HOCHSCHULE
MITTWEIDA
UNIVERSITY OF
APPLIED SCIENCES

BACHELORARBEIT

Frau
Laura Moritz

Programmiersprachen für Smart Contracts und ihre Eignung zur Entwicklung sicherer und kostengünstiger Blockchainanwendungen

Mittweida, 2018

Fakultät Angewandte Computer- und Biowissenschaften

BACHELORARBEIT

Programmiersprachen für Smart Contracts und ihre Eignung zur Entwicklung sicherer und kostengünstiger Blockchainanwendungen

Autor:

Frau Laura Moritz

Studiengang:
Angewandte Informatik, IT-Sicherheit

Seminargruppe:
IF15wI-B

Erstprüfer:
Prof. Dr. – Ing. Andreas Ittner

Zweitprüfer:
M. Sc. Steffen Kux (Slock.it)

Einreichung:
Mittweida, 04.10.2018

Verteidigung/Bewertung:
Mittweida, 2018

Faculty Applied Computer Sciences &
Biosciences

BACHELOR THESIS

Smart Contract Programming Languages and Their Suitability For Secure and Cost-Saving Development of Blockchain Applications

author:

Ms. Laura Moritz

course of studies:
Applied Computer Sciences, IT-Security

seminar group:
IF15wI-B

first examiner:
Prof. Dr. – Ing. Andreas Ittner

second examiner:
M. Sc. Steffen Kux (Slock.it)

submission:
Mittweida, 04.10.2018

defence/ evaluation:
Mittweida, 2018

Bibliografische Beschreibung:

Moritz, Laura:
Programmiersprachen für Smart Contracts und ihre Eignung zur Entwicklung sicherer und kostengünstiger Blockchainanwendungen. - 2018. - VI, 89, XVIII S.
Mittweida, Hochschule Mittweida, Fakultät Angewandte Computer- & Biowissenschaften, Bachelorarbeit, 2018

Referat:

Diese Arbeit beschäftigt sich mit Programmiersprachen für die Smart Contract - Entwicklung auf der Ethereum Blockchain. Diese sollen vorgestellt sowie anhand eines ERC20 Contracts verglichen und bewertet werden. Zusätzlich soll ihre Sicherheit im Hinblick auf häufige Gefahren der Contract - Entwicklung untersucht und sprachspezifische Unsicherheiten gesammelt werden. Ziel ist es, festzustellen, welche der Sprachen die geeignetste für die Entwicklung sicherer und kostengünstiger Smart Contracts ist, wobei auch Bytecodegrößen und Deploymentkosten einbezogen werden.

Diese Bachelorarbeit wurde betreut von der Slock.it GmbH.



