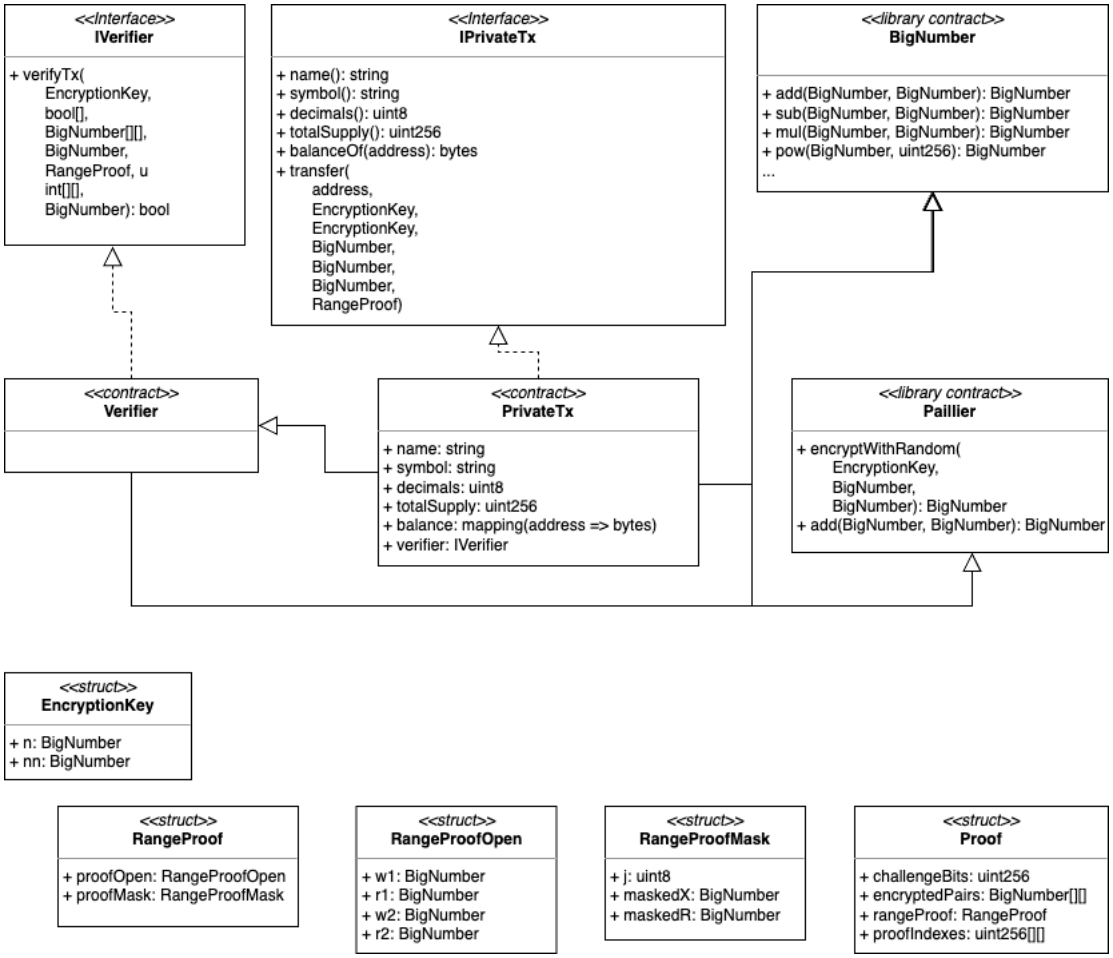

Abbildung 5.2: Smart contract architecture
Quelle: Eigene Bilder

6 Implementierung

In diesem Abschnitt werden die spezifischen technischen Aspekte der Implementierung vertraulicher Transaktionen für tokenisierte Assets behandelt. Dies umfasst Details zu den verwendeten Programmiersprachen, Frameworks und Bibliotheken. Besonders wird auf die Integration von ZKPs und homomorpher Verschlüsselung eingegangen. Die Implementierungsschritte werden systematisch erläutert, um einen klaren Einblick in den technischen Entwicklungsprozess zu gewährleisten.

6.1 Code-Struktur

Der vollständige Code wird in einem Verzeichnis namens `confidential_transactions` abgelegt. Dieses Verzeichnis enthält einen Unterordner namens `packages`, der wiederum in drei Unterordner unterteilt ist: `blockchain`, `paillier` und `stealthAddresses`. Die Bezeichnungen dieser Unterordner geben Aufschluss über den enthaltenen Code. Der Unterordner `blockchain` beherbergt den Code für Smart Contracts und zugehörige Tests. Der Unterordner `paillier` enthält den Code für die Implementierung der Paillier-Verschlüsselung und die Generierung von ZKRPs. Der Unterordner `stealthAddresses` enthält den Code für die Generierung von Stealth-Adressen. Diese organisatorische Struktur erleichtert die Verwaltung und das Auffinden spezifischer Funktionen und Implementierungen im Gesamtprojekt.

6.2 Implementierung von Stealth Adressen

Die Umsetzung der Stealth-Adressen orientiert sich an dem in [5] für secp256k1 beschriebenen Algorithmus ist. Die Implementierung der Stealth-Adressen erfolgte mithilfe von JavaScript. Dabei kamen folgende Bibliotheken zum Einsatz:

1. js-sha3: Diese Bibliothek bietet eine unkomplizierte SHA-3 / Keccak / Shake-Hash-Funktion für JavaScript mit Unterstützung für UTF-8-Kodierung[27].
2. @noble/secp256k1: Diese Bibliothek stellt die schnellste JavaScript-Implementierung der elliptischen Kurve secp256k1 bereit[28].
3. ethereum-public-key-to-address: Diese Bibliothek ermöglicht die Umwandlung eines Ethereum-Öffentlichen Schlüssels in eine zugehörige Adresse[29].

Diese Bibliotheken wurden ausgewählt, um sicherzustellen, dass die Generierung von Stealth-Adressen in JavaScript effizient und zuverlässig erfolgt. Die Implementierung umfasst insgesamt fünf Hauptfunktionen:

1. *generateStealthMetaAddress*: Diese Funktion erstellt Dateien für eine Stealth-Meta-Adresse, bestehend aus einem Spending-Keypair, einem Viewing-Keypair und einer Stealth-Meta-Adresse.
2. *generateStealthInfo*: Diese Funktion akzeptiert eine Stealth-Meta-Adresse als Parameter und erzeugt Informationen über die Stealth-Adresse. Diese umfassen die Stealth-Adresse selbst, den Ephemeral-Public-Key, den ViewTag und den Hashed-Shared-Secret.
3. *parseStealthAddress*: Akzeptiert den Ephemeral-Public-Key, den Spending-Public-Key, den Viewing-Private-Key und den ViewTag als Parameter. Generiert daraus die Stealth-Adresse und das Hashed-Shared-Secret.

4. *computeStealthKey*: Nimmt den Spending-Private-Key und das Hashed-Shared-Secret an und generiert den Private-Key für die Stealth-Adresse.

Die Umsetzung dieser Funktionen zielt darauf ab, die Generierung, Analyse und Verarbeitung von Stealth-Adressen in einem JavaScript-Umfeld effizient und zuverlässig zu ermöglichen. Jede Funktion erfüllt dabei spezifische Aufgaben im Gesamtprozess der Stealth-Adressverwaltung. Eine ausführliche Implementierung von Stealth-Adressen kann im Anhang D.1 eingesehen werden.

