Analyse und Vergleich von Smart Contract Sprachen der Blockchain-Plattform Ethereum

Fakultät Design, Medien und Information
der Hochschule für Angewandte Wissenschaften Hamburg

Abschlussarbeit

im Studiengang Media Systems
zur Erlangung des akademischen Grades
Bachelor of Science

vorgelegt von

Fabian Hensel
[redacted]

im Februar 2022

Erstprüfer:	Prof. Dr. Andreas Plaß
Zweitprüferin:	Prof. Dr. Larissa Putzar

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Abschlussarbeit selbstständig und nur unter Verwendung der von mir angegebenen Quellen und Hilfsmittel verfasst zu haben. Sowohl inhaltlich als auch wörtlich entnommene Inhalte wurden als solche kenntlich gemacht. Die Arbeit hat in dieser oder vergleichbarer Form noch keinem anderem Prüfungsgremium vorgelegen. Ich bin damit einverstanden, dass die Arbeit veröffentlicht wird.

Datum: _____ Unterschrift: _____

Kurzzusammenfassung

Smart Contracts haben in den letzten Jahren, insbesondere durch die Blockchain-Plattform Ethereum, zunehmend an Bedeutung gewonnen. Der daraus folgende rapide Anstieg an Smart Contract Programmiersprachen macht es schwer die richtige Auswahl unter diesen zu treffen. Daher ist es notwendig aufzuzeigen, welche Smart Contract Sprachen es gibt und zu analysieren über welche Eigenschaften diese verfügen.

Das Ziel in der vorliegenden Arbeit ist es zu beantworten, welche der gegebenen Smart Contract Sprachen die optimale Wahl zur Erstellung von Smart Contracts ist. Dazu wird folgende Frage gestellt: „Welche Smart Contract Sprache ist für den Einstieg und für die allgemeine Entwicklung am besten geeignet?“

Um die Forschungsfrage zu beantworten, wurde sich auf die Blockchain-Plattform Ethereum begrenzt und anhand einer Literaturrecherche alle für diese gegebenen Smart Contract Sprachen identifiziert. Die Auswahl zur Analyse wurde schlussendlich auf drei Sprachen begrenzt. Die ausgewählten Sprachen Solidity, Vyper und Fe wurden anhand eines ERC20-Tokens analysiert und verglichen. Dabei wurde zu den jeweiligen Sprachen mittels einer Nutzwertanalyse auf die Kriterien Lesbarkeit, Funktionalitäten, Verfügbarkeit, Lernaufwand, Sicherheit und Effizienz eingegangen. Die Kriterien wurden ebenfalls anhand der Literaturrecherche und in Bezug auf herkömmliche Programmiersprachen prozentual zueinander gewichtet. An jedes Kriterium wurden Punkte von 1 bis 5 vergeben, um anschließend zu jeder Sprache eine gewichtete Endsumme als Ergebnis zu erhalten.

Die Ergebnisse der Analyse zeigen, dass für den Einstieg in die Smart Contract Programmierung die Sprache Vyper am besten geeignet ist. Vyper hat bei den Kriterien Lesbarkeit, Lernaufwand, Sicherheit und Effizienz die besten Ergebnisse erzielen können. Dies macht sie zur optimalen ersten Wahl, um grundlegende Konzepte der Blockchain-Plattform Ethereum und von Smart Contracts zu verinnerlichen. Für die allgemeine Entwicklung lässt sich keine optimale Wahl ermitteln, da keine der Sprachen alle Kriterien perfekt erfüllt hat. Die Auswahl hängt hier von den Präferenzen des jeweiligen Entwicklers ab.

Abstract

Smart contracts have become increasingly important in recent years, especially due to the Ethereum blockchain platform. The resulting rapid increase of smart contract programming languages makes it difficult to make the right choice among them. Therefore, it is necessary to show which smart contract languages exist and to analyze which features they have.

The goal of this paper is to answer which of the given smart contract languages is the optimal choice for creating smart contracts. To this end, the following question is posed: "Which smart contract language is the best for getting started with smart contract development and for general development of smart contracts?"

In order to answer the research question, the focus was limited to the Ethereum blockchain platform and a literature research was used to identify all smart contract languages given for this platform. The selection for analysis was ultimately limited to three languages. The selected languages Solidity, Vyper and Fe were analyzed and compared using an ERC20 token. A utility analysis was performed on the criteria of readability, functionality, availability, learning effort, security, and efficiency for each language. The criteria were weighted likewise on the basis of the literature research and in relation to conventional programming languages proportionally to each other. To each criterion points were assigned from 1 to 5, in order to receive afterwards to each language a weighted final sum as result.

The results of the analysis show that Vyper is the most suitable for getting started with smart contract programming. Vyper has been able to achieve the best results on the criteria of readability, learning effort, security and efficiency. This makes it the optimal first choice for internalizing basic concepts of the Ethereum blockchain platform and smart contracts. For general development, it is not possible to identify an optimal choice, as none of the languages perfectly met all the criteria. The choice here depends on the preferences of the respective developer.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

ABI Application Binary Interface

DApp Dezentrale Applikation

EIP Ethereum Improvement Proposals

EOA Externally Owned Account

ERC Ethereum Request for Comments

ETH Ether

EVM Ethereum Virtual Machine

EWasm Ethereum (flavored) WebAssembly

GWei GigaWei

NFT Non-fungible Token

PoS Proof-of-Stake

PoW Proof-of-Work

P2P Peer to Peer

Wasm WebAssembly

Glossar

Application Binary Interface (ABI): Die »Binärschnittstelle« zwischen zwei Computerprogrammen auf Maschinenebene. Nicht zu verwechseln mit dem API, welches die »Quellschnittstelle« auf höherer Ebene (Quellenebene) definiert.[1]

Account: Bei Ethereum und anderen Blockchain-Plattformen ein Konstrukt bestehend aus Adresse, Guthaben und Nonce, welches optional noch Speicher und Code beinhaltet. Ein Account ist entweder ein Externally Owned Account oder ein Contract Account.[2]

Adresse: Ein mittels kryptografischen Hashfunktionen aus dem öffentlichen Schlüssel abgeleiteter Hash. Dient dem Empfangen und Senden von Transaktionen.

Bitcoin: Die erste weltweite Kryptowährung auf Grundlage einer Blockchain als Datenbankmanagementsystem.

Block: Ein strukturierter Datensatz, der beliebige Transaktionen mit Daten enthalten kann.[3]

Blockchain: Eine dezentrale, chronologisch aktualisierte Datenbank mit einem aus dem Netzwerk hergestellten Konsensmechanismus zur dauerhaften digitalen Verbriefung von Eigentumsrechten.[4]

Bytecode: Ein für eine virtuelle Maschine lesbarer Programmcode. Für die EVM in zwei Varianten unterteilt, dem Creation Bytecode und dem Runtime Bytecode.

Contract Account: Ein von der Logik eines Smart Contracts gesteuerter Account. Er verfügt über keinen privaten Schlüssel.

DApp: Eine dezentrale Applikation bestehend aus einem Smart Contract (Backend) und einer Web-Benutzerschnittstelle.

Difficulty: Bestimmt den Schwierigkeitsgrad und Zeitaufwand für die Ermittlung des richtigen Hashs eines Blocks.[5]

Digitale Signatur: Stellt die eindeutige Identifizierung des Kommunikationspartners sicher. Wird im Zusammenhang einer Blockchain für die Verifizierung von Transaktionen benötigt.

[1] vgl. QA Stack, 2021
[2] vgl. Antonopoulos und Wood, 2019, Glossar
[3] vgl. Westfälische Hochschule, 2021
[4] Mitschele, 2016
[5] Bitpanda, 2021

Elliptic Curve Cryptography (ECC): Die von Bitcoin, Ethereum und co. verwendete Kryptografie zur asymmetrischen Verschlüsselung.

Elliptic Curve Digital Signature Algorithm (ECDSA): Das von Bitcoin, Ethereum und co. verwendete kryptografische System zur digitalen Signatur von Transaktionen.

Ethereum Request for Comments (ERC): Eine Reihe von Standardisierungen für die Erstellung von Tokens. Es gibt Standards sowohl für fungible Tokens als auch für NFTs.

Ether: Die Kryptowährung der Plattform Ethereum.

Ethereum: Eine dezentralisierte Open-Source-basierte Plattform, welche die Ausführung von Smart Contracts ermöglicht.

Ethereum Virtual Machine (EVM): Eine virtuelle Maschine zur Ausführung von für diese übersetzten Bytecode. Sie berechnet die Aktualisierung eines neu entstandenen Zustands.

Ethereum WebAssembly (EWasm): Eine auf WebAssembly aufbauende Alternative zur EVM. Geplant für Ethereum 2.0.

Externally Owned Account (EOA): Ein Account, der von oder für einen Benutzer erstellt wird, um von außen mit der Blockchain interagieren zu können. Verfügt über einen privaten Schlüssel und einen öffentlichen Schlüssel sowie über eine Adresse.[6]

Fe: Eine sich im Alpha-Stadium (Stand: 02.12.2021) befindende Programmiersprache zur Entwicklung von Smart Contracts.

Gas: Der »Treibstoff« zur Ausführung von Smart Contracts, welcher unter anderem Endlosschleifen im Ethereum-Netzwerk verhindern soll.

GasLimit: Die maximale Anzahl an Gas die eine Transaktion verbrauchen darf.

GasPrice: Die Anzahl an Gas, die der Urheber der Transaktion bereit ist für diese (in Wei) zu bezahlen.

Genesis Block: Der erste Block einer Blockchain. Er wird manchmal auch als Block Null bezeichnet.

Global State: Der globale Zustand von Ethereum (auch World State genannt). Eine Abbildung aller im Ethereum-Netzwerk existierenden Accounts.

GWei: Die Ether-Unterwährung, welche für die anfallenden Kosten des aufgebrachten Gases angegeben wird.

Hash: Das Ergebnis einer Eingabe, welche einer Hashfunktion übergeben wurde. Wird in der digitalen Welt als »Fingerabdruck« verwendet.

[6] vgl. Antonopoulos und Wood, 2019, S. 17

Keccak-256: Die von Ethereum verwendete kryptografische Hashfunktion.

Konsensmodell: Ein Modell zur Findung eines gemeinsamen Konsens in einem Blockchain-System. Die bekanntesten Konsensmodelle sind PoW und PoS.

Mempool: Der Wartebereich für Transaktionen, welche noch nicht einem Block hinzugefügt wurden.

Messages: Ein Konstrukt, welches von Smart Contracts verwendet wird, um miteinander zu kommunizieren.

Mining: Der Prozess, durch welchen unverpackte Transaktionen in Blöcke gepackt und der Blockchain hinzugefügt werden. Das Mining kommt im Zusammenhang mit PoW vor.

Non-fungible Token (NFT): Repräsentiert eine einmalige Sonderanfertigung in seinem Ökosystem.

Node: Ein Knoten im Netzwerk, welcher eine vollständige Kopie der Blockchain enthält.

Nonce: Die Antwort auf ein mathematisches Rätsel zur Validierung eines Blocks. Bei einem EOA die Anzahl an verschickten Transaktionen. Bei einem Contract Account die Anzahl an Interaktionen mit anderen Contracts.[7]

Nutzwertanalyse: Ein Analyseverfahren zur Ermittlung der optimalen Option aus einer Menge mehrerer sich ähnelnden Optionen.

Opcode: Wird auch als »operation code« bezeichnet. Stellt die Nummer für einen Maschinenbefehl der EVM (bzw. eines bestimmten Prozessortyps) dar.[8]

Proof-of-Stake (PoS): Ein Konsensmechanismus zur Findung eines gemeinsamen Konsens in einem dezentralen Netzwerk. Beruht auf dem größten Besitz im Netzwerk.

Proof-of-Work (PoW): Ein Konsensmechanismus zur Findung eines gemeinsamen Konsens in einem dezentralen Netzwerk. Beruht auf der stärksten Rechenleistung im Netzwerk.

P2P-Netzwerk: Ein Netzwerk ohne zentrale Instanz. Jeder Teilnehmer kann von »Person zu Person« kommunizieren.

Receipt: Speichert die Ergebnisse einer Transaktion.[9]

Ropsten: Ein Ethereum Testnetzwerk für Entwickler zum Testen ihrer Anwendungen. Basiert wie das Ethereum Mainnet auf dem PoW. Weitere Ethereum Testnetzwerke sind Rinkeby, Kovan und Goerli.[10]

[7] vgl. Fertig und Schütz, 2019, S. 118
[8] vgl. Wikipedia, 2019
[9] vgl. ebd., S. 123
[10] vgl. Ziv, 2019

Satoshi Nakamoto: Ein Pseudonym für die Person oder Gruppe, welche die Entwicklung von Bitcoin vorangetrieben hat/haben.

Smart Contract: Ein intelligenter, sich selbst ausführender »Vertrag«.

Solidity: Eine »Curly-Bracket« Programmiersprache zur Entwicklung von Smart Contracts.

Token: Werden im Zusammenhang mit der Blockchain-Technologie dazu verwendet, um bspw. Wertgegenstände zu repräsentieren oder Währungen darzustellen.

Transaktion: Bei Ethereum handelt es sich dabei um signierte Nachrichten von EOAs. Werden zur Übertragung von Nutzdaten oder zum Austausch von Ether verwendet.

Vitalik Buterin: Der Begründer der Blockchain-Plattform Ethereum und Mitentwickler der Programmiersprache Vyper.

Vyper: Eine »Python-artige« Programmiersprache zur Entwicklung von Smart Contracts.

Wei: Die kleinste Ether-Einheit. 1 Wei = 10^{-18} Ether.

Yul/Yul+: Yul ist eine Zwischensprache für die EVM. Bei Yul+ handelt es sich um eine Erweiterung dieser Programmiersprache.

Öffentlicher Schlüssel (Public Key): Der aus dem privaten Schlüssel abgeleitete Schlüssel zum verschlüsseln und entschlüsseln von Nachrichten.

1 Einleitung

Motivation

Seit Einführung des Bitcoins im Jahr 2009 durch Satoshi Nakamoto hat die Blockchain- Technologie einen rasanten Aufstieg erlebt. Im nachfolgenden Jahrzehnt wurden zahlreiche Weiterentwicklungen vorgenommen. Heute werden viele dieser Weiterentwicklungen unter dem Sammelbegriff Blockchain 2.0 zusammengefasst. Einer der bekanntesten und zugleich auch ersten Vertreter der Blockchain 2.0 ist Ethereum. Mit Ethereum wurden völlig neue Möglichkeiten zur Nutzung der Blockchain geschaffen. Eine dieser Möglichkeiten ist die Erstellung sog. Smart Contracts. Diese sind mittlerweile ein vieldiskutiertes Thema und werden von vielen als vielversprechende Zukunftstechnologie angesehen, da sie vereinfacht ausgedrückt, Verträge ohne Mittelsmann ermöglichen. Zur Erstellung von Smart Contracts wurden/werden insbesondere von der Ethereum Foundation spezielle, dafür ausgelegte Programmiersprachen konzipiert. Allein für die Ethereum-Plattform gibt es eine Vielzahl verschiedener Smart Contract Sprachen, Tendenz steigend. Beispielsweise hat die Ethereum Foundation den Elite Universitäten Yale und Columbia Forschungsgelder bereitgestellt für die Entwicklung einer neuen Smart Contract Sprache.[1] Da die Blockchain und insbesondere Smart Contracts noch verhältnismäßig junge Technologien sind, gibt es viele Einsteiger und Umsteiger für deren Programmierung. Die Nachfrage von Firmen nach Smart Contract- bzw. Blockchain-Programmierern ist hoch, weshalb die Erlernung sinnvoll ist. Allerdings stellt sich für viele nach wie vor die Frage um was genau es sich bei der Blockchain überhaupt handelt. Ebenso was Smart Contracts sind und insbesondere welche der für die Ethereum-Plattform gegebenen Programmiersprachen am ehesten erlernt werden sollte, um sichere und effiziente Smart Contracts auf dieser zu programmieren.

Zielsetzung

Das Ziel dieser Arbeit ist es zu beantworten, welche Smart Contract Programmiersprache die optimale Wahl zur Erstellung von Smart Contracts ist. Die Kernfrage lautet dabei: „Welche Smart Contract Sprache ist für den Einstieg und für die allgemeine Entwicklung am besten geeignet?“ Dabei wird im Speziellen auf die Programmierung von Smart Contracts auf der Blockchain-Plattform Ethereum eingegangen, da diese im Sektor Smart Contract Plattformen die weiteste Verbreitung findet. Die Sprachen werden anhand eines Beispiel-ERC-Token analysiert und verglichen, außerdem wird ein allgemeiner Überblick über sie verschafft. Anhand einer Nutzwertanalyse werden die Kriterien Lesbarkeit, Funktionalitäten, Verfügbarkeit, Lern-

[1] vgl. Huillet, 2019

aufwand, Sicherheit und Effizienz gewichtet und im Bezug auf die jeweiligen Smart Contract Sprachen bewertet. Des Weiteren wird anhand einer Literaturrecherche ein allgemeiner Überblick über die Blockchain-Technologie und insbesondere über die Blockchain-Plattform Ethereum gegeben.

Gliederung

Zunächst werden in Kapitel 2 die Grundlagen zur Blockchain und insbesondere zu Ethereum vorgestellt. In Abschnitt 2.1 wird dabei einleitend auf die kryptografischen Grundlagen wie asymmetrische Verschlüsselung und kryptografische Hashfunktionen eingegangen, um ein besseres Verständnis für die nachfolgenden Abschnitte und Kapitel zu verschaffen. Abschnitt 2.2 dient dazu, um aufzuklären was genau eine Blockchain ist. Dabei wird im Speziellen auf die Blockchain-Datenstruktur, das Blockchain-System und auf den Mining-Prozess eingegangen. In Abschnitt 2.3 wird die Blockchain-Plattform Ethereum erläutert. Dazu werden Konzepte wie Ether, Gas, Accounts, Transaktionen, Receipts, Blöcke, die EVM und ERC-Tokenstandards in dieser Reihenfolge vorgestellt und ihre Zusammenhänge erläutert. Der letzte Abschnitt stellt Smart Contracts vor sowie alle Smart Contract Sprachen, die zur Programmierung auf der Ethereum-Plattform existieren.

In Kapitel 3 wird in Abschnitt 3.1 das für diese Arbeit zur Analyse verwendete Bewertungssystem vorgestellt. Des Weiteren werden in Abschnitt 3.2 die Bewertungskriterien erläutert und gewichtet. Den Abschluss dieses Kapitels bildet ein kurzes Beispiel zum Bewertungssystem.

In Kapitel 4 erfolgt eine Analyse und ein Vergleich der für diese Arbeit gewählten Smart Contract Programmiersprachen. In Abschnitt 4.1 wird zuerst die Vorgehensweise zur Erstellung und zum Deployment des Beispiel-ERC-Token erläutert. In Abschnitt 4.2 wird die Programmiersprache Solidity anhand der gewählten Kriterien analysiert und bewertet. Der gleiche Prozess erfolgt nachfolgend in Abschnitt 4.3 für die Programmiersprache Vyper und in Abschnitt 4.4 für die Programmiersprache Fe. In den Analyse-Abschnitten werden darüber hinaus noch die Transaktionsdaten, Receipts und Blockinformationen zu den jeweiligen Token-Deployments aufgelistet. Abschließend werden die gewonnenen Ergebnisse aufgezeigt und erläutert.

2 Grundlagen

2.1 Kryptografische Grundlagen

Die Kryptografie stellt einen unerlässlichen Bestandteil der Blockchain-Technologie dar. Ihre mathematischen Verfahren ermöglichen die effiziente Verschlüsselung von Daten (Public-Key Verschlüsselung), die Überprüfung der Authentizität von Daten (Hashfunktionen) sowie die Überprüfung um das Wissen eines »Geheimnisses«, ohne dieses preisgeben zu müssen (digitale Signaturen).[1] Wie diese Verfahren im Zusammenhang mit der Blockchain eingesetzt werden, wird nachfolgend beschrieben.

2.1.1 Asymmetrische Verschlüsselung

Bei der asymmetrischen Verschlüsselung (bzw. Public-Key Verschlüsselung) werden zwei Schlüssel zur geheimen Übermittlung von Daten verwendet. Möchte bspw., wie in Abbildung 2.1 zu sehen ist, eine Person A eine geheime Nachricht an eine Person B versenden, so generiert Person A zu Beginn einen privaten und öffentlichen Schlüssel. Der öffentliche Schlüssel wird dabei aus dem privaten Schlüssel abgeleitet. Dies erfolgt mithilfe von mathematischen Einwegfunktionen, d.h. der öffentliche Schlüssel kann mit vergleichsweise einfachen Mitteln aus dem privaten Schlüssel abgeleitet werden. Umgekehrt ist es allerdings nahezu unmöglich mit dem öffentlichen Schlüssel wieder den privaten Schlüssel zu erhalten. Der öffentliche Schlüssel wird nun an die Kommunikationspartner verteilt, also in diesem Fall an Person B, C und D. Alle Personen haben nun die Möglichkeit eine Nachricht mit diesem Schlüssel zu verschlüsseln und an Person A zu schicken. Im Fall von Abbildung 2.1 erfolgt dieser Prozess durch Person B. Die Nachricht kann nur mithilfe des privaten Schlüssels entschlüsselt werden. Mit dem öffentlichen Schlüssel ist, dank der mathematischen Gesetze, nur die Verschlüsselung möglich.[2] Dies ist auch zwingend notwendig. Angenommen weitere Personen (hier Person C und D) würden den öffentlichen Schlüssel besitzen, so soll es für sie schließlich nicht möglich sein die Nachricht zwischen A und B ebenfalls entschlüsseln zu können.

Bei Ethereum und vielen weiteren Blockchain-Plattformen bilden der private und öffentliche Schlüssel zusammen einen Account (um genau zu sein einen Externally Owned Account, mehr dazu in Abschnitt 2.3.3), welcher mithilfe einer aus dem öffentlichen Schlüssel abgeleiteten Adresse zugänglich gemacht wird. Dies geschieht mit kryptografischen Hashfunktionen um welche es in Abschnitt 2.2.2 geht. Vergleicht man so einen Ethereum-Account mit einem Bankkonto,

[1] vgl. Antonopoulos und Wood, 2019, S. 61
[2] vgl. Fertig und Schütz, 2019, S. 69f

so entspräche der private Schlüssel der PIN und der öffentliche Schlüssel der Kontonummer.[3] Sowohl Ethereum als auch Bitcoin verwenden für die asymmetrische Verschlüsselung die Elliptic Curve Cryptography (ECC). Dabei handelt es sich um eine Einwegfunktion, welche die Eigenschaften von elliptischen Kurven verwendet. Genaueres dazu ist hier zu finden: `https://cryptobook.nakov.com/asymmetric-key-ciphers/elliptic-curve-cryptography-ecc`.

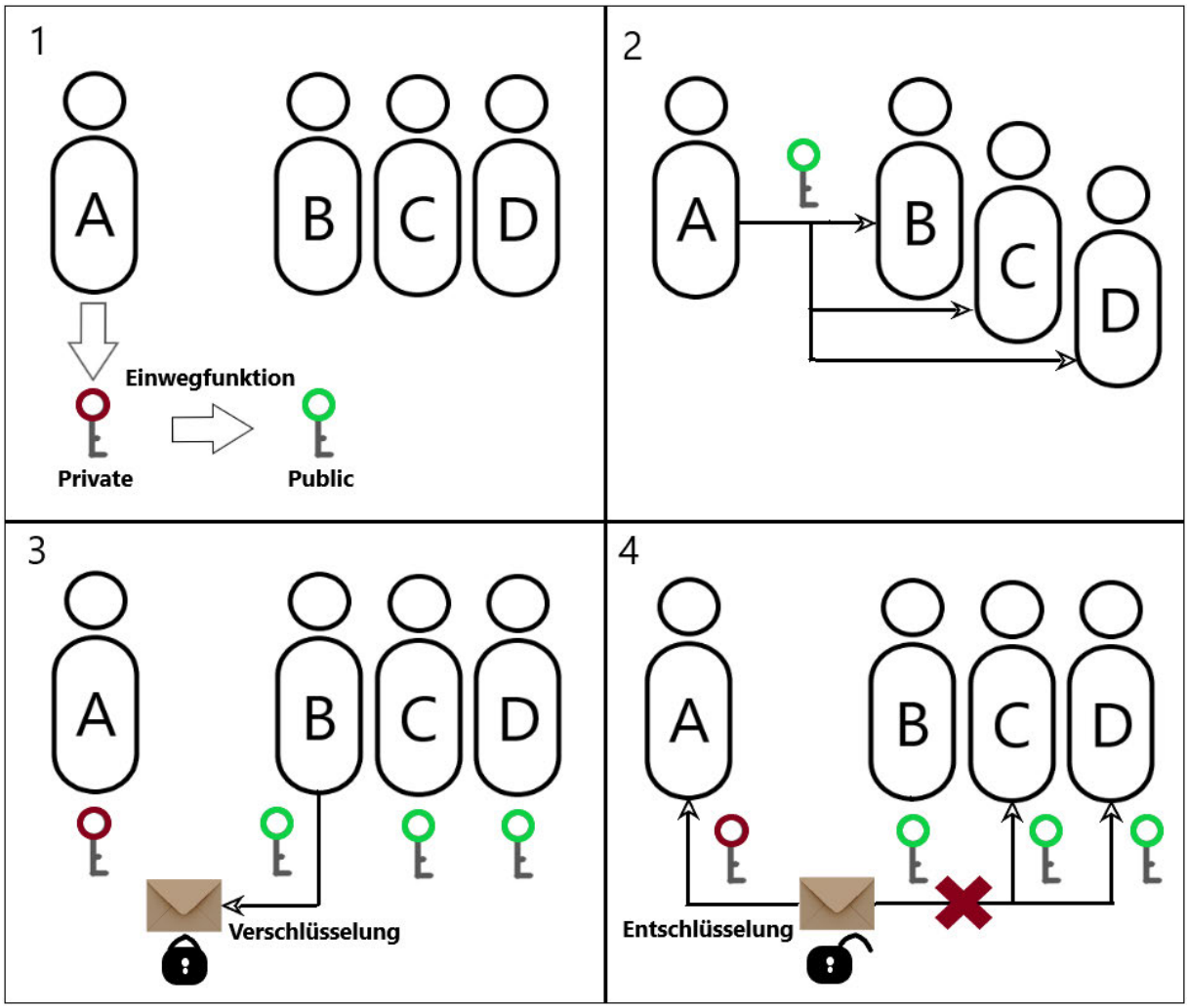

Abbildung 2.1: Funktionsweise der Verschlüsselung und Entschlüsselung eines asymmetrischen Kryptografiesystems.[3]

Ein weiterer Nutzen durch die asymmetrische Verschlüsselung sind sog. digitale Signaturen. Digitale Signaturen ermöglichen die eindeutige Identifizierung des Kommunikationspartners. Möchte im obigen Beispiel Person A eine Nachricht verschicken, dann kann sie diese Nachricht mit dem privaten Schlüssel verschlüsseln. Die Entschlüsselung dieser Nachricht ist nur mit einem aus dem privaten Schlüssel abgeleiteten öffentlichen Schlüssel möglich. D.h. Person B, C und D können sich sicher sein, dass eine Nachricht von Person A auch wirklich von dieser verschickt wurde, wenn sie diese mit ihrem öffentlichen Schlüssel entschlüsseln können.[4]

[3] vgl. Antonopoulos und Wood, 2019, S. 64
[4] vgl. Fertig und Schütz, 2019, S. 70

Bei Ethereum und für die Blockchain-Technologie allgemein werden digitale Signaturen verwendet, um sicher zu stellen, dass eine Transaktion nur von ihrem rechtmäßigen Eigentümer durchgeführt wird.[5] Das hierfür verwendete Verfahren wird im Zusammenhang mit ECC verwendet und nennt sich Elliptic Curve Digital Signature Algorithm (ECDSA). Weiterführende Informationen unter: `https://en.wikipedia.org/wiki/Elliptic_Curve_Digital_Signature_Algorithm`.