Slock.it

Inhalt

Inhalt ..... **I**

Abbildungsverzeichnis ..... **III**

Tabellenverzeichnis ..... **V**

Abkürzungsverzeichnis ..... **VI**

1 Übersicht **1**

1.1 Einleitung *1*

1.2 Kapitelübersicht *1*

2 Grundlagen **3**

2.1 Ethereum *3*

2.2 Die Ethereum Virtual Machine *9*

2.3 Smart Contracts *10*

2.4 Smart Contract Programmiersprachen *11*

2.4.1 Solidity 11

2.4.2 Vyper 16

2.4.3 LLL 21

2.4.4 Bamboo 25

2.4.5 eWASM 28

3 Vergleich anhand eines ERC20 Contracts **30**

3.1 Der ERC20 Tokenstandard *30*

3.2 ERC20 in Solidity *32*

3.3 ERC20 in Vyper *36*

3.4 ERC20 in LLL *39*

3.5 ERC20 in Bamboo *45*

3.6 Deployment: Codegrößen und Kosten im Vergleich *48*

3.7 Zusammenfassung und Einschätzung der Unterschiede *56*

4 Sicherheitsaspekte und häufige Gefahren **63**

4.1 Typische Gefahren in der Smart Contract Entwicklung 63

4.2 Sicherheitsmechanismen und Gefahren der Sprachen 67

4.3 Vor- und Nachteile des Einsatzes der EVM als Laufzeitumgebung 80

4.4 Zusammenfassung und Einschätzung 83

5 Fazit und Ausblick 87

6 Literatur 90

Anlagen 102

Anlagen, Kapitel 2 I

Anlagen, Kapitel 3 V

Selbstständigkeitserklärung XIX

Abbildungsverzeichnis

Abbildung 1: Aufbauprinzip der Blockchain [86] 4

Abbildung 2: State-Übergang in Ethereum [5] 5

Abbildung 3: Kategorien der Anwendungsfälle [86] 7

Abbildung 4: Beispielcontract Solidity 13

Abbildung 5: Solidity Funktionsaufbau [vgl. 23] 15

Abbildung 6: Vyper Contract Aufbau 18

Abbildung 7: Benutzerdefinierte Einheiten Vyper 19

Abbildung 8: Vyper Funktionen 20

Abbildung 9: Aufbau LLL Contract [40] 23

Abbildung 10: LLL Makro 24

Abbildung 11: Aufbau Bamboo Contract 26

Abbildung 12: ERC20 Interface [51] 31

Abbildung 13: Remix DIE 49

Abbildung 14: Ganache 50

Abbildung 15: ABI 50

Abbildung 16: Ganache Ausgabe 51

Abbildung 17: Gasschätzung Remix 51

Abbildung 18: Transferdetails Solidity 51

Abbildung 19: Vyper Onlinecompiler 53

Abbildung 20: Ablauf Deployment Vyper 54

Abbildung 21: Gaskosten Deployment 61

Abbildung 22: Gaskosten Transfer 61

Abbildung 23: Transaktionskosten in Euro 62

Abbildung 24: Addition 2er 32 Bit Integer [79] 81

Abbildung 25: Addition 2er 256 Bit Integer [79] 81

Abbildung 26: Einordnung der Sprachen 88

Tabellenverzeichnis

Tabelle 1: Spracheigenschaften im Vergleich 59

Tabelle 2: Sicherheitsaspekte der Sprachen 85

Abkürzungsverzeichnis

ABI	Application Binary Interface
CA	Contract Account
DoS	Denial of Service
EOA	Externally Owned Account
ENS	Ethereum Name Service
ERC	Ethereum Request for Comments
EVM	Ethereum Virtual Machine
eWASM	Ethereum flavoured Web Assembly
ICO	Initial Coin Offering
OWASP	Open Web Application Security Project
WASM	Web Assembly

1 Übersicht

In diesem Kapitel soll zunächst die Thematik und Zielsetzung der Arbeit erläutert und zusätzlich ein Überblick über die Kapitel gegeben werden.

1.1 Einleitung

Ethereum ist ein dezentrales System, welches wie Bitcoin über eine eigene Kryptowährung verfügt und auf einer Blockchain basiert. Diese noch sehr junge Technologie unterliegt einer ständigen Entwicklung. Ethereum besitzt, nach Bitcoin, die zweitgrößte Marktkapitalisierung [vgl. 1], verfügt jedoch neben einer Währung über die Möglichkeit mithilfe sogenannter „Smart Contracts“ digitale Verträge und Anwendungen auf der Blockchain abzubilden, was in diesem Netzwerk eine zentrale Rolle spielt.

Als Programmiersprache für die Entwicklung dieser Anwendungen hat sich „Solidity“ durchgesetzt. Sie ist sehr gut dokumentiert und wird von den meisten Entwicklern bevorzugt. Jedoch gibt es zunehmend kritische Stimmen, die besonders die Fehleranfälligkeit durch den großen Funktionsumfang hervorheben. Aus diesem Grund sollen in der vorliegenden Arbeit weniger bekannte und neue Sprachen der Smart Contract Entwicklung auf ihre Eignung untersucht und mit Solidity verglichen werden. Dabei soll ein ERC20 Token Smart Contract als Vergleichsbasis dienen. Neben Lese- und Programmierfreundlichkeit werden Fehleranfälligkeit, inhärente Sicherheitsmechanismen bezüglich häufiger Gefahren sowie der Funktionsumfang als auch Bytecodegrößen und Deploymentkosten betrachtet.

Ziel ist es, zu untersuchen, ob die Sprachen eine Alternative zu Solidity darstellen, beziehungsweise besser zur Implementation sicherer und kostengünstiger Blockchainanwendungen geeignet sind.

1.2 Kapitelübersicht

Diese Arbeit besteht aus fünf Kapiteln, deren Aufbau im Folgenden beschrieben wird.

Nach diesem einleitenden Kapitel werden in Kapitel zwei die Grundlagen, auf denen die Arbeit basiert, erläutert. Dazu gehören Ethereum, die Ethereum Virtual Machine, Smart Contracts sowie die Programmiersprachen Solidity, Vyper, LLL und Bamboo. Neben ihrem Aufbau werden sprachliche Besonderheiten und Hintergründe beschrieben. Es folgt ein Ausblick auf den neuen Ansatz „eWASM“.

Im Anschluss erfolgt in Kapitel drei eine vergleichende Auseinandersetzung mit den Sprachen anhand eines ERC20 Contracts, wobei zunächst der ERC20 Standard erläutert und anschließend die Contracts in den jeweiligen Sprachen betrachtet werden. Weiterhin gilt es, die Bytecode-Größen nach dem Kompilieren zu vergleichen und die Kosten für das Hochladen auf die Blockchain zu bestimmen. Abschließend folgt eine Zusammenfassung.

Im vierten Kapitel werden typische Gefahren in der Smart Contract Entwicklung erläutert und die Sprachen auf vorhandene Sicherheitsmechanismen bezüglich dieser untersucht sowie sprachspezifische Unsicherheiten aufgezeigt. Danach werden die Vor- und Nachteile der Ethereum Virtual Machine als Laufzeitumgebung für diese Anwendungen abgewogen. Zum Schluss erfolgt eine Zusammenfassung.

Das fünfte Kapitel soll die Erkenntnisse zusammenfassen und einen Ausblick auf die weitere Entwicklung von Ethereum liefern.

2 Grundlagen

Dieses Kapitel befasst sich mit den Grundlagen, auf denen die Arbeit basiert. Zunächst soll die Ethereum Plattform beschrieben werden. Danach folgt ein Abschnitt zur Ethereum Virtual Machine und zu Smart Contracts. Schließlich werden die zu betrachtenden Programmiersprachen sowie eWASM vorgestellt.