6.3 Implementierung von ZKRP

Die Umsetzung des Paillier-Kryptosystems und der Zero-Knowledge Range Proofs (ZKRP) erfolgte in der Programmiersprache Rust. Hierbei wurde die Bibliothek `zk-paillier`[30] von ZengoX[1] geforkt und entsprechend leicht angepasst. Diese Bibliothek stellt eine Sammlung von Paillier-Kryptosystem-Nullwissen-Beweisen bereit. Es wurden jedoch nur zwei ZKRP beibehalten: der ZKRP, der nachweist, dass ein Paillier-Ciphertext im Intervall [0, q] liegt, und der Non-Interactive ZKRP, der dasselbe nachweist. Alle anderen wurden aus der Bibliothek entfernt.

Die Umsetzung des ZKRP beruht auf dem Algorithmus, der im Anhang A, von [31] ausführlich beschrieben ist. Zudem wurde der Ansatz aus [20] (Abschnitt 1.2.2) herangezogen. Dieser Algorithmus besagt, dass wenn $x < \frac{q}{3}$ ist, dann liegt $E(x)$ im Intervall [0, q]. Im Verlauf der Implementierung wurde q auf $2^{256} - 1$ festgelegt.

Um diese Implementierung in eine Non-Interactive Zero-Knowledge Range Proof (NIZKRP) zu überführen, wurde die Fiat-Shamir-Heuristik gemäß [32] angewandt. Der erste Schritt des Algorithmus im Anhang A entfällt, und stattdessen generiert der Beweiser die Challenge-Bits e wie folgt:

$$e = \text{sha256}(N, c_1^1, \ldots, c_1^t, c_2^1, \ldots, c_2^t), \tag{6.1}$$

wobei N der Paillier-Offentlicher-Schlüssel ist und $c_1^1, \ldots, c_1^t, c_2^1, \ldots, c_2^t$ sind zuffälige verschlüsslte Zahlen.
Der Sicherheitsparameter t im Anhang A gibt die Anzahl der Challenge-Bits an, die bei der Generierung und Verifizierung des Proofs verwendet werden. Im Fall des ZKRP beträgt t 40, jedoch sollte dieser Wert beim NIZKRP auf 128 erhöht werden, um die Sicherheit zu gewährleisten[32].

Das Neon-Framework[33] wurde eingesetzt, um aus dem Rust-Code eine Node.js-Bibliothek zu erstellen. Neon erleichtert die Integration von Rust-Funktionalitäten in JavaScript, was die reibungslose Nutzung des Codes aus `zk-paillier` in einer Node.js-Umgebung ermöglicht. Die entwickelte Bibliothek bietet folgende Funktionen:

1. *random(bit_len):* Generiert eine Zufallszahl der Bitlänge *bit_len*.
2. *sample_below(max):* Generiert eine Zufallszahl, die kleiner als *max* ist.
3. *keypair(bit_len):* Generiert ein Paillier-Key-Paar der Bitlänge *bit_len*.
4. *add(n, nn, ciphertext1, ciphertext2):* Addiert zwei Ciphertexts.

[1] ZengoX ist eine Forschungsgruppe, die sich aus Forschern aus dem akademischen Bereich und der Industrie zusammensetzt und sich zum Ziel gesetzt hat, die Schwellenwertkryptographie von der Theorie auf mobiles Gerät (oder jede andere Software/Hardware für Verbraucher) zu bringen.
URL:https://zengo.com/research/

5. *encrypt(n, nn, message):* Verschlüsselt die Nachricht unter Verwendung des Verschlüsselungsschlüssel (n, nn).
6. *encrypt_with_randomness(n, nn, message, randomness):* Verschlüsselt die Nachricht mit der gegebenen Zufälligkeit unter Verwendung des Verschlüsselungsschlüssels (n, nn).
7. *decrypt(p, q, ciphertext):* Entschlüsselt den Ciphertext unter Verwendung des Entschlüsselungsschlüssels (p, q).
8. *prove(n, nn, number, error_factor):* Generiert einen Zero-Knowledge Range Proof (ZKRP), der nachweist, dass der Ciphertext, der die Zahl repräsentiert, im Bereich [0, $2^{256} - 1$] liegt. Der initial erstellte Beweis wurde daraufhin in zwei Strukturen, nämlich `RangeProofOpen` und `RangeProofMask`, aufgeteilt, die den Schritten 4.a bzw. 4.b im Anhang A entsprechen. Hierbei wurden die Indizes der einzelnen Beweiselemente gespeichert. Diese Vorgehensweise ermöglicht die Rekonstruktion des ursprünglichen Beweises mithilfe der Challenge-Bits. Die Unterteilung des Beweises in Elemente gleicher Struktur bzw. Typ erleichtert eine spätere effiziente Verifizierung in Solidity (Solidity unterstützt nur Arrays des gleichen Typs).

Alle Parameter werden im Hexadezimalformat angegeben, mit Ausnahme von *bit_len*, der eine Dezimalzahl ist. Der Build-Prozess resultiert in der Erstellung einer Datei mit dem Namen *index.node*, welche anschließend in Node.js mittels des require-Befehls eingebunden werden kann.

6.4 Implementierung von Smart Contracts

Die Smart Contracts wurden unter Verwendung von Solidity 0.8.17 implementiert, und für Testzwecke kam das Framework Hardhat zum Einsatz.
Die Smart Contracts sind in drei Hauptkategorien unterteilt: Bibliotheken, Schnittstellen und die Smart Contracts selbst. Die Bibliotheken dienen als grundlegende Bausteine und können von anderen Smart Contracts oder Schnittstellen genutzt werden. Die Schnittstellen dienen als klar definierte Verträge, welche von anderen Smart Contracts implementiert werden können. Schließlich umfasst die Kategorie der Smart Contracts jene, die entweder auf Bibliotheken zurückgreifen und/oder eine Schnittstelle implementieren. Diese Strukturierung ermöglicht eine klare und modulare Organisation der Smart Contracts und fördert die Wiederverwendbarkeit von Bibliotheken und Schnittstellen.

Es wurden zwei Bibliotheken verwendet:

1. BigNumbers.sol: Diese Bibliothek stellt eine Reihe von arithmetischen Operationen für große Zahlen bereit. Da alle Berechnungen im Protokoll auf großen Zahlen durchgeführt werden müssen und Solidity nur Zahlen bis 256 Bits unterstützt, wird auf diese Bibliothek zurückgegriffen, um erweiterte numerische Operationen zu ermöglichen.
2. Paillier.sol: Diese Bibliothek wurde eigenständig entwickelt und bietet zwei Funktionen:
   - encrypt_with_random(EncryptionKey ek, BigNumber message, BigNumber random): Diese Funktion verschlüsselt die Nachricht mithilfe des Verschlüsselungsschlüssels und der Zufälligkeit.
   - add(BigNumber n, BigNumber ciphertext1, BigNumber ciphertext2): Diese Funktion addiert zwei verschlüsselte Ciphertexts.

6.4.1 ConfidentialTx.sol

Dieser Smart Contract bietet eine Implementierung der Funktionen, die in `IConfidentialTx.sol` definiert sind. Bei der Bereitstellung des Smart Contracts sind Angaben wie der Name, das Symbol, die Dezimalstellen, die Gesamtanzahl der Token (Total Supply), sowie der Ciphertext des Total Supply erforderlich. Der Ciphertext des Total Supply sollte dabei mit dem Verschlüsselungsschlüssel des Smart Contract Besitzers verschlüsselt sein. Es ist wichtig zu betonen, dass der Total Supply kleiner als $\frac{q}{3}$ sein muss. Dieser Smart Contract wurde im Sepolia-Testnetzwerk unter der Adresse `0xAae689aeA7695aC064a73eBAaA312a0F92525e15` bereitgestellt.