2.1.2 Kryptografische Hashfunktionen

Bei Hashfunktionen handelt es sich um Einwegfunktionen, d.h. wie in Abschnitt 2.2.1 bereits beschrieben, um Funktionen die einfach in eine Richtung zu berechnen sind, aber nahezu unmöglich in die andere Richtung. Hashfunktionen werden verwendet, um einen beliebig großen Eingabe-Datensatz in einen Hash-Datensatz fester Größe umzuwandeln. Bei kryptografischen Hashfunktionen handelt es sich um eine spezielle Ausprägung dieser. Sie müssen eine Reihe von Eigenschaften mitbringen, die für die Blockchain von besonderer Wichtigkeit sind. Diese sind:[6]

Determinismus ⇒ Aus einer gleichen Eingabe erfolgt immer der gleiche Hash.

Verifizierbarkeit ⇒ Die Berechnung des Hashs einer Eingabe erfolgt effizient.

Nicht-Korrelation ⇒ Kleinste Änderungen der Eingabe ändern diese so weitgehend, dass sie in keiner Weise mit der unveränderten Eingabe korreliert werden kann.

Unumkehrbarkeit ⇒ Die Berechnung aus dem Hash zurück zur Eingabe ist unmöglich.

Kollisionsschutz ⇒ Unterschiedliche Nachrichten müssen immer zu einem unterschiedlichen Hash führen.

Abbildung 2.2 stellt den Sachverhalt grafisch dar. Als Hashfunktion wurde hier Keccak-256[7] verwendet. Angenommen es erfolgt zweimal die Eingabe »Bachelorarbeit«, dann muss für diese auch der identische Hash (in der Abbildung grün dargestellt) errechnet werden. Falls eine kleine Änderung vorgenommen wird, also in diesem Fall das Ausrufezeichen in »Bachelorarbeit!«, dann muss hierfür ein Hash errechnet werden, welcher der Ursprungsform nicht annähernd ähnelt (in der Abbildung rot dargestellt). Hat man nun eine weitere Eingabe »Hausarbeit!«, dann darf für diese nicht der gleiche Hash wie für eine bereits vorhandene Eingabe errechnet werden.

Die Blockchain hat mehrere Anwendungsfelder für kryptografische Hashfunktionen. Eines davon ist die bereits erwähnte Ableitung der Adresse aus einem öffentlichen Schlüssel oder im Falle von Ethereum auch aus einem Smart Contract. Ein weiteres ist die Verkettung der Blöcke zu einer Blockchain. Dieser Prozess wird noch einmal in Abschnitt 2.2 erläutert. Bitcoin verwendet als Hashfunktion SHA-256. Als Input bekommt diese Funktion einen 512 Bit großen Datensatz und gibt als Output einen 256 Bit großen Hash zurück. Genaueres zu SHA-256 (auch als

[5] vgl. Kerem, 2021
[6] vgl. Antonopoulos und Wood, 2019, S.74
[7] `https://emn178.github.io/online-tools/keccak_256.html`

SHA-2 bekannt) ist hier zu finden: `https://de.wikipedia.org/wiki/SHA-2`. Ethereum verwendet die Hashfunktion Keccak-256. Sie hat die gleichen Input und Output Werte wie SHA-256. Weiterführende Informationen zu Keccak-256 (auch als SHA-3 bekannt) sind hier zu finden: `https://de.wikipedia.org/wiki/SHA-3`

Abbildung 2.2: Veranschaulichung der Eigenschaften einer kryptografischen Hashfunktion.

Der vollständige Ablauf zur Erzeugung des privaten und öffentlichen Schlüssels sowie der Adresse könnte bspw. wie in Abbildung 2.3 dargestellt aussehen. In Schritt 1 erzeugt ein Zufallszahlengenerator eine Zahl zwischen 1 und $2^{256}-1$. Ethereum verwendet dafür den Zufallszahlengenerator des jeweiligen Betriebssystem auf welchem die Ethereum-Software läuft. Die Zufallszahl wird in Schritt 2 einer Hashfunktion übergeben. In diesem Fall wäre das die Funktion Keccak-256. Das Resultat ist der private Schlüssel. In Schritt 3 wird mittels ECC der öffentliche aus dem privaten Schlüssel abgeleitet. In Schritt 4 wird nun der öffentliche Schlüssel der Hashfunktion Keccak-256 übergeben.[8] Die letzten 20 Bytes, des daraus folgenden Resultats, stellen eine Adresse bei Ethereum dar. Einer Adresse wird in den meisten Fällen ein 0x vorangestellt.[9] Eine gültige Ethereum-Adresse sieht also wie folgt aus: **0xDd7D08B014Cd8E58c86cE71c5094C7aa20BA957c**

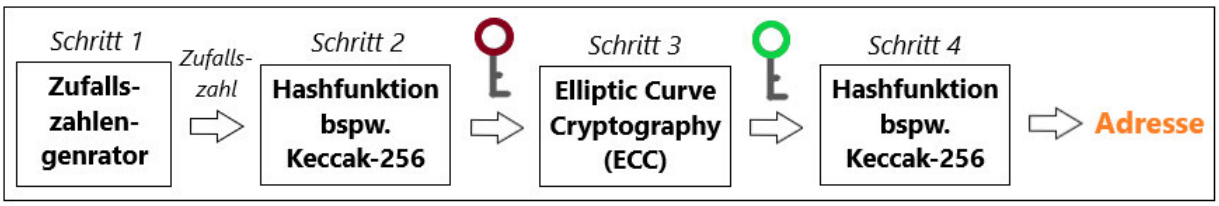

Abbildung 2.3: Vorgehensweise zur Erzeugung des privaten und öffentlichen Schlüssels sowie der Adresse bei Ethereum.

[8] vgl. Antonopoulos und Wood, 2019, S. 65f
[9] vgl. ebd., S. 76

2.2 Was ist eine Blockchain?

Die Ursprünge der Blockchain-Technologie reichen zurück bis in das Jahr 1991, in welchem die ersten Grundlagen zur Schaffung einer Blockchain spezifiziert wurden. Ihren heutigen Bekanntheitsgrad hat die Blockchain allerdings vor allem durch die Einführung des Bitcoins erlangt. Der Bitcoin nutzt als erste Anwendung überhaupt das Konzept der Blockchain als verteiltes Datenbankmanagementsystem.[10] Die Idee dahinter stammt von Satoshi Nakamoto. Dieser wollte mit Bitcoin nach der Weltwirtschaftskrise 2008/09 ein von finanziellen Instituten unabhängiges Zahlungssystem erschaffen. Innerhalb vergleichsweise kurzer Zeit hat der Bitcoin mit explosionsartigen Kursanstiegen auf sich aufmerksam gemacht. Am Ende des Jahres 2009 besaß ein Bitcoin gerade einmal einen Wert von 0,08 US-Dollar.[11] Heute (Stand: Dezember 2021) notiert er einen Wert von ca. 56000 US-Dollar. Das entspricht einem Kursanstieg von ca. 700.000% in gerade einmal 12 Jahren. Hätte man also Ende 2009 für einen Dollar Bitcoin gekauft (ungefähr 12,5 BTC), so hätten diese heute einen Wert von ca. 700.000 US-Dollar. Aus diesem Grund hat der Bitcoin und somit auch die Blockchain-Technologie schnell nicht nur das Interesse von Spekulanten geweckt, sondern auch das von Firmen, Banken und sogar ganzen Staaten. Hinter dieser Technologie steckt allerdings deutlich mehr als nur ein Zahlungssystem. Konkret formuliert handelt es sich bei der Blockchain um eine dezentrale/verteilte, öffentliche Datenbank. Sie besteht aus einer Datenstruktur (der Blockchain) und aus einem System zur Verwaltung dieser Datenstruktur (das Blockchain-System).

Die Datenstruktur besteht aus chronologisch verketteten Blöcken (siehe Abbildung 2.1). Bei den Blöcken handelt es sich um einen strukturierten Datensatz. Ein Block besteht grob formuliert aus einem Blockheader und einem Blockkörper (Body). Der Blockheader enthält dabei wichtige Informationen wie die Versionsnummer der Software, die Nonce, das Ziel zur Bestimmung der aktuellen Schwierigkeit (engl. Difficulty), den Zeitstempel, den Hash des vorangegangenen Blockheaders, die Zusammenfassung aller Transaktionen die sich im Blockkörper befinden und (bei Ethereum) noch einige weitere Komponenten.[12] Innerhalb des Körpers werden die von den Teilnehmern erstellten und signierten Transaktionen gespeichert. Die Signatur erfolgt dabei durch asymmetrische Kryptografiesysteme, welche in Abschnitt 2.1.1 beschrieben wurden. Ohne gültige Signatur kann eine Transaktion nicht in der Blockchain aufgenommen werden. Transaktionen können Daten wie Geldeinheiten, Bytecode (Smart Contracts) oder Sensordaten enthalten.[13] Sie bestehen, wie der Block, aus verschiedenen Komponenten, welche sich von Blockchain zu Blockchain unterscheiden können. Da in dieser Arbeit die Plattform Ethereum im Vordergrund steht, werden die genaueren Details von Transaktionen im Zusammenhang mit dieser später erläutert (siehe Abschnitt 2.3.4). Die Blöcke werden durch kryptografische Hashfunktionen, welche in Abschnitt 2.1.2 beschrieben wurden, miteinander verknüpft. Dabei wird der Hash des Blockheaders eines vorangegangenen Blocks beim nachfolgenden Block in den Blockheader eingetragen. Die

[10] vgl. Wikipedia, 2021
[11] vgl. Schinko, 2021
[12] vgl. Roos, 2018
[13] vgl. Westfälische Hochschule, 2021

Verkettung sorgt dafür, dass die Blöcke aufeinander aufbauen und Veränderungen im Konstrukt sofort auffallen würden. D.h. nachträgliche Veränderungen an einem früheren Block würden dazu führen, dass alle nachfolgenden Blöcke ungültig werden.[14] Angenommen jemand würde bei Block 33 in Abbildung 2.4 nachträglich eine Transaktion verändern. Dadurch das sich auch eine Zusammenfassung der Transaktionen im Header befindet, würde das Hashverfahren bei dessen Header nun zu einer anderen Ausgabe führen. Da dieser Hash wiederum in den nachfolgenden Blockheader eingetragen wird, würde sich auch dessen Hash (des Headers) ändern usw. Den Startpunkt einer Blockchain stellt der sog. Genesis Block dar. Da dieser keinen vorherigen Block besitzt mit dem er verkettet ist, wird er fest in die Software der Anwendungen eingebunden, welche seine Blockchain nutzen.[15]

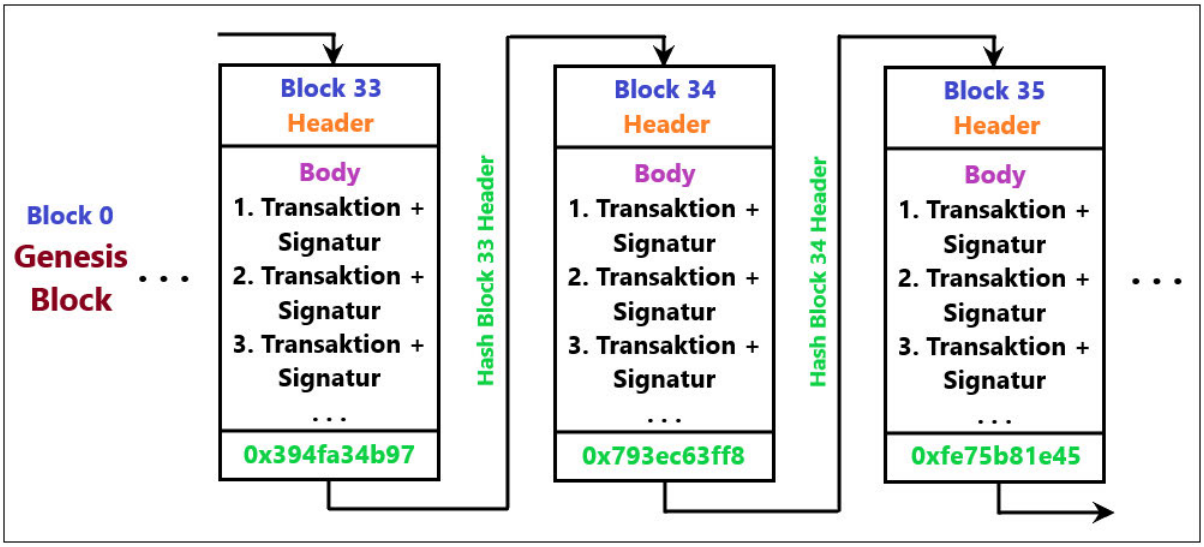

Abbildung 2.4: Grobe Zusammensetzung der Datenstruktur einer Blockchain.

Das Blockchain-System ist ein (im besten Falle) weltweites Netz aus vielen verteilten Knoten. Bei diesem Netz handelt es sich um ein P2P-Netzwerk (Peer to Peer) in dem jeder Knoten gleichberechtigt ist und direkt mit einem anderen Knoten interagieren kann (siehe Abbildung 2.5). D.h. P2P-Netze benötigen keine zentrale Instanz.[16] Ein Knoten in diesem Netzwerk kann theoretisch alles mit genug Rechenleistung und Speicherplatz sein. Jeder Teilhaber am Netzwerk besitzt eine Kopie der vollständigen Blockchain. Mit anderen Worten, sämtliche Daten der Blockchain werden auf jeden im Netzwerk vorhandenen Knoten redundant gespeichert. Diese gewollte Redundanz sorgt für ein wichtiges Sicherheitsmerkmal der Blockchain, da so der Manipulation von Daten durch mächtige einzelne Instanzen oder dem Verlust durch technische Probleme vorgebeugt wird. Angenommen in Tokio werden durch ein Erdbeben sämtliche Knoten zerstört. So würde das für die Blockchain kein großes Problem darstellen, da sie ja auf weiteren Knoten im Rest der Welt verteilt gespeichert ist. Für die Kommunikation zwischen den Knoten wird bei Bitcoin das Transmission Controll Protocol (TCP) und bei Ethereum eine eigens entwickelte

[14] vgl. Fertig und Schütz, 2019, S. 27
[15] vgl. Schiller, 2018
[16] vgl. Bosch, 2021

Protokollfamilie namens RLPx, welche auf TCP basiert, verwendet.[17]

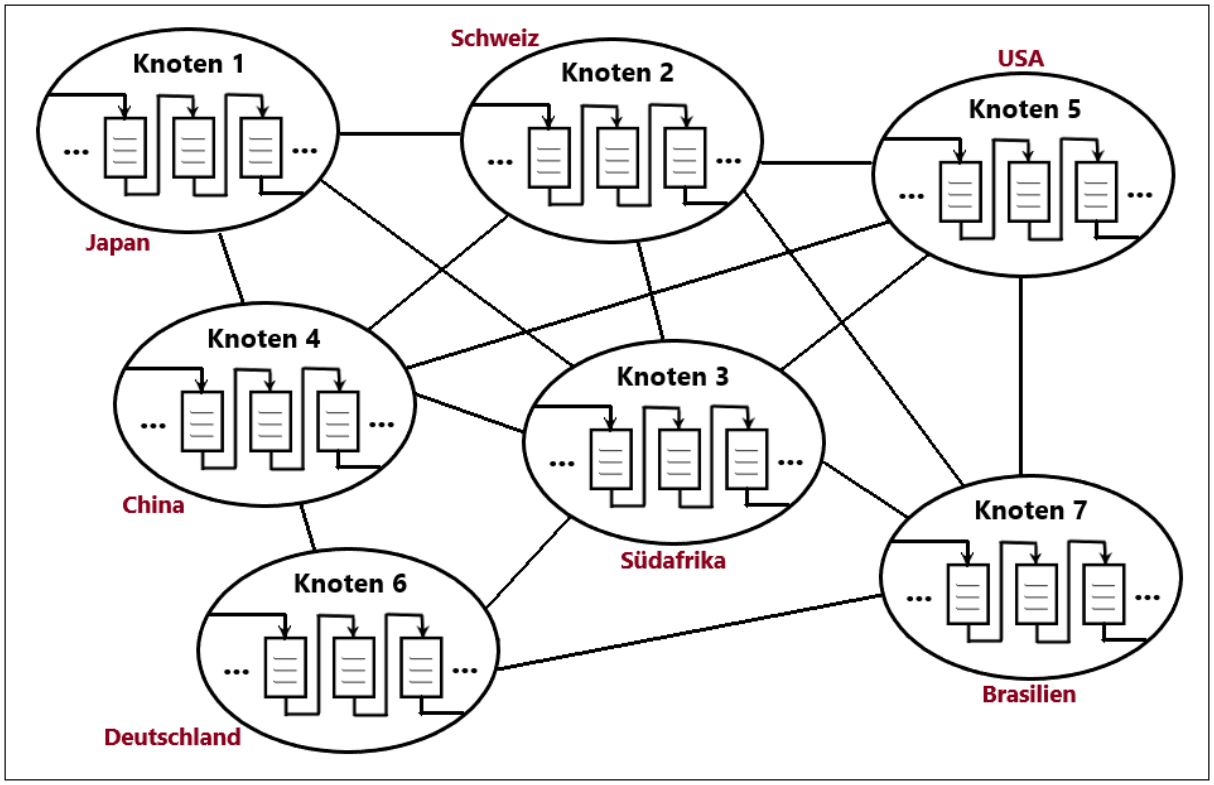

Abbildung 2.5: Veranschaulichung der P2P-Vernetzung sowie der möglichen Verteilung der Knoten eines Blockchain-Systems.

Um neue Blöcke für die Blockchain zu generieren und zu validieren sowie um sich auf eine identische Version der Blockchain im dezentralen Netzwerk zu einigen, wird ein sog. Konsensmechanismus verwendet. Bei Bitcoin und Ethereum wird dafür der Proof-of-Work (PoW) verwendet. Ethereum wird im Laufe seiner Entwicklung noch auf den Proof-of-Stake (PoS) wechseln. Da allerdings beide derzeit PoW verwenden (Stand: Dezember 2021) wird nur dieser weiter erläutert. Im Zusammenhang mit PoW wird oft von Mining und Minern gesprochen. Durch das Mining werden Transaktionen, welche sich noch nicht in Blöcken befinden, in eben diese gepackt und der Blockchain hinzugefügt. Die unverpackten/unverifizierten Transaktionen befinden sich zuvor im sog. Mempool. Dabei handelt es sich quasi um einen Wartebereich für die Transaktionen, bis ein Miner sie für einen Block auswählt.[18] Damit der Miner den Block der Blockchain hinzufügen darf, muss der Block nach bestimmten Parametern gültig sein. Um einen gültigen Block zu finden muss, vereinfacht ausgedrückt, eine Rechenaufgabe gelöst werden. Dabei gilt es einen Hash vom Blockheader des noch ungültigen Blocks zu erstellen. Das vorher bereits erwähnte Ziel zur Bestimmung der aktuellen Schwierigkeit (engl. Difficulty) erschwert diese Aufgabe künstlich. Das Ziel stellt nämlich eine Einschränkung dafür dar, wie dieser Hash auszusehen hat. Der zu berechnende Hash muss kleiner oder gleich dem Wert des Ziels sein. Der Wert des Ziels wird

[17] vgl. Fertig und Schütz, 2019, S. 93ff
[18] vgl. Blocktrainer, 2021

dabei durch einen 256-Bit-Integer-Wert repräsentiert.[19] Die Difficulty stellt das Maß dafür da, wie schwierig es ist den passenden Hash zu diesem Ziel zu finden. Die Schwellengrenze der Difficulty muss dabei immer wieder angepasst werden, da es bei bspw. Bitcoin nur alle zehn Minuten möglich sein soll einen Block der Blockchain hinzuzufügen. Je mehr Miner im Netzwerk (mit einer umso höheren Rechenleistung) versuchen die Rechenaufgabe zu lösen, umso niedriger muss die Grenze für die Difficulty sein und umgekehrt.[20]

Eine weitere wichtige, bereits erwähnte Komponente beim Mining ist die Nonce. Diese stellt quasi die Lösung der kryptografischen Rechenaufgabe dar. Sie wird vom Miner gewählt und in den Blockheader eingetragen, der gehasht werden soll. Ist der Wert des gehashten Blockheaders größer als er Wert des Ziels, dann muss eine neue Nonce vom Miner gewählt werden. Dabei hat er 2^{32} Möglichkeiten eine passende Nonce zu finden.[21] Für das Netzwerk ist es nun nicht mehr schwer festzustellen welche die richtige Blockchain ist. Meistens wird sich auf diejenige Version geeinigt, in welche die größte Rechenleistung geflossen ist. Um die Teilnehmer zu motivieren ihre Rechenleistung für das Minen bereitzustellen, durch welches ja der gemeinsame Konsens geschaffen wird, werden diese beim erfolgreichen Erstellen und Hinzufügen eines Blocks vom Netzwerk mit der Ausschüttung von BTCs bei Bitcoin oder ETHs bei Ethereum entschädigt. Die Entschädigung bzw. Belohnung wird auch als Block Reward bezeichnet und besteht aus dem Blockanteil und den Transaktionsgebühren. Der Blockanteil besteht dabei aus neu generierten Coins und macht den größeren Anteil aus. Zusätzlich erhält ein Miner noch die Gebühren, welche von den Transaktionen die dem Block hinzugefügt wurden, gezahlt werden. Wenn alle Coins gemined wurden (bei Bitcoin 21 Mio.), dann erhalten die erfolgreichen Miner nur noch die Transaktionsgebühren.[22]

2.3 Die Blockchain-Plattform Ethereum

Bei Ethereum handelt es sich nicht um eine reine Kryptowährung wie bspw. Bitcoin. Ethereum ist eine auf der Blockchain-Technologie aufgebaute Open-Source-basierte Plattform für die Entwicklung von Smart Contracts und dezentralen Applikationen (dApps). Auf einer etwas tieferen Ebene formuliert ist Ethereum ein transaktionsbasierter Zustandsautomat. D.h. Ethereum befindet sich in einem einzigen globalen Zustand, welcher durch Transaktionen in einen neuen aktualisierten Zustand übergeht. So gesehen kann Ethereum als ein Weltcomputer betrachtet werden, da jede noch so kleine Aktion eines externen Akteurs dafür sorgt, dass sich für jeden Teilnehmer im Netzwerk der gesamte Zustand von Ethereum aktualisiert.[23] Die Idee hinter Ethereum stammt von Vitalik Buterin (und später Gavin Wood), welcher erst 19 Jahre alt war, als er das erste Whitepaper dazu verfasste. Ethereum wird auch häufig als Blockchain 2.0 bezeichnet, da es sich um eine Erweiterung der ursprünglichen Blockchain von Bitcoin handelt.

[19] vgl. Fertig und Schütz, 2019, S. 97f
[20] vgl. Bitpanda, 2021
[21] vgl. Fertig und Schütz, 2019, S. 97f
[22] vgl. BTC Acadamy, 2022
[23] vgl. Fertig und Schütz, 2019, S. 113

2.3.1 Ether

Die innerhalb des Ethereum-Netzwerks verwendete Kryptowährung heißt Ether (kurz ETH). Ether dient als Zahlungsmittel zur Ausführung von Smart Contracts und DApps im Netzwerk. Des Weiteren wird Ether, wie auch Bitcoin, als Belohnung für einen erfolgreichen Miner verwendet. Diese erhalten für das Schürfen und Hinzufügen eines Blocks zur Ethereum-Blockchain 3 ETH als Belohnung (Stand: Dezember 2021, vermutlich wird diese mit dem Übergang zu Ethereum 2.0 irgendwann auf 0,6 ETH sinken[24]). Ether lässt sich auch in kleineren Einheiten darstellen. Die kleinste Einheiten-Stückelung für Ether ist Wei. 1 Wei ist entspricht 10^{-18} ETH. Wei lässt sich nicht weiter stückeln und auch nicht als Kommazahl darstellen. Somit ist die kleinstmögliche Summe bei Ethereum 1 Wei. Eine weitere häufig vorkommende Stückelung ist GWei (GigaWei). In GWei werden häufig die für das Gas anfallenden Kosten angegeben. 1 GWei ist umgerechnet 10^{-9} ETH.[25]. Die nachfolgende Tabelle zeigt weitere Ether-Stückelungen auf.[26] Ein simpler Einheiten-Rechner dazu ist hier zu finden: `https://converter.murkin.me/`.

Einheit	**Wert (in Wei)**	**Exponent**	**Name**
Wei	1	1	Wei
Kwei	1.000	10^3	Babbage
Mwei	1.000.000	10^6	Lovelace
Gwei	1.000.000.000	10^9	Shannon
Mikroether	1.000.000.000.000	10^{12}	Szabo
Milliether	1.000.000.000.000.000	10^{15}	Finney
Ether	1.000.000.000.000.000.000	10^{18}	Ether
Kether	1.000.000.000.000.000.000.000	10^{21}	Grand
Mether	1.000.000.000.000.000.000.000.000	10^{24}	
Gether	1.000.000.000.000.000.000.000.000.000	10^{27}	

Tabelle 2.1: Ether-Stückelungen[26]

2.3.2 Gas

Eine weitere wichtige Einheit, die bei Ethereum vorkommt, ist das sog. Gas. Es gibt zwei Gründe für die Verwendung von Gas. Erstens soll es dazu dienen einen Puffer zwischen dem volatilen Ether-Preis und der Belohnung für die Miner zu schaffen. Der zweite Grund ist der Schutz vor Denial-Of-Service (DoS) Attacken. Letzterer hängt mit der zur Ausführung von Smart Contracts verwendeten virtuellen Maschine, die Ethereum Virtual Machine (EVM, siehe Abschnitt 2.3.7) zusammen. Da die EVM Turing-Vollständig ist, d.h. mit ihr theoretisch jedes Programm berechnet werden kann, welches auch mit einer Turingmaschine berechnet werden kann, wird eine »Lösung« für das Halteproblem auf dem Weltcomputer benötigt. Vereinfacht ausgedrückt wäre es auf Grund des Halteproblems möglich Ethereum durch eine Endlosschleife unbrauchbar

[24] vgl. BlockchainCenter.net, 2018
[25] vgl. Antonopoulos und Wood, 2019, S. 15f
[26] vgl. Frankenfield, 2021

zu machen. Dessen neuer aktualisierter Zustand würde dadurch nämlich nie erreicht werden. Dabei ist es völlig egal, ob dies bewusst oder unbewusst verursacht wird. Beide Fälle gilt es zu verhindern. Hier kommt das Gas ins Spiel.[27]

Das Gas legt fest wie viel Rechenleistung (und auch Speicherressourcen) für ein Programm bzw. eine Aktion maximal von der EVM aufgebracht werden soll, bevor die Ausführung von dieser abgebrochen wird. Dies löst das Problem mit der Endlosschleife, da ein Smart Contract mit so einem Konstrukt beim Erreichen des vorher vom Urheber festgelegten Gas-Limits durch die EVM abgebrochen wird und der vorherige Zustand des Weltcomputers beibehalten wird. Der Initiator einer Transaktion muss eine kleine Menge Ether für das Gas bezahlen, welches zur Ausführung seiner Aktionen der Transaktion benötigt wird. Dabei benötigt jede von einem Smart Contract ausgeführte Operation Gas. Beispielsweise benötigt ein bitweises UND 3 Gas, eine ganzzahlige Division 5 Gas, eine Addition wieder 3 Gas, das Erstellen eines Contract Accounts (ein Smart Contract) 32000 Gas und das Senden einer Transaktion 21000 Gas.[28] Mehr dazu in Abschnitt 2.3.7.