2.1 Ethereum

Die Ethereum Plattform ist ein verteiltes, dezentrales System auf Basis einer Blockchain. Sie besteht seit 2015 und ist somit noch eine sehr junge Technologie. Die formale Spezifikation wurde 2014 von Gavin Wood veröffentlicht. Neben ihm wurde Ethereum von Vitalik Buterin und Jeffrey Wilcke begründet. [vgl. 1]

Der Vorreiter Bitcoin kann, im Gegensatz zu Ethereum, nur Token transferieren und einfache Skripte ausführen. Deshalb begannen 2014 die Ethereum-Gründer auf der Grundlage von Bitcoin die Arbeit an der Ethereum-Plattform, die zusätzlich zu den Tokentransfers in der Krypotwährung „Ether“ die Ausführung von Programmen ermöglicht. [vgl. 2] Als Laufzeitumgebung dient dabei die Ethereum Virtual Machine, die in Abschnitt 2.2 noch genauer beschrieben werden soll.

Als Kryptowährung bezeichnet man hierbei ein digitales Zahlungsmittel, welches durch kryptographische Verfahren erstellt und mithilfe der Blockchain transferiert wird, was in hohem Maße zu ihrer Sicherheit beiträgt. [vgl. 3]

Der Begriff Blockchain wurde erstmalig 2008 im Zuge von Bitcoin durch Satoshi Nakamoto beschrieben. Sie bezeichnet ein verteiltes System beziehungsweise Hauptbuch, in dem alle Teilnehmer (Knoten) dieselben Transaktionen ausführen und aufzeichnen. Alle Daten auf der Blockchain werden dabei dauerhaft und unveränderbar in die Historie aufgenommen. Die Transaktionen auf der Blockchain beinhalten Zahlungen oder Nachrichten zwischen den Teilnehmern. Sie werden von bestimmten Knoten in Blöcke zusammengefasst und diese aneinandergereiht. Daher der Name Blockkette. Um diese Blöcke zu erzeugen ist auf der Ethereum-Blockchain das Lösen eines mathematischen Problems, des „Proof of Work“, notwendig. Ist dies geschehen, wird der neue Block von den Netzwerkteilnehmern validiert. In Ethereum entsteht so in etwa alle 15 Sekunden ein neuer Block, wohingegen es bei Bitcoin zehn Minuten dauert. Verbunden werden die Blöcke über Hashzeiger zu einer Kette, indem ein neuer Block stets auf seinen Vorgänger verweist. Der Hashzeiger entsteht aus der Eingabe des Kopfes des vorherigen Blocks in eine Hashfunktion. Es kann nur ein Block auf einmal angehangen werden. Dadurch entsteht ein Wettbewerb um die Erzeugung des nächsten Blocks. Jeder Teilnehmerknoten kann unabhängig vonei-

nander die neuen Blöcke validieren, die durch einen komplexen Algorithmus erzeugt werden. Durch die Komplexität der Erzeugung, aber leichten Validierung, entsteht Konsens im Netzwerk. Das System wird durch starke Kryptographie abgesichert, um Authentizität und Integrität sicherzustellen. [vgl. 3, 4]

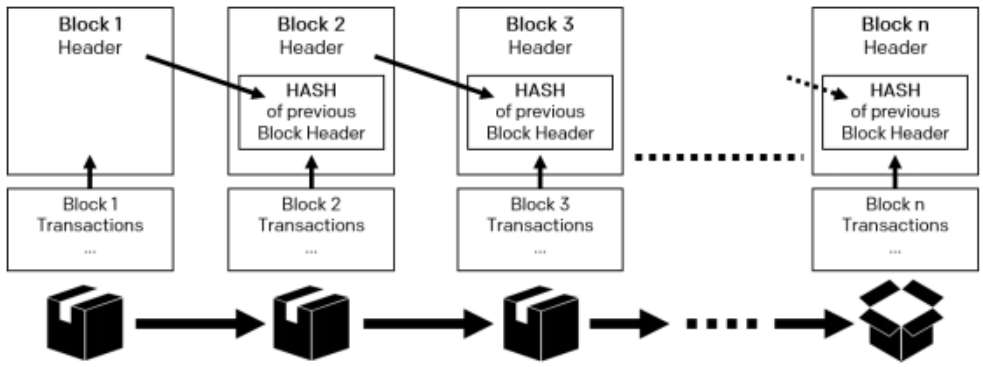


Abbildung 1: Aufbauprinzip der Blockchain [86]

Design Prinzipien

Ethereums Design basiert auf fünf Prinzipien:

1. Zum einen soll das Protokoll so einfach wie möglich sein und das Hinzufügen von Komplexität soll möglichst vermieden werden.
2. Zweitens soll das Protokoll universell sein, sodass es jedem möglich ist, Smart Contracts zu erstellen, die tun, was mathematisch definierbar ist.
3. Drittens ist gewünscht, dass die Teile des Protokolls modular und separierbar sind, sodass Änderungen sich nicht auf alle Teile auswirken.
4. Das vierte Prinzip ist Flexibilität. Eine Anpassbarkeit des Protokolls zwecks Sicherheitsverbesserungen soll möglich sein.
5. Schließlich soll es, als fünftes Prinzip, keiner Zensur unterliegen. Anwendungen zu regulieren ist nicht erwünscht. [vgl. 5]

Ethereum Accounts

In Ethereum gibt es nicht wie in Bitcoin nur Adressen als Referenz, vielmehr spricht man hier von Accounts. Man unterscheidet zwei Arten: „Externally Owned Accounts“ (EOA) und „Contract Accounts“ (CA).