Die Hauptfunktion ist die *Transfer*-Funktion, die eine vertrauliche Überweisung von Vermögen von einem Konto auf ein anderes ermöglicht. Die *Transfer*-Funktion akzeptiert die folgenden Parameter:

1. Die Adresse des Empfängers.
2. Der Verschlüsselungsschlüssel des Senders.
3. Der Verschlüsselungsschlüssel des Empfängers.
4. Der Ciphertext des zu sendenden Betrags. Der Betrag muss mit dem Verschlüsselungsschlüssel des Empfängers verschlüsselt sein.
5. Der multiplikative Inverse des Ciphertexts des zu sendenden Betrags. Der Betrag muss jedoch diesmal mit dem Verschlüsselungsschlüssel des Senders verschlüsselt und dann die multiplikative Inverse berechnet werden. Die multiplikative Inverse ist erforderlich, um eine Subtraktion durchzuführen.
6. Der Ciphertext der Differenz zwischen dem Guthaben des Senders und dem zu sendenden Betrag. Die Differenz muss mit dem Verschlüsselungsschlüssel des Senders verschlüsselt sein.
7. Der Range Proof, der aus zwei Komponenten besteht: `RangeProofOpen` und `RangeProofMask`
8. Hilfsdaten, die aus einem zweidimensionalen Array von Bytes (siehe Schritt 4.d im Anhang A) und einem zweidimensionalen Array von `uint256` (Indizes der Beweiselemente) bestehen.

Eine ausführliche Implementierung vom ConfidentialTx.sol kann im Anhang B.1 eingesehen werden.

6.4.2 VerifierRangeProof.sol

Dieser Smart Contract bietet eine Implementierung der Funktionen, die in `IVerifierRangeProof.sol` definiert sind. Zusätzlich enthält der Smart Contract drei Hilfsfunktionen, die im Folgenden ohne Angabe der Parameter erläutert werden:

1. *computeChallenge()*: Diese Funktion berechnet die Challenge e.
2. *verifyRangeProofOpen()*: Diese Funktion entspricht dem Verifikationsschritt 1 im Anhang A.
3. *verifyRangeProofMask()*: Diese Funktion entspricht dem Verifikationsschritt 2 im Anhang A.

Die *verifyTx*-Funktion stellt die Hauptfunktionalität bereit und überprüft eine Range-Proof auf Korrektheit. Diese Funktion erwartet die folgende Eingaben:

1. Der Verschlüsselungsschlüssel des Senders.
2. Der Ciphertext des zu beweisenden Werts. Dieser Wert entspricht in unserem Fall dem Differenzbetrag.
3. Der Range-Proof.

4. Die Hilfsdaten
5. Das Intervall q.
6. Der Wert, der $(\frac{q}{3})$ entspricht.

Das Ergebnis der Funktion ist entweder wahr oder falsch, abhängig davon, ob die Verifizierung erfolgreich war oder nicht.
Eine ausführliche Implementierung vom VerifierRangeProof.sol kann im Anhang C.1 eingesehen werden.
Dieser Smart Contract wurde im Sepolia-Testnetzwerk unter der Adresse `0x997dA2CC9d0dD4cCaA43DF15Cb5aA` bereitgestellt.

7 Auswertung und Ausblick

In diesem Abschnitt erfolgt die eingehende Vorstellung und kritische Diskussion der Resultate des implementierten Protokolls, begleitet von detaillierten Erwägungen zur möglichen Optimierung. Die Analyse konzentriert sich präzise auf drei wesentliche Dimensionen: Kosten, Skalierbarkeit, sowie Sicherheits- und Zuverlässigkeitsaspekte.

7.1 Kosten

Die Kostenstruktur dieses Protokolls setzt sich aus zwei Hauptkomponenten zusammen: den Verifikationskosten der Non-Interactiv Zero-Knowledge Range Proofs (NIZKRP) und den Kosten im Zusammenhang mit der Aktualisierung der Kontostände. Es sei darauf hingewiesen, dass die vorliegende Analyse explizit die Betrachtung der Kosten für die Aktualisierung der Kontostände ausklammert.

Die Gasverbrauchsmessungen wurden sorgfältig auf einem lokalen Ethereum-Node durchgeführt, um eine genaue Bewertung des Protokolls bei verschiedenen Sicherheitswerten zu ermöglichen. Dabei wurden die Transaktionskosten in Bezug auf das Gaslimit und andere relevante Parameter analysiert. Die Ergebnisse dieser Messungen bieten wichtige Einblicke in die Leistungsfähigkeit und Effizienz des entwickelten Protokolls unter verschiedenen Sicherheitseinstellungen.

Die Tabelle 7.1 präsentiert eine Zusammenfassung dieser Gasverbrauchsmessungen und verdeutlicht, wie sich der Sicherheitswert auf die Transaktionskosten auswirkt. Diese Daten sind entscheidend, um die praktische Anwendbarkeit des Protokolls zu bewerten.

solidity 0.8.17 optimizer=false	t	avg gas
verifyTx	40	5.405.846
	80	17.127.797
	128	47.851.157

Tabelle 7.1: Gasverbrauch des Protokolls

Bei einem Sicherheitswert von 128 überschreitet die Transaktion das Gaslimit pro Block, und es ist erforderlich, manuell zusätzliches Gas bereitzustellen. Um die Effizienz des entwickelten Protokolls besser zu verstehen, wurde es mit einem anderen Protokoll verglichen, das Bulletproofs als Zero-Knowledge Range Proofs (ZKRP) implementiert[34].
Die Smart Contracts in [34] wurden ursprünglich in Solidity 0.4.24 implementiert. Um eine genauere Gasabschätzung zu ermöglichen, wurden sie auf Solidity 0.8.17 portiert, was zu einem durchschnittlichen Gasverbrauch von 5.363.161 Gas führte. Dieser Wert entspricht ungefähr dem Gasverbrauch bei einem Sicherheitswert von 40.
Es ist anzumerken, dass [32] einen Sicherheitswert von 128 für Non-Interactive Zero-Knowledge Range Proofs (NIZKRP) empfiehlt. In Anbetracht dieser Empfehlung scheint Bulletproofs eine vielversprechendere Alternative zu sein, da sie eine höhere Sicherheit bei geringerem Gasverbrauch bieten.

Diese Erkenntnisse legen nahe, dass zukünftige Arbeiten darauf abzielen könnten, Bulletproofs oder ähnliche Ansätze zu verwenden, um die Sicherheit zu erhöhen, ohne dabei die Skalierbarkeit zu beeinträchtigen. Eine eingehende Analyse und Optimierung des entwickelten Protokolls könnte ebenfalls erforderlich sein, um die Anforderungen an Sicherheit und Effizienz besser zu balancieren.

7.2 Skalierbarkeit

Die Erreichung von Skalierbarkeit wird durch die Implementierung von Stealth-Adressen erleichtert. Für jede Transaktion wird eine neue Stealth-Adresse für den Empfänger generiert. Es ist jedoch auch möglich, das Protokoll mit herkömmlichen Ethereum-Adressen zu verwenden.In dieser Situation wird die Skalierbarkeit erheblich beeinträchtigt, da effektiv nur eine Transaktion pro Block durchgeführt werden kann. Dies resultiert aus der Tatsache, dass der ZKRP in Bezug auf einen bestimmten Zustand des Smart Contracts generiert wird. Zether[26] beschreibt dieses Problem als `Front-running`.
Angenommen, Alice überweist zweimal innerhalb eines Blocks Vermögen an Bob (Ethereum-Adresse). Die Transaktion, die zuerst durchgeführt wird, enthält korrekte Daten. Nach der Ausführung haben sich die Kontostände von Alice und Bob geändert. Wenn die zweite Transaktion durchgeführt wird, sind die Daten nicht mehr korrekt, und das Ergebnis nach der Ausführung ist dementsprechend inkorrekt. Das entwickelte Protokoll bietet keinen Mechanismus zur Lösung dieses Problems.