Der nachfolgende Smart Contract Donation.sol (Listing 2.1), welcher in Solidity geschrieben wurde, hat beim Deployen 221952 Gas benötigt. Der Urheber dieses Contracts muss also:

221952 Gas * 2338872469 Wei ≈ 0.00051912 ETH

dafür bezahlen. Beim ersten Wert handelt es sich um das sog. GasUsed (siehe Abschnitt 2.3.5) und beim zweiten Wert um den sog. GasPrice (siehe Abschnitt 2.3.4). Der Contract dient im Übrigen lediglich dazu, um eine beliebige Summe an Ether an die beim Deployen festgelegte Adresse *_to* spenden zu können. Um Platz zu sparen wurde hier das Versionspragma und der SPDX-License-Identifier weggelassen. Beides wird im Verlauf der Arbeit (Abschnitt 4.2 Solidity) noch erläutert. Dennoch sei erwähnt, dass für diesen Contract Version 0.7.0 bis 0.8.9 zulässig sind und das es sich bei der Lizenz um die MIT-Lizenz handelt. Das Deployment fand auf dem Ethereum-Testnetzwerk Ropsten statt, d.h. bei den Kosten handelt es sich nicht um echte Ether.

```
1  contract Donation {
2
3      address payable private _to;
4
5      constructor(address payable to_) {
6          _to = to_;
7      }
8
9      function donate() public payable {
10         require(_to != address(0), "Transfer to the zero address!");
11         _to.transfer(msg.value);
12     }
13
```

[27] vgl. Antonopoulos und Wood, 2019, S. 312f
[28] vgl. ebd., S. 313

```
14      function getAddress() public view returns(address) {
15          return _to;
16      }
17  }
```

Listing 2.1: Solidity-Quellcode Donation.sol

2.3.3 Accounts

In Ethereum gibt es zwei Arten von Accounts: Die Externally Owned Accounts (EOA) und die Contract Accounts. Erstere werden von externen Nutzern (menschliche Nutzer) verwendet, um von außen mit der Ethereum-Plattform zu interagieren. Ein EOA entspricht dem Account, welcher in Abschnitt 2.1.1 beschrieben wird. D.h. der private und öffentliche Schlüssel bilden solch einen EOA und die Adresse macht diesen zugänglich. Contract Accounts werden im Netzwerk verwendet, um Smart Contracts darzustellen. Sie besitzen keinen privaten Schlüssel, da sie vom Programmcode des Smart Contracts gesteuert werden und nicht wie EOAs von einem menschlichen Nutzer. Eine Adresse besitzen sie allerdings, um Ether empfangen und senden zu können.[29]

Beide Account-Typen bestehen aus folgenden Komponenten: *Account Balance*, *Nonce*, *Storage-Hash* und *Code-Hash*. Die *Account Balance* gibt den Kontostand eines Accounts in Wei an. Die *Nonce* ist in diesem Fall nicht die Lösung eines kniffligen Rätsels, sondern die Anzahl an von diesem Account getätigten Transaktionen. Bei Contract Accounts stellt die *Nonce* die Anzahl an Interaktionen mit anderen Contract Accounts dar. Der *Storage-Hash* ist nur bei Contract Accounts von Relevanz, da dieser die angelegten Variablen eines Smart Contracts enthält. Genau genommen handelt es sich dabei nur um den Hash der Wurzel des Storage Tries. Der Storage Trie ist eine Baumstruktur zur Abspeicherung der Variablendaten. EOAs besitzen diese Komponente zwar auch, allerdings bleibt diese leer, da EOAs keinen Speicher benötigen. Auch der *Code-Hash* findet nur bei Contract Accounts Anwendung. Dabei handelt es sich um einen Hash des Bytecodes für die Ethereum Virtual Machine (EVM). Auch diese Komponente ist unnötigerweise bei EOAs vorhanden. In ihr befindet sich bei diesen nämlich nur der Hash eines leeren Strings.[30]

Zwischen EOAs und Contract Accounts gibt es drei weitere gravierende Unterschiede. Erstens kostet die Erstellung eines EOAs im Vergleich zum Contract Account nichts. Der Grund für die Kosten bei einem Contract Account ist die Nutzung des Netzwerk-Speichers von Ethereum. Zweitens kann ein EOA Transaktionen nicht nur erhalten sondern auch initiieren. Der Contract Account kann Transaktionen nur als Antwort auf eine erhaltene Transaktion senden. Drittens können zwischen EOAs nur Ether bzw. Tokens ausgetauscht werden. Eine Transaktion zwischen einem EOA (natürlich vom EOA initiiert) und einem Contract Account kann zur Ausführung von Code führen, welcher wiederum verschiedene Aktionen ausführen kann. Solche Aktionen

[29] vgl. ebd., S. 28
[30] vgl. Fertig und Schütz, 2019, S. 117f

können bspw. das einfache Übertragen von Tokens sein oder aber auch die Erstellung eines völlig neuen Smart Contracts.[31]

Abbildung 2.6 soll das Konzept von Accounts in Ethereum noch einmal verdeutlichen. Der EOA besitzt hier ein Guthaben von 100000 Wei und hat bereits 47 Transaktionen durchgeführt. Der *Storage-Hash* ist leer und der *Code-Hash* ist der Hash eines leeren Strings (angedeutet durch Hash("")). Der EOA wird vom Besitzer des jeweiligen privaten Schlüssels kontrolliert. Der Contract Account besitzt hier ein Guthaben von 50000 Wei und hat bereits mit 66 anderen Contract Accounts interagiert. Der *Storage-Hash* ist der Hash der Storage Root des zugehörigen Storage Tries. Der *Code-Hash* ist angedeutet mit Hash(Code). Der Contract Account wird vom vorgegebenen Programmcode kontrolliert. Alle im Ethereum-Netzwerk existierenden Accounts bilden zusammen den globalen Zustand (World State oder Global State) von Ethereum. Der Global State wird in Form eines Tries (dem sog. State Trie) auf jedem Knoten im Netzwerk gespeichert. Trie ist im Übrigen kein Rechtschreibfehler, sondern die Kurzform für den englischen Fachbegriff Information Retrieval (engl. für Informationsrückgewinnung). Dabei handelt es sich um eine von Ethereum zur Speicherung verschiedenster Daten verwendete Baumstruktur (die Ähnlichkeit zum Wort Tree ergibt also Sinn), nämlich den sog. »Merkle Patricia Trie« bzw. »Modified Merkle Patricia Trie«.[32] Tatsächlich werden auch beide Begriffe im Zusammenhang mit dieser Baumstruktur verwendet. Weiterführende Information unter: `https://eth.wiki/fundamentals/patricia-tree`

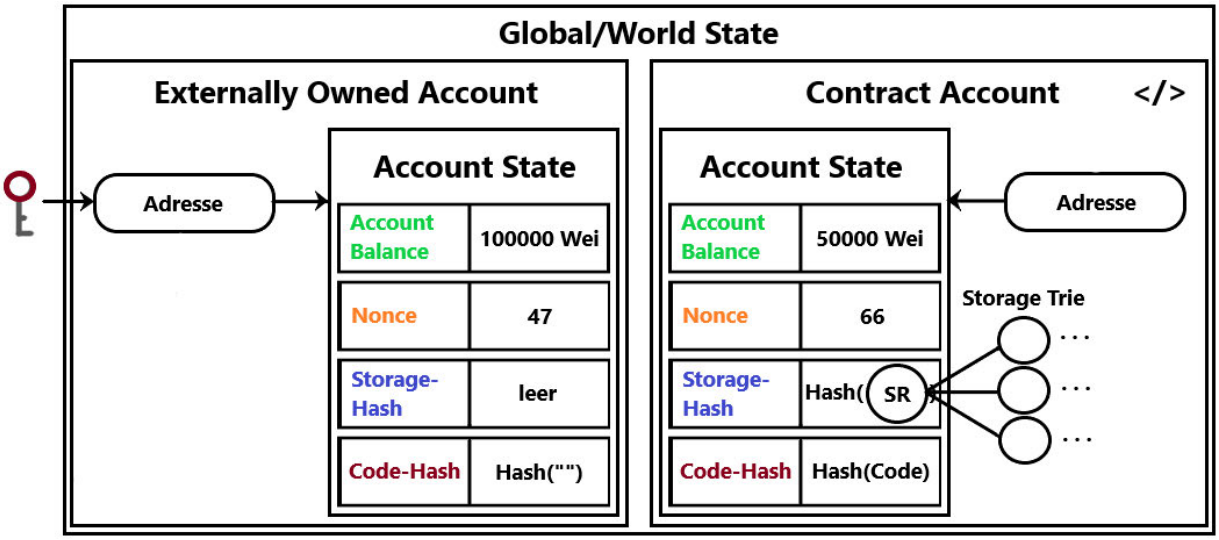

Abbildung 2.6: Darstellung des Global States und der diesen bildenden Account-Typen.

2.3.4 Transaktionen

Transaktionen haben bei Ethereum mehrere Verwendungszwecke. Bei Bitcoin dienen sie lediglich dazu, die namensgebende Währung zwischen zwei Akteuren auszutauschen. Dies ist auch ein

[31] vgl. Ethereum, 2021
[32] vgl. Fertig und Schütz, 2019, S. 114

Anwendungsfall bei Ethereum und entspräche dem Versenden von Ether zwischen zwei EOAs. Ein weiterer Anwendungsfall ist eine Transaktion zwischen einem EOA und einem Contract Account, welche dazu führt, dass der Programmcode des Contract Accounts aufgerufen wird. Diese beiden Anwendungsfälle werden auch als Message Call eines EOAs bezeichnet. Der letzte Anwendungsfall ist die Erschaffung eines Contract Accounts (also das Deployen eines Smart Contracts) mittels einer Transaktion.[33] Dabei handelt es sich um die sog. Kontrakterzeugungstransaktion. Die Zieladresse einer solchen Transaktion ist 0x0 (die Nulladresse). Der einzige Nutzen dieser Adresse ist, wie sich unschwer vermuten lässt, die Erzeugung eines Contract Accounts.[34]

Es gibt zwei Formate für den Aufbau von Transaktionen. Das erste Format ist die sog. LegacyTransaction, welches das Ursprungsformat für Transaktionen bei Ethereum ist.[35] Die LegacyTransaction besteht aus mehreren Komponenten. Diese sind: *Nonce*, *GasPrice*, *GasLimit*, *To*, *Value*, *Data/Init* und *v,r,s*. Die *Nonce* stellt die Nonce aus dem Account des Senders dar. Der *GasPrice* gibt an, wie viel der Urheber einer Transaktion bereit ist, für eine Gaseinheit zu bezahlen (in Wei). Je höher der *GasPrice* angegeben wird, umso wahrscheinlicher ist es, dass die Transaktion schneller als andere Transaktionen abgewickelt wird. Der Grund dafür ist, dass beim Ethereum-PoW die Gebühren, die für das Gas anfallen, dem jeweiligen Miner der Transaktion zugeschrieben werden. Diese suchen sich demnach zuerst diejenigen Transaktionen mit hohen Gas-Preisen aus, um sie einem Block hinzuzufügen.[36] Das *GasLimit* legt die maximale Anzahl an Gas fest, welches der Urheber bereit ist für die Durchführung der Transaktion aufzuwenden. Das *To*-Feld gibt die Empfänger-Adresse der Transaktion an. *Value* gibt die Anzahl an Ether (in Wei) an, die an den Empfänger gesendet werden soll. *Data/Init* wiederum enthalten Nutzdaten, die an den Empfänger gesendet werden sollen. *Init* wird nur beim Initialisieren eines Contract Accounts (also bei Deployen eines Smart Contracts) angegeben. Die Komponente enthält dabei den EVM-Bytecode, welcher an die Nulladresse gesendet wird. *Data* wird wiederum nur bei einem Message Calls angegeben. Die Komponente enthält bspw. die erforderlichen Argumente zum Aufruf einer Funktion eines Smart Contracts, welche an die Adresse des Contract Accounts gesendet werden. Schlussfolgernd kann eine Transaktion nie beide Felder gleichzeitig enthalten. Die Variablen *v,r,s* sind ein Teil des ECDSA-Verfahren, um die Transaktion zu signieren.[37]

Das zweite Format ist die sog. TypedTransaction, welches durch EIP-2718 definiert wurde. EIP (Ethereum Improvement Proposals) beschreibt verschiedene Standards für die Ethereum-Plattform.[38] Die TypedTransaction beinhaltet den TransactionType und den TransactionPayload. Der TransactionType gibt den jeweiligen Typ der Transaktion an und der TransactionPayload gibt die zu diesem Typ gehörenden Transaktionskomponenten an.[39] Zur Verdeutlichung,

[33] vgl. Takenobu, 2018
[34] vgl. Antonopoulos und Wood, 2019, S. 112
[35] vgl. MyCrypto, 2021
[36] vgl. Computerbild, 2021
[37] vgl. Antonopoulos und Wood, 2019, S. 99ff
[38] vgl. Ethereum, 2021
[39] vgl. MyCrypto, 2021

wurde mittels des aus Listing 2.1 erstellten Smart Contracts »Donation.sol« 0.1 ETH von Adresse »0xDd7D08B014Cd8E58c86cE71c5094C7aa20BA957c« an Adresse »0x40C6E889f87dc5b6054 68B93A4EF816b6dDb00fB« gespendet. Die Daten zur Transaktion dieses Vorgangs sind in Listing 2.2 zu sehen. Bei dieser Transaktion (sowie bei allen weiteren dieser Arbeit) handelt es sich um den EIP-1559 Transaktionstyp. Dieser beinhaltet neben den LegacyTransaction-Komponenten noch folgende: *AccessList*, *ChainId*, *MaxFeePerGas*, *MaxPriorityFeePerGas* und *Type*. Die *AccessList* beinhaltet Zugangsdaten in Form von Tupeln, welche wiederum je eine Account-Adresse und eine Liste der Storage Keys beinhalten. Die *ChainId* gibt die ID der jeweiligen Blockchain an, auf welcher die Transaktion gültig ist. Das *MaxFeePerGas* ist identisch mit dem *GasPrice*. Das *MaxPriorityFeePerGas* gibt die Menge an Gas an, die der jeweilige Miner der Transaktion als »Trinkgeld« erhalten soll. Der *Type* gibt den jeweiligen Transaktionstyp an.[40] Das zur Ausgabe der Daten verwendete Tool gibt die Komponente *GasLimit* als *Gas* und die Komponente *Data/Init* für beides immer als *Input* an. Darüber hinaus werden vom Tool unnötigerweise sowohl der *GasPrice* als auch *MaxFeePerGas* angegeben. Diese Transaktion hat im Übrigen:

57891 Gas * 2338872469 Wei ≈ 0.0001354 ETH

gekostet. Sowohl für den *GasPrice* als auch für das *GasLimit* wurden in diesem Fall die voreingestellten Werte von MyEtherWallet (dazu später mehr) verwendet.

```
accessList: [],
chainId: '0x3',
gas: 60146,
gasPrice: '2338872469',
hash: '0xb410c3aa275e7d07874fb8d2f405311967a05e5db2eeefb091ac4622654f8786',
input: '0xed88c68e',
maxFeePerGas: '2338872469',
maxPriorityFeePerGas: '2338872469',
nonce: 1,
r: '0x64f5391b916444fccbea3a2d42298c7845b0488d37be0d49de73602e9abb108f',
s: '0x43edb6075f5fb88cb2dcf18e9b288fa1f75e15cf4abd134060f30f3cedca05ae',
to: '0x9A0B96750D1CCF7d5bE4e23c7199687bD5A97C63',
type: 2,
v: '0x1',
value: '100000000000000000'
```

Listing 2.2: Daten einer Transaktion

Bei Ethereum gibt es neben Transaktionen ein weiteres wichtiges Konstrukt zum Datenaustausch, nämlich die Messages. Diese werden von Contract Accounts verwendet, um miteinander zu kommunizieren. Möchte bspw. ein Contract Account eine Funktion eines anderen Contract Accounts aufrufen, dann geschieht dies mithilfe einer Message. Der ersten Message zwischen zwei Contract Accounts geht immer eine Transaktion eines EOAs voraus, d.h. sie können nicht

[40] vgl. Wood, 2022

initial verwendet werden. Anders als Transaktionen werden Messages nicht in Blöcke gepackt und sind somit kein Teil der Blockchain. Die Komponenten der Message sind identisch mit den Komponenten einer Transaktion. Kleine Unterschiede sind, dass die Werte *v,r,s* wegfallen (auf Grund des fehlenden privaten Schlüssels wird keine Signatur benötigt), der *GasPrice* nicht benötigt wird (dieser wird in der vorausgehenden Transaktion bereits festgelegt) und eine Message die Adresse des Senders einer Nachricht besitzt.[41]

Wie bereits weiter oben beschrieben, sorgen die Transaktionen für eine Zustandsänderung (bei Messages ist das nicht der Fall) des Global State, da jede Transaktion zur Veränderung eines Accounts führen kann. Durch die Transaktionen wird also ein neuer »finaler« Zustand des Weltcomputers erreicht. Abbildung 2.7 soll zur Veranschaulichung dienen. Des Weiteren werden sie ebenfalls in einem Trie (dem sog. Transaction Trie) gespeichert.[42]

Abbildung 2.7: Vereinfachte Darstellung der Zustandsüberleitung in einen neuen globalen Zustand durch Transaktionen.

2.3.5 Receipts

Receipts (engl. für Quittungen) speichern die Ergebnisse einer Transaktion. Sie werden benötigt, um die Auswirkungen einer Transaktion, bzw. die daraus folgende Veränderung des Global States, nachvollziehen zu können. Auch sie werden in einem separaten Trie (dem sog. Receipt Trie) gespeichert.

Die Receipts zu den LegacyTransactions bestehen aus den folgenden Komponenten: *Logs*, *Error*, *ExecutionResult*, *LogsBloom*, *CumulativeGas*, *GasUsed*, *Status* und generelle Daten. Die *Logs* geben die Liste der Log-Einträge an, die von einer Transaktion verursacht wurden. Log-Einträge werden von Smart Contracts für Transaktionen erstellt, wenn die Transaktion ein Event auslöst. Das *Error*-Feld vermerkt den Fehler, der beim Aufrufen einer Funktion durch eine Transaktion evtl. entstanden ist. Das *ExecutionResult*-Feld vermerkt das Ergebnis einer Funktion, die von der Transaktion aufgerufen wurde. *LogsBloom* ist ein Filter der auf die *Logs* angewandt wird, um diese aufzubereiten und so das Durchsuchen nach ähnlichen Inhalten innerhalb dieser erleichtert. Das *CumulativeGas* gibt die Summe des Gases an, welches die zu dem Receipt gehörige Transaktion und alle im Block vor dieser befindlichen Transaktionen verbraucht haben. Das

[41] vgl. Fertig und Schütz, 2019, S. 122f
[42] vgl. Antonopoulos und Wood, 2019., S. 120

GasUsed-Feld gibt an, wie viel Gas eine Transaktion verbraucht hat. Der *Status* gibt an, ob eine Transaktion erfolgreich war oder nicht. Dabei bedeutet 1 (bzw. true) das die Transaktion erfolgreich war und 0 (bzw. false) das sie nicht erfolgreich war. Bei den generellen Daten handelt sich sich um Daten wie *BlockHash*, *BlockNumber*, *ContractAddress*, *TransactionHash*, *TransactionIndex*, *From* und *To*. Diese dienen zur Lokalisierung einer Transaktion.[43]. Nachfolgend ist zur Verdeutlichung in Listing 2.3 das Receipt zur Transaktion aus Listing 2.2 dargestellt. Um Platz zu sparen wurde *LogsBloom* auf 0x0 gekürzt (dies wird in allen weiteren Listings dieser Arbeit ebenfalls so gehandhabt). Das *CumulativeGas* wird hier als *CumulativeGasUsed* bezeichnet. Da es sich hier um das Receipt zum EIP-1559 Transaktionstyp handelt, fallen die Komponenten *ExecutionResult* und *Error* weg. Dafür wird wiederum der *Type* angegeben.

```
blockHash: '0x8210209cc2531f82e1a6cf473d2dd104798a4382702236d048beb8ed92d1f604',
blockNumber: 11960431,
contractAddress: null,
cumulativeGasUsed: 573193,
effectiveGasPrice: '0x8b685c95',
from: '0xdd7d08b014cd8e58c86ce71c5094c7aa20ba957c',
gasUsed: 57891,
logs: [],
logsBloom: '0x0',
status: true,
to: '0x9a0b96750d1ccf7d5be4e23c7199687bd5a97c63',
transactionHash:
  '0xb410c3aa275e7d07874fb8d2f405311967a05e5db2eeefb091ac4622654f8786',
transactionIndex: 3,
type: '0x2'
```

Listing 2.3: Daten des Receipts einer Transaktion

2.3.6 Blöcke

In Abschnitt 2.2 wurde bereits beschrieben das ein Block aus einem Blockheader und einem Blockkörper besteht und welche Komponenten sich in diesen befinden. Bei Ethereum befinden sich, insbesondere im Blockheader, noch einige weitere Komponenten. Diese sind: *Beneficiary*, *ExtraData*, *GasLimit*, *GasUsed*, *MixHash*, *UnclesHash*, *StateRoot*, *TransactionRoot*, *ReceiptsRoot*, *Number* und *LogsBloom*. Die Komponente *Beneficiary* gibt dabei die Adresse des Accounts an, welcher erfolgreich einen Block geschürft hat und somit eine Belohnung erhalten soll. In *ExtraData* können Miner optionale Daten wie Nachrichten eintragen. Im Block stellt das *GasLimit* eine Begrenzung für die maximale Blockgröße bzw. der Anzahl an Transaktionen im Block dar, d.h. wie viel Gas von allen Transaktionen im Block verbraucht werden darf. Das *GasUsed*-Feld gibt hier die insgesamt verbrauchte Menge Gas an, die von allen im Block befindlichen Transaktionen benötigt wurde. Bei Ethereum wird zusätzlich zur Nonce der *MixHash* verwendet, um zu beweisen, dass die vom Miner verwendete Rechenleistung ausreichend für die Erstellung des

[43] vgl. Fertig und Schütz, 2019, S. 124f

Blocks war. Der *UnclesHash* ist ein Hash der Uncles-Liste. Dabei handelt es sich um eine Liste von verwaisten Blöcken, welche parallel gemined wurden, aber kein Teil der längsten Kette wurden. Die *StateRoot* ist die Wurzel des im Abschnitt 2.3.3 vorgestellten State Tries. Genauso verhält es sich mit der *TransactionRoot* und der *ReceiptsRoot*. Das *Number*-Feld gibt die Anzahl der in der Blockchain befindlichen Vorgängerblöcke aus Sicht des jeweiligen Blocks an. Wie in der Informatik üblich wird ab null beginnend (dem Genisis-Block bzw. Block Null) gezählt. Genauso wie die Receipts enthält auch der Header einen *LogsBloom*. Allerdings enthält dieser hier die Log-Daten aller Transaktionen im Block und nicht nur der Transaktion zum jeweiligen Receipt.[44] Listing 2.4 zeigt die Details zum Block, in welchem sich die Transaktion aus Listing 2.2 befindet. Auch hier wurde, um Platz zu sparen, *LogsBloom* auf 0x0 gekürzt. Die Komponente *Beneficiary* wird hier als *Miner* bezeichnet.

```
baseFeePerGas: 8,
difficulty: '4699838284',
extraData: '0x68747470733a2f2f7069636f706f6f6c2e6f7267',
gasLimit: 8000000,
gasUsed: 709950,
hash:
  '0x8210209cc2531f82e1a6cf473d2dd104798a4382702236d048beb8ed92d1f604',
logsBloom: '0x0',
miner: '0x66Fff7Bb63e8521180234F03F8F407E49347213C',
mixHash: '0x9f6f8e70c32ef9b112da5829cf4f3272e519e95ed1f244de8b13d341d434641f',
nonce: '0xe266ba5d41966aaf',
number: 11960431,
parentHash:
  '0xe13a58b63b926b45e36d79bebd3e8315577971a114fa4c9f41d6fd2d03991410',
receiptsRoot:
  '0xb2e000da742d4a2c18d78205de1f1822f4ec3869a64eb1332ce649bb961f1653',
sha3Uncles:
  '0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347',
size: 2542,
stateRoot:
  '0x941caed2d0507a2e31cc2355032c74eed8b3b0966b3f4c7f70a3288c10812ca5',
timestamp: 1644760972,
totalDifficulty: '39562003252866217',
transactions: [
                        ...
  '0xb410c3aa275e7d07874fb8d2f405311967a05e5db2eeefb091ac4622654f8786',
  '0x005bb0ca3382c1a510a4801f6c608c4d7f62cb524647d9eb5965339937f23946'
],
transactionsRoot:
  '0xc9fe07ba1507bd2752b58ac9586d460481d0d9859c4d0d896791c300653cf5e7',
uncles: []
```

Listing 2.4: Block zur Transaktion aus Listing 2.2

[44] vgl. ebd., S. 127f

Im Blockkörper befindet sich bei Ethereum ebenfalls eine weitere Komponente. Neben der Transaktionsliste enthält dieser noch die bereits zuvor erwähnte Uncles-Liste. Diese wird manchmal auch als Ommers-Liste bezeichnet. Abbildung 2.8[45] zeigt noch einmal die Zusammenhänge der vier zuvor beschriebenen Abschnitte auf. Die Komponente ParentHash enthält hier im Übrigen den Hash des vorangegangenen Blockheaders.