EOAs werden durch ein Schlüsselpaar aus privatem und öffentlichem Schlüssel kontrolliert. Dabei wird der öffentliche vom privaten Schlüssel abgeleitet und von diesem die 20 Byte lange Adresse des Accounts. Der private Schlüssel dient, wie auch bei Bitcoin, dem Signieren von Transaktionen. Ein solcher Account besteht neben der Adresse aus einer Nonce, die dessen Transaktionen zählt und einem Kontostand, der die Menge an Ether im Besitz beschreibt. Diese Attribute definieren den Zustand (State) des Accounts. Betrachtet

man alle Ethereum Accounts zusammen, so spricht man dann vom „World State“. [vgl. 5, 6]

CAs besitzen zusätzlich zu den genannten Attributen einen Schlüssel-Wert-Speicher, der 256-Bit Wörter auf 256 Bit Speicherslots (Storage) abbildet. Außerdem verfügen sie über Code, der sie kontrolliert. Dieser Code stellt die Smart Contracts dar. Die Adresse eines Contract Accounts wird bei dessen Erzeugung aus der Adresse des Erzeugers und seiner Nonce abgeleitet, da ein CA immer durch eine Transaktion erzeugt wird. [vgl. 7]

Jede Transaktion, wie beispielsweise ein Funktionsaufruf eines CA, der zur Ausführung des Smart Contracts führt oder eine Übersendung von Ether verändert die Attribute und somit den State der betreffenden Accounts und den World State. Da jeder Netzwerkknoten diese Transaktionen im Zuge der Validierung unabhängig voneinander ausführt, verfügt jeder Knoten dadurch über den aktuellen Zustand. Die Ethereum Blockchain enthält somit eine Historie der Zustände, wohingegen Bitcoin eine Historie an Transaktionsausgängen (Unspent Transaction Outputs) aufweist. Die folgende Abbildung zeigt beispielhaft einen Zustandsübergang.

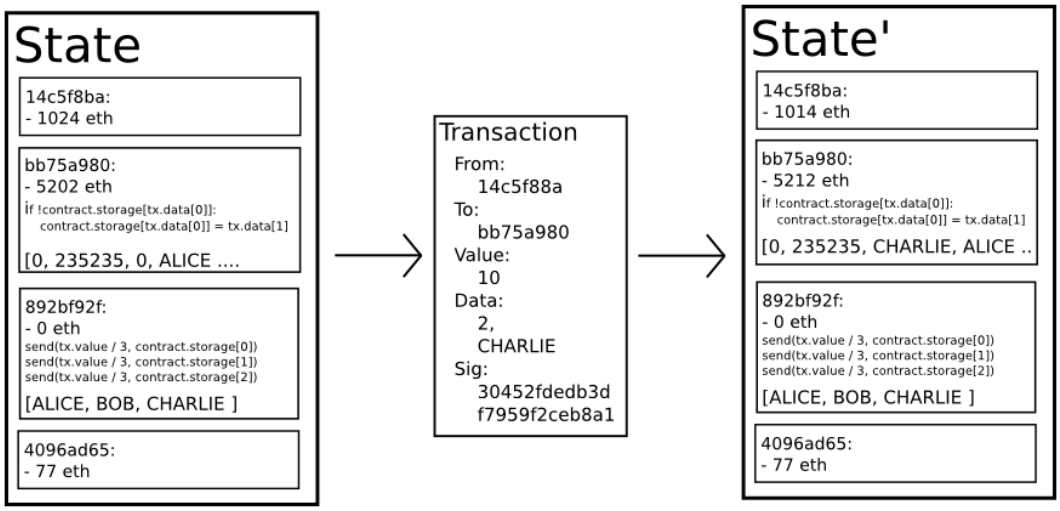


Abbildung 2: State-Übergang in Ethereum [5]

Die States der Abbildung beschreiben jeweils den World State der die Zustände der verschiedenen Accounts umfasst. Eine Transaktion des oberen EOA an den CA darunter führt zu einem neuen Zustand. Die Transaktion enthält dabei Absender und Zieladresse, den gesendeten Wert, ein Datenfeld mit Informationen für den Contract und die Signatur des Absenders. Die Transaktion an einen CA löst dabei immer die Ausführung des Codes aus, wobei das Senden dieser ebenfalls von einem anderen CA möglich ist.

Mining

Die Transaktionen im Netzwerk werden von den sogenannten „Minern“ in Blöcke zusammengefasst. Dafür müssen sie ein mathematisches Problem lösen, was nur durch massenhaftes Probieren möglich ist. Dies bezeichnet man als „Proof-of-Work“. In Ethereum heißt dieser Algorithmus „Ethash“. Wichtig dabei ist, dass das Problem schwer zu lösen, jedoch leicht zu validieren ist, damit die Knoten neu zusammengestellte Blöcke einfach prüfen können. Wird ein neuer Block erzeugt und von der Mehrheit des Netzwerkes als Nachfolger des letzten anerkannt, erhält der Miner dafür eine Belohnung von derzeit fünf Ether. Bei Bitcoin liegt sie bei 12,5 Bitcoin. Neben der Blockbelohnung erhalten sie die Transaktionsgebühren, die für jede Transaktion gezahlt werden müssen. Damit wird der Miner für die investierte Rechenleistung sowie den Speicher entlohnt, welche für die Validierung und Verbreitung benötigt werden. [vgl. 4]

Anders als bei Bitcoin kann der Miner in seinem Block sogenannte „Uncles“ referenzieren. Dies sind gültige Blöcke, die parallel zum vorhergehenden Block entstanden sind, es aber nicht geschafft haben, die Mehrheit der Stimmen im Netzwerk zu erreichen. Dafür erhalten sowohl der Miner des aktuellen Blocks als auch die der Uncles eine Belohnung. Dies wird als GHOST-Protokoll bezeichnet. [vgl. 4]