Diese Limitierung in Bezug auf die Skalierbarkeit erfordert eine sorgfältige Abwägung zwischen der Verwendung von Stealth-Adressen und herkömmlichen Ethereum-Adressen. Es ist wichtig, sich der potenziellen Auswirkungen auf die Skalierbarkeit bewusst zu sein und gegebenenfalls geeignete Maßnahmen zu ergreifen, um diesen Herausforderungen zu begegnen. Dies könnte die Integration zusätzlicher Sicherheitsmechanismen oder die Anpassung des Protokolls an verschiedene Adresstypen umfassen.

7.3 Sicherheit und Zuverlässigkeit

Die Sicherheit des Protokolls ist in erster Linie abhängig von der Integrität des Beweisers, der für die korrekte und ehrliche Berechnung des Zero-Knowledge Range Proofs (ZKRP) verantwortlich ist. Diese Abhängigkeit von der Ehrlichkeit der involvierten Parteien stellt in der blockchain-basierten Umgebung eine Herausforderung dar, da die Voraussetzung der grundsätzlichen Ehrlichkeit der Benutzer nicht immer erfüllt werden kann.

Ein zusätzlicher Sicherheitsaspekt liegt in der Möglichkeit der Generierung von falsch negativen Beweisen während des ZKRP-Prozesses. Diese potenziellen Unstimmigkeiten erhöhen das Risiko, dass legitime Transaktionen fälschlicherweise als ungültig interpretiert werden. Die Wahrscheinlichkeit solcher Fehler bei der ZKRP-Generierung erfordert eine kritische Überprüfung und möglicherweise eine Optimierung des Prozesses, um die Robustheit des Protokolls zu gewährleisten.

Gleichzeitig erweist sich der Smart Contract als zuverlässig in seiner Funktionalität. Falsche ZKRP werden korrekt als solche erkannt und verifiziert, während korrekte Beweise ordnungsgemäß validiert werden. Diese Unterscheidung unterstreicht die essentielle Rolle der präzisen Berechnung und Überprüfung der ZKRP für die Integrität und Sicherheit des Protokolls.

Im Kontext der Sicherheitsaspekte erweist sich eine umfassende Evaluierung möglicher Gegenmaßnahmen als imperative Notwendigkeit, um die Gesamtzuverlässigkeit und Sicherheit des Protokolls zu verstärken. Diese proaktive Herangehensweise könnte die Integration zusätzlicher Sicherheitsmechanismen und die Anwendung anspruchsvollerer kryptografischer Techniken beinhalten, um potenzielle Schwächen zu adressieren und die Integrität des Systems zu gewährleisten.

Einige spezifische Überlegungen in dieser Hinsicht umfassen:

1. **Abstrahierung des ZKRP-Prozesses vom Benutzer**
   Eine vielversprechende Strategie besteht in der Abstrahierung der Generierung von ZKRP von der direkten Benutzerinteraktion, insbesondere in Wallet-Anwendungen. Dies ermöglicht eine automatische Erstellung der ZKRP durch das Wallet, wodurch Benutzer von technischen Details entlastet werden und die Sicherheit durch standardisierte, automatisierte Prozesse gesteigert wird.
2. **Redundante Berechnungen**
   Zur Minimierung von Fehlern in den ZKRP besteht die Option der redundanten Berechnung durch mehrere unabhängige Instanzen. Dieser Ansatz stärkt die Zuverlässigkeit des Protokolls, indem die Konsistenz der generierten Beweise überprüft wird. Diese Redundanz trägt dazu bei, potenzielle Anomalien in den Ergebnissen zu identifizieren und zu korrigieren.

Diese Überlegungen reflektieren das Bestreben, nicht nur die technologische Robustheit, sondern auch die Anwendbarkeit und Benutzerfreundlichkeit des Protokolls zu optimieren.

8 Fazit

Im Verlauf dieser Arbeit wurde ein Protokoll zur Durchführung vertraulicher Transaktionen für Vermögenswerte auf der Ethereum-Plattform entworfen und erfolgreich implementiert. Das entwickelte Protokoll basiert auf einer kontenbasierten Struktur und macht Gebrauch vom Paillier-Verschlüsselungssystem, um die Vertraulichkeit der Kontoguthaben sicherzustellen. Weiterhin integriert das Protokoll Non-Interactive Zero-Knowledge Range Proofs (NIZKRP), um zu gewährleisten, dass der Restbetrag während einer Transaktion positiv bleibt.

Die Implementierung erfolgte auf der Ethereum-Blockchain und wurde auf einem lokalen Node ausgiebig getestet. Die Ergebnisse dieser Tests zeigen, dass das Protokoll unter Berücksichtigung eines Sicherheitsparameters von 40 für ZKRP vergleichbare Transaktionskosten aufweist wie andere Protokolle, die Bulletproofs implementieren. Diese Feststellung unterstreicht die Effizienz und Wettbewerbsfähigkeit des entwickelten Protokolls im Kontext der geprüften Sicherheitsparameter.

Es ist jedoch zu beachten, dass das Protokoll bei einem Sicherheitsparameter von 128, wie von Non-Interactive Zero-Knowledge Range Proofs (NIZKRP) empfohlen, nicht anwendbar ist. Diese Einschränkung wirft die Frage nach weiteren Optimierungsmöglichkeiten und Anpassungen auf, um die Skalierbarkeit und Anwendbarkeit des Protokolls bei höheren Sicherheitsparametern zu verbessern.

Insgesamt zeigt die Forschungsarbeit, dass das entwickelte Protokoll eine robuste Grundlage für vertrauliche Transaktionen auf Ethereum bildet, wobei jedoch weiterführende Untersuchungen und Optimierungen erforderlich sind, um den Einsatzbereich des Protokolls zu erweitern und den Empfehlungen für höhere Sicherheitsparameter gerecht zu werden.
Der vollständige Quellcode des entwickelten Protokolls ist auf https://bitbucket.org/kemlohko/masterarbeit/src/master/ verfügbar.

Anhang A: Zero-Knowledge Range Proof, dass ein Wert$x < \frac{q}{3}$ im Intervall $[0, q]$ liegt

For the sake of completeness, in this appendix, we present the ZK-proof that $x \in \mathbb{Z}_q^*$ where $c = \text{Enc}_{pk}(x)$. The value sid is a unique session identifier obtained from the application. The proof that we use proves for $x \in \{0, \ldots, \lfloor \frac{q}{3} \rfloor\}$ that it is in the range

$$\left[-\frac{q}{3}, \frac{2q}{3}\right]$$

Let $\ell = \lfloor \frac{q}{3} \rfloor$. Stated differently, the input is $x \in \{0, \ldots, \ell\}$, and the proof guarantees that $x \in [-\ell, 2\ell]$. In order to use this proof, we begin by subtracting $\frac{q}{3}$ from x. This suffices since we need completeness for $x \in \mathbb{Z}_q^*$, and the proof that we use works with completeness for $x \in \{0, \ldots, \frac{q}{3}\}$ and soundness for $x \in \{-\frac{q}{3}, \ldots, \frac{2q}{3}\}$.