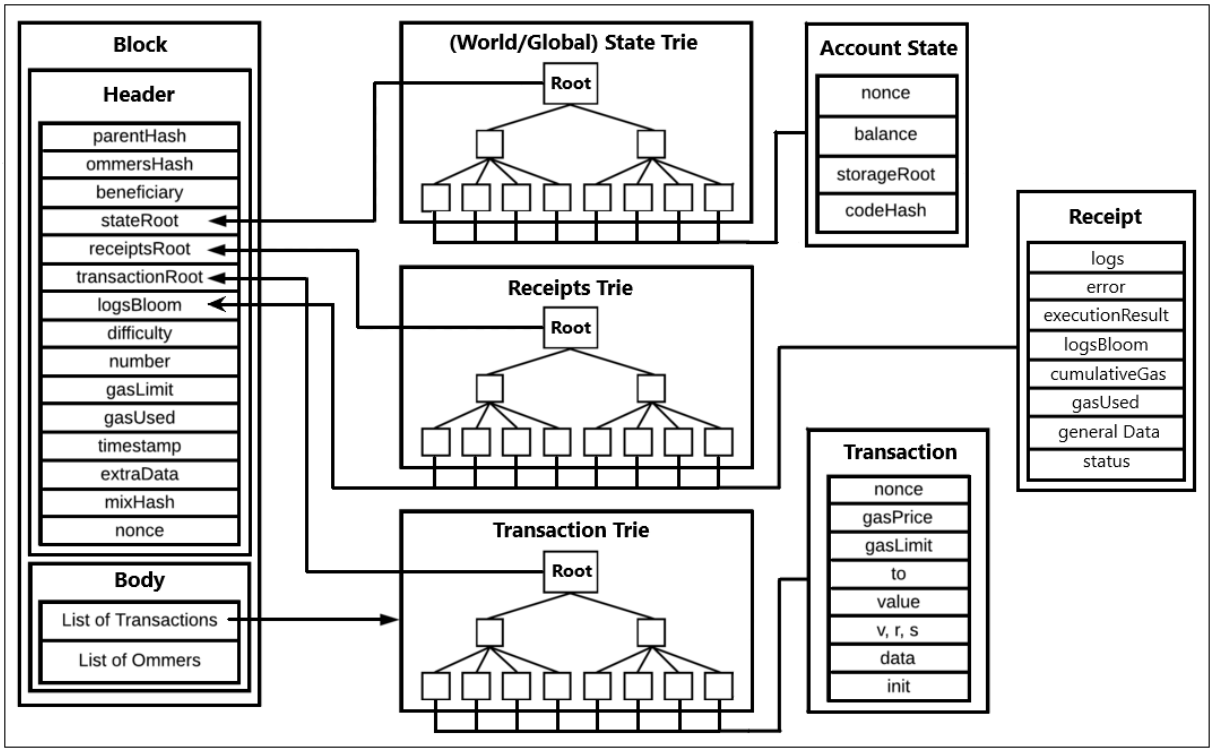

Abbildung 2.8: Vollständige Zusammensetzung eines Blocks in Ethereum.[45]
Originalquelle: `https://blog.csdn.net/inthat`

2.3.7 Ethereum Virtual Machine

Die EVM ist eine Stack- und Zustandsbasierte virtuelle Maschine. Stack-basiert bedeutet, dass ihre Speicherstruktur in Form von Stacks (engl. für Stapel) organisiert ist. Die daraus entstehenden Vorteile sind eine höhere Effizienz und verminderte Speicheranforderungen. Vereinfacht ausgedrückt werden Operanden (z.B. zwei zu addierende Zahlen) vom Stack der EVM genommen und das Ergebnis wird wieder auf den Stack geschoben. Zustandsbasiert bedeutet, dass sie einen Input bekommt und in einen neuen aktualisierten Zustand über geht, basierend auf diesem Input. Grob formuliert besteht die EVM aus dem bereits zuvor beschriebenen Global State, also dem globalen Zustand von Ethereum, welcher der Speicher aller Accounts (*Storage*) des Netzwerks ist. Eine weitere Komponente ist der *Machine State*, also der Maschinenzustand. Dieser beinhaltet den *Program Counter* (engl. für Programmzähler), das *verfügbare Gas* (für die Ausführung des jeweiligen Smart Contracts), den *Stack* und *Memory*. Die letzte Komponente

[45] Bhat, 2021

ist ein unveränderliches *virtuelles ROM*, welches mit dem Bytecode des auszuführenden Smart Contract geladen wird.[46] Abbildung 2.9 zeigt das Schema der EVM.[47]

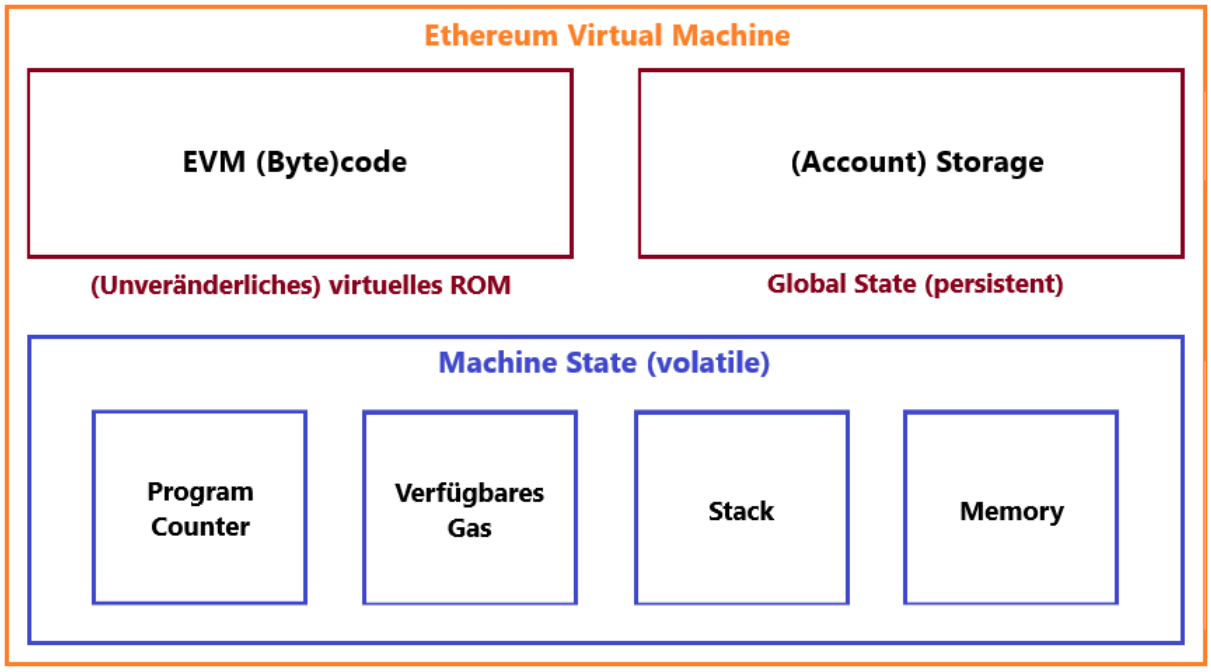

Abbildung 2.9: Schematische Zusammensetzung der Ethereum Virtual Machine.[47]

Die Aufgabe der EVM ist das Deployment und die Ausführung der Smart Contracts. Eine lokale Instanz der EVM befindet sich auf jedem Knoten im Ethereum-Netzwerk. Jede dieser Instanzen arbeitet mit dem gleichen Ausgangszustand und erzeugt den gleichen Endzustand. Die EVM berechnet die Aktualisierung des durch einen Smart Contract neu entstandenen Zustands. Mit anderen Worten, als in den vorherigen Abschnitten vom Weltcomputer die Rede war, war damit die EVM gemeint, da diese als Ganzes als ein Weltcomputer arbeitet.[48]

Grob vergleichen lässt sie sich unter anderem auch mit der Java Virtual Machine. Die JVM abstrahiert die zugrunde liegende Hardware eines Systems und bietet so die Kompabilität mit vielen verschiedenen Systemen. Der Code von Programmiersprachen wie Java (und allen weiteren Sprachen, welche die JVM nutzen) wird in JVM-Bytecode kompiliert. Dieser Bytecode wird dann von der JVM ausgeführt. Die EVM führt ebenfalls einen für sie kompilierten Bytecode aus. Dieser wird bei der Kompilierung (mit dem jeweiligen Compiler) des Quellcodes von Smart Contract Programmiersprachen wie Solidity und Vyper erzeugt.[49]. Listing 2.5 zeigt ein Beispiel für EVM-Bytecode im Hexadezimalformat, welcher zum Solidity-Quellcode »Donation.sol« aus Listing 2.1 gehört. Erwähnenswert ist an dieser Stelle noch, dass es sich dabei (um Platz zu sparen) um eine optimierte Variante des sog. Runtime-Bytecodes handelt. Dieser wird aus dem sog. Creation Bytecode erzeugt und enthält im Gegensatz zu diesem nicht die Konstruktor-Logik

[46] vgl. Schwarz, 2019
[47] vgl. Ethereum Wackerow, 2021
[48] vgl. Antonopoulous und Wood, 2019, S. 128
[49] vgl. ebd., S. 299

eines Contracts.[50] Optimiert wird für den Contract beim Deployen weniger Gas benötigt.

```
// Optimierter Runtime-Bytecode
0x6080604052600436106026576000356e01c806338cc483114602b578063ed88c68e146056575
b600080fd5b34801560365760008fd5b5060005460408051600160a01b0390921682525190
8190036020019f35b605c605e565b005b600054600160a01b031660b95760405162461bcd6
0e51b8152602060048201526001d60248201527f5472616e7366657220746f20746865207a65726f
206164647265737321000000604482015260640160405180910390fd5b60008054600516001600
160a01b0390911691348015610fc02929091818181858888f193505050501580156f2573d6000
803e3d6000fd5b5056fea2646970667358221220a2dc03dc04037ddc6413a7e00c98d2389ab238f
965b2a5b6f7a98d058f07b69764736f6c63430008070033
```

Listing 2.5: Optimierter Runtime-Bytecode zu Donation.sol

Wie die JVM (und generell Prozessoren) besitzt natürlich auch die EVM einen Befehlssatz aus Maschinenbefehlen. Jeder dieser Befehle enthält dabei einen sog. Opcode. D.h. eine Zahl, welche die Nummer eines Maschinenbefehls der EVM angibt.[51] Der EVM-Befehlssatz umfasst folgende Operationen: Arithmetische Operationen, Stack-Operationen, Operationen zur Programmsteuerung, Systemoperationen, Logische Operationen, Umgebungsoperationen und Blockoperationen. Die arithmetischen Operationen, die Operationen zur Programmsteuerung und die logischen Operationen sind selbsterklärend. Die Stack-Operationen sind Instruktionen zur Speicherverwaltung. Die Systemoperationen sind Instruktionen für das jeweilige System, welches das Programm ausführt. Die Umgebungsoperationen sind Instruktionen für den Umgang mit Informationen der Ausführungsumgebung. Die Blockoperationen sind Instruktionen für den Zugriff auf Informationen zum aktuellen Block.[52] Tabelle 2.2 zeigt zu den jeweiligen Operationen Beispiele für ein paar EVM-Befehle, deren Opcodes, die Mnemonics (kurzes, beschreibendes, meist abgekürztes Wort) zu den Opcodes und deren Gaskosten.[53] Die vollständige Auflistung aller Opcodes ist hier zu finden: `https://github.com/crytic/evm-opcodes`. Eine detaillierte Auflistung (Inputs, Outputs, Expressions usw.) ist hier zu finden: `https://ethervm.io/`. Listing 2.6 zeigt die (optimierten) Opcodes zum Quellcode aus Listing 2.1.

```
// Optimierter Operationscode
PUSH1 0x80 PUSH1 0x40 MSTORE PUSH1 0x4 CALLDATASIZE LT PUSH1 0x26 JUMPI PUSH1
0x0 CALLDATALOAD PUSH1 0xE0 SHR DUP1 PUSH4 0x38CC4831 EQ PUSH1 0x2B JUMPI DUP1
PUSH4 0xED88C68E EQ PUSH1 0x56 JUMPI JUMPDEST PUSH1 0x0 DUP1 REVERT JUMPDEST
CALLVALUE DUP1 ISZERO PUSH1 0x36 JUMPI PUSH1 0x0 DUP1 REVERT JUMPDEST POP PUSH1
0x0 SLOAD PUSH1 0x40 DUP1 MLOAD PUSH1 0x1 PUSH1 0x1 PUSH1 0xA0 SHL SUB SWAP1
SWAP3 AND DUP3 MSTORE MLOAD SWAP1 DUP2 SWAP1 SUB PUSH1 0x20 ADD SWAP1 RETURN
JUMPDEST PUSH1 0x5C PUSH1 0x5E JUMP JUMPDEST STOP JUMPDEST PUSH1 0x0 SLOAD PUSH1
0x1 PUSH1 0x1 PUSH1 0xA0 SHL SUB AND PUSH1 0xB9 JUMPI PUSH1 0x40 MLOAD PUSH3
0x461BCD PUSH1 0xE5 SHL DUP2 MSTORE PUSH1 0x20 PUSH1 0x4 DUP3 ADD MSTORE PUSH1
0x1D PUSH1 0x24 DUP3 ADD MSTORE PUSH32 0x5472616E7366657220746F20746865207A65726
F206164647265737321000000 PUSH1 0x44 DUP3 ADD MSTORE PUSH1 0x64 ADD PUSH1 0x40
```

[50] vgl. Fontaine, 2019
[51] vgl. Wikipedia, 2019
[52] vgl. ebd., S. 299ff
[53] Computerality, 2021

```
MLOAD DUP1 SWAP2 SUB SWAP1 REVERT JUMPDEST PUSH1 0x0 DUP1 SLOAD PUSH1 0x40 MLOAD
PUSH1 0x1 PUSH1 0x1 PUSH1 0xA0 SHL SUB SWAP1 SWAP2 AND SWAP2 CALLVALUE DUP1
ISZERO PUSH2 0x8FC MUL SWAP3 SWAP1 SWAP2 DUP2 DUP2 DUP2 DUP6 DUP9 DUP9 CALL
SWAP4 POP POP POP POP ISZERO DUP1 ISZERO PUSH1 0xF2 JUMPI RETURNDATASIZE PUSH1
0x0 DUP1 RETURNDATACOPY RETURNDATASIZE PUSH1 0x0 REVERT JUMPDEST POP JUMP
INVALID LOG2 PUSH5 0x6970667358 0x22 SLT KECCAK256 LOG2 0xDC SUB 0xDC DIV SUB
PUSH30 0xDC6413A7E00C98D2389AB238F965B2A5B6F7A98D058F07B69764736F6C63 NUMBER
STOP ADDMOD SMOD STOP CALLER
```

Listing 2.6: Optimierte Opcodes zu Donation.sol

Mnemonic	**Opcode**	**Gas**	**Beschreibung**
Bsp. arithmetische Operationen			
ADD	0x01	3	Addition
MUL	0x02	5	Multiplikation
Bsp. logische Operationen			
EQ	0x14	3	Gleich
XOR	0x18	3	Bitweises Exklusiv-Oder
Bsp. Umgebungsoperationen			
ADDRESS	0x30	2	Abruf Adresse
BALANCE	0x31	400	Guthaben abrufen
Bsp. Blockoperationen			
NUMBER	0x43	2	Abruf Nummer des Blocks
DIFFICULTY	0x44	2	Abruf Difficulty des Blocks
GASLIMIT	0x45	2	Abruf Gaslimits des Blocks
Bsp. Stack-Operationen			
POP	0x50	2	Operand vom Stack entfernen
PUSHx	0x60-0x7f	3	x-Byte Element auf Stack ablegen
DUPx	0x80-0x8f	3	x-stes Stack-Element duplizieren
SWAPx	0x90-0x9f	3	Erstes und x-stes Stack-Element vertauschen
Bsp. Operationen zur Programmsteuerung			
JUMP	0x56	8	Programmzähler ändern
JUMPI	0x57	10	Bedingte Änderung Programmzähler
Bsp. Systemoperationen			
CREATE	0xf0	32000	Neuen Contract Account erzeugen
RETURN	0xf3	0	Ausführung anhalten, Ausgabe zurückgeben
SELFDESTRUCT	0xff	5000*	Ausführung anhalten, Account löschen

Tabelle 2.2: Beispiele für EVM-Befehle und deren Gaskosten.[53]

Neben dem EVM-Bytecode gibt es eine weitere wichtige Komponente für das »Deployen« von Smart Contracts auf der EVM, nämlich das Ethereum Application Binary Interface (Ethereum Contract ABI, engl. für Binärschnittstelle). Bei einem ABI handelt es sich um eine Schnittstelle

zwischen zwei Computerprogrammen auf Maschinenebene. Häufig ist eines der Programme das Betriebssystem und das andere sind beliebige Benutzerprogramme.[54] Bei Ethereum dient das ABI dazu, die Funktionen und Events des Contracts zu definieren und die passende Beschreibung zu geben, wie Argumente von diesen akzeptiert und Ergebnisse zurückgeliefert werden. Es wird von DApp-Anwendungen wie bspw. Wallets benötigt, um Transaktionen zu konstruieren, welche die Funktionen mit den richtigen Argumenten und Datentypen aufrufen. Das ABI ist ein JSON-Array aus den Funktions- und Eventbeschreibungen.[55] Listing 2.7 zeigt das ABI, welches zu »Donation.sol« aus Listing 2.1 gehört.

```
[
  {
    "inputs": [
      {
        "internalType": "address payable",
        "name": "to_",
        "type": "address"
      }
    ],
    "stateMutability": "nonpayable",
    "type": "constructor"
  },
  {
    "inputs": [],
    "name": "donate",
    "outputs": [],
    "stateMutability": "payable",
    "type": "function"
  },
  {
    "inputs": [],
    "name": "getAddress",
    "outputs": [
      {
        "internalType": "address",
        "name": "",
        "type": "address"
      }
    ],
    "stateMutability": "view",
    "type": "function"
  }
]
```

Listing 2.7: ABI zu Donation.sol

Im Laufe der Entwicklung zu Ethereum 2.0 möchte Ethereum von der EVM auf EWasm (Ethereum WebAssembly) umsteigen. Auf der Webseite zu WebAssembly steht folgendes (Übersetzt

[54] vgl. Wikipedia, 2021
[55] vgl. Antonopoulos und Wood, 2019, S. 134f

aus dem Englischen): »WebAssembly (abgekürzt Wasm) ist ein binäres Befehlsformat für eine stapelbasierte virtuelle Maschine. Wasm ist als portables Kompilierungsziel für Programmiersprachen konzipiert und ermöglicht den Einsatz im Web für Client- und Serveranwendungen.«[56]. Wasm wird als eine ernstzunehmende Alternative zu Javascript entwickelt. Die Ziele von Wasm sind es eine hohe Effizienz, Geschwindigkeit, Portabilität und Sicherheit zu schaffen. Bei EWasm handelt es sich um eine angepasste Form von Wasm für Ethereum, die wiederum vollständig kompatibel zu Wasm ist und eine Schnittstelle zur Blockchain bietet. Mit EWasm will man einige Schwächen der EVM beheben. Darunter fallen bspw. die schlechte Performance und Portabilität. Mit EWasm wird es auch möglich sein mit Programmiersprachen wie C++ oder Rust Smart Contracts für Ethereum zu schreiben, da diese bereits für WebAssembly kompilieren können.[57]

2.3.8 ERC-Tokenstandards

Tokens stellen einen wesentlichen Bestandteil für die Existenz von Ethereum dar. Sie sind laut dem Ethereum Gründer Vitalik Buterin »eine der offensichtlichsten und nützlichsten Anwendungen einer generalisierten programmierbaren Blockchain wie Ethereum.«[58] Tokens werden im Zusammenhang mit der Blockchain-Technologie bspw. dazu verwendet, um Währungen (der häufigste Anwendungsfall) darzustellen, um Aktienanteile zu repräsentieren oder um Vermögenswerte wie z.B. Gold abzubilden. Sie stellen also ein virtuelles Asset dar. Sie haben noch einige weitere Anwendungsfälle. Einzelne Token können auch mehrere dieser Anwendungsfälle repräsentieren. Tokens lassen sich unterscheiden in fungible und non-fungible Tokens (NFT). Fungible bedeutet, dass sich solche Tokens mit anderen austauschen lassen ohne einen Unterschied im Wert oder der Funktion zu erhalten. D.h. sie besitzen keine einmaligen Eigenschaften. Die Fungibilität spielt bspw. auch bei Börsen eine wichtige Rolle, da nur fungible Werte an einer Börse gehandelt werden können. Non-fungible bedeutet, dass solch ein Token nicht austauschbar ist. Sie repräsentieren quasi für sich eine Sonderanfertigung, welche sich nicht im Wert oder der Funktion mit einem Token des gleichen Systems austauschen lässt.[59]

Währungen wie Ether, Bitcoin oder Dogecoin sind keine Tokens, sondern (wie der Name der letzteren Beiden bereits verrät) Coins. Der Unterschied besteht darin, dass sich Tokens nur auf das Ökosystem beziehen in dem sie existieren, während Coins auch außerhalb des Systems verwendet werden können.[60] Außerdem besitzen Tokens keine eigene Blockchain und werden ausschließlich mithilfe von Smart Contracts erstellt. Sie haben auch, wie zuvor bereits beschrieben, mehrere Anwendungsfelder, während Coins überwiegend als Währung zum Bezahlen dienen.[61] Die Begriffe sind zum Teil etwas irreführend. Beispielsweise begann EOS als Token und wurde später zu einem Coin, da die Entwickler später ihre eigene Blockchain erstellten.

[56] WebAssembly, 2022
[57] vgl. Schwarz, 2019
[58] Antonopoulos und Wood, 2019, S. 227
[59] vgl. ebd., S. 223
[60] vgl. Schmitz, 2019
[61] vgl. Kabakci, 2021

Viele Projekte die auf der Blockchain-Technologie beruhen, haben ihren Ursprung in Ethereum als ein ERC-Token. ERC steht für »Ethereum Request for Comments« und stellt eine Reihe verschiedener Standardisierungen für die Entwicklung von Tokens auf Ethereum dar. Diese ermöglichen die einfache Wiederverwendung der Tokens durch verschiedene Anwendungen (bspw. Wallets) indem sie eine gemeinsame Schnittstelle für Smart Contracts zur Implementierung eben dieser definieren. Die Schnittstellen dieser Standards setzen sich aus mehreren verschiedenen Funktionen und Events zusammen. Es gibt Standards sowohl für die Entwicklung von fungiblen Tokens als auch für NFTs.[62]

ERC20

Der bekannteste und am weitesten verbreitete fungible Tokenstandard ist ERC20. Die für diesen Standard vorgegebenen Funktionen, optionalen Funktionen und Events sind:[63]

totalSupply ⇒ Gibt die gesamte verfügbare Anzahl des Token zurück.

balanceOf ⇒ Gibt den Kontostand/das Guthaben der *_owner* Adresse zurück.

transfer ⇒ Transferiert einen Wert *_value* an Tokens zu Adresse *_to*.

transferFrom ⇒ Transferiert einen Wert *_value* an Tokens von Adresse *_from* zu Adresse *_to*.

approve ⇒ Erlaubt *_spender* mehrere Abhebungen vom Konto des Urhebers der Bestätigung (Approval) bis zum Wert *_value*.

allowance ⇒ Gibt den Betrag zurück, den *_spender* noch von *_owner* abheben darf.

Transfer ⇒ Muss ausgeführt werden wenn Tokens transferiert werden.

Approval ⇒ Muss ausgeführt werden wenn die Funktion approve erfolgreich aufgerufen wurde.

name ⇒ Gibt den Namen des Token an.

symbol ⇒ Gibt das Symbol des Token an.

decimals ⇒ Gibt die Dezimalstellen zurück, durch die die Token-Beträge geteilt werden.

Die vollständige Implementierung der ERC20-Schnittstelle, inklusive optionaler Funktionen, sieht in Solidity geschrieben also wie folgt aus (Listing 2.8):[64]

```
interface IERC20 {
    // Funktionen
    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);
```

[62] Antonopoulos und Wood, 2019, S. 228
[63] vgl. Lightclient, 2021
[64] ebd., 2021

```
7     function approve(address _spender, uint256 _value) external returns (bool
      success);
8     function allowance(address _owner, address _spender) external view returns (
      uint256 remaining);
9
10    // Events
11    event Transfer(address indexed _from, address indexed _to, uint256 _value);
12    event Approval(address indexed _owner, address indexed _spender, uint256
      _value);
13
14    // Optionale Funktionen
15    function name() external view returns (string memory);
16    function symbol() external view returns (string memory);
17    function decimals() external view returns (uint8);
18 }
```

Listing 2.8: ERC20-Implementierung in Solidity[64]

Hier wurde der ERC20-Standard als Interface implementiert. In der eigentlichen Spezifikation ist dies nicht der Fall, da zum Zeitpunkt der Erstellung dieses Standards (Ende 2015) noch keine Interfaces von Solidity unterstützt wurden. Erwähnenswert ist auch noch, dass es mittlerweile eine verbesserte Version dieses Standards gibt, nämlich den ERC777-Standard. Dieser schützt unter anderem vor dem versehentlichen Versenden von Tokens. Beim ERC20-Standard sind dadurch Millionen Verluste entstanden, da solche Tokens selbsterklärend für immer verloren sind. Eine weitere Besonderheit des ERC777-Standard ist, dass er Abwärtskompatibel mit ERC20 ist. Genauere Details zu diesem Standard sind hier zu finden: `https://github.com/0xjac/ERC777/blob/devel/eip-777.md`

ERC721

Unter den NFT-Standards hat ERC721 die weiteste Verbreitung. Auch dieser Standard hat vorgegebene Funktionen und Events. Diese sind:[65]

balanceOf ⇒ Gibt die Anzahl aller NFTs zurück die dem *_owner* zugewiesen sind.

ownerOf ⇒ Gibt die Adresse des Besitzers der NFTs *_tokenId* zurück.

safeTransferFrom ⇒ Überträgt das Eigentum an einem NFT von Adresse *_from* zu Adresse *_to*. Ermöglicht das Übertragen von zusätzlichen Daten *data*.

transferFrom ⇒ Überträgt das Eigentum an einem NFT von Adresse *_from* zu Adresse *_to*.

approve ⇒ Ändert oder bestätigt die genehmigte Adresse *_approved* eines NFT.

setApprovalForAll ⇒ Ermöglicht die Aktivierung oder Deaktivierung der Genehmigung für die Verwaltung durch Dritte.

getApproved ⇒ Gibt die genehmigte Adresse für den NFT *_tokenId* zurück.

[65] vgl. Davidbrai, 2021

isApprovedForAll ⇒ Fragt ab, ob eine Adresse ein autorisierter *_operator* für eine andere Adresse ist.

supportsInterface ⇒ Überprüft ob der Smart Contract ein Interface implementiert.

Transfer ⇒ Muss ausgeführt werden beim Besitzwechsel eines NFT durch eine der Transfer-Funktionen.

Approval ⇒ Muss bei der Änderung oder Bestätigung einer genehmigten Adresse eines NFTs ausgeführt werden.

ApprovalForAll ⇒ Wird ausgelöst, wenn ein *_operator* für einen Besitzer aktiviert oder deaktiviert wird.

Des Weiteren gibt es beim ERC721 noch optionale Schnittstellen. Deren Funktionen sind:

name ⇒ Gibt den Namen des NFT an.

symbol ⇒ Gibt das Symbol des NFT an.

tokenURI ⇒ Ein eindeutiger Uniform Resource Identifier (URI) für ein bestimmtes Asset *_tokenId*.

totalSupply ⇒ Gibt die Anzahl der durch diesen Smart Contract erfassten NFTs zurück.

tokenByIndex ⇒ Zählt die gültigen NFTs auf.

tokenOfOwnerByIndex ⇒ Zählt die NFTs auf, die dem *_owner* zugeordnet sind.

Die vollständige Implementierung der ERC721-Schnittstelle, inklusive der optionale Schnittstellen, sieht in Solidity geschrieben wie folgt aus (Listing 2.9):[66]

```
interface IERC721 /* is IERC165 */ {
    // Funktionen
    function balanceOf(address _owner) external view returns (uint256);
    function ownerOf(uint256 _tokenId) external view returns (address);
    function safeTransferFrom(address _from, address _to, uint256 _tokenId,
    bytes memory data) external payable;
    function transferFrom(address _from, address _to, uint256 _tokenId) external
     payable;
    function approve(address _approved, uint256 _tokenId) external payable;
    function setApprovalForAll(address _operator, bool _approved) external;
    function getApproved(uint256 _tokenId) external view returns (address);
    function isApprovedForAll(address _owner, address _operator) external view
    returns (bool);

    // Events
    event Transfer(address indexed _from, address indexed _to, uint256 indexed
    _tokenId);
    event Approval(address indexed _owner, address indexed _approved, uint256
    indexed _tokenId);
```

[66] ebd., 2021

```
15      event ApprovalForAll(address indexed _owner, address indexed _operator, bool
        _approved);
16  }
17  interface IERC165 {
18      function supportsInterface(bytes4 interfaceID) external view returns (bool);
19  }
20  // Optionale Schnittstelle Metadaten
21  interface IERC721Metadata /* is IERC721 */ {
22      function name() external view returns (string memory _name);
23      function symbol() external view returns (string memory _symbol);
24      function tokenURI(uint256 _tokenId) external view returns (string memory);
25  }
26  // Optionale Schnittstelle Enumerierung
27  interface IERC721Enumerable /* is IERC721 */ {
28      function totalSupply() external view returns (uint256);
29      function tokenByIndex(uint256 _index) external view returns (uint256);
30      function tokenOfOwnerByIndex(address _owner, uint256 _index) external view
        returns (uint256);
31  }
```

Listing 2.9: ERC721-Implementierung in Solidity[66]

In Kapitel 4, wird der ERC20-Token als Beispiel-Token mit jeder zu analysierenden Programmiersprache implementiert. Der Grund für die Wahl dieses Standards zur Analyse ist, dass dieser nach wie vor die weiteste Verbreitung hat. Des Weiteren ist er im Vergleich zu ERC777 oder ERC721 relativ simpel zu implementieren, da es zu quasi jeder Ethereum Smart Contract Programmiersprache bereits vorgefertigte Contracts zu diesem gibt und der Standard sogar die Verwendung dieser Contracts empfiehlt. Es müssen nur noch Kleinigkeiten für die eigene Verwendung angepasst werden (z.B. der Token-Name muss gewählt werden).