Ether

In Ethereum gibt es die Kryptowährung Ether. Sie verfügt über einige weitere Einheiten, wobei die kleinste „Wei“ ist. So entsprechen 10^{18} Wei einem Ether. Im Gegensatz zu Bitcoin, wo es ein Tokenlimit von 21 Millionen gibt, existiert ein solches bei Ethereum nicht. Im Gegensatz zum deflationären Bitcoin, ist Ether eine inflationäre Währung. So wird algorithmisch jedes Jahr eine feste Menge Ether erzeugt. Dabei sinkt die Inflation stetig und geht gegen Null, ohne diese jedoch zu erreichen. Der prozentuale Anteil neuer Ether verringert sich in Bezug auf die bereits vorhandene Menge somit jährlich. [vgl. 87]

Für jede Transaktion muss eine kleine Gebühr gezahlt werden, dabei gilt, je mehr man zahlt, desto wahrscheinlicher wird die Transaktion in den nächsten Block aufgenommen. Die Einheit, die dabei eine wesentliche Rolle spielt, heißt „Gas“. Jede Rechenoperation benötigt eine festgelegte Menge an Gas, sodass sich für Transaktionen die notwendige Gesamtmenge kalkulieren lässt. Man kann als Absender selbst bestimmen, wie viel Wei pro Gas man bezahlen möchte:

$$Transaktionskosten = \frac{Wei}{Gas} * Gaseinheiten$$

Das Gas hat zudem die Funktion, Endlosschleifen in den Smart Contracts zu verhindern. Dazu folgendes Beispiel: wird ein Contract ausgeführt, so verbraucht jede Operation eine bestimmte Menge Gas. Diese wird vom vorhandenen Betrag abgezogen. Kommt es nun

durch einen Fehler zur Entstehung einer Dauerschleife, wird, nach Aufbrauchen des mitgelieferten Gas, die entsprechende Transaktion abgebrochen und alle Änderungen umgekehrt. Ebenso gibt es für jeden Block ein Gaslimit, das derzeit bei etwa acht Millionen Gas liegt. [vgl. 8, 9]

Zur Zeit des Schreibens werden im Mittel in etwa 14 GWei pro Gas gezahlt, das wären 0.000000014 Ether. [10]

Anwendungsmöglichkeiten

Für Ethereum existieren eine Reihe von Anwendungsfällen, die sich in folgende Kategorien einordnen lassen:

1. Einsatz als Kryptowährung: So lassen sich zum Beispiel Token erstellen.
2. Einsatz in der Verwaltung / Führung: Dabei sind beispielsweise Crowdfunding oder Wahlen denkbar. Insbesondere da die Blockchain eine hohe Sicherheit gegen Manipulation und Fälschung bietet. Ein weiteres Beispiel wäre die Realisierung von Multi-Signature-Accounts, die es erlauben, dass mehrere Accounts gemeinsam Entscheidungen treffen.
3. Einsatz für Handel / Miete / Teilen von Objekten: Hier sind zum Beispiel E-Versicherungen oder Mietanwendungen für IoT-Geräte möglich. Letzteres bietet beispielsweise die Slock.it GmbH mit ihrem Universal Sharing Network. Es erlaubt das Vermieten von Objekten auf der Blockchain, inklusive Zahlung in Ether. [vgl. 11]
4. Einsatz als digitale Registrierung: Auch sind Anwendungsfälle im Bereich Supply Chain beziehungsweise Logistik naheliegend.



Abbildung 3: Kategorien der Anwendungsfälle [86]

Insgesamt ist, wie bereits erwähnt, alles möglich, was mathematisch definierbar ist und von den Vorzügen der Blockchain bezüglich Authentizität, Integrität, Persistenz und Dezentralität profitieren will.

Herausforderungen

Die Blockchaintechnologie birgt als dezentrales System auch eigene Gefahren. Beispielsweise besteht in Systemen, die Konsens durch Mehrheitsentscheidung erreichen, die Gefahr einer 51% - Attacke. Dabei kann ein Angreifer, dem es gelingt, mindestens 51 Prozent des Netzwerkes zu kontrollieren, die Anerkennung von Blöcken und Transaktionen beeinflussen. Das bedeutet, er kann im Zuge eines „Double Spending“ Angriffs Token mehrfach ausgeben, indem er im Hintergrund eine längere Kette als die aktuelle Hauptkette mined und publiziert. Dadurch wird seine Kette die neue gültige und er kann Token, die er auf dem anderen Zweig ausgegeben hatte, nun auf seiner Kette erneut ausgeben. Da er die Mehrheit der Hashleistung besitzt, kann er dies mit hoher Wahrscheinlichkeit erreichen. Außerdem könnte er die Aufnahme bestimmter Transaktionen in Blöcke herauszögern. Dies würde dem System erheblich schaden. Allerdings benötigt ein Angreifer dafür eine sehr große Hashrate, damit er schneller mehr Blöcke als die Hauptkette erstellen kann. Dies ist jedoch mit hohen Kosten verbunden und erfordert viel Rechenleistung, sodass es praktisch nur großen Miningpools, Zusammenschlüssen von Minern, möglich wäre. [vgl. 12] Eine solche Attacke würde, Schätzungen zufolge, im Ethereum-Netzwerk etwa 400 000 Dollar pro Stunde kosten. [22]