Input: The prover P has input (c, x, r) where $c = \text{Enc}_{pk}(x; r)$ and the Paillier key-pair $(N, \phi(N))$; the verifier V has input c and the Paillier public key N. Both parties have q and $\ell = \lfloor \frac{q}{3} \rfloor$. Both parties have a parameter $t = 40$.

The protocol: P computes $x \leftarrow x - \frac{q}{3}$, and both P and V compute $c \leftarrow c \oplus \frac{q}{3}$ (with the latter operation being the homomorphic property of Paillier to subtract the constant $\frac{q}{3}$).

1. **V's first message:** V chooses a random $e \leftarrow \{0,1\}^t$, computes com $=$ commit(e, sid), and sends com to P. Denote $e = e_1, \ldots, e_t$.

2. **P's first message:**
   (a) P chooses random $w_1^1, \ldots, w_1^t \leftarrow \{\ell, \ldots, 2\ell\}$ and computes $w_2^i = wi_1^i - \ell$ for every $i = 1, \ldots, t$.
   (b) For every $i = 1, \ldots, t$, P switches the values of wi_1^i and wi_2^i with probability $\frac{1}{2}$ (independently for each i).
   (c) For every $i = 1, \ldots, t$, P computes $c_1^i = \text{Enc}_{pk}(w_1^i; r_1^i)$ and $c_2^i = \text{Enc}_{pk}(w_2^i; r_2^i)$, where $r_i^1, r_i^2 \leftarrow \mathbb{Z}_N$ are the randomness used in Paillier encryption.
   (d) P sends $c_1^1, c_2^1, \ldots, c_1^t, c_2^t$ to V.

3. **V's second message:** Upon receiving $c_1^1, c_2^1, \ldots, c_1^t, c_2^t$, V decommits to com, revealing (e, sid) to P.

4. **P's second message:** For $i = 1, \ldots, t$:
   (a) If $e_i = 0$, then P sets $z_i = (w_1^i, r_1^i, w_2^i, r_2^i)$.
   (b) If $e_i = 1$, then P sets z_i as follows. Let $j \in \{1, 2\}$ be the unique value of j such that $x + w_j^i \in \{\ell, \ldots, 2\ell\}$. Then, P sets $z_i = (j, x + w_j^i, r \cdot r_j^i \mod N)$.
   (c) P sends $z_1, \ldots, z_t$ to V.

- **V's output:** V parses z_i appropriately according to the value of e_i. Then, for $i = 1, \ldots, t$:

1. If $e_i = 0$, then V checks that $c_1^i = \mathsf{Enc}_{pk}(w_1^i; r_1^i)$ and $c_2^i = \mathsf{Enc}_{pk}(w_2^i; r_2^i)$ and that one of $w_1^i, w_2^i \in \{\ell, \ldots, 2\ell\}$ while the other is in $\{0, \ldots, \ell\}$, where $z_i = (w_1^i, r_1^i, w_2^i, r_2^i)$.
2. If $e_i = 1$, then V checks that $c \oplus c_j^i = \mathsf{Enc}_{pk}(w_i; r_i)$ and $w_i \in \{\ell, \ldots, 2\ell\}$, where $z_i = (j, w_i, r_i)$.

V outputs 1 if and only if all of the checks pass.

Anhang B: ConfidentialTx.sol

```
1  //SPDX-License-Identifier: MIT
2  pragma solidity 0.8.17;
3
4  import "./libraries/BigNumbers.sol";
5  import "./interfaces/IConfidentialTx.sol";
6  import "./interfaces/IVerifierRangeProof.sol";
7  import "./Utils.sol";
8  import "./libraries/Paillier.sol";
9  import "hardhat/console.sol";
10
11 contract ConfidentialTx is IConfidentialTx{
12   using BigNumbers for BigNumber;
13   /// @notice the max uint256 value (2^256-1)
14   uint256 public constant MAX = 2**256-1;
15
16   string private _name;
17   string private _symbol;
18   uint8 private _decimals;
19   uint256 private _totalSupply;
20   mapping(address => bytes) private _balance;
21   IVerifierRangeProof public verifier;
22
23   event Transfer(address sender, address receiver, bytes amount);
24
25   constructor(
26   string memory __name,
27   string memory __symbol,
28   uint8 __decimals,
29   uint256 _sup,
30   bytes memory _encryptedTotalSupply,
31   address _verifier)
32   {
33     require(_sup <= (MAX / 3), "The total supply must be lesser
         than MAX/3");
34     _name = __name;
35     _symbol = __symbol;
36     _decimals = __decimals;
37     _totalSupply = _sup;
38     _balance[msg.sender] = _encryptedTotalSupply;
39     verifier = IVerifierRangeProof(_verifier);
40
41   }
42
43
44   function name() external view returns (string memory) {
45     return _name;
46   }
47
48   function symbol() external view returns (string memory) {
49     return _symbol;
```

```
50  }
51
52  function decimals() external view returns (uint8) {
53    return _decimals;
54  }
55
56  function totalSupply() external view returns (uint256){
57    return _totalSupply;
58  }
59
60  function balanceOf(address _owner) external view returns (bytes
        memory balance){
61    balance = _balance[_owner];
62  }
63
64  /**
65  * @notice Transfers funds from one account to another while
       ensuring the confidentiality of the sent amount.
66  * @param _to The receiver address
67  * @param _ekSender The sender encryption key
68  * @param _ekReceiver The receiver encryption key
69  * @param _amountToAdd The encrypted amount to send to the
       receiver. Should be encrypted using the receiver encryption
       key
70  * @param _amountToSubtract The multiplicative inverse of the
       encrypted amount to subtract from the sender account. Should
        be encrypted using the sender encryption key
71  * @param _difference The encrypted difference (sender balance -
        amount). encrypted using the sender encryption key
72  * @param _proof The range proof [RangeProofOpen[],
       RangeProofMask[]]
73  * @param _helpData The help data
74  */
75  function transfer(
76  address _to,
77  Utils.EncryptionKey memory _ekSender,
78  Utils.EncryptionKey memory _ekReceiver,
79  BigNumber memory _amountToAdd,
80  BigNumber memory _amountToSubtract,
81  BigNumber memory _difference,
82  Utils.RangeProof memory _proof,
83  Utils.VerificationHelpData memory _helpData
84  ) external returns (bool success)
85  {
86    require(_balance[msg.sender].length > 0, 'Balance to low');
87    // verify the zk proof
88    require(verifier.verifyTx(
89    _ekSender,
90    _difference,
91    _proof,
92    _helpData,
93    BigNumbers.init(MAX/3, false), // rangeScaledThird
94    BigNumbers.init(MAX, false)  // range
```

```
95      ), 'The encrypted value is not in [0, 2^256-1]');
96
97      // Update balances
98      _balance[msg.sender] = (Paillier.add(_ekSender.nn, BigNumbers
            .init(_balance[msg.sender], false), _amountToSubtract)).
            val;
99
100     if(_balance[_to].length == 0) _balance[_to] = _amountToAdd.
            val;
101     else _balance[_to] = (Paillier.add(_ekReceiver.nn, BigNumbers
            .init(_balance[_to],false), _amountToAdd)).val;
102
103     emit Transfer(msg.sender, _to, _amountToSubtract.val);
104     return true;
105   }
106
107   /**
108   * @notice Convert a bytes to a BigNumber
109   * @param _value The bytes to convert.
110   * @return BN A BigNumber representing the value
111   */
112   function toBN(bytes memory _value) public view returns (
          BigNumber memory BN){
113     BN = BigNumbers.init(_value, false);
114   }
115
116 }
```