2.4 Smart Contracts

Smart Contracts (engl. für intelligenter Vertrag/Kontrakt) sind selbstausführende Verträge zwischen mehreren Parteien, die mittels verschiedener Programmiersprachen erstellt werden können. Es handelt sich dabei nicht um einen Vertrag im rechtlichen Sinne. Sie sind somit natürlich nicht rechtlich bindend. Das Wort Vertrag bzw. Kontrakt ist deshalb ein wenig unglücklich gewählt. Etwas anders und vereinfacht ausgedrückt handelt es sich dabei um kleine Computerprogramme, die auf einem dezentralisierten Weltcomputer (der EVM) laufen. Da sie eben auf der Blockchain-Technologie aufbauen, nutzen sie auch deren Vorteile wie z.B. Transparenz und Fälschungssicherheit.[67]

Letztendlich sind Smart Contracts nichts weiter als »Wenn-Dann-Regeln«. D.h. möchte beispielsweise eine Person A von Person B ein Auto kaufen, so legt Person B vorher in einem Smart

[67] vgl. Antonopoulos und Wood, 2019, S. 127f

Contract fest ab welchem Verkaufspreis Person A das Auto erhält. **Wenn** Person A die geforderte Summe aufbringt, **dann** wird ihr automatisch das Recht am Besitz des Autos zugeschrieben. Vereinfacht ausgedrückt folgt bei einem Smart Contract einer erfüllten Bedingung immer eine automatisierte Konsequenz.[68]

2.4.1 Programmiersprachen

Die Ethereum Foundation hat für die Entwicklung von Smart Contracts mehrere spezielle Programmiersprachen konzipiert. Im Englischen setzt sich für einige dieser Sprachen sogar vermehrt ein eigener Begriff durch, nämlich »Contract-oriented programming language« (zu deutsch: Kontraktorientierte Programmiersprache). Theoretisch ist es auch möglich mit Hochsprachen wie Java, Javascript oder C++ Smart Contracts zu entwickeln (dies ist bei anderen Smart Contract Plattformen wie Cardano oder Moonriver auch durchaus der Fall und wird mittels EWasm auch früher oder später bei Ethereum der Fall sein). Allerdings ist der Aufwand für die Anpassung dieser Sprachen zur Ausgabe von EVM-Bytecode aufwändiger als direkt eine neue Sprache für diese Aufgabe zu entwickeln. Diese spezialisierten Programmiersprachen verfügen über eine Reihe von Eigenschaften die für die Entwicklung auf der EVM bzw. einer Blockchain von hoher Relevanz sind. Dennoch sind sie im Vergleich zu herkömmlichen Programmiersprachen stark abgespeckt, da ihre Programme auf einer hochgradig beschränkten Ausführungsumgebung (nämlich der EVM) laufen.[69] Die Ethereum-Sprachen sind: LLL, Serpent, Mutan, Solidity, Vyper, Bamboo, Yul/Yul+ und Fe.

LLL steht für Low Level Lisp und ist die erste, von der Ethereum Foundation, entwickelte Programmiersprache für die Erstellung von Smart Contracts auf der EVM. LLL ist deklarativ, sehr minimalistisch und besitzt eine Lisp-artige Syntax.[70] In der Ethereum Homestead Documentation wird sie wie folgt beschrieben (Übersetzt aus dem Englischen): »Low Level Lisp (LLL) ist eine Low-Level-Sprache, ähnlich wie Assembler. Sie soll sehr einfach und minimalistisch sein. Im Wesentlichen ist sie nur ein kleiner Wrapper für die direkte Kodierung in der EVM.«[71]

Bei Serpent handelt es sich um eine imperative, statisch typisierte (d.h. die Datentypen der Variablen müssen zur Übersetzungszeit bekannt sein) Programmiersprache mit einer Python-artigen Syntax. Serpent zielt darauf ab, möglichst simplen und sauberen Code zu erstellen. Die Ersteller wollten die Effizienzvorteile einer Low-Level-Sprache mit der Benutzerfreundlichkeit der Python-artigen Syntax kombinieren. Serpent ist der Vorläufer zu Vyper.[72]

Bei Mutan handelt es sich ebenfalls um eine imperative, statisch typisierte Programmiersprache, allerdings mit einer C-artigen Syntax. Auf der GitHub Seite zu Mutan steht folgendes (Übersetzt

[68] vgl. Schiller, 2018
[69] vgl. Antonopoulos und Wood, 2019, S. 129ff
[70] vgl. LLL Compiler Documentation, 2017
[71] Ethereum Homestead Documentation, 2016
[72] vgl. Ethereum Wiki, 2021

aus dem Englischen): »Mutan ist eine C-ähnliche Sprache für das Ethereum Projekt. Mutan unterstützt eine vollständige, statisch typisierte Hochsprache, die zu EVM Bytecode kompiliert.«[73]

Solidity ist die derzeit am Häufigsten vorkommende Programmiersprache für die Erstellung von Smart Contracts auf der EVM. Es handelt sich dabei um eine imperative, statisch typisierte Sprache, welche die bekannten Mechanismen zur Objektorientierung unterstützt. Sie wurde von mehreren Sprachen beeinflusst. Den größten Einfluss hatten Javascript, C++ und Python.[74]

Bei Vyper handelt es sich um den direkten Nachfolger von Serpent. Wie diese ist Vyper eine imperative, statisch typisierte Programmiersprache mit einer Python-artigen Syntax. Vyper zielt vor allem auf eine erhöhte Sicherheit bei der Erstellung von Smart Contracts ab. Um dies zu erreichen verzichtet Vyper unter anderem auf einige objektorientierte Funktionalitäten wie bspw. die Vererbung oder Polymorphie. Das Wegfallen dieser, und noch einiger weiterer Funktionalitäten, soll es erschweren fehlerhaften Code zu schreiben.[75]

Gute Quellen zu Bamboo sind leider kaum vorhanden. Der Entwickler Yoichi Hirai hat die Arbeit an seiner Programmiersprache eingestellt. Der Grund dafür sind vermeintlich rechtliche Probleme in Japan.[76] Auf der GitHub Seite zu Bamboo steht noch folgendes (Übersetzt aus dem Englischen): »Bamboo ist eine Programmiersprache für Ethereum-Verträge. Bamboo macht Zustandsübergänge explizit und vermeidet standardmäßig Reentrance-Probleme.«[77]

Yul ist eine Zwischensprache für die EVM. D.h. sie befindet sich auf einer Abstraktionsebene zwischen einer höheren Sprache wie Solidity und dem EVM-Bytecode. Der Yul-Code wird mit dem Solidity-Compiler zu EVM-Bytecode kompiliert. Yul-Code kann auch als Inline-Assembly innerhalb von Solidity-Code verwendet werden.[78] Yul+ kann als eine Erweiterung zu Yul gesehen werden, mit welcher neue Funktionalitäten hinzugefügt wurden. Zu den Erweiterungen gehören bspw. Enums, Konstanten, Booleans und ein integrierter Über-/Unterlauf Schutz. Beide »Versionen« sind imperativ und dynamisch typisiert. Yul+ wird im Laufe seiner Entwicklung noch auf die statische Typisierung umgestellt. Des Weiteren können beide als Stand-alone verwendet werden.[79]

Bei Fe handelt es sich um eine im Alpha-Stadium befindende Programmiersprache. Sie ist ebenfalls imperativ und statisch typisiert. Das Ziel von Fe ist es, möglichst leicht erlernbar zu sein. Selbst Nicht-Programmierer sollen mit dieser Sprache in der Lage sein, die Smart Contract Entwicklung schnell zu erlernen. Beeinflusst wurde Fe hauptsächlich von Rust und Python. Der

[73] Obscuren, 2015
[74] vgl. Solidity Documentation, 2021
[75] vgl. Vyper Documentation, 2020
[76] vgl. Reddit RealGoat, 2018
[77] Pirapira, 2018
[78] vgl. Yul Documentation, 2021
[79] vgl. Adlerjohn, 2021

Alpha Release war im Januar 2021.[80]

In dieser Arbeit werden nur die Programmiersprachen Solidity, Vyper und und Fe genauer untersucht. Alle anderen hier erwähnten Sprachen werden kaum noch verwendet bzw. gewartet (mit Ausnahme von Yul/Yul+) und sind deshalb für die Analyse nicht weiter relevant. Dennoch sei hier erwähnt, dass einige dieser Sprachen noch eine Daseinsberechtigung haben und ein Blick auf diese sich durchaus lohnen kann. Der Grund für die Analyse ausgerechnet dieser drei Sprachen ist simpel. Solidity und Vyper haben bereits eine große Verbreitung im Sektor der Smart Contract Programmierung, weshalb eine Analyse dieser beiden Sprachen sinnvoll erscheint. Fe wiederum ist eine neue, sehr junge Sprache und somit für die Analyse besonders interessant. Des Weiteren sind dies drei der vier von Ethereum für die Smart Contract Programmierung empfohlenen Sprachen. Die vierte empfohlene Sprache ist Yul bzw. Yul+. Der Grund warum Yul nicht in der Analyse vorkommt ist, dass diese als Zwischensprache eine sehr kryptische Syntax hat und generell von Umsteigern oder Einsteigern in die Smart Contract Programmierung nicht verwendet werden sollte. Yul wird von Ethereum nur für sehr erfahrene Blockchain-Programmierer empfohlen, die bereits ein tiefgreifendes Verständnis für die Smart Contract Programmierung besitzen. Für diese bringt die Sprache tatsächlich auch einige Vorteile mit, unter anderem lässt sich mit ihr, auf Grund ihrer Nähe zur EVM, sehr effizienter Code schreiben.

[80] vgl. Ethereum Languages, 2021

3 Kriterien zur Analyse

3.1 Das Bewertungssystem

Als Bewertungssystem bzw. Analyseverfahren wurde die Nutzwertanalyse, welche auch als Scoring-Modell bezeichnet wird, gewählt. Dabei handelt es sich um eine subjektives Entscheidungsverfahren zur Ermittlung der optimalen Option aus einer Menge gegebener Optionen. Dieses Verfahren wird unter anderem von der Stiftung Warentest zur Benotung der von dieser getesteten Produkte verwendet. Das Verfahren eignet sich besonders gut, um aus sich ähnelnden Optionen einen Spitzenreiter zu finden. Somit eignet es sich perfekt für die gegebene Fragestellung dieser Arbeit.

Die Funktionsweise ist sehr simpel. Zu Beginn müssen natürlich verschiedene Entscheidungsalternativen vorliegen, also in diesem Fall die verschiedenen Smart Contract Sprachen. Als nächstes ist es wichtig klare Bewertungskriterien festzulegen und diese zu gewichten. Die Erläuterung und Gewichtung zu den Kriterien erfolgt in Abschnitt 3.2. Zur Interpretierung der Ergebnisse muss noch ein Bewertungsmaßstab festgelegt werden. Im Falle dieser Arbeit wurde sich für einen Maßstab von 1 bis 5 Punkte entschieden, wobei gilt:

1 = sehr schlecht 2 = schlecht 3 = befriedigend 4 = gut 5 = sehr gut

In den letzten Schritten erfolgt nun die Bewertung der Entscheidungsmöglichkeiten, dies geschieht in Kapitel 4. Dabei werden zu jedem Kriterium je Alternative die Punkte vergeben. Abschließend werden diese Einzelgewichtungen zu einer gewichteten Punktzahl pro Alternative aufsummiert. Gemäß der Nutzwertanalyse ist die Alternative mit der höchsten Punktzahl die beste Wahl.[1]

3.2 Erläuterung zu den Kriterien

Da es sich bei der Nutzwertanalyse um ein rein subjektives Entscheidungsverfahren handelt, ist es an dieser Stelle besonders wichtig die jeweiligen Kriterien für diese zu erläutern sowie deren Gewichtung genau zu spezifizieren. Dabei wurde sich unter anderem anhand der vorhergegangen Literaturrecherche orientiert, um am Ende ein möglichst aussagekräftiges Ergebnis zu erhalten. Die Kriterien sind Lesbarkeit, Funktionalitäten, Verfügbarkeit, Lernaufwand, Sicherheit und Effizienz. Die Intentionen dahinter werden nachfolgend beschrieben. Einige Kriterien werden als Unterpunkte dargestellt.

[1] vgl. Studyflix, 2018

Code-Aufbau

Dieser Punkt dient dazu eine kurze Übersicht über die Syntax sowie dem Aufbau und der Struktur des Codes der jeweiligen Smart Contract Sprache zu verschaffen. Des Weiteren wird hier die Lesbarkeit des Codes beurteilt sowie die Codelänge für die Erstellung der Beispiel-Token ausgewertet. Die Lesbarkeit kann besonders für Programmieranfänger ausschlaggebend sein. Außerdem ist sie bei der Fehlerfindung relevant. Umso übersichtlicher und strukturierter der Code ist, desto leichter ist es evtl. Fehler zu finden. Deshalb wird dieses Kriterium mit 15% gewichtet.

Spracheigenschaften

Dieser Punkt soll die Spracheigenschaften der jeweiligen Smart Contract Sprache kurz vorstellen. Dabei wird nur erläutert welche Funktionalitäten es gibt. Anhand dessen soll hier der Umfang an Funktionalitäten im Verhältnis zu den jeweils anderen Sprachen beurteilt werden. Programmiersprachen besitzen unterschiedlich viele Funktionalitäten zur Erstellung von Code, ebenso ist es bei Smart Contract Sprachen. Viele Funktionalitäten bedeuten meistens eine schnellere und einfachere Lösung zum Ziel, bzw. mehrere Wege zum Ziel. Für Programmieranfänger können zu viele Funktionalitäten erschlagend wirken, für fortgeschrittene Programmierer können sie allerdings sehr hilfreich sein. Dieses Kriterium wird deshalb mit 20% gewichtet. Dieser Punkt wird insgesamt nur sehr oberflächlich behandelt, d.h. die Funktionalitäten im Einzelnen werden nur kurz beschrieben und manche auch gar nicht. Es geht hier nur darum, einen ungefähren Überblick über den Sprachumfang zu verschaffen.

Verfügbarkeit

Dieses Kriterium soll die Verbreitung der jeweiligen Smart Contract Sprache beurteilen. Unter diesen Punkt fällt auch die Portabilität, d.h. ob die jeweilige Smart Contract Sprache auch für andere Blockchain-Plattformen kompiliert werden kann. Eine gute Portabilität erhöht natürlich auch die Verbreitung. Die Verfügbarkeit ist nicht irrelevant, denn je höher die Verbreitung einer Sprache ist, umso unwahrscheinlicher ist es, dass diese »ausstirbt«. Kaum jemand will etwas erlernen, was nach einem Jahr evtl. nicht mehr von nutzen ist. Dieses Kriterium wird mit 10% gewichtet.

Lernaufwand

Dieser Punkt dient dazu, um zu beurteilen wie viel Lernaufwand proportional zu den anderen Programmiersprachen benötigt wird, um diese zu erlernen. D.h. welche der Smart Contract Sprachen eignet sich am besten, um schnell, ohne großes Lernen, Smart Contracts zu erstellen. Die Schwierigkeit hinter der Erlernung einer Programmiersprache wird durch gute Hilfestellungen verringert. Diese können ein wahrer Segen bei der schnellen und leichten Erlernung sowie bei der generellen Verwendung einer Programmiersprache sein. Bei jungen Programmiersprachen bilden sich solche Gegebenheiten allerdings teilweise erst über Jahre hinweg. D.h. Tutorials, eine gut geschriebene, ausführliche Dokumentation (bei der Bewertung wird auch der Dokumentationsumfang der jeweiligen Sprache mit einbezogen) oder vorgefertigte Code-Lösungen durch Foren wie StackOverflow sind bei jungen Sprachen zum Teil kaum vorhanden. Da alle Smart Contract

Sprachen die hier analysiert werden noch vergleichsweise jung sind, gilt es auch diesen Punkt mit in die Bewertung einzubeziehen. Hilfestellungen sind sowohl für Programmieranfänger als auch für fortgeschrittene Programmierer sehr relevant, da man beim Programmieren nie auslernt. Insgesamt wird dieses Kriterium mit 10% gewichtet.

Sicherheit

Der Punkt Sicherheit dient dazu, um zu bewerten wie fehleranfällig die jeweilige Smart Contract Sprache ist. Insbesondere Einsteiger sind mit den vielen verschiedenen Optionen einer Programmiersprache schnell überfordert und verursachen so ungewollt schwerwiegende Fehler im Programmcode. Da Smart Contracts zum Teil große Summen an Geld verwalten und steuern, ist die Sicherheit dieser von besonders hoher Relevanz. Kleinste Fehler können zu hohem Verlust führen. Dieses Kriterium wird deshalb mit 25% gewichtet.

Effizienz

Das Kriterium Effizienz ist selbsterklärend. Es bezieht sich darauf, wie viel Gas ein vollständig implementierter Token beim Deployment benötigt. Wie bereits in Abschnitt 2.3.2 beschrieben, hat das Gas bei Ethereum eine wichtige Rolle. Je mehr Gas ein Smart Contract verbraucht, desto mehr muss für diesen auch bezahlt werden. Viel Gas bedeutet, dass die EVM viel Rechenleistung und Speicherressourcen zur Ausführung aufwenden musste. Da die Compiler der Sprachen die Möglichkeit bieten den EVM-Bytecode zu optimieren, wird auch der anfallende Gasverbrauch für die optimierten Token in die Bewertung miteinbezogen. Um den Gasverbrauch ermitteln und die Gaskosten errechnen zu können, muss der Code natürlich zuerst kompiliert und deployed werden. Deshalb werden unter diesem Punkt auch die Schritte zum Kompilieren erläutert. Da der Gasverbrauch eine wichtige Rolle bei Ethereum spielt, wird dieses Kriterium mit 20% gewichtet.

Ein Beispiel zur Verdeutlichung der gesamten Methodik ist in Tabelle 3.2 dargestellt. Ausgehend von drei Programmiersprachen Alpha, Beta und Gamma, wäre Programmiersprache Gamma hier der Sieger mit einem Ergebnis von 3,1 Punkten. Auf Platz 2 wäre Programmiersprache Alpha mit 2,6 Punkten als Ergebnis. Auf dem letzten Platz wäre Programmiersprache Beta mit 2,25 Punkten als Ergebnis.

	Alpha	**Beta**	**Gamma**
Lesbarkeit (15%)	1	4	3
Funktionalitäten (20%)	5	3	3
Verfügbarkeit (10%)	2	2	1
Lernaufwand (10%)	3	2	4
Sicherheit (25%)	3	1	3
Effizienz (20%)	1	2	4
Bewertung	**2,6**	**2,25**	**3,1**

Tabelle 3.1: Beispiel zur Nutzwertanalyse

4 Analyse und Vergleich

Für die Analyse wurden die Dokumentationen der jeweiligen Smart Contract Sprachen als Hilfe verwendet. Diese sind im Literaturverzeichnis unter dem jeweiligen Namen der Sprache verlinkt.

4.1 Der Beispiel-ERC20-Token

Bei der Implementierung des beispielhaften ERC20-Token (mit den jeweiligen Smart Contract Sprachen), wurde sich an der Implementierungsvariante von OpenZeppelin orientiert. Es wurde quasi eine vereinfachte und etwas angepasste Version dieser verwendet. Die eigentliche OpenZeppelin-Variante ist von Ethereum als Grundlage für einen eigenen Token zur Verwendung empfohlen (Stand: Januar 2022). OpenZeppelin erweitert den ERC20-Standard noch um ein paar nützliche Funktionalitäten. Des Weiteren wird hier der ERC20-Standard als Interface vom eigentlichen Token-Contract implementiert. Eine Neuerung im Vergleich zu älteren Implementierungsvarianten, bei welchen es für z.B. Solidity noch keine Interfaces gab. Interfaces werden nach Konvention mit einem I voran benannt, also in diesem Fall IERC20 (wie auch in Abschnitt 2.3.8 bereits gezeigt).

In der eigentlichen OpenZeppelin-Variante sind drei Imports vorhanden, nämlich Context.sol, IERC20.sol und SafeMath.sol sowie optional noch ERC20Detailed.sol. Im Fall des Beispiel-Tokens zur Analyse ist nur IERC20.sol vorhanden und relevant. Bei Context.sol handelt es sich um eine Definition um OpenGSN zu benutzen. OpenGSN erlaubt einem externen Nutzer ohne den Besitz von Ether die Ethereum-Blockchain zu nutzen. Bei SafeMath.sol handelt es sich um eine Bibliothek, die dazu führt, dass Additionen und Subtraktionen ohne Überlauf durchgeführt werden. Sowohl Context.sol als auch SafeMath.sol werden für die Implementierung nicht zwingend benötigt, insbesondere SafeMath.sol sollte aber bei der Erstellung eines echten Tokens vorhanden sein, da dadurch eine gefährliche Sicherheitslücke behoben wird.[1] Bei ERC20Detailed.sol handelt es sich lediglich um die optionalen Funktionen von ERC20, also *name*, *symbol* und *decimals*.[2] Diese wurden im Falle des Beispiel-Tokens bereits in IERC20.sol implementiert.

Die über den Basisstandard hinausgehenden Funktionen sind: *increaseAllowance*, *decreaseAllowance*, *_transfer* und *_approve*. Bei den ersten beiden Funktionen handelt es sich um Zusätze für eine erhöhte Sicherheit. Beide Funktionen können alternativ zu *approve* verwendet werden,

[1] vgl. Ethereum, 2021
[2] vgl. OpenZeppelin Docs, 2021

um ein potentielles Problem dieser Funktion zu umgehen. Bei Verwendung von *approve* zur Änderung der Allowance kann es nämlich passieren, dass jemand bei einer unvorteilhaften Transaktionsreihenfolge sowohl die alte, als auch die neue Allowance verwendet. Die beiden Funktionen *_transfer* und *_approve* übernehmen den Großteil der Arbeit im Contract. Sie sind das interne Äquivalent zu *transfer* und *approve* und werden von diesen Funktionen jeweils verwendet. Des Weiteren werden beide in *transferFrom* verwendet und *_approve* wird noch in den beiden zuvor erwähnten Funktionen *increaseAllowance* und *decreaseAllowance* verwendet. Erwähnenswert ist noch, dass in der eigentlichen OpenZeppelin-Variante noch drei weitere Funktionen vorkommen, nämlich *_mint*, *_burn* und *_burnFrom*. Diese werden dazu verwendet, um das Gesamtvorkommen an Tokens zu modifizieren.[3] Mehr zum OpenZeppelin-ERC20 ist hier zu finden: `https://docs.openzeppelin.com/contracts/2.x/api/token/erc20#ERC20`. An dieser Stelle sei erwähnt, dass es verschiedene Möglichkeiten zur Implementierung des OpenZeppelin-ERC20 gibt.

Beim Schreiben der OpenZeppelin Beispiel-Token wurde zum Teil der Code der GitHub-Nutzer v0xat[4], sbillig[5] und skellet0r[6] übernommen. Des Weiteren wurde das Walk-Through von Ethereum zur Hilfe genommen.[7] Die Token wurden identisch, in der jeweiligen Smart Contract Sprache, zu den anderen Sprachen abgeglichen und strukturiert. Nur eben mit den zur Verfügung stehenden Mitteln der jeweiligen Sprache. Sämtliche Einrückungen, Leerzeilen, Kommentare, Variablen- und Funktionsbezeichnungen sind möglichst identisch verfasst worden.

Der Code aller Token wurde mithilfe des Texteditors Visual Studio Code bearbeitet. Dabei wurden Extensions für die Sprachen installiert, um bspw. Syntax-Highlighting verwenden zu können. Die Token wurden mithilfe von MyEtherWallet (`https://www.myetherwallet.com/wallet/deploy`) und MetaMask (`https://metamask.io/`) auf dem Ethereum Testnetzwerk Ropsten (hier sind Statistiken zu Ropsten: `https://teth.bitaps.com/`) deployed. Um einen Contract auf Ropsten (und allen anderen Testnetzwerken sowie dem Mainnet) deployen zu können, wird der kompilierte EVM-Bytecode und das Ethereum Contract ABI (siehe Abschnitt 2.3.7) benötigt. Diese wurden mithilfe des jeweiligen Compilers einer Sprache über die Kommandozeile erstellt. Beides muss in das dafür vorgesehene Feld bei MyEtherWallet eingetragen werden. Bevor die Token deployed werden können, muss noch jeweils der Name, das Symbol, das Token-Gesamtvorkommen und die Dezimalstellen eingetragen werden. Diese Schritte sind bei allen Token identisch. Für das GasLimit wurden für alle Token die voreingestellten Werte von MyEtherWallet übernommen. Dieser Wert ist so gut wie immer identisch mit dem tatsächlich verwendeten Gas des Contracts, da MyEtherWallet schon vorab die Gaskosten ermittelt und das GasLimit direkt auf diesen Wert setzt. Für den GasPrice wurde ein Betrag von 2.5 GWei pro Gaseinheit für alle Token gewählt. Der GasPrice im Mainnet kann im Übrigen wesentlich höher ausfallen.

[3] vgl. ebd.
[4] V0xat, 2022
[5] Sbillig, 2022
[6] Skellet0r, 2021
[7] Ethereum, 2021

Um die Transaktionen bezahlen zu können, wurden von einem sog. Ropsten-Faucet Test-ETHs angefordert. Um die genauen Transaktionsdaten, Receipts und Blockinformationen zu den jeweiligen Token zu erhalten, wurde Web3.js (`https://web3js.readthedocs.io/en/v1.7.0/`), Node.js (`https://nodejs.org/en/`), Infura (`https://infura.io/`) und Truffle (`https://trufflesuite.com/docs/truffle/reference/configuration.html`) verwendet. Listing 4.1 zeigt den JS-Code der dafür geschrieben wurde.

```
const HDWalletProvider = require('@truffle/hdwallet-provider');
const Web3 = require('web3');

const provider = new HDWalletProvider(
  // Das Mnemonic des jeweilig verwendeten Metamask Accounts
    'zwoelf englische worte in eine Reihe geschrieben bilden das Mnemonic eines Accounts',
    'https://ropsten.infura.io/v3/ba5fd02fbb234761b174d5c9d20e6938'
);

const web3 = new Web3(provider);

// Der Transaktions-Hash der Transaktion aus Listing 2.2
const transactionHash = '0xb410c3aa275e7d07874fb8d2f405311967a05e5db2eeefb091ac4622654f8786';
web3.eth.getTransaction(transactionHash, function (error, result) {
    console.log(result);
});

web3.eth.getTransactionReceipt(transactionHash, function(error, result) {
    console.log(result);
});

// Die Block-Nummer in welchem sich die Transaktion aus Listing 2.2 befindet
const blockNumber = '11960431';
web3.eth.getBlock(blockNumber, function(error, result) {
    console.log(result);
});
provider.engine.stop();
```

Listing 4.1: JS-Code für Transaktionsdaten, Receipt und Blockinformation

4.2 Solidity

Der Code zum Solidity-ERC20-Token ist in Anhang A zu finden.