Neben konkreten Gefahren gibt es weitere Herausforderungen für das Ethereum-Netzwerk und die Blockchaintechnologie im Allgemeinen [vgl. 86]:

1. *Skalierbarkeit*: Es können aktuell nur eine sehr limitierte Menge an Transaktionen verarbeitet werden, welche mit circa 15 Transaktionen pro Sekunde im Vergleich zu Visa nicht mithalten kann, das 2000 in der Zeit handhaben kann.
2. *Privatheit und Anonymität*: Transaktionen können auf der offiziellen Blockchain nachverfolgt werden, auch wenn Sender und Empfänger anonym sind. Wenn Transaktionen jedoch vertrauliche Informationen enthalten, müssen „Zero-Knowledge“-Verfahren eingesetzt werden.
3. *Langsamkeit*: Man kann nie davon ausgehen, dass eine Transaktion bereits in den nächsten Block aufgenommen wird. So ist zwar die Blockzeit mit durchschnittlich 15 Sekunden kürzer als bei Bitcoin, trotzdem können einige Minuten Wartezeit entstehen. Um sicherzugehen, dass eine Transaktion mit hoher Wahrscheinlichkeit aufgenommen ist, sollte man zusätzlich weitere drei bis sechs Blöcke warten. Dies erhöht die Wartezeit zusätzlich.
4. *Hohe Kosten*: Für jede Transaktion muss eine Gebühr bezahlt werden, die für das Übersenden sehr kleiner Beträge unrentabel sein kann. Zudem verursacht das Speichern von Daten auf der Blockchain hohe Kosten.
5. *Softwaresicherheit*: Smart Contracts auf der Blockchain können nachträglich nicht korrigiert werden. Fehler können hohe finanzielle Verluste bedeuten. Dies macht ein ausgiebiges Testen vor dem Deployment erforderlich.
6. *Energieverbrauch:* Der Proof-of-Work benötigt sehr viel Energie, da so schnell wie möglich so viele Hashwerte wie möglich durchprobiert werden. Dies ist zudem notwendig, um den beim Mining erzeugten Token, der Minerbelohnung, ihren Wert

zu geben. Jedoch verbraucht das Bitcoin-System im Jahr bereits mehr Strom als die Schweiz. [vgl. 13] Auch Ethereum hat 2017 schon so viel Strom wie Zypern verbraucht. [vgl. 14] Aus diesem Grund wird bereits an einer Alternative für Proof-of-Work gearbeitet, dem „Proof-of-Stake“. Dabei hängt das Recht zu Minen vom Etherbesitz ab. Dies soll weitaus umweltverträglicher sein.

7. *Sicherheit*: Trotz guter Absicherung des Systems durch Kryptografie, kann es im Zusammenhang mit den Nutzern zu Sicherheitsproblemen kommen. So führt der Verlust eines privaten Schlüssels auch zum Verlust der Kontrolle über den entsprechenden Account. Es ist nicht möglich, einen neuen dafür zu erhalten. Außerdem kann die Kompromittierung des Schlüssels einem Angreifer unerwünschten Zugriff auf das Guthaben eines Accounts ermöglichen. Das Sicherheitsrisiko stellen hierbei somit die Nutzer selbst dar.

Einigen dieser Punkte soll zukünftig noch begegnet werden, wohingegen andere mit der Blockchaintechnologie selbst zusammenhängen.

2.2 Die Ethereum Virtual Machine

Die Ethereum Virtual Machine (EVM) ist die Laufzeitumgebung, die es erlaubt, die für die Blockchain entwickelten dezentralen Applikationen auszuführen. Sie läuft auf jedem Netzwerkknoten. Sie ist Turing-vollständig und unterstützt somit Schleifen, die jedoch durch das bereits beschriebene Gaslimit beschränkt werden. [vgl. 2]

Um die Smart Contracts auszuführen, müssen sie zuvor in EVM Bytecode übersetzt werden, welcher dann von der virtuellen Maschine verstanden wird. Die EVM arbeitet stackbasiert. Das bedeutet, sie nimmt die obersten Elemente vom Stack und legt, je nach Operation, das Ergebnis wieder obenauf. Die maximale Stacktiefe liegt bei 1024 Elementen, wobei nur die obersten 16 direkt erreichbar sind. Der Stack ist der einzige Speicher, der nichts kostet. Die Slots haben 256 Bit Weite.

Neben dem Stack gibt es noch Memory und Storage. Memory ist wie Hauptspeicher nicht persistent und wird für jeden Contract-Aufruf bereinigt. Er ist linear angeordnet und in 256-Bit Weite lesbar, beschrieben werden kann er in 8 oder 256 Bit Stücken. Seine Reservierung ist mit Kosten verbunden und immer nur in 256-Bit-Schritten möglich. Die Kosten steigen quadratisch zur Größe des allokierten Speichers. [vgl. 15]

Der Storage ist persistent und Teil des States des CA. Schreibzugriffe sind hier besonders teuer. Er enthält beispielsweise die globalen Variablen der Smart Contracts. Auch hier werden Daten in 32 Byte Stücken angesprochen. Programme auf der Blockchain können nicht gegenseitig auf ihre Zustände zugreifen. [vgl. 15, 16]

Die virtuelle Maschine unterstützt eine Reihe von Opcodes, die im Yellow Paper beschrieben werden, darunter spezielle wie SHA3 und CALLVALUE. Zudem gibt es eine

Reihe von Programmiersprachen, deren Compiler EVM Bytecode erzeugt. Solidity, Vyper, LLL und Bamboo sollen im Folgenden noch vorgestellt werden.