Quelltext B.1: ConfidentialTx.sol

Anhang C: VerifierRangeProof.sol

```
1  //SPDX: License-Identifier:UNLICENSED
2  pragma solidity 0.8.17;
3  import './libraries/BigNumbers.sol';
4  import './Utils.sol';
5  import './libraries/Paillier.sol';
6  import './interfaces/IVerifierRangeProof.sol';
7  import "hardhat/console.sol";
8
9  contract VerifierRangeProof is IVerifierRangeProof {
10   using BigNumbers for BigNumber;
11
12   uint8 public SECURITY_PARAMETER = 40; //128;
13   address public owner;
14
15   constructor() {
16     owner = msg.sender;
17   }
18
19   /**
20   *  Zero-knowledge range proof that a value x<q/3 lies in
        interval [0,q].
21   * @param _ek The paillier encryption key (public key) that
        encrypt the ciphertest
22   * @param _ciphertext The ciphertest encrypted with _ek
23   * @param _proof The range proof ([proofOpen, proofMask])
24   * @param _helpData Some help data for the verification ({
        challenge, encrypted_pairs, proofIndexes})
25   * @param _rangeScaledThird The range/3
26   * @param range The range
27   * @return True if the verification is successful otherwise
        False
28   */
29   function verifyTx(
30   Utils.EncryptionKey memory _ek,
31   BigNumber memory _ciphertext,
32   Utils.RangeProof memory _proof,
33   Utils.VerificationHelpData memory _helpData,
34   BigNumber memory _rangeScaledThird,
35   BigNumber memory range
36   //uint8 errorFactor
37   ) public returns (bool)
38   {
39     // We cannot directly compute range/3 because division are
        very expensive onchain.
40     // Check rangeScaledThird = range / 3
41     require(range.divVerify(BigNumbers.init(3, false),
        _rangeScaledThird));
42     BigNumber memory rangeScaledTwoThirds = _rangeScaledThird.mul
        (BigNumbers.init(BigNumbers.TWO, false));
43     bool[] memory verifications = new bool[](SECURITY_PARAMETER);
```

```
44      uint256 challenge = computeChallenge(_ek.n.val, _helpData.
            encryptedPairs[0], _helpData.encryptedPairs[1]) ;
45
46      for (uint8 i = 0; i < SECURITY_PARAMETER; i++) {
47        bool ei = challenge >> uint8(255-i) & 1 == 1; // Check if
              the bit at index i is set or not.
48        if ( !ei && _helpData.proofIndexes[0][i] != 130 ){ // ei =
              0, 130 is considered a sentinel
49          verifications[i] = verifyProofOpen(
50          _ek,
51          _proof.proofOpen[_helpData.proofIndexes[0][i]].w1,
52          _proof.proofOpen[_helpData.proofIndexes[0][i]].r1,
53          _proof.proofOpen[_helpData.proofIndexes[0][i]].w2,
54          _proof.proofOpen[_helpData.proofIndexes[0][i]].r2,
55          _helpData.encryptedPairs[0][i],
56          _helpData.encryptedPairs[1][i],
57          _rangeScaledThird,
58          rangeScaledTwoThirds
59          );
60        }
61
62        else if ( ei && _helpData.proofIndexes[1][i] != 130){ // ei
              = 1
63          bytes memory c1_2i;
64          if (_proof.proofMask[_helpData.proofIndexes[1][i]].j ==
              1) c1_2i = _helpData.encryptedPairs[0][i];
65          else c1_2i = _helpData.encryptedPairs[1][i];
66          verifications[i] = verifyProofMask(
67          _ek,
68          _ciphertext,
69          c1_2i,
70          _proof.proofMask[_helpData.proofIndexes[1][i]].maskedX,
71          _proof.proofMask[_helpData.proofIndexes[1][i]].maskedR,
72          _rangeScaledThird,
73          rangeScaledTwoThirds
74          );
75        }
76      }
77      for (uint256 i = 0; i < verifications.length; i++) {
78        if (!verifications[i]) {
79          return false;
80        }
81      }
82      return true;
83    }
84
85    // Helper functions
86    /**
87    * @notice Compute the challenge
88    * @param _n The n component of the encryption key
89    * @param _c1 The first encrypted pair
90    * @param _c2 The second encrypted pair
91    * @return A uint256 representing the challenge
```

```
92    */
93    function computeChallenge(bytes memory _n, bytes[] memory _c1,
          bytes[] memory _c2) public view returns(uint256) {
94      bytes memory output = abi.encodePacked(_n);
95      for(uint256 i = 0; i < _c1.length; i++) {
96        output = abi.encodePacked(output, _c1[i]);
97      }
98      for(uint256 i = 0; i < _c2.length; i++) {
99        output = abi.encodePacked(output, _c2[i]);
100     }
101     return uint256(sha256(output));
102   }
103
104   /**
105   * @notice Verify the proof open part.
106   * @param _ek The encryption key
107   * @param _w1 A BigNumber representing the w1 compoment of the
          ProofOpen
108   * @param _r1 A BigNumber representing the r1 compoment of the
          ProofOpen
109   * @param _w2 A BigNumber representing the w2 compoment of the
          ProofOpen
110   * @param _r2 A BigNumber representing the r2 compoment of the
          ProofOpen
111   * @param _c1i A bytes representing c1[i]. c1 => first encrypted
          pair
112   * @param _c2i A bytes representing c2[i]. c2 => second
          encrypted pair
113   * @param _rangeScaledThird A BigNumber representing range/3
114   * @param _rangeScaledTwoThirds A BigNUmber representing (range
          /3)*2
115   * @return True if the verification is successful otherwise
          False
116   */
117   function verifyProofOpen(
118   Utils.EncryptionKey memory _ek,
119   BigNumber memory _w1,
120   BigNumber memory _r1,
121   BigNumber memory _w2,
122   BigNumber memory _r2,
123   bytes memory _c1i,
124   bytes memory _c2i,
125   BigNumber memory _rangeScaledThird,
126   BigNumber memory _rangeScaledTwoThirds
127   ) private returns(bool)
128   {
129     bool success = true;
130     BigNumber memory expectedC1i = Paillier.encrypt_with_random(
          _ek, _w1, _r1);
131     BigNumber memory expectedC2i = Paillier.encrypt_with_random(
          _ek, _w2, _r2);
132
```

```
133     if (!( expectedC1i.eq(BigNumbers.init(_c1i, false)) ||
            expectedC2i.eq(BigNumbers.init(_c2i, false)))) success =
            false;
134
135     if (!((_w2.lt(_rangeScaledThird) && _w1.gt(_rangeScaledThird)
             && _w1.lt(_rangeScaledTwoThirds)) ||
136     (_w1.lt(_rangeScaledThird) && _w2.gt(_rangeScaledThird) &&
            _w2.lt(_rangeScaledTwoThirds))))
137     {
138       success= false;
139     }
140     return success;
141   }
142
143   /**
144   * @notice Verify the proof mask part.
145   * @param _ek The encryption key that encrypt the ciphertest
146   * @param _ciphertext A BigNumber representing the encrypted
        value
147   * @param _c1_2i A bytes taken from either c1 or c2 (c1 => first
        encrypted pair, c2 => second encrypted pair)
148   * @param _maskedX A BigNumber representing the maskedX
        compoment of the ProofMask
149   * @param _maskedR A BigNumber representing the maskedR
        compoment of the ProofMaskedThird
150   * @param _rangeScaledTwoThirds A BigNUmber representing (range
        /3)*2
151   * @return True if the verification is successful otherwise
        False
152   */
153   function verifyProofMask(
154   Utils.EncryptionKey memory _ek,
155   BigNumber memory _ciphertext,
156   bytes memory _c1_2i,
157   BigNumber memory _maskedX,
158   BigNumber memory _maskedR,
159   BigNumber memory _rangeScaledThird,
160   BigNumber memory _rangeScaledTwoThirds
161   ) private returns(bool)
162   {
163     bool success = true;
164     BigNumber memory c = BigNumbers.init(_c1_2i, false).modmul(
            _ciphertext, _ek.nn);
165     BigNumber memory encZi = Paillier.encrypt_with_random(_ek,
            _maskedX, _maskedR);
166     if (!c.eq(encZi)) success = false;
167     if (_maskedX.lt(_rangeScaledThird) || _maskedX.gt(
            _rangeScaledTwoThirds)) success = false;
168
169     return success;
170   }
171
172   /**
```

```
173     * @notice Update the security parameter
174     * @dev A higher security parameter enhances security but comes
            with increased costs. The recommended value is 128.
175     * @param _errorFactor A uint8 representing the new security
            pparameter.
176     */
177     function updateErrorFactor(uint8 _errorFactor) public {
178       require(msg.sender == owner, 'You are not allowed to perfom
              this action.');
179       SECURITY_PARAMETER = _errorFactor;
180     }
181
182 }
```