Code-Aufbau

Ein Solidity Contract sollte immer mit einem sog. »SPDX-License-Identifier« beginnen. Dieser wird dafür genutzt, um die genaue Identifizierung der Lizenz von Open-Source-Software einhalten zu können. Im Beispiel-Token wird die MIT-Lizenz verwendet. Als Nächstes folgt das

Versionspragma. Mit diesem wird die Version des Contracts für die verschiedenen Compiler-Versionen festgelegt. Im Beispiel-Token wäre die Version ab 0.7.0 bis 0.8.9 zum Kompilieren zulässig. Nach dem Versionspragma folgen evtl. Import-Anweisungen. Das können bspw. Erweiterungen, Bibliotheken oder Interfaces sein. Nun beginnt der eigentliche Contract. Dieser ist ähnlich zu einer Klasse in Objektorientierten Programmiersprachen. Innerhalb dieses Konstrukts befinden sich zu Beginn die Zustandsvariablen. Diese sind quasi die Instanzvariablen der Smart Contract Programmierung. Als Nächstes folgt optional genau ein Konstruktor. Ein Solidity-Contract darf nur einen Konstruktor haben, d.h. das Überladen von Konstruktoren ist nicht möglich. Nach dem Konstruktor folgen die Funktionen/Methoden und die Events. Vor dem Konstruktor, den Funktionen und den Events können optional noch selbstdefinierte Modifikatoren stehen.

Die Syntax von Solidity ähnelt der Syntax von Programmiersprachen wie Javascript, C++ oder Java. Wie auch bei diesen Programmiersprachen werden Anweisungsblöcke in geschweiften Klammern geschrieben und jede Anweisung endet mit einem Semikolon. Strings werden in doppelten oder einfachen Anführungsstrichen geschrieben und Kommentare beginnen mit »//« bzw. mit »/* « und enden mit »*/« für mehrzeilige Kommentare. Des Weiteren werden Imports durch das Schlüsselwort »import« gekennzeichnet. Die Syntax für die Kontrollstrukturen ist identisch mit der Syntax von Java oder Javascript. Der Contract wird eingeleitet mit dem Schlüsselwort »contract« gefolgt vom jeweiligen Contract-Namen. Das Schlüsselwort entspricht hier also dem, was in Programmiersprachen wie Java »class« entspricht. Die (Zustands)variablen-Deklaration beginnt mit dem jeweiligen Datentyp, gefolgt von den Modifikatoren und dem Variablennamen. Im Beispiel-Token sind vier Zustandsvariablen vorhanden, nämlich *_totalSupply*, *_decimals*, *_name* und *_symbol*. Der Konstruktor wird durch das Schlüsselwort »constructor« gekennzeichnet. In älteren Solidity-Versionen wird der Konstruktor unter anderem noch mit dem Schlüsselwort »function« gefolgt vom Contract-Namen gekennzeichnet. Im Beispiel-Token werden dem Konstruktor der Name *name_*, das Symbol *symbol_*, das Gesamtvorkommen der Token *total_* und die möglichen Nachkommastellen *decimals_* übergeben. Für Funktionsdeklarationen gilt, erst das Schlüsselwort »function«, dann der Funktionsname, dann die Modifikatoren sowie zum Schluss ggf. eine Liste, gekennzeichnet durch das Schlüsselwort »returns«, mit den zurückzugebenden Datentypen. Dem Funktionsnamen folgt die Parameterliste. Falls keine Parameter vorhanden sind, werden wie üblich in der Programmierung lediglich die runden Klammern angegeben. Im Beispiel-Token sind die Funktionen natürlich die in Abschnitt 2.3.8 »ERC-Tokenstandards« beschriebenen Funktionen für einen ERC20-Contract. Die in Abschnitt 2.3.5 erwähnten Events werden in Solidity mit dem Schlüsselwort »event« eingeleitet, gefolgt vom Event-Namen und in runden Klammern den Event-Parametern. Ausgelöst werden sie mit dem Schlüsselwort »emit« gefolgt vom jeweiligen Event. Die Events sind im Interface IERC20.sol definiert. Wie bei jeder Programmiersprache gibt es auch bei Solidity Konventionen nach denen programmiert wird. Nach Konvention werden Contract-Namen großgeschrieben und Zustandsvariablennamen, Modifikatoren-Namen sowie Funktionsnamen kleingeschrieben. Event-Namen werden wiederum großgeschrieben. Modifikatoren werden vor dem Konstruktor und den Funk-

tionen im Code geschrieben. Des Weiteren beginnen Zustandsvariablen mit einem Unterstrich und die Parameter des Konstruktors enden mit einem Unterstrich.

Strukturiert geschrieben und ohne Kommentare beträgt der Code für den Beispiel-Token 110 Zeilen. Hält man sich an die Konventionen für Einrückungen und Zeilenumbrüche, so ist der Code insgesamt gut übersichtlich. Abzüge gibt es für die Modifikatoren der Funktionen, da mehrere hintereinander gereiht zu Unübersichtlichkeit führen. Im Beispiel-Token besitzt bspw. eine simple Getter-Funktion wie *_name* bereits vier Modifikatoren. Geschweifte Klammern und Semikolon sind Geschmackssache, allerdings führen diese Konstrukte nicht selten zu schwer auffindbaren Fehlern und tragen nicht unbedingt zur Lesbarkeit bei. Insgesamt erhält das Kriterium Lesbarkeit des Solidity-Codes 3 Punkte.

Spracheigenschaften

Bei der Deklaration einer Variable stehen bei Solidity mehrere Datentypen zur Verfügung. Diese sind Integers, Booleans und Strings. Es gibt noch keine Gleitkommazahlen, allerdings sind Festkommazahlen verfügbar. Diese werden mit dem Schlüsselwort »fixedMxN« oder »ufixedMxN« (für unsigned) erstellt, wobei M und N für die Vor- und Nachkommastellen stehen. Integers werden mit »intx«, bzw. »uintx« gekennzeichnet. Das x steht dabei für eine optionale Größe in Bits (in 8er Schritten). Der Boolean wird mit »bool« und Strings werden mit »string« gekennzeichnet. Für die Typumwandlung gilt, sowohl die explizite als auch implizite Umwandlung ist bei Solidity möglich. Es gibt noch einen weiteren besonderen Datentyp, der für die Smart Contract Programmierung von hoher Relevanz ist. Nämlich ein Datentyp für Adressen. Dieser wird mit dem Schlüsselwort »address« kenntlich gemacht. Mit dem »address«-Datentyp wird eine Ethereum-Adresse abgespeichert.

Wie die meisten Programmiersprachen besitzt auch Solidity noch verschiedene Datentypen für das Abspeichern von mehreren Werten/Objekten. Nämlich Arrays, Structs, Enums und Mappings. Arrays werden dabei unterschieden in feste und dynamische Arrays. Arrays, Structs und Enums kommen im Beispiel-Token nicht vor. Mappings kommen in Zeile 13 und 14 vor. Booleans, Integers, Adressen, Festkommazahlen und Enums gehören bei Solidity zu den sog. Value Types. D.h. sie werden beim Verwenden durch eine Funktion einfach kopiert. Alle anderen hier erwähnten Datentypen sind bei Solidity Reference Types. Diese sind größer als 32 Byte, weswegen nur deren Referenz auf ihren Speicherort verwendet wird.

Solidity verfügt über alle gängigen Kontrollstrukturen der imperativen Programmierung. Diese werden mit den dafür üblichen Schlüsselwörtern »if« und »else« für bedingte Anweisungen und mit »while«, »do« und »for« für Schleifen und mit »break«, »continue« und »return« für Abbruch, Fortführen und Zurückgeben eingeleitet. Im Beispiel-Token kommt lediglich eine if-Bedingung in Zeile 69 vor und mehrere return-Anweisungen.

Wie in Abschnitt 2.3.7 bereits gezeigt, besitzt die EVM drei Speicherbereiche. Diese sind: Sto-

rage, Memory und Stack. Solidity bietet die Möglichkeit auf zwei dieser Bereiche explizit mit den Schlüsselwörtern »storage« (teuer in der Nutzung) und »memory« (günstig in der Nutzung) zuzugreifen. Im Storage wird der Zustand eines Contracts gespeichert, also unter anderem die Zustandsvariablen. Die Lebensdauer ist dabei auf die Laufzeit des Contracts beschränkt. Theoretisch können im Storage alle Datentypen gespeichert werden, allerdings wäre das äußerst ineffizient. Das Memory beinhaltet die temporären Variablen, deren Lebensdauer auf einen externen Funktionsaufruf beschränkt ist. Der Stack wird nur implizit für lokale Value Type Variablen verwendet und ist in der Nutzung quasi kostenlos. Darüber hinaus gibt es noch das Schlüsselwort »calldata« mit welchem auf den speziellen Speicherort der Funktionsargumente zugegriffen werden kann. Calldata verhält sich ähnlich zu Memory. Im Beispiel-Token ist bspw. in Zeile 23 vor den Konstruktor-Parametern *_name* und *_symbol* das Schlüsselwort »memory«. D.h. an den Konstruktor übergebene String-Argumente werden im Memory gespeichert. Für Reference Types muss der Speicherort immer explizit angegeben werden. Dieser sollte dabei immer Memory sein, da dieser deutlich günstiger in der Nutzung ist als Storage. Für Value Types muss keine explizite Angabe gemacht werden.

Solidity besitzt mehrere Sichtbarkeiten zur Datenkapselung. Diese sind »public«, »private«, »external« und »internal«. Im Beispiel-Token sind die Zustandsvariablen und Mappings mit »private« deklariert. Alle Funktionen bis auf *_transfer* und *_approve* (diese sind intern, d.h. nur der Eigene und erbende Contracts dürfen auf sie zugreifen) sind mit »public« deklariert. Die Funktionen im Interface IERC20.sol sind mit »external« deklariert (externe Funktionen können nur extern aufgerufen werden, bzw. im selben Contract nur mit dem Schlüsselwort »this« vorweg).

Neben den Sichtbarkeitsmodifikatoren gibt es in Solidity noch weitere Modifikatoren. Diese sind: Zustandsmodifikatoren und Event-Modifikatoren. Des Weiteren bietet Solidity die Möglichkeit eigene Funktionsmodifikatoren mittels des Schlüsselworts »modifier« zu erstellen. Bei den Zustandsmodifikatoren handelt es sich um »view« (nur lesender Zugriff auf Storage), »constant« (final-Deklaration für Zustandsvariablen) »pure« (weder lesender noch schreibender Zugriff auf Storage) und »payable« (einer Funktion, einem Konstruktor oder einer Adresse können Ether übermittelt werden). Im Beispiel-Token ist bspw. die Funktion *name()* in Zeile 32 mit »view« deklariert, da es sich bei dieser um eine Getter-Funktion handelt. D.h. sie soll nur den Namen des Tokens aus dem Storage auslesen und zurückgeben. Bei den Event-Modifikatoren handelt es sich um »anonymous« (Event nicht filterbar in den Logs) und »indexed« (Event erhält zusätzliches Topic in den Logs). Im Beispiel-Token kommen auch mehrfach die Schlüsselwörter »virtual« und »override« vor. Diese werden im Zusammenhang mit einer zu überschreibenden Funktion verwendet.

Bei Solidity werden wichtige Konzepte der Objektorientierten Programmierung unterstützt. D.h. Vererbung (darüber hinaus wird sogar die Mehrfachvererbung unterstützt), Polymorphie (auch das Überladen von Operatoren ist möglich) und, wie zuvor beschrieben, Datenkapselung sind

möglich. Mittels des Schlüsselworts »is« wird einem Contract kenntlich gemacht, dass er von einem anderen Contract erbt. Das Überladen von Funktionen ist bei Solidity äquivalent zur Programmiersprache Java, abgesehen natürlich von der Möglichkeit Operatoren zu überladen.

Zur Fehlerbehandlung gibt es bei Solidity die eingebauten Funktionen »assert« (Ausführung wird beendet bei nicht Erfüllung einer Bedingung), »require« (arbeitet auf die gleiche Weise wie »assert«) und »revert« (Ausführung des Contracts wird bei nicht Erfüllung einer Bedingung gestoppt und Zustandsänderungen zurückgenommen). Im Beispiel-Token kommt bspw. in Zeile 98 »require« vor. In diesem Fall, um sicherzustellen, dass es sich beim Sender nicht um die Nulladresse handelt. Bei Solidity können auch (nur) für externe Funktionsaufrufe und Contract-Erzeugungsaufrufe die Konstrukte »try« und »catch« zur Ausnahmebehandlung verwendet werden.

In Solidity kann auch mittels Inline-Assembly programmiert werden. Dies ermöglicht die Programmierung von Code auf niedriger Ebene der EVM im Solidity-Code. Die Programmiersprache die dafür verwendet wird heißt Yul (siehe Abschnitt 2.4.1). Um dem Solidity-Compiler kenntlich zu machen, dass es sich um Yul-Code handelt, wird das Schlüsselwort »assembly« verwendet. Im darauf folgenden Anweisungsblock kann dann der Yul-Code geschrieben werden.

Solidity verfügt außerdem noch über zahlreiche fest eingebaute Funktionalitäten für Adressen, Block- und Transaktionsinformationen, sowie über Funktionen für bspw. Keccak-256 oder SHA256. Insgesamt handelt es sich bei Solidity im Kontext der Smart Contract Programmierung um eine sehr vielseitige und komplexe Programmiersprache. Deshalb erhält Solidity für das Kriterium Funktionalitäten 5 Punkte.

Verfügbarkeit

Solidity existiert seit 2015 und hat seitdem eine hohe Verbreitung in der Smart Contract Programmierung erlangt. In zahlreichen Quellen und Büchern (die unter anderem für diese Arbeit verwendet wurden) wird Solidity sogar als die mit Abstand verbreitetste Smart Contract Sprache betitelt. Darüber hinaus kamen mittlerweile zahlreiche Verbesserungen und Fehlerbehebungen für Solidity. Die Sprache besitzt auch eine eigene Web-IDE namens Remix (`https://remix.ethereum.org/`). Ein besonderer Punkt ist außerdem die Tatsache, dass Solidity nicht nur auf der Blockchain-Plattform Ethereum verwendet werden kann, sondern auch auf weiteren Plattformen wie bspw. Tron. Darüber hinaus gibt es für alle gängigen Betriebssysteme Implementierungen des Solidity-Compilers und mehrere Online-Compiler (bspw. Remix oder Codedamn). Die Sprache ist somit aus der Smart Contract Entwicklung nicht mehr wegzudenken. Solidity erhält für dieses Kriterium deshalb 5 Punkte.

Lernaufwand

Auf Grund des zuvor vorgestellten hohen Umfangs an Funktionalitäten ist natürlich auch der Lernaufwand für Solidity erheblich. Alle Konzepte zu verstehen wird eine gewisse Zeit in An-

spruch nehmen. Allerdings gibt es für Solidity durch die hohe Verbreitung bereits sehr viele Tutorials (bspw. auf Youtube und Udemy). Die Ethereum Foundation hat auf ihrer Webseite sieben Bootcamps und zahlreiche Tutorials zu Solidity verlinkt (die Bootcamps sind allerdings nicht kostenlos). Des Weiteren gibt es schon zahlreiche vorgefertigte Code-Lösungen auf Programmierer-Foren. Die Tatsache das Ethereum eine eigene Web-IDE besitzt, gibt auch Plus-Punkte. Remix hilft beim Lernen von Solidity ungemein, da hier bspw. automatisch kompiliert wird, simpel Bytecode optimiert werden kann, das Deployen stark vereinfacht ist, viele Erklärungen und Beispiel-Skripte existieren, ein Debugger vorhanden ist, Code in Echtzeit deployed werden kann und noch vieles mehr. Eine perfekte Hilfestellung ist außerdem noch die Solidity-Dokumentation, welche zu allen Solidity-Versionen und deren Funktionalitäten Erklärungen bietet. Die Dokumentation zur jüngsten Solidity-Version umfasst insgesamt 360 Seiten. Das ist im Verhältnis zu den anderen Sprachen sehr ausführlich, was aber auch an dem deutlich größeren Umfang an Funktionalitäten liegt. Da Solidity zwar eine vergleichsweise komplexe Smart Contract Sprache ist, aber bereits viele Hilfestellungen besitzt (letztendlich kommt es auch auf die Motivation des Lernenden an) wird der Lernaufwand als moderat eingeschätzt. Deshalb erhält die Sprache für dieses Kriterium 3 Punkte.

Sicherheit

Auch auf den Punkt Sicherheit hat der Umfang an Funktionalitäten von Solidity eher einen schlechten Einfluss. Viele Sicherheitslücken entstehen durch eine fehlerhafte Programmierung. Die fehlerhafte Programmierung wiederum kann durch ein fehlendes Verständnis für die Funktionalitäten, bzw. durch die Überforderung mit diesen entstehen. Für Solidity gibt es zahlreiche Beispiele für gehackte Contracts durch eine fehlerhafte Programmierung. Das bekannteste Beispiel ist der DAO-Angriff. Bei diesem haben Angreifer 150 Million Dollar aus dem Contract für eine DAO (Decentralized Autonomous Organization) erbeuten können. Eine sehr bekannte Sicherheitslücke von Solidity, die unter anderem auch beim Beipiel-Token ausgenutzt werden könnte, ist der arithmetische Über-/Unterlauf. Dazu kommt es wenn eine Operation eine Zahl in einer Variablen speichern soll, aber der Wertebereich der Variablen nicht mit dem der Zahl übereinstimmt. Vereinfacht ausgedrückt, kann auf Grund dieses Problems ein Angreifer die Logik eines Contracts manipulieren. Die Lösung für dieses Problem ist die Verwendung einer SafeMath-Bibliothek. Wie in Abschnitt 4.1 beschrieben, sollte diese auch verwendet werden. Viele Sicherheitslücken von Solidity wurden bereits entdeckt und lassen sich durch Bibliotheken oder Best Practices einfach vermeiden. Allerdings wurden diese Sicherheitslücken nicht zwingend von guten Instanzen entdeckt. Durch Solidity-Contracts ist viel Geld verloren gegangen. Auf Grund dessen steht die Sprache auch in der Kritik. Des Weiteren ist die Tatsache, dass für den Schutz vor arithmetischen Fehlern eine extra Bibliothek benötigt wird eher abschreckend. Für Programmieranfänger kann auch das Einhalten von Best Practices, neben dem Erlernen der vielen Funktionalitäten, sehr überfordernd sein. Insgesamt erhält Solidity für das Kriterium Sicherheit nur 2 Punkte.

Effizienz

Für das Kompilieren des Solidity-Codes, wurde der Solidity-Compiler »solcjs« via Kommandozeile verwendet. Dieser konnte ganz simpel mit dem Package-Manger NPM installiert werden. Dafür wird der Befehl »npm install -g solc« eingegeben. Mittels des Befehls »solcjs –bin ERC20.sol« erhält man den EVM-Bytecode und mittels des Befehls »solcjs –abi ERC20.sol« das ABI zum Quellcode. Der nicht-optimierte Bytecode des Solidity-ERC20-Tokens hat für das Deployment 1038000 Gas benötigt. Die Gaskosten belaufen sich somit auf:

1038000 Gas * 2500000000 Wei ≈ 0.002595 ETH

Der Gasverbrauch und die daraus folgenden Kosten für den nicht-optimierten Solidity-Bytecode sind im Vergleich zu den nicht-optimierten Bytecodes der anderen Sprachen, mit Abstand am höchsten. Für das Deployment des optimierten Bytecodes des Solidity-ERC20-Tokens wurden nur 560790 Gas benötigt. Die Gaskosten belaufen sich hier also auf:

560790 Gas * 2500000000 Wei ≈ 0.00140197 ETH

Der Gasverbrauch und die Kosten des optimierten Bytecodes sind somit um fast die Hälfte reduziert und sind nahe der Effizienz von Vyper. Allerdings ist der optimierte Bytecode bei Solidity dennoch ineffizienter als der nicht-optimierte Vyper-Bytecode. Deshalb und auf Grund des schlechten Ergebnisses des nicht-optimierten Bytecodes erhält Solidity nur 3 Punkte. Listing 4.2 zeigt sämtliche Daten zum nicht-optimierten Solidity-ERC20-Token.

```
// Transaktionsdaten
accessList: [],
chainId: '0x3',
gas: 1038000,
gasPrice: '2500000000',
hash: '0x85b104ee1a81a287cc5b1cda164820fd0fccb5003e1024462d197f89d21a2cb5',
input: 'EVM-Bytecode',
maxFeePerGas: '2500000000',
maxPriorityFeePerGas: '2500000000',
nonce: 10,
r: '0x2cbaeb24071eefcfa5f8430a186a6ac3833c09311a4ceb038167027c57b0f436',
s: '0x604fe240cc136d96d3a8ea29415d4ac635610c7c409151d1a5883ac2986606fb',
to: null,
type: 2,
v: '0x0',
value: '0'

// Receipt zur Transaktion
blockHash: '0x5099de559f7c23b39f6924bef14aecdddac97e99fd9a358edc4a429ad87636ad',
blockNumber: 11993927,
contractAddress: '0x9622a96f941F0f76f70eacA43749C5A046f4cF73',
cumulativeGasUsed: 1167597,
effectiveGasPrice: '0x9502f900',
from: '0xdd7d08b014cd8e58c86ce71c5094c7aa20ba957c',
gasUsed: 1038000,
```

```
logs: [],
logsBloom: '0x0',
status: true,
to: null,
transactionHash: '0
    x85b104ee1a81a287cc5b1cda164820fd0fccb5003e1024462d197f89d21a2cb5',
transactionIndex: 2,
type: '0x2'

// Block zur Transaktion
baseFeePerGas: 10,
difficulty: '3025961700',
extraData: '0x726f707374656e303101024b7b',
gasLimit: 8000000,
gasUsed: 1501642,
hash: '0x5099de559f7c23b39f6924bef14aecdddac97e99fd9a358edc4a429ad87636ad',
logsBloom: '0x0',
miner: '0x43262A12d8610AA70C15DbaeAC321d51613c9071',
mixHash:
  '0xafa29d31f2f56b6ec81dcd0be7f282979d0fdb6659ce9861385759d2ae0cebfb',
nonce: '0x5bc898da4d7c1c26',
number: 11993927,
parentHash:
  '0xb6142aa5f4bfac99f80541c54ac78cb21ebfa67903005a94722ee1a4593e0094',
receiptsRoot:
  '0xeb26e60f7d5edd27b20c25d28afc04a920d8e30b66418c6dd93d233ba1094da3',
sha3Uncles:
  '0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347',
size: 8722,
stateRoot:
  '0x7d4f3bcd08042f37945d843f172a1d98a6953d38739d332cd7062e3826846247',
timestamp: 1645371186,
totalDifficulty: '39698768399580277',
transactions: [
                          ...
  '0x85b104ee1a81a287cc5b1cda164820fd0fccb5003e1024462d197f89d21a2cb5',
                          ...
],
transactionsRoot:
  '0x77d9d037a2a30a33f528af5c2e51936f41767553b874585795c4ba2f982e45b7',
uncles: []
```

Listing 4.2: Transaktionsdaten, Receipt und Blockinformationen zum Solidity-Token

4.3 Vyper

Der Code zum Vyper-ERC20-Token ist in Anhang B zu finden.

Code-Aufbau

Der Vyper-Code beginnt genauso wie der Solidity-Code mit einer Lizenz-Angabe. Auch hier wird die MIT-Lizenz verwendet. Als Nächstes folgt auch hier das Versionspragma. Im Beispiel-Token wird die Vyper-Version 0.3.1 verwendet. Dann folgen die Import-Anweisungen. Eine Besonderheit bei Vyper ist, dass es hier Built-In Interfaces gibt. D.h. die Sprache stellt wichtige Standards wie bspw. ERC20 von Haus aus als Import bereit. Im Fall des Beispiel-Token wird dies genutzt, um das ERC20-Interface zu importieren. Da die Events *Transfer* und *Approval* nicht im bereitgestellten ERC20-Interface implementiert sind, müssen diese extra implementiert werden. Nun folgen die für den Contract relevanten Zustandsvariablen. Danach werden die internen Funktionen *_transfer* und *_approve* implementiert. Bei Vyper müssen Funktionen, welche innerhalb anderer Funktionen im selben Contract aufgerufen werden, vor diesen im Code implementiert werden. Als Nächstes folgt auch hier der optionale Konstruktor. Anschließend werden die für den ERC20-Token relevanten Funktionen implementiert.

Die Syntax von Vyper ähnelt am ehesten der von Python. Wie auch bei Python werden Anweisungsblöcke durch entsprechende Einrückungen dem Compiler kenntlich gemacht. Es gibt keine geschweiften Klammern oder Semikolons hinter Anweisungen. Der Doppelpunkt spielt bei der Definition von Funktionen, des Konstruktors und von Variablen eine wichtige Rolle. Strings können, wie auch bei Solidity, sowohl in einfachen als auch doppelten Anführungsstrichen geschrieben werden. Kommentare beginnen bei Vyper Python-typisch mit der Raute »#«. Imports werden durch das Schlüsselwort »import« gekennzeichnet und müssen mit dem Schlüsselwört »as« benannt werden. Mit dem Schlüsselwort »from« kann aus einer Vyper-Bibliothek importiert werden. Implementiert wird mit dem Schlüsselwort »implements«. Für die Kontrollstrukturen verwendet Vyper die identische Syntax zu Python. Bei Vyper gibt es kein Schlüsselwort, welches den Contract einleitet. Bei der (Zustands)variablen-Deklaration wird zuerst der Variablenname angegeben, gefolgt vom gewünschten Datentyp. Der Konstruktor wird durch das Schlüsselwort »___init___« gekennzeichnet. Funktionen werden durch das Schlüsselwort »def« definiert. Nach dem Schlüsselwort folgt der Funktionsname und optional die Parameterliste. Des Weiteren kann optional noch ein (oder mehrere) Rückgabetyp angegeben werden. Dies geschieht mit dem »->« Symbol und rechts davon dem jeweiligen Datentyp, bzw. in runden Klammern den Datentypen. Modifikatoren werden beginnend mit dem »@« Zeichen eine Zeile vor der jeweiligen Funktion oder dem Konstruktor platziert. Falls eine Funktion mehrere Modifikatoren benötigt, dann werden diese in mehreren Zeilen übereinander geschrieben. Bei Vyper werden Events mit dem Schlüsselwort »event« eingeleitet, gefolgt vom Event-Namen und (nach dem Doppelpunkt) den Parametern. Ausgelöst wird das Event durch das Schlüsselwort »log« gefolgt vom jeweiligen Event. Auch hier werden nach Konvention Funktionsnamen und Zustandsvariablen kleingeschrieben und Event-Namen großgeschrieben. Des Weiteren wurde sich auch hier

an die Konvention gehalten, Zustandsvariablen mit einem Unterstrich beginnen zu lassen und Konstruktor-Parameter mit einem Unterstrich enden zu lassen.

Der Vyper-Code für den Beispiel-Token beträgt abzüglich der Kommentare 116 Zeilen. Theoretisch könnte man noch die Events abziehen, da das Interface auch manuell implementiert werden könnte und die Events sich somit in diesem befinden würden. Der Code würde sich dann auf 106 Zeilen belaufen. Insgesamt lässt sich der Vyper-Code sehr gut lesen, da Konstrukte wie Semikolons und geschweifte Klammern völlig wegfallen und die Modifikatoren eigene Zeilen im Code erhalten. Ein weiterer Plus-Punkt sind die Built-In Interfaces. Der Vyper-Code des Beispiel-Token erhält für das Kriterium Lesbarkeit 5 Punkte.

Spracheigenschaften

Bei Vyper stehen ebenfalls die Datentypen Integer, Boolean und String zur Verfügung. Darüber hinaus verfügt Vyper über den Datentyp Decimal für Festkommazahlen. Gleitkommazahlen gibt es hier ebenfalls nicht. Integers und Booleans werden identisch zu Solidity gekennzeichnet. Allerdings stehen bei den vorzeichenbehafteten Integers nur 128 Bit und 256 Bit zur Verfügung und bei den vorzeichenlosen nur 8 Bit und 256 Bit. Strings haben eine feste Größe und werden mit »String[x]« gekennzeichnet. Das x steht dabei für die Größe des Strings. Der Datentyp Decimal wird mit »decimal« gekennzeichnet. In Vyper kann ein Typ nur explizit umgewandelt werden. Dafür wird die Built-in Funktion »convert()« genutzt. Des Weiteren verfügt auch Vyper über den Datentyp »address« für Adressen. Bei diesen Datentypen (auch der String) handelt es sich bei Vyper um Value Types.