Zur Interaktion mit dem Ethereum-Netzwerk kommen verschiedene Clients zum Einsatz – implementiert in C++, JavaScript, Go und anderen Sprachen. Sie alle verfügen über die EVM.

Projekte, wonach anstelle der EVM Web Assembly eingesetzt werden soll, was in Browsern Anwendung findet, sind bereits vorhanden. Dies soll jede etablierte Programmiersprache zur Entwicklung von Smart Contracts nutzbar machen, inklusive ihrer Bibliotheken. Dies könnten zum Beispiel C, C++ und Java sein. Ein Transcompiler, der EVM Bytecode zu eWASM kompiliert, existiert bereits, befindet sich jedoch noch in der Entwicklung. [vgl. 88]

2.3 Smart Contracts

Der Begriff „Smart Contract“ hat sich erstmals 1994 durch Nick Szabo etabliert, im Zusammenhang mit autonomen rechtlichen Verträgen. Ein Smart Contract bezeichnet ein Computerprogramm, welches Verträge digital abbildet oder unterstützt und somit den Verzicht auf schriftliche Abschlüsse, insbesondere über Mittelsmänner, ermöglichen kann. Smart Contracts sollen teilweise oder komplett selbstständig agieren können, im Falle von Ethereum bedeutet dies, dass sie durch Funktionsaufrufe beziehungsweise Transaktionen ausgelöst werden. [vgl. 17]

Smart Contracts besitzen einige Vorteile in Bezug auf Verträge. So arbeiten sie immer so wie programmiert, da sie deterministisch sind. So können bei der Ausführung keine menschlichen Fehler auftreten. Alle Interaktionen mit ihnen sind dauerhaft auf der Blockchain nachweisbar. Es wird jede Interaktion mit den Contracts von allen Knoten des Netzwerkes gesehen und validiert, trotzdem können sie nahezu anonymisiert erfolgen, da lediglich die Adressen der Akteure mit publiziert werden. Natürlich sind sie selbst dafür zuständig, die Zugehörigkeit zu ihnen zu verbergen, wenn sie anonym bleiben wollen.

Smart Contracts sind in Ethereum, wie bereits erwähnt, der Code eines Contract Accounts. Sie können Nachrichten durch Funktionsaufrufe erhalten sowie selbst welche versenden oder für andere Contracts als Bibliothek fungieren. Jeder kann mit dem Code eines Smart Contracts auf der Blockchain interagieren, deshalb sind bei der Entwicklung entsprechend Sicherheitsmaßnahmen umzusetzen.

Das Hochladen auf die Blockchain, was als deployen bezeichnet wird, erfordert nach dem Kompilieren eine Transaktion. Dafür müssen anfallende Gaskosten bezahlt werden. Es können zusätzliche Daten für die Contract-Erzeugung mitgesendet werden. Ist ein Contract auf der Blockchain deployed, so kann er nachträglich aufgrund der Immutabilität

der Blockchain nicht mehr bearbeitet oder gelöscht werden. Deshalb ist Sorgfalt bei der Implementation sowie eine ausgiebige Testphase notwendig.

Zum Kompilieren gibt es sowohl Onlinecompiler als auch Compiler für die Kommandozeile. Auch einige Entwicklungsumgebungen haben diese integriert.

Ein Problem bei Smart Contracts ist ihre ungeklärte gesetzliche Grundlage. Zum einen muss für eine wirksame Willenserklärung der Vertrag auch für Laien verständlich sein. Dies ist jedoch bei Quellcode schwierig, da viele Menschen keine Programmierkenntnisse besitzen. Daher werden die Contracts aktuell eher als Mittel zur Umsetzung eines Vertrags anstatt zu dessen Abschluss angesehen. Zum anderen ist die Haftung bei Programmierfehlern aber auch das Widerrufsrecht noch nicht geklärt. Außerdem gilt, dass der Zugriff auf einen Contract niemandem untersagt werden kann. Er wird immer ausgeführt, wenn er aufgerufen wird. Daher gibt es an sich keinen unzulässigen Zugriff. [vgl. 18]

Schließlich sind reine Smart Contracts von DApps, dezentralen Applikationen, abzugrenzen. Im Gegensatz zum Smart Contract umfasst eine DApp zusätzlich zum Contract-Code noch eine Nutzerschnittstelle, wie eine Applikation oder ein Front-End für den Browser und eventuell ein Datenbank-Backend. Sie enthalten somit Teile auf und außerhalb der Blockchain. Ein Smart Contract dagegen ist die reine Logik, also der Code auf der Blockchain. [vgl. 19, S.174f.]

2.4 Smart Contract Programmiersprachen

Im Folgenden sollen nun die Programmiersprachen zur Smart Contract Entwicklung vorgestellt werden. Dabei werden neben dem typischen Aufbau der Contracts auch sprachliche Besonderheiten erläutert. Darüber hinaus soll ein Einblick in ein neues Prinzip als Alternative zur EVM, dem „Ethereum flavoured Web Assembly“ (eWASM), gegeben werden.