Quelltext C.1: VerifierRangeProof.sol

Anhang D: Stealthaddresses.mjs

```
1
2  import pkg from 'js-sha3';
3  const { keccak256 } = pkg;
4  import * as secp from '@noble/secp256k1'; // Only support: 1)
       browsers 2) node.js 19+ 3) deno, other envs with crypto
5  import publicKeyToAddress from 'ethereum-public-key-to-address';
6
7
8  export const generateStealthMetaAddress = () => {
9    const spendingPrivateKey = randomPrivateKey();
10   const viewingPrivateKey = randomPrivateKey();
11   const spendingPublicKey = uintArrayToHex(secp.getPublicKey(
         spendingPrivateKey, true));
12   const viewingPublicKey = uintArrayToHex(secp.getPublicKey(
         viewingPrivateKey, true));
13   const stealthMetaAddress = "st:eth:0x"+spendingPublicKey+
         viewingPublicKey;
14
15   return {
16     spendingPrivateKey: "0x"+spendingPrivateKey.toString(16),
17     viewingPrivateKey: "0x"+viewingPrivateKey.toString(16),
18     spendingPublicKey: "0x"+spendingPublicKey,
19     viewingPublicKey: "0x"+viewingPublicKey,
20     stealthMetaAddress
21   }
22 }
23
24 export const generateStealthInfo = stealthMetaAddress => {
25   if (!stealthMetaAddress.startsWith('st:eth:0x')) {
26     throw 'Wrong address format; Address must start with `st:eth
         :0x...``';
27   }
28
29   const R_pubkey_spend = secp.ProjectivePoint.fromHex(
         stealthMetaAddress.slice(9, 75));
30   //console.log('spend pub', R_pubkey_spend);
31
32   const R_pubkey_view = secp.ProjectivePoint.fromHex(
         stealthMetaAddress.slice(75,));
33   //console.log('view pub', R_pubkey_view);
34
35   const ephemeralPrivatKey = randomPrivateKey();
36
37   const ephemeralPublicKey = secp.getPublicKey(ephemeralPrivatKey
         , true);
38   //console.log('ephemeral pub', ephemeralPublicKey)
39
40   const sharedSecret = secp.getSharedSecret(ephemeralPrivatKey,
         R_pubkey_view.toHex(), true);
41   //console.log('shared secret', sharedSecret);
```

```
42
43    const hashedSharedSecret = keccak256(Buffer.from(sharedSecret.
          slice(1)));
44    //console.log('hashed shared secret', hashedSharedSecret);
45
46    const ViewTag = hashedSharedSecret.slice(0,2);
47
48    const hashedSharedSecretPoint = secp.ProjectivePoint.
          fromPrivateKey(Buffer.from(hashedSharedSecret, "hex"));
49    //console.log('hashed secret point', hashedSharedSecretPoint);
50
51    const stealthPublicKey = R_pubkey_spend.add(
          hashedSharedSecretPoint);
52
53    const stealthAddress = publicKeyToAddress(stealthPublicKey.
          toHex());
54    //console.log('stealth address:', stealthAddress);
55
56    return {
57      stealthAddress: stealthAddress,
58      ephemeralPublicKey: "0x"+Buffer.from(ephemeralPublicKey).
            toString('hex'),
59      ViewTag: "0x"+ViewTag.toString('hex'),
60      HashedSecret: hashedSharedSecret
61    };
62
63  }
64
65  export const parseStealthAddress = (
66  ephemeralPublicKey_hex,
67  //stealthAddress_given,
68  spendingPublicKey_hex,
69  viewingPrivateKey,
70  viewTag_given
71  ) => {
72
73    const ephemeralPublicKey = secp.ProjectivePoint.fromHex(
          ephemeralPublicKey_hex.slice(2));
74
75    const spendingPublicKey = secp.ProjectivePoint.fromHex(
          spendingPublicKey_hex.slice(2), true);
76
77    const sharedSecret = secp.getSharedSecret(BigInt(
          viewingPrivateKey), ephemeralPublicKey.toHex());
78
79    const hashedSharedSecret = keccak256(Buffer.from(sharedSecret.
          slice(1)));
80    //console.log('hash secret', hashedSharedSecret);
81
82    const viewTag = hashedSharedSecret.slice(0, 2).toString(16);
83    //console.log('view', viewTag);
84
85    if ('0x'+viewTag !== viewTag_given) {
```

```
86      return false;
87    }
88
89    const hashedSharedSecretPoint = secp.ProjectivePoint.
          fromPrivateKey(Buffer.from(hashedSharedSecret, 'hex'));
90
91    const stealthPublicKey = spendingPublicKey.add(
          hashedSharedSecretPoint);
92
93    const stealthAddress = publicKeyToAddress(stealthPublicKey.
          toHex());
94    return {
95      stealthAddress,
96      //ephemeralPublicKey_hex,
97      hashedSharedSecret: "0x" + hashedSharedSecret.toString('hex')
98    };
99    /*
100   if (stealthAddress === stealthAddress_given) {
101     return {
102       stealthAddress,
103       //ephemeralPublicKey_hex,
104       hashedSharedSecret: "0x" + hashedSharedSecret.toString('hex
              ')
105     };
106   }
107   return false;
108   */
109
110 }
111
112 export const computeStealthKey = (spendingPrivateKey,
        hashedSharedSecret) => {
113   return BigInt(spendingPrivateKey) + BigInt(hashedSharedSecret);
114 }
115
116 export const toEthAddress = (publicKey) => publicKeyToAddress(
        publicKey);
117
118 // Helper Functions
119 const randomPrivateKey = () => {
120   var randPrivateKey = secp.utils.randomPrivateKey();
121   return BigInt(`0x${Buffer.from(randPrivateKey, "hex").toString(
          'hex')}`);
122 }
123
124 const uintArrayToHex = (uintArray) => {
125   return secp.etc.bytesToHex(uintArray);
126 }
```

Quelltext D.1: generateStealthAddresses.mjs

Literaturverzeichnis

[1] S. Nakamoto, „Bitcoin whitepaper“, *URL: https://bitcoin. org/bitcoin. pdf-(: 17.07. 2019)*, 2008.