Für das Abspeichern mehrerer Werte verfügt Vyper über Arrays, Listen, Structs und Mappings. Bei den Listen handelt es sich um sog. Fixed-size Listen. D.h. sie können nur eine vorbestimmte Anzahl an Elementen enthalten. Bei den Arrays handelt es sich um dynamische Arrays. Listen, Arrays und Structs kommen nicht im Beispiel-Token vor. Mappings kommen in Zeile 20 und 21 im Beispiel-Token vor. Diese Datentypen sind Reference Types.

Vyper verfügt nicht über alle typischen Kontrollstrukturen der imperativen Programmierung. Die while-Schleife und die do-while-Schleife fehlen. Alle anderen Konstrukte sind vorhanden. Diese werden mit den Schlüsselwörtern »if«, »elif« und »else« für bedingte Anweisungen, mit dem Schlüsselwort »for« für Schleifen und mit den Schlüsselwörtern »break«, »continue« und »return« für Abbruch, fortführen und zurückgeben eingeleitet. Bei Vyper spielen auch die Schlüsselwörter »in« und »pass« ein wichtige Rolle. Ersteres wird bei for-Schleifen zum Iterieren verwendet und das Zweite dient dazu, um einer Funktion klar zu machen, dass sie keinen Funktionskörper besitzt. Im Beispiel-Token kommen nur return-Anweisungen vor.

Bei Vyper gibt es keine Schlüsselwörter um explizit auf die Speicherbereiche der EVM zuzugreifen. Die Speicherbereiche werden implizit vom Vyper-Compiler vergeben. Wie genau der Compiler dabei vorgeht, ist aus der Dokumentation nicht ersichtlich.

Vyper besitzt nur zwei Sichtbarkeitsmodifikatoren. Diese sind »internal« und »external«. Ersterer hat die gleiche Bedeutung wie »private« und das zweite Schlüsselwort hat die gleiche Bedeutung wie »public« bei Solidity. Darüber hinaus kann die Sichtbarkeit »public()« als Built-in Funktion verwendet werden. Sie kann nur bei Zustandsvariablen eingesetzt werden. Dadurch werden vom Compiler automatisch die dazugehörigen Getter-Funktionen erstellt. Das Gleiche gilt im Übrigen auch für öffentliche Zustandsvariablen bei Solidity.

Vyper besitzt neben den Sichtbarkeitsmodifikatoren nur noch die Zustandsmodifikatoren. Diese sind identisch mit denen von Solidity. Es ist nicht möglich eigene Modifikatoren zu erstellen. Event-Modifikatoren sind allerdings als Built-in Funktionen implementiert. Im Beipiel-Token bspw. in Zeile 10 die Funktion »indexed()«.

Für die Fehlerbehandlung gibt es bei Vyper die Schlüsselwörter »assert« (identisch zu assert bei Solidity) und »raise« (ähnlich zu revert bei Solidity). Im Beispiel-Token kommt bspw. in Zeile 32 »assert« vor. Des Weiteren gibt es auch bei Vyper die Schlüsselwörter »try« und (Python-typisch) »except«.

Auch Vyper verfügt darüber hinaus noch über zahlreiche Built-in Funktionen für verschiedene Anwendungsfälle. Im Gesamtergebnis schneidet Vyper bei den Funktionalitäten allerdings sehr schlecht ab. Es gibt keine implizite Typumwandlung, keine selbstdefinierten Modifikatoren, keine Vererbung, keine Polymorphie, kein Inline-Assembly, keine rekursiven Aufrufe von Funktionen und keine Möglichkeit explizit auf die Speicherbereiche der EVM zuzugreifen. Das sind nur die Dinge, welche während der Erstellung dieser Arbeit herausgefunden wurden. Deshalb erhält Vyper für das Kriterium Funktionalitäten nur einen Punkt.

Verfügbarkeit

Vyper existiert seit 2017 und hat seitdem ebenfalls eine hohe Verbreitung erlangt. Sie steht aber dennoch im Schatten von Solidity. In den meisten Fachbüchern und Tutorials die speziell die Smart Contract Programmierung erläutern, wird auf Solidity zurückgegriffen. Vyper wird wenn überhaupt nur beiläufig erwähnt. Außerdem besitzt Vyper leider keine eigene IDE. Derzeit ist es auch noch nicht möglich Remix für Vyper zu verwenden, da Remix nur einen Solidity-Compiler und Yul-Compiler besitzt. Interessanterweise steht in der Vyper-Dokumentation, dass dies möglich sein soll. Des Weiteren wird Vyper zurzeit (Stand: Februar 2022) noch von keiner Blockchain-Plattform neben Ethereum unterstützt. Allerdings existieren, wie auch bei Solidity, Implementierungen des Vyper-Compilers für alle gängigen Betriebssysteme, sowie Online-Compiler (bspw. Etherscan). Für dieses Kriterium erhält Vyper insgesamt 3 Punkte.

Lernaufwand

Da Vyper auf viele komplexe Funktionalitäten verzichtet, ist natürlich auch der Lernaufwand bedeutend geringer als bei Solidity. Des Weiteren gibt es auch für Vyper schon zahlreiche Tutorials, allerdings nicht in dem Ausmaß wie bei Solidity. Auf der Ethereum Webseite sind ebenfalls meh-

rere Bootcamps und Tutorials zu Vyper verlinkt. Durch die verhältnismäßig gute Verbreitung gibt es auch hier schon einige vorgefertigte Code-Lösungen auf den gängigen Foren. Des Weiteren ist die Vyper-Dokumentation gut strukturiert und verständlich, allerdings werden einige Funktionalitäten nur bedingt erläutert. Die Dokumentation zur jüngsten Vyper-Version umfasst 148 Seiten und ist damit für die Verhältnisse dieser Sprache recht umfangreich. Insgesamt ist Vyper eine sehr kompakte und leicht zu erlernende Smart Contract Sprache, was auch an ihrer Syntax liegt. Der Lernaufwand wird als sehr gering eingeschätzt. Deshalb erhält Vyper für dieses Kriterium 5 Punkte.

Sicherheit

Bei diesem Punkt glänzt Vyper besonders, da die Sprache speziell dafür entworfen wurde, sicherer in der Programmierung von Smart Contracts zu sein. Dies ist schon allein daran erkennbar, dass eben auf viele komplexe Funktionalitäten, die zu einer deutlich erhöhten Fehleranfälligkeit führen, verzichtet wurde. Des Weiteren hat Vyper bestimmte Sicherheitsmaßnahmen wie bspw. SafeMath zur Vermeidung von arithmetischen Über-/Unterlaufen von Haus aus implementiert. Dies und noch weitere Punkte machen Vyper zu einer sehr sicheren Smart Contract Sprache. Deshalb erhält die Sprache für das Kriterium Sicherheit 5 Punkte.

Effizienz

Für das Kompilieren des Vyper-Codes, wurde der Vyper-Compiler »vyper« ebenfalls via Kommandozeile verwendet. Auch dieser konnte sehr simpel mithilfe des Package-Manager PIP installiert werden. Um Vyper installieren zu können, wird Python 3.6 oder höher benötigt. Der Befehl »pip install vyper« installiert den Vyper-Compiler. Mittels der Befehle »vyper ERC20.vy« und »vyper -f abi ERC20.vy« erhält man zum einen den EVM-Bytecode und zum anderen die ABI des Quellcodes. Der nicht-optimierte Bytecode des Vyper-ERC20-Tokens hat für das Deployment 541040 Gas benötigt. Die Kosten belaufen sich auf:

541040 Gas * 2500000000 Wei ≈ 0.0013526 ETH

Somit war der Gasverbrauch und die Kosten bei der Ausführung dieses Smart Contracts um fast die Hälfte reduziert im Vergleich zum nicht-optimierten Bytecode des Solidity-Contracts. Für das Deployment des optimierten Bytecodes des Vyper-ERC20-Tokens wurden 525003 Gas benötigt. Die Gaskosten belaufen sich also auf:

525003 Gas * 2500000000 Wei ≈ 0.00131251 ETH

Vyper hat somit sowohl für den nicht-optimierten als auch für den optimierten Bytecode den geringsten Gasverbrauch und die daraus folgend geringsten Kosten benötigt. Vyper erhält deshalb für das Kriterium Effizienz 5 Punkte. Listing 4.3 zeigt sämtliche Daten zum nicht-optimierten Vyper-ERC20-Token.

```
1 // Transaktionsdaten
2 accessList: [],
3 chainId: '0x3',
```

```
gas: 541040,
gasPrice: '2500000000',
hash: '0x8dca7ca491c61be9cc403c0ddeb00b58252d4079f8cc26acb7566353c6d2a97b',
input: 'EVM-Bytecode'
maxFeePerGas: '2500000000',
maxPriorityFeePerGas: '2500000000',
nonce: 3,
r: '0xee620c3b2f65a36709e672d7fb3bdc8c9476463262d40c0a052008bfb9ab207b',
s: '0x6da723e367c30c061df05e2adbd662ac5c05ddb97a03225afc376fb7b23bb573',
to: null,
type: 2,
v: '0x0',
value: '0'

// Receipt zur Transaktion
blockHash: '0x84ccc92c69f0825fac75693a3ed0f36751a4eebca34be396111e50f36b416645',
blockNumber: 11965070,
contractAddress: '0x5CC54a31Bd249E49c38F37a55948a5c5ae1f5242',
cumulativeGasUsed: 7902051,
effectiveGasPrice: '0x9502f900',
from: '0xdd7d08b014cd8e58c86ce71c5094c7aa20ba957c',
gasUsed: 541040,
logs: [],
logsBloom: '0x0',
status: true,
to: null,
transactionHash: '0
    x8dca7ca491c61be9cc403c0ddeb00b58252d4079f8cc26acb7566353c6d2a97b',
transactionIndex: 7,
type: '0x2'

// Block zur Transaktion
baseFeePerGas: 13,
difficulty: '4988615112',
extraData: '0xd883010a0f846765746888676f312e31372e33856c696e7578',
gasLimit: 8000000,
gasUsed: 7989022,
hash: '0x84ccc92c69f0825fac75693a3ed0f36751a4eebca34be396111e50f36b416645',
logsBloom: '0x0',
miner: '0xd8731106Fd3637E4DB29bc3FA0fC3b3490cd1E92',
mixHash:
  '0xde0d24c9ec2c3e75c9beb49849f75d39607c76f279245cce42e1465b69d92977',
nonce: '0x50d0bf2a756d68bd',
number: 11965070,
parentHash:
  '0x78aa07abccb0cbbc1d2ee35e50549610a937c7a7cbb754186646671178638eb6',
receiptsRoot:
  '0x90e26c339f3347ad7e361b9f0ff575f5fe1a610a159754167c99c99f62dec2d0',
sha3Uncles:
  '0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347',
```

```
size: 37476,
stateRoot:
  '0x922ff8b8a34c960c2be8985087418a38efde29b4a10fb14c736110f14a7a2102',
timestamp: 1644839898,
totalDifficulty: '39583420253186396',
transactions: [
                                ...
  '0x8dca7ca491c61be9cc403c0ddeb00b58252d4079f8cc26acb7566353c6d2a97b',
                           ...
],
transactionsRoot:
  '0x23eef60fa67d6d18fa577576a22bb5dbd2940f28e3345c078171b50acf79b227',
uncles: []
```

Listing 4.3: Transaktionsdaten, Receipt und Blockinformationen zum Vyper-Token

4.4 Fe

Der Code zum Fe-ERC20-Token ist in Anhang C zu finden.

Code-Aufbau

Bei Fe ist zurzeit noch nicht genau spezifiziert, ob eine Lizenz angegeben werden muss. Dies kann sich mit dem Fortschreiten der Sprache aber noch ändern. Deshalb beginnt der Code hier mit dem Versionspragma. Im Beispiel-Token wird die Alpha-Version 0.13.0-alpha des Fe-Compilers angegeben. Import-Anweisungen sind nicht vorhanden, da hier kein Interface verwendet wurde. Genau genommen widerspricht dies der OpenZeppelin-Implementierungsvariante, allerdings scheint es zurzeit noch keine Interfaces bei Fe zu geben. Nun beginnt auch bei Fe der Contract. Innerhalb des Contracts werden zuerst die benötigten Events und dann die Zustandsvariablen deklariert. Danach folgt der optionale Konstruktor. Anschließend folgen die benötigten ERC20-Funktionen.

Die Syntax von Fe ähnelt am ehesten der Syntax von Rust und Python. Auch hier werden Anweisungsblöcke durch Einrückungen dem Compiler kenntlich gemacht und es gibt keine geschweiften Klammern oder Semikolons. Bei Fe hat der Doppelpunkt ebenfalls eine tragende Rolle bei der Definition von Funktionen, des Konstruktors und von Variablen. Strings werden in der Dokumentation nur mit doppelten Anführungszeichen markiert, vermutlich ist auch nur das zu Darstellung dieser erlaubt, da die Rust-Syntax für Strings ebenfalls nur doppelte Anführungszeichen erlaubt. Kommentare beginnen ebenfalls Python-typisch mit der Raute »#«. Imports scheinen noch nicht zu existieren, zumindest gibt es keine Beschreibung dazu in der Dokumentation. Für die Kontrollstrukturen verwendet Fe ebenfalls die Python-typische Syntax. Bei Fe wird ein Contract mit dem Schlüsselwort »contract«, gefolgt vom Contract-Namen eingeleitet. Eine Contract-Definition ist hier also ähnlich zu einer Klassendefinition in Python, bei welcher das Schlüsselwort »class« verwendet wird. Auch hier wird für die (Zustands)variablen-

Deklaration zuerst der Variablenname angegeben, gefolgt vom jeweiligen Datentyp. Ebenso wird der Konstruktor durch das Schlüsselwort »___init___« gekennzeichnet. Allerdings muss hier vor dem Konstruktor noch das Schlüsselwort »fn« stehen. Dieses wird auch verwendet um Funktionen zu definieren. Bei diesen folgt nach dem Schlüsselwort der Funktionsname und optional die Parameterliste. In der Parameterliste muss immer zuerst das Schlüsselwort »self« stehen. Das gilt auch für den Konstruktor. Auch hier kann noch ein (oder mehrere) Rückgabetyp angegeben werden. Dies geschieht ebenso wie bei Vyper mit der Zeichenfolge »->« und rechts davon dem Datentyp. Modifikatoren werden hier zu Beginn einer Funktionsdefinition geschrieben. Dies gilt auch für den Konstruktor. Die Definition eines Events ist identisch zu Vyper und das Auslösen ist identisch zu Solidity. Für Fe gelten die gleichen Konventionen wie bei Solidity und Vyper.

Der Fe-Code für den Beispiel-Token beträgt abzüglich der Kommentare 92 Zeilen. Allerdings muss hier bedacht werden, dass weder ein License-Identifier oder Imports angegeben wurden. Dennoch fällt der Code im Verhältnis zu den anderen Token kürzer aus. Hier gilt ähnliches wie bei Vyper. Eine sehr gute Übersichtlichkeit, da Semikolons und geschweifte Klammern nicht benötigt werden. Unschön ist, dass in jeder Parameterliste das Schlüsselwort »self« angegeben werden muss. Des Weiteren werden Schlüsselwörter Rust-typisch teils stark abgekürzt, wodurch gerade für Programmieranfänger die Bedeutung nicht mehr direkt ersichtlich sein kann. Die Lesbarkeit des Fe-Codes erhält deshalb insgesamt 4 Punkte.

Spracheigenschaften

Bei Fe gibt es Datentypen für Integers, Booleans, Adressen und Strings. Es gibt derzeit weder Gleitkommazahlen noch Festkommazahlen. Der Boolean und die Adresse wird identisch zu Solidity und Vyper gekennzeichnet. Integers werden mit »ix«, bzw. »ux« (unsigned) gekennzeichnet. Für das x können dabei in beiden Fällen die Werte 8, 16, 32, 64, 128 und 256 (in Bits) eingesetzt werden. Strings werden mit »String<x>«. Auch hier steht das x für eine feste Größe. Bei diesen Datentypen, ausgenommen des Strings, handelt es sich um Value Types.

Für das Abspeichern mehrerer Werte gibt es bei Fe Tuples, Arrays, Stucts, Enums und Maps. Bei den Arrays handelt es sich um Arrays mit fester Größe. Bis auf Maps kommt keiner dieser Datentypen im Beispiel-Token vor. Bei allen hier erwähnten Datentypen handelt es sich um Reference Types.

Fe verfügt über alle gängigen Kontrollstrukturen der imperativen Programmierung. Diese werden hier ebenfalls mit den typischen Schlüsselwörtern »if«, »elif« und »else« für bedingte Anweisungen, mit den Schlüsselwörtern »while« und »for« für Schleifen und mit den Schlüsselwörtern »break«, »continue« und »return« für Abbruch, fortführen und zurückgeben gekennzeichnet. Genauso wie bei Vyper spielen auch bei Fe die Schlüsselwörter »in« und »pass« eine wichtige Rolle. Die Verwendung ist identisch zu Vyper bzw. Rust und Python. Im Beispiel-Token kommen auch hier nur return-Anweisungen vor.

Bei Fe gibt es die Möglichkeit explizit auf das Memory zuzugreifen. Dies geschieht mit der Methode »to_mem()«. Der Storage und der Stack werden implizit vom Fe-Compiler vergeben. Im Memory können hier nur die Sequence Types, also Tuples, Arrays, Strings, Structs und Enums gespeichert werden. Das heißt Maps, und somit nicht alle Reference Types (wie bei Solidity), können nicht in Memory gespeichert werden. Im Beispiel-Token wird die Funktion bspw. in Zeile 35 verwendet.

Auch Fe besitzt nur zwei Sichtbarkeitsmodifikatoren. Diese sind »private« und »public« (nach Syntax abgekürzt zu »pub«). Ersterer ist die Standardsichtbarkeit und hat die gleiche Bedeutung wie »private« bei Solidity. Die zweite Sichtbarkeit hat ebenfalls die gleiche Bedeutung wie ihr Namensvetter bei Solidity. Für den Event-Modifikator Indexed gibt es das Schlüsselwort »idx«. Des Weiteren scheint es auch die Zustandsmodifikatoren »payable« und »nonpayable« zu geben. Zumindest sind diese in der Liste der Keywords der Dokumentation vorhanden.

Für die Fehlerbehandlung gibt es hier die Schlüsselwörter »assert« und »revert«. Beide werden identisch zu ihren Namensvettern bei Solidity verwendet. Ersteres kommt im Beispiel-Token bspw. in Zeile 63 vor. In der Dokumentation gibt es keinen Eintrag zu den Schlüsselwörtern »try« und »catch/except«. Deshalb wird davon ausgegangen, dass es diese Konstrukte derzeit noch nicht gibt.

Natürlich gibt es auch zu Fe noch zahlreiche Built-in Funktionen. Allerdings schneidet diese Sprache insgesamt im Punkt Funktionalitäten sehr schlecht ab. Aus der Dokumentation gehen viele Dinge leider auch nicht wirklich hervor, d.h. es ist nicht klar ob objektorientierte Mechanismen, Inline-Assembly, rekursive Aufrufe, implizite Typumwandlung oder selbstdefinierte Modifikatoren unterstützt werden. Deshalb wird davon ausgegangen, dass diese Techniken nicht existieren. Laut der Fe-Webseite strebt Fe aber danach, eine möglichst umfangreiche Standardbibliothek anzubieten. Außerdem gibt es Ansätze für explizite Speicherzugriffe. Deswegen und weil Fe noch in der frühen Entwicklungsphase ist und vermutlich noch um einige weitere Features ergänzt wird, erhält die Sprache für das Kriterium Funktionalitäten dennoch 2 Punkte.

Verfügbarkeit

Fe existiert erst seit Frühjahr 2021 und hat dementsprechend noch kaum eine große Verbreitung. Allerdings ist die Sprache unter anderem schon bei Ethereum verlinkt. Natürlich besitzt sie auch keine eigene IDE und kann auch nicht unter Remix verwendet werden. Des Weiteren wird auch Fe (Stand: Februar 2022) noch von keiner weiteren Blockchain neben Ethereum unterstützt. Darüber hinaus existiert noch kein Fe-Compiler unter Windows und kein Online-Compiler. Zurzeit gibt es nur unter Linux und MacOS Implementierungen eines Fe-Compilers. Positiv ist allerdings die Tatsache, dass der Fe-Compiler die gleiche Zwischensprache wie Solidity unterstützt, nämlich Yul. Davon ausgehend, dass die Sprache durch ihren offiziellen Release in den nächsten Jahren rasant an Verbreitung gewinnt, erhält sie für dieses Kriterium dennoch 2 Punkte.

Lernaufwand

Da bei Fe ebenfalls viele komplexe Funktionalitäten nicht vorhanden sind, ist der Lernaufwand natürlich, wie bei Vyper, deutlich geringer. Zu Fe gibt es allerdings bisher kaum Tutorials, Foren-Einträge oder Fachbücher. Die Dokumentation ist zwar gut strukturiert und verständlich, einige wichtige Informationen fehlen aber noch. Mit 131 Seiten ist sie dennoch schon sehr umfangreich, wenn man bedenkt das Fe gerade mal ein Jahr alt ist. Auf Grund der leicht verständlichen Syntax und der dennoch bereits gegebenen Dokumentation wird der Lernaufwand von Fe insgesamt als gering eingeschätzt. Deshalb erhält Fe für dieses Kriterium 4 Punkte.

Sicherheit

Derzeit lässt sich zu Fe zu diesem Punkt nicht viel herausfinden. Das einzig Ersichtliche ist, das die Sprache ebenfalls sehr kompakt ist und somit die Fehleranfälligkeit, auf Grund Fehler-provozierender Funktionalitäten, reduziert. Aus der Dokumentation geht leider nicht hervor, ob Fe eingebaute Sicherheitsmaßnahmen o.ä. besitzt. Deshalb erhält die Sprache hier 3 Punkte.

Effizienz

Das Kompilieren des Fe-Codes ist ein wenig komplizierter. Da es derzeit noch keinen Fe-Compiler unter Windows gibt, muss bei diesem via Oracle VirtualBox ein virtuelles Linux-System installiert werden. Die Linux-Distribution, die dafür verwendet wurde ist Kali-Linux. Auf diesem System wurde der Compiler fe_amd64 von der Fe-Webseite (`https://github.com/ethereum/fe/releases`) heruntergeladen. Um diesen verwenden zu können, muss die Datei zuerst in eine Executable-Datei umgewandelt werden. Dies geschieht unter Linux mit dem Befehl »chmod +x fe_amd64«. Nun kann der Fe-Compiler verwendet werden. Mittels des Befehls »./fe_amd64 ERC20.fe« erhält man sowohl den EVM-Bytecode als auch die ABI des Quellcodes in einem dafür angelegten Ordner Output. Bei der Erstellung des Fe-Codes muss penibel darauf geachtet werden, dass für die Anweisungsblöcke, Einrückungen und Umbrüche nur der Tabulator und Zeilenumbrüche verwendet werden. Kommen in der Fe-Datei zwischen diesen oder an Stelle dieser Leerzeichen vor, dann wird der Fe-Parser dies als Syntax-Fehler behandeln, da bspw. der Newline-Token erwartet wird. Der nicht-optimierte Bytecode des Fe-ERC20-Tokens hat für das Deployment insgesamt 807822 Gas benötigt. Die Kosten belaufen sich somit auf:

807822 Gas * 2500000000 Wei ≈ 0.00201955 ETH

Der Gasverbrauch und die Kosten liegen damit im unteren Mittelfeld der drei Sprachen bezüglich der nicht-optimierten Token. Der optimierte Bytecode des Fe-ERC20-Tokens hat 788030 Gas für das Deployment benötigt. Die Kosten belaufen sich in diesem Fall also auf:

788030 Gas * 2500000000 Wei ≈ 0.00197007 ETH

Somit sind der Gasverbrauch und die Kosten des optimierten Fe-Bytecodes mit Abstand am höchsten. Deshalb erhält Fe insgesamt für das Kriterium Effizienz nur 2 Punkte. Listing 4.4 zeigt sämtliche Daten zum nicht-optimierten Fe-ERC20-Token.

```
// Transaktionsdaten
accessList: [],
chainId: '0x3',
gas: 807822,
gasPrice: '2500000000',
hash: '0x706b40085ba44f2a1799292d10564331e552dc60bd9680c8ef3b033660aa4327',
input: 'EVM-Bytecode'
maxFeePerGas: '2500000000',
maxPriorityFeePerGas: '2500000000',
nonce: 6,
r: '0x57abaa2c7996a3102b7dc59283f75ce7ebe7223d095f20ac7d929cb9b6daf0f',
s: '0x1cbb3c6bd81ca7627306287ed2aee56c994308dcf45d7f9b761d9b0500f84c36',
to: null,
type: 2,
v: '0x1',
value: '0'

// Receipt zur Transaktion
blockHash: '0x853ef9881c36edc3deb603a6c3ef852ceb1854aa2404515c626850e6135a1337',
blockNumber: 11983976,
contractAddress: '0xC094D1a1c5757Cc2E8e07aB5C13FFD8A22cAd1a8',
cumulativeGasUsed: 807822,
effectiveGasPrice: '0x9502f900',
from: '0xdd7d08b014cd8e58c86ce71c5094c7aa20ba957c',
gasUsed: 807822,
logs: [],
logsBloom: '0x0',
status: true,
to: null,
transactionHash: '0
    x706b40085ba44f2a1799292d10564331e552dc60bd9680c8ef3b033660aa4327',
transactionIndex: 0,
type: '0x2'

// Block zur Transaktion
baseFeePerGas: 2379,
difficulty: '3664512775',
extraData: '0xd883010a0f846765746888676f312e31372e35856c696e7578',
gasLimit: 8000000,
gasUsed: 7359002,
hash: '0x853ef9881c36edc3deb603a6c3ef852ceb1854aa2404515c626850e6135a1337',
logsBloom: '0x0',
miner: '0xfbb61B8b98a59FbC4bD79C23212AddbEFaEB289f',
mixHash:
  '0xd8c1b75cc3c6512027048aaffb5817817fd941ff89f843f5749cf0efc4fd3124',
nonce: '0x1fc961c35245ea63',
number: 11983976,
parentHash:
  '0x79ea74a1d27f3022947e96b3c2317661ac9cfb80ac96a753ecd0e31753a628b1',
receiptsRoot:
```

```
  '0x6a77a2224ce3bcf53c7db1956e08af7d9293fbe3ec205e6255c1c047c6add4d4',
sha3Uncles:
  '0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347',
size: 33792,
stateRoot:
  '0x9bf6a49e2fdd7e2adfffb5c0f9d94a1b968e97c910cf2daaa37add032210e408',
timestamp: 1645185006,
totalDifficulty: '39661418811677707',
transactions: [
  '0x706b40085ba44f2a1799292d10564331e552dc60bd9680c8ef3b033660aa4327',
                            ...
],
transactionsRoot:
  '0x29534519e12c7117488a217bf441491d674099ac020854039636c369d336bb66',
uncles: []
```

Listing 4.4: Transaktionsdaten, Receipt und Blockinformationen zum Fe-Token

4.5 Diskussion der Analyseergebnisse

	Solidity	**Vyper**	**Fe**
Lesbarkeit (15%)	3	5	4
Funktionalitäten (20%)	5	1	2
Verfügbarkeit (10%)	5	3	2
Lernaufwand (10%)	3	5	4
Sicherheit (25%)	2	5	3
Effizienz (20%)	3	5	2
Bewertung	**3,35**	**4**	**2,75**

Tabelle 4.1: Endergebnisse der Analyse

Im Endergebnis schneidet Vyper mit einer Gesamtpunktzahl von 4 Punkten am besten ab. Auf dem zweiten Platz ist Solidity mit einer Gesamtpunktzahl von 3.35 Punkten. Auf dem letzten Platz ist Fe mit einer Gesamtpunktzahl von 2.75 Punkten. Die Ergebnisse zeigen klar, dass wenn man eine sichere und effiziente Smart Contract Sprache erlernen will, Vyper die beste Wahl ist. Auch beim Lernaufwand und der Lesbarkeit glänzt Vyper durch Dinge wie eine gute Syntax. Allerdings muss die Sprache stark an Funktionalitäten einbüßen, um Kriterien wie ein gute Sicherheit zu erreichen. Will man auf viele Funktionalitäten nicht verzichten und möchte darüber hinaus noch eine große Community im Rücken haben, dann ist Solidity die beste Wahl. Allerdings muss man hier bereit sein, einen signifikant höheren Lernaufwand zu betreiben. Nicht zuletzt deswegen, um die mangelnde Sicherheit durch bspw. Best Practices auszugleichen.