[2] V. Buterin. „Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform“. Zugriffsdatum: 8.08.2023. (2014), Adresse: https://ethereum.org/669c9e2e2027310b6b3cdce6e1c52962/Ethereum_Whitepaper_-_Buterin_2014.pdf.

[3] F. Vogelsteller und V. Buterin. „ERC-20: Token Standard“. (Nov. 2015), Adresse: https://eips.ethereum.org/EIPS/eip-20.

[4] ethereum.org. „EIP Etehereum Improvment Proposal“. gelesen am: 11.10.2023. (2015), Adresse: https://eips.ethereum.org.

[5] T. Wahrstätter, M. Solomon, B. DiFrancesco und V. Buterin. „ERC-5564: Stealth Addresses [DRAFT]“. (Aug. 2022), Adresse: https://eips.ethereum.org/EIPS/eip-5564.

[6] V. Buterin u. a. „ERC-4337: Account Abstraction Using Alt Mempool [DRAFT]“. (Sep. 2021), Adresse: https://eips.ethereum.org/EIPS/eip-4337.

[7] R. L. Rivest, A. Shamir und Y. Tauman, „How to leak a secret“, in *Advances in Cryptology—ASIACRYPT 2001: 7th International Conference on the Theory and Application of Cryptology and Information Security Gold Coast, Australia, December 9–13, 2001 Proceedings 7*, Springer, 2001, S. 552–565.

[8] C. Gentry, „Fully homomorphic encryption using ideal lattices“, in *Proceedings of the forty-first annual ACM symposium on Theory of computing*, 2009, S. 169–178.

[9] T. P. Pedersen, „Non-interactive and information-theoretic secure verifiable secret sharing“, in *Annual international cryptology conference*, Springer, 1991, S. 129–140.

[10] T. ElGamal, „A public key cryptosystem and a signature scheme based on discrete logarithms“, *IEEE transactions on information theory*, Jg. 31, Nr. 4, S. 469–472, 1985.

[11] 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.

[12] R. D. Carmichael, „On composite numbers P which satisfy the Fermat congruence a p-1≡ 1 mod P“, *The American Mathematical Monthly*, Jg. 19, Nr. 2, S. 22–27, 1912.

[13] S. Goldwasser, S. Micali und C. Rackoff, „The Knowledge Complexity of Interactive Proof-Systems“, in *Proceedings of the 17th Annual ACM Symposium on Theory of Computing*, 1985.

[14] M. Blum, P. Feldman und S. Micali, „Non-Interactive Zero-Knowledge and Its Applications“, in *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, Ser. STOC '88, Chicago, Illinois, USA: Association for Computing Machinery, 1988, S. 103–112, ISBN: 0897912640. DOI: 10.1145/62212.62222. Adresse: https://doi.org/10.1145/62212.62222.

[15] N. Bitansky, R. Canetti, A. Chiesa und E. Tromer, „From Extractable Collision Resistance to Succinct Non-Interactive Arguments of Knowledge, and Back Again“, in *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, Ser. ITCS '12, Cambridge, Massachusetts: Association for Computing Machinery, 2012, S. 326–349, ISBN: 9781450311151. DOI: 10.1145/2090236.2090263. Adresse: https://doi.org/10.1145/2090236.2090263.

[16] A. Banerjee, M. Clear und H. Tewari, „Demystifying the Role of zk-SNARKs in Zcash“, in *2020 IEEE Conference on Application, Information and Network Security (AINS)*, 2020, S. 12–19. DOI: 10.1109/AINS50155.2020.9315064.

[17] E. Ben-Sasson, I. Bentov, Y. Horesh und M. Riabzev, „Scalable, transparent, and post-quantum secure computational integrity“, *Cryptology ePrint Archive*, 2018.

[18] I. B. Damgård, „Practical and provably secure release of a secret and exchange of signatures“, in *Workshop on the Theory and Application of of Cryptographic Techniques*, Springer, 1993, S. 200–217.

[19] E. Fujisaki und T. Okamoto, „Statistical zero knowledge protocols to prove modular polynomial relations“, in *Advances in Cryptology—CRYPTO'97: 17th Annual International Cryptology Conference Santa Barbara, California, USA August 17–21, 1997 Proceedings 17*, Springer, 1997, S. 16–30.

[20] F. Boudot, „Efficient proofs that a committed number lies in an interval“, in *International Conference on the Theory and Applications of Cryptographic Techniques*, Springer, 2000, S. 431–444.

[21] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille und G. Maxwell, „Bulletproofs: Short proofs for confidential transactions and more“, in *2018 IEEE symposium on security and privacy (SP)*, IEEE, 2018, S. 315–334.

[22] R. S. Alexey Pertsev Roman Semenov. „Tornado Cash Privacy Solution“. gelesen am: 20.10.2023. (2019), Adresse: https://web.archive.org/web/20211026053443/https://tornado.cash/audits/TornadoCash_whitepaper_v1.4.pdf.

[23] M. Albrecht, L. Grassi, C. Rechberger, A. Roy und T. Tiessen, „MiMC: Efficient encryption and cryptographic hashing with minimal multiplicative complexity“, in *International Conference on the Theory and Application of Cryptology and Information Security*, Springer, 2016, S. 191–219.

[24] Z. J. Williamson. „The AZTEC Protocol“. gelesen am: 20.10.2023. (2018), Adresse: https://github.com/AztecProtocol/aztec-v1/blob/master/AZTEC.pdf.

[25] M. D. et al. „fhEVM: Confidential EVM Smart Contract using Fully Homomorphic Encryption“. Zugriffsdatum: 20.08.2023. (2023), Adresse: https://github.com/zama-ai/fhevm/blob/main/fhevm-whitepaper.pdf.

[26] B. B. et al. „Zether: Towards Privacy in a Smart Contract World“. Zugriffsdatum: 10.06.2023. (2019), Adresse: https://crypto.stanford.edu/~buenz/papers/zether.pdf.

[27] Y.-C. Chen. „js-sha3“. (), Adresse: https://www.npmjs.com/package/js-sha3.

[28] P. Miller. „@noble/secp256k1“. (), Adresse: https://www.npmjs.com/package/@noble/secp256k1.

[29] „ethereum-public-key-to-address“. (), Adresse: https://www.npmjs.com/package/ethereum-public-key-to-address.

[30] Z. Team. „ZenGo-X/zk-paillier“. Zugriffsdatum: 05.07.2023. (2021), Adresse: https://github.com/ZenGo-X/zk-paillier.

[31] Y. Lindell, „Fast secure two-party ECDSA signing“, in *Advances in Cryptology–CRYPTO 2017: 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20–24, 2017, Proceedings, Part II 37*, Springer, 2017, S. 613–644.

[32] A. Fiat und A. Shamir, „How to prove yourself: Practical solutions to identification and signature problems“, in *Conference on the theory and application of cryptographic techniques*, Springer, 1986, S. 186–194.

[33] T. N. Contributors. „Neon“. Zugriffsdatum: 05.07.2023. (2023), Adresse: https://neon-bindings.com/.

[34] S. V. Alex Vlasov. „BANKEY/BulletproofJS“. Zugriffsdatum: 05.07.2023. (2018), Adresse: https://github.com/BANKEX/BulletproofJS.

Eidesstattliche Erklärung

Hiermit versichere ich – Alex Kemloh Kouyem – an Eides statt, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Sämtliche Stellen der Arbeit, die im Wortlaut oder dem Sinn nach Publikationen oder Vorträgen anderer Autoren entnommen sind, habe ich als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt oder anderweitig veröffentlicht.

Mittweida, 08. Dezember 2023	
Ort, Datum	BSc. Alex Kemloh Kouyem