Warum Solidity und Fe für das Kriterium Effizienz so schlecht abschneiden, ist nicht ganz ersichtlich. Vermutlich liegt der hohe Gasverbrauch beim nicht-optimierten Solidity-Contract

daran, dass bei diesem eine Interface-Datei zusätzlich zum ERC20-Contract kompiliert werden musste. Bei Vyper ist dies zwar auch der Fall, allerdings als Built-in Interface. Wahrscheinlich sorgt dieses für die deutlich bessere Effizienz. Unklar ist warum Vyper hier dennoch deutlich besser abgeschnitten hat als Fe. Bei Fe wurde schließlich in keiner Form ein Interface verwendet. Eine Vermutung ist, dass dies mit den impliziten Speicherzugriffen der jeweiligen Compiler zu tun. Eventuell sind diese beim Fe-Compiler noch nicht ganz ausgereift. Dies könnte auch der Grund für die großen Unterschiede des Gasverbrauchs der optimierten Contracts sein. Dieser ist bei Fe wesentlicher höher als bei Solidity und Vyper, welche wiederum schon ausgereiftere Compiler besitzen. Des Weiteren wird auch die Länge des Bytecodes eine gewisse Rolle spielen, da es sich dabei um Nutzdaten handelt, welche natürlich ebenfalls bezahlt werden müssen. Der letzte Punkt allein erklärt aber nicht die überragenden Ergebnisse nach dem Optimierungsprozess des Solidity-Bytecodes. Mit dem optimierten EVM-Bytecode ist der Gasverbrauch bei allen Sprachen signifikant geringer. Bei Solidity muss auch noch ergänzt werden, dass es möglich ist mit Inline-Assembly zu programmieren, was wiederum zu einer besseren Effizienz führen würde.

Fe war in einigen Kriterien schwer zu bewerten. Was sich jetzt schon klar sagen lässt ist, dass die Syntax verständlich und der Lernaufwand vergleichsweise gering ist. Alle weiteren Kriterien werden sich vermutlich im Laufe der Entwicklung von Fe noch zu sehr verändern, um deren Bewertung als final anzusehen. Dennoch wird die Sprache besonders für Vyper in der Zukunft vermutlich ein ernstzunehmender Konkurrent, da sie ebenfalls auf eine hohe Sicherheit und leichte Erlernbarkeit bei der Contract Erstellung abzielt. An dieser Stelle muss noch einmal erwähnt werden, dass die gesamte Bewertung zum Teil sehr subjektiv ist. Besonders Kriterien wie Lesbarkeit und Lernaufwand sind Ansichtssache, weshalb deren Bewertung unter anderen Augen gänzlich anders ausfallen kann. Des Weiteren hat auch die Gewichtung der Kriterien einen erheblichen Einfluss auf die Gesamtbewertung. Erfahrene Smart Contract Entwickler könnten hier evtl. eine andere Verteilung anstreben.

5 Fazit und Ausblick

Die Ergebnisse dieser Arbeit zeigen, dass für den Einstieg in die Smart Contract Programmierung die Programmiersprache Vyper am ehesten zu empfehlen ist. Die Sprache ist leicht zu erlernen, hat eine gute Überprüfbarkeit durch eine simple Syntax, hat integrierte Sicherheitskonzepte und ist im Vergleich zu den anderen Sprachen sehr effizient. All dies sind Eigenschaften die besonders für Anfänger in der (Smart Contract) Programmierung sehr hilfreich sind. Was ebenfalls aus der Analyse hervorgeht ist, dass die Frage welche Sprache für die allgemeine Entwicklung am besten geeignet ist, nicht so leicht beantwortet werden kann. Die Auswahl hängt hier unter anderem von den Präferenzen des jeweiligen Entwicklers ab. Das heißt möchte man eine Sprache mit vielen Funktionalitäten oder lieber eine Sprache die von Haus aus eine hohe Sicherheit bietet verwenden. Beides lässt sich, wie gezeigt, schwer kombinieren, da die gute Sicherheit vor allem durch den Verzicht auf Funktionalitäten einhergeht. Generell ist es demnach schwierig eine Sprache zu finden, die alle Kriterien perfekt erfüllt. Um dennoch eine Antwort auf die Frage zu geben, ist als beste Alternative, um die Kombination aus beidem zu erreichen, Solidity zu empfehlen. Der Umfang an Funktionalitäten ist im Verhältnis zu den anderen Sprachen enorm. Fehlende Sicherheitsaspekte können wiederum durch Best Practices und vorgefertigte geprüfte Code-Lösungen ausgeglichen werden. Außerdem können Bibliotheken implementiert werden, welche bestimmte Sicherheitslücken schließen. Die Sprache wird allerdings nur für fortgeschrittenere Smart Contract Programmierer empfohlen. Kritisch ist die Analyse zu Fe zu betrachten, da die Sprache noch sehr jung ist, kann sich die Bewertung einiger Kriterien noch ändern.

Die vorgestellten Ergebnisse werfen auch weiterführende Fragen auf. Zum einen ist unklar wie es zu den signifikanten Unterschieden in der Ausführungseffizienz kommt. In diesem Zusammenhang wäre es sinnvoll, den Bytecode der jeweiligen Sprache sowie die Compiler genauer zu untersuchen. Darüber hinaus wäre es sinnvoll, die Optimierungsprozesse der jeweiligen Compiler zu analysieren. Außerdem betrachtet diese Arbeit nur die Programmiersprachen Solidity, Vyper und Fe. Welche Vorteile beispielsweise die Programmiersprache Yul+ bietet wird nicht beantwortet. Generell ist die Beschäftigung mit weiteren Smart Contract Sprachen durchaus zu empfehlen, da jede für sich entscheidende Vorteile bieten könnte. Ethereum ist in einem ständigen Prozess der Entwicklung. Neuentwicklungen wie EWasm könnten die Frage nach der richtigen Smart Contract Sprache völlig neu aufwerfen. Deshalb ist es sinnvoll sich im Bezug auf die Smart Contract Programmierung auch mit dieser Technologie genauer auseinanderzusetzen.

Literaturverzeichnis

Adlerjohn. (2021, 27. Dezember). Yul+ GitHub. Zuletzt abgerufen am 06.01.2022 von: https://github.com/fuellabs/yulp

Antonopoulos, M. und Wood, G. (2019). Ethereum Grundlagen und Programmierung. Heidelberg, Deutschland. O'Reilly.

Benjaminion. (2017, 11. August). LLL (Low-Level-Lisp). Zuletzt abgerufen am 16.12.2021 von: https://github.com/benjaminion/LLL_erc20

Bhat, A. (2021, 17. August). Ethereum World State. Zuletzt abgerufen am 05.01.2022 von: https://medium.com/@bhatasif17/ethereum-world-state-c38007dd8138

Bitpanda. (2021). Was versteht man unter dem Begriff "Mining Difficulty"? Zuletzt abgerufen am 13.12.2021 von: https://www.bitpanda.com/academy/de/lektionen/was-versteht-man-unter-dem-begriff-mining-difficulty/

BlockchainCenter.net. (2018, 06. Juni). Block Reward. Zuletzt abgerufen am 14.01.2022 von: https://www.blockchaincenter.net/wiki/block-reward/

Blocktrainer. (2021). Was ist ein "Mempool"? Zuletzt abgerufen am 14.12.2021 von: https://www.blocktrainer.de/blocktrainer-1x1/was-ist-ein-mempool/

Bosch. (2021). Blockchain einfach erklärt. Zuletzt abgerufen am 07.12.2021 von: https://www.bosch.com/de/stories/blockchain-einfach-erklaert/

BTC Acadamy. (2022). Block Reward. Zuletzt abgerufen am 14.01.2022 von: https://www.btc-echo.de/academy/bibliothek/block-reward/

Computerality. (2021, 13. September). Ethereum VM (EVM) Opcodes und Instruction Reference. Zuletzt abgerufen am 14.01.2022 von: https://github.com/crytic/evm-opcodes

Computerbild. (2021, 12. März). Ethereum: Gas-Preis einstellen. Zuletzt abgerufen am 13.12.2021 von: https://www.computerbild.de/artikel/cb-Tipps-Finanzen-ethereum-gas-preis-29920563.html

Davidbrai. (2021, 29. Oktober). ERC721-Tokenstandard GitHub. Zuletzt abgerufen am 17.12. 2021 von: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-721.md

Ethereum. (2021, 09. März). ERC-20 Contract Walk-Through. Zuletzt abgerufen am 08.02.2022 von: https://ethereum.org/de/developers/tutorials/erc20-annotated-code/

Ethereum. (2021, 10. Dezember). Ethereum Accounts. Zuletzt abgerufen am 10.12.2021 von: https://ethereum.org/en/developers/docs/accounts/

Ethereum. (2021). Ethereum Improvement Proposals. Zuletzt abgerufen am 14.12.2021 von: https://eips.ethereum.org/#:~:text=EIPs,the%20Ethereum%20Project%20Management%20repository.

Ethereum Homestead Documentation. (2016). Contracts - LLL. Zuletzt abgerufen am 04.01.2022 von: https://www.ethdocs.org/en/latest/contracts-and-transactions/contracts.html#id4

Ethereum Languages. (2021, 22. Dezember). Smart Contract Languages. Zuletzt abgerufen am 04.01.2022 von: https://ethereum.org/en/developers/docs/smart-contracts/languages/

Ethereum Wackerow. (2021, 22. Dezember). Ethereum Virtual Machine (EVM). Zuletzt abgerufen am 07.01.2022 von: https://medium.com/chainsafe-systems/ethereum-2-0-a-complete-guide-ewasm-394cac756baf

Ethereum Wiki. (2021). Serpent. Zuletzt abgerufen am 04.01.2022 von: https://eth.wiki/archive/serpent

Fe Documentation. (2021). The Fe Guide. Zuletzt abgerufen am 08.02.2022 von: https://fe-lang.org/docs/spec/visibility_and_privacy.html

Fertig, T. und Schütz, A. (2019). Blockchain für Entwickler. Bonn, Deutschland. Rheinwerk Verlag.

Fontaine, S. (2019, 29. September). Understanding Bytecode on Ethereum. Zuletzt abgerufen am 11.02.2022 von: https://medium.com/authereum/bytecode-and-init-code-and-runtime-code-oh-my-7bcd89065904

Frankenfield, J. (2021, 27. Mai). Wei. Zuletzt abgerufen am 14.01.2022 von: https://www.investopedia.com/terms/w/wei.asp

Huillet, M. (2019, 07. März). Smart-Contract-Sprache: Ethereum Foundation finanziert Forschung an der Columbia und Yale. Zuletzt abgerufen am 25.11.2021 von: https://de.cointelegraph.com/news/ethereum-foundation-funds-columbia-yale-researchers-work-on-smart-contract-language

Kabakci, I. (2021, 03. Juli). Was ist der Unteschied zwischen einem Token und einer Kryptowährung? Zuletzt abgerufen am 16.12.2021 von: https://www.futurezone.de/digital-life/article232677981/was-ist-der-unterschied-zwischen-einem-token-und-einer-kryptowaehrung.html

Kerem, G. (2021, 28. Mai). Was ist Hashing und digitale Signatur in der Blockchain? Zuletzt abgerufen am 30.11.2021 von: https://de.techbriefly.com/was-ist-hashing-und-digitale-signatur-in-der-blockchain-tech-32383/

Lightclient. (2021, 15. August). ERC20-Tokenstandard GitHub. Zuletzt abgerufen am 16.12.2021 von: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md

LLL Compiler Documentation. (2017). LLL Introduction. Zuletzt abgerufen am 04.01.2022 von: https://lll-docs.readthedocs.io/en/latest/lll_introduction.html

Mitschele, A. (2016, November). Blockchain Definition: Was ist "Blockchain"? Zuletzt abgerufen am 30.11.2021 von: https://wirtschaftslexikon.gabler.de/definition/blockchain-54161

MyCrypto. (2021, 07. Mai). New Transaction Types on Ethereum. Zuletzt abgerufen am 14.12.2021 von: https://blog.mycrypto.com/new-transaction-types-on-ethereum

Obscuren. (2015, 07. Juli). Mutan GitHub. Zuletzt abgerufen am 04.01.2022 von: https://github.com/obscuren/mutan

OpenZeppelin Docs. (2021). ERC20. Zuletzt abgerufen am 08.02.2022 von: https://docs.openzeppelin.com/contracts/2.x/api/token/erc20#ERC20

Pirapira. (2018, 2. November). Bamboo GitHub. Zuletzt abgerufen am 06.01.2022 von: https://github.com/pirapira/bamboo

QA Stack. (2021). Unterschied zwischen API und ABI. Zuletzt abgerufen am 14.01.2022 von: https://qastack.com.de/programming/3784389/difference-between-api-and-abi

Reddit Real-Goat. (2018). Bamboo is a programming language for Ethereum. Zuletzt abgerufen am 06.01.2022 von: https://www.reddit.com/r/ethereum/comments/9ufttu/bamboo_is_a_programming_language_for_ethereum/

Roos, A. (2018, 29. Dezember). Was ist ein Block in der Blockchain? Zuletzt abgerufen am 09.12.2021 von: https://www.btc-echo.de/news/was-ist-ein-block-in-der-blockchain-65579/

Sbillig. (2022, 13. Februar). Fe OpenZeppelin-ERC20 GitHub. Zuletzt abgerufen am 28.01.2022 von: https://github.com/ethereum/fe/blob/master/crates/test-files/fixtures/demos/erc20_token.fe

Schiller, K. (2018, 30. April). Genisis Block | Anfang einer Kryptowährung. Zuletzt abgerufen am 14.12.2021 von: https://blockchainwelt.de/genesis-block-bitcoin-ethereum-blockchain-kryptowaehrung/

Schiller, K. (2018, 24. November). Was sind Smart Contracts? Definition und Erklärung. Zuletzt abgerufen am 07.01.2022 von: https://blockchainwelt.de/smart-contracts-vertrag-blockchain/

Schinko, C. (2021, 24. Februar). So reich wären Sie heute, wenn Sie 2009 in Bitcoin investiert hätten. Zuletzt abgerufen am 02.12.2021 von: https://www.cancom.info/2021/02/so-reich-waeren-sie-heute-wenn-sie-2009-in-bitcoin-investiert-haetten/

Schmitz, P. (2019, 30. August). Was ist ein Token? Zuletzt abgerufen am 16.12.2021 von: https://www.blockchain-insider.de/was-ist-ein-token-a-854928/

Schwarz, C. (2019, 31. August). Ethereum 2.0: A Complete Guide. Ewasm. Zuletzt abgerufen am 07.01.2022 von: https://medium.com/chainsafe-systems/ethereum-2-0-a-complete-guide-ewasm-394cac756baf

Skellet0r. (2021, 22. Oktober). Vyper OpenZeppelin-ERC20 GitHub. Zuletzt abgerufen am 24.01.2022 von: https://github.com/vyperlang/vyper/blob/master/examples/tokens/ERC20.vy

Solidity Documentation. (2021). Solidity. Zuletzt abgerufen am 07.02.2022 von: https://docs.soliditylang.org/en/latest/

Studyflix. (2018, 29. November). Nutzwertanalyse: Erklärung und Beispiel. Zuletzt abgerufen am 01.12.2021 von: https://studyflix.de/wirtschaft/nutzwertanalyse-315

Takenobu, T. (2018, 03. März). Ethereum EVM illustrated. Zuletzt abgerufen am 03.02.2022 von: https://takenobu-hs.github.io/downloads/ethereum_evm_illustrated.pdf

Vyper Documentation. (2020). Vyper. Zuletzt abgerufen am 07.02.2022 von: https://vyper.readthedocs.io/en/stable/

V0xat. (2022, 13. Februar). Solidity OpenZeppelin-ERC20 GitHub. Zuletzt abgerufen am 20.01.2022 von: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol

Westfälische Hochschule. (2021). Wie ist eine Blockchain aufgebaut? Zuletzt abgerufen am 08.12.2021 von: https://www.internet-sicherheit.de/forschung/connectemscherlippe/blockchain/wie-ist-die-blockchain-aufgebaut.html

Wikipedia. (2021, 02. Dezember). Binärschnittstelle. Zuletzt abgerufen am 10.02.2022 von: https://de.wikipedia.org/wiki/Bin%C3%A4rschnittstelle

Wikipedia. (2021, 10. November). Blockchain. Zuletzt abgerufen am 02.12.2021 von: https://de.wikipedia.org/wiki/Blockchain

Wikipedia. (2019, 07. Januar). Opcode. Zuletzt abgerufen am 12.02.2022 von: https://de.wikipedia.org/wiki/Opcode

Wood, G. (2022, 13. Februar). Ethereum Yellow Paper. Zuletzt abgerufen am 30.01.2022 von: https://ethereum.github.io/yellowpaper/paper.pdf

Yul Documentation. (2021). Yul. Zuletzt abgerufen am 06.01.2022 von: https://docs.soliditylang.org/en/latest/yul.html

Ziv, O. (2019, 31. Dezember). What is Ropsten ETH and how can I get some? Zuletzt abgerufen am 13.12.2021 von: https://www.2key.network/blog-posts/what-is-ropsten-eth-and-how-can-i-get-some

Anhang A (Solidity-Code)

```
// SPDX-License-Identifier: MIT

// Versionspragma
pragma solidity >=0.7.0 <0.9.0;

// Imports
import "./IERC20.sol";

// Contract Beginn
contract ERC20 is IERC20 {

    // Mappings
    mapping(address => uint256) private _balances;
    mapping(address => mapping(address => uint256)) private _allowances;

    // Zustandsvariablen
    uint256 private _totalSupply;
    uint8 private _decimals;
    string private _name;
    string private _symbol;

    // Konstruktor
    constructor(string memory name_, string memory symbol_, uint256 total_,
    uint8 decimals_) {
        _balances[msg.sender] = total_;
        _name = name_;
        _symbol = symbol_;
        _totalSupply = total_;
        _decimals = decimals_;
    }

    // Funktionen
    function name() public view virtual override returns (string memory) {
        return _name;
    }

    function symbol() public view virtual override returns (string memory) {
        return _symbol;
    }

    function decimals() public view virtual override returns (uint8) {
        return _decimals;
```

```
42      }
43
44      function totalSupply() public view virtual override returns (uint256) {
45          return _totalSupply;
46      }
47
48      function balanceOf(address account) public view virtual override returns (
        uint256) {
49          return _balances[account];
50      }
51
52      function transfer(address recipient, uint256 amount) public virtual override
         returns (bool) {
53          _transfer(msg.sender, recipient, amount);
54
55          return true;
56      }
57
58      function allowance(address owner, address spender) public view virtual
        override returns (uint256) {
59          return _allowances[owner][spender];
60      }
61
62      function approve(address spender, uint256 amount) public virtual override
        returns (bool) {
63          _approve(msg.sender, spender, amount);
64
65          return true;
66      }
67
68      function transferFrom(address sender, address recipient, uint256 amount)
        public virtual override returns (bool) {
69          if (_allowances[sender][msg.sender] != type(uint256).max) {
70              require(_allowances[sender][msg.sender] >= amount);
71              unchecked {
72                  _approve(sender, msg.sender, _allowances[sender][msg.sender] -
        amount);
73              }
74          }
75
76          _transfer(sender, recipient, amount);
77
78          return true;
79      }
80
81      // OpenZeppelin Funktionen
82      function increaseAllowance(address spender, uint256 addedValue) public
        virtual returns (bool) {
83          _approve(msg.sender, spender, _allowances[msg.sender][spender] +
        addedValue);
```

```
84
85          return true;
86      }
87
88      function decreaseAllowance(address spender, uint256 subtractedValue) public
    virtual returns (bool) {
89          require(_allowances[msg.sender][spender] >= subtractedValue);
90          unchecked {
91              _approve(msg.sender, spender, _allowances[msg.sender][spender] -
    subtractedValue);
92          }
93
94          return true;
95      }
96
97      function _transfer(address sender, address recipient, uint256 amount)
    internal virtual {
98          require(sender != address(0));
99          require(recipient != address(0));
100
101         require(_balances[sender] >= amount);
102         unchecked {
103             _balances[sender] -= amount;
104         }
105         _balances[recipient] += amount;
106
107         emit Transfer(sender, recipient, amount);
108     }
109
110     function _approve(address owner, address spender, uint256 amount) internal
    virtual {
111         require(owner != address(0));
112         require(spender != address(0));
113
114         _allowances[owner][spender] = amount;
115
116         emit Approval(owner, spender, amount);
117     }
118 }
```

Anhang B (Vyper-Code)

```
# License: MIT

# @version ^0.3.1

from vyper.interfaces import ERC20

implements: ERC20

event Transfer:
    _from: indexed(address)
    _to: indexed(address)
    _value: uint256

event Approval:
    _owner: indexed(address)
    _spender: indexed(address)
    _value: uint256

# Mappings
_balances: HashMap[address, uint256]
_allowances: HashMap[address, HashMap[address, uint256]]

# Zustandsvariablen
_totalSupply: uint256
_decimals: uint8
_name: String[64]
_symbol: String[32]

# Interne OpenZeppelin Funktionen
@internal
def _transfer(_sender: address, _recipient: address, _amount: uint256):
    assert _sender != ZERO_ADDRESS
    assert _recipient != ZERO_ADDRESS

    assert self._balances[_sender] >= _amount
    self._balances[_sender] -= _amount
    self._balances[_recipient] += _amount

    log Transfer(_sender, _recipient, _amount)

@internal
def _approve(_owner: address, _spender: address, _amount: uint256):
```

```
43      assert _owner != ZERO_ADDRESS
44      assert _spender != ZERO_ADDRESS
45
46      self._allowances[_owner][_spender] = _amount
47
48      log Approval(_owner, _spender, _amount)
49
50  # Konstruktor
51  @external
52  def __init__(name_: String[64], symbol_: String[32], total_: uint256, decimals_:
        uint8):
53      self._balances[msg.sender] = total_
54      self._name = name_
55      self._symbol = symbol_
56      self._totalSupply = total_
57      self._decimals = decimals_
58
59  # Funktionen
60  @external
61  @view
62  def name() -> String[64]:
63      return self._name
64
65  @external
66  @view
67  def symbol() -> String[32]:
68      return self._symbol
69
70  @external
71  @view
72  def decimals() -> uint8:
73      return self._decimals
74
75  @external
76  @view
77  def totalSupply() -> uint256:
78      return self._totalSupply
79
80  @external
81  @view
82  def balanceOf(_account: address) -> uint256:
83      return self._balances[_account]
84
85  @external
86  def transfer(_recipient: address, _amount: uint256) -> bool:
87      self._transfer(msg.sender, _recipient, _amount)
88
89      return True
90
91  @external
```

```
92  @view
93  def allowance(_owner: address, _spender: address) -> uint256:
94      return self._allowances[_owner][_spender]
95
96  @external
97  def approve(_spender: address, _amount: uint256) -> bool:
98      self._approve(msg.sender, _spender, _amount)
99
100     return True
101
102 @external
103 def transferFrom(_sender: address, _recipient: address, _amount: uint256) ->
        bool:
104     assert self._allowances[_sender][msg.sender] >= _amount
105     self._approve(_sender, msg.sender, self._allowances[_sender][msg.sender] -
        _amount)
106     self._transfer(_sender, _recipient, _amount)
107
108     return True
109
110 # OpenZeppelin Funktionen
111 @external
112 def increaseAllowance(_spender: address, _addedValue: uint256) -> bool:
113     self._approve(msg.sender, _spender, self._allowances[msg.sender][_spender] +
         _addedValue)
114
115     return True
116
117 @external
118 def decreaseAllowance(_spender: address, _subtractedValue: uint256) -> bool:
119     assert self._allowances[msg.sender][_spender] >= _subtractedValue
120     self._approve(msg.sender, _spender, self._allowances[msg.sender][_spender] -
         _subtractedValue)
121
122     return True
```

Anhang C (Fe-Code)

```
pragma ^0.13.0-alpha

contract ERC20:

    event Transfer:
        idx _from: address
        idx _to: address
        _value: u256

    event Approval:
        idx _owner: address
        idx _spender: address
        _value: u256

    # Mappings
    _balances: Map<address, u256>
    _allowances: Map<address, Map<address, u256>>

    # Zustandsvariablen
    _totalSupply: u256
    _decimals: u8
    _name: String<100>
    _symbol: String<100>

    # Konstruktor
    pub fn __init__(self, name_: String<100>, symbol_: String<100>, total_: u256
    , decimals_: u8):
        self._balances[msg.sender] = total_
        self._name = name_
        self._symbol = symbol_
        self._totalSupply = total_
        self._decimals = decimals_

    # Funktionen
    pub fn name(self) -> String<100>:
        return self._name.to_mem()

    pub fn symbol(self) -> String<100>:
        return self._symbol.to_mem()

    pub fn decimals(self) -> u8:
        return self._decimals
```

```
42
43      pub fn totalSupply(self) -> u256:
44          return self._totalSupply
45
46      pub fn balanceOf(self, account: address) -> u256:
47          return self._balances[account]
48
49      pub fn transfer(self, recipient: address, amount: u256) -> bool:
50          self._transfer(msg.sender, recipient, amount)
51
52          return true
53
54      pub fn allowance(self, owner: address, spender: address) -> u256:
55          return self._allowances[owner][spender]
56
57      pub fn approve(self, spender: address, amount: u256) -> bool:
58          self._approve(msg.sender, spender, amount)
59
60          return true
61
62      pub fn transferFrom(self, sender: address, recipient: address, amount: u256)
     -> bool:
63          assert self._allowances[sender][msg.sender] >= amount
64          self._transfer(sender, recipient, amount)
65          self._approve(sender, msg.sender, self._allowances[sender][msg.sender] -
     amount)
66
67          return true
68
69      # OpenZeppelin Funktionen
70      pub fn increaseAllowance(self, spender: address, addedValue: u256) -> bool:
71          self._approve(msg.sender, spender, self._allowances[msg.sender][spender]
     + addedValue)
72
73          return true
74
75      pub fn decreaseAllowance(self, spender: address, subtractedValue: u256) ->
    bool:
76          assert self._allowances[msg.sender][spender] >= subtractedValue
77          self._approve(msg.sender, spender, self._allowances[msg.sender][spender]
     - subtractedValue)
78
79          return true
80
81      fn _transfer(self, sender: address, recipient: address, amount: u256):
82          assert sender != address(0)
83          assert recipient != address(0)
84
85          assert self._balances[sender] >= amount
86          self._balances[sender] -= amount
```

```
87          self._balances[recipient] += amount
88
89          emit Transfer(sender, recipient, amount)
90
91      fn _approve(self, owner: address, spender: address, amount: u256):
92          assert owner != address(0)
93          assert spender != address(0)
94
95          self._allowances[owner][spender] = amount
96
97          emit Approval(owner, spender, amount)
```