2.4.1 Solidity

Einführung

Bei Solidity handelt es sich um die am häufigsten zur Smart Contract Entwicklung eingesetzten Programmiersprache. Sie ist gut dokumentiert und es finden sich auch in der Ethereum Community vielerlei Hilfestellungen. Solidity basiert auf JavaScript und wird vom Compiler zu EVM Bytecode kompiliert. Es handelt sich um eine statisch typisierte Sprache, was bedeutet, dass zur Kompilierzeit alle Variablentypen bekannt sein müssen. Die erste Spezifikation der Sprache erfolgte 2014 durch Gavin Wood, seitdem wird ihre Entwicklung durch die Ethereum Foundation fortgeführt. Neben dem Compiler für die Kommandozeile, gibt es beispielsweise auch die Möglichkeit, Entwicklungsumgebungen zu nutzen, um Contracts zu schreiben und zu deployen. [vgl. 20]

Die Sprache unterstützt Mechanismen der Objektorientierten Programmierung, zum Beispiel Vererbung, Bibliotheken und nutzerdefinierte Typen wie „Structs“ und „Enums“. Für einige Entwicklungsumgebungen werden bereits Syntaxhervorhebungen angeboten. [vgl. 21]

Aufbau

Ein Solidity Contract beginnt immer mit dem sogenannten „pragma“, welches die verwendete Compilerversion angibt, zwecks Kompatibilitätsinformationen mit anderen Anwendungen. Danach können weitere Solidity-Dateien importiert werden. Darauf folgt nun der eigentliche Körper des Contract, der vom „contract“-Keyword eingeleitet wird.

Im Contract-Körper werden üblicherweise zunächst globale Variablen deklariert, gefolgt von Events, die in Funktionen ausgelöst werden können und in diesem Zuge den darauf lauschenden Web3-Schnittstellen übermittelt werden. Zusätzlich besteht die Möglichkeit, sogenannte „Modifier“ zu definieren, die an Funktionsköpfe angefügt werden können, um ihr Verhalten zu beeinflussen. Sie ähneln Makros in C.

Schließlich erfolgt die Definition der Funktionen des Contracts, die zwecks Zugriff und Sichtbarkeit mit bestimmten Keywords modifiziert werden können. Die folgenden Abbildungen zeigen den Aufbau eines Modifiers sowie eines Contracts im Allgemeinen.

```
modifier onlyOwner {
        require(msg.sender == owner);
        _;  }
```

Der Modifier prüft, ob derjenige, der mit der Contractfunktion interagiert, der Besitzer des Contracts ist. Wenn nicht, werden durch das require() alle Aktionen gestoppt und Änderungen rückgängig gemacht. Der Unterstrich bedeutet, dass nach erfolgreicher Prüfung dort mit der Funktion fortgefahren wird.

```
pragma solidity ^0.4.22;  //gibt Compilerversion an

contract Example {

    address public owner;     // statische Variablen
    uint public changeTime;
    uint public fee;

    // Event
    event OwnerChanged(address indexed new, address indexed old);

    // Konstruktor, wird seit Kompilerversion 0.4.22 explizit so
    // bezeichnet. Vorher hätte man Example() geschrieben.
    constructor (uint _fee) public {
        owner = msg.sender;
        fee = _fee;
        changeTime = now;
    }

    // Fallback-Fkt., akzeptiert keine Ether. Kehrt bei Aufruf alles um.
    function() {
        revert();
    }

    // Akzeptiert Ether, von außen aufrufbar.
    function changeOwner(address _new) external payable {
        require (msg.value >= fee);
        address old = owner;
        owner = _new;
        changeTime = now;
        emit OwnerChanged(_new, old);
    }

    // Getter für den Besitzer, wird auch automatisch für public
    // Variablen generiert.
    function getOwner() public view returns (address) {
        return owner;
    }
    //.....
}
```

Abbildung 4: Beispielcontract Solidity

Dieser Contract enthält Beispielvariablen, -events und -funktionen. Wenn vorhanden, wird als erste Funktion typischerweise der Konstruktor definiert. Diese spezielle Funktion sowie die sogenannte Fallback-Funktion werden im Folgenden noch erläutert.

Datentypen und Funktionen

In Solidity gibt es viele Datentypen, die auch in anderen Sprachen üblich sind. Boolean (bool), Ganzzahlen von acht bis 256 Bit mit Vorzeichen (int) und ohne (uint) sowie auch Zeichenketten (Strings). Festkommazahlen werden noch nicht unterstützt. Die Sprache bietet zudem die üblichen arithmetischen und logischen Operationen sowie Vergleiche. Strings können jedoch nur anhand ihres Hashwertes verglichen werden. [vgl. 23]

Zusätzlich gibt es spezielle Datentypen wie „address“. Dies sind 20 Byte Hexadezimalstrings mit führendem 0x. Sie sind für Accountadressen gedacht, daher kann man von ihnen das Guthaben (balance) abfragen sowie über „<address>.transfer(…)“ Wei versenden. Wenn dies an einen CA erfolgt, wird in diesem Zuge dessen Code ausgeführt. Weiterhin gibt es Bytearrays fester Länge – ein bis 32 Byte – auf deren Bytes man wie bei Arrays über Indices zugreifen kann und dynamische Bytearrays, wozu „string“ und „bytes“ gehören. [vgl. 23]

Es gibt in Solidity ebenfalls Referenztypen wozu auch „bytes“ und „string“ zählen. Sie werden, im Gegensatz zu den einfachen Datentypen nicht direkt als Wert übergeben, sondern per Referenz, da sie nicht in einen Speicherslot passen. Referenztypen sind hier Arrays, Structs, Mappings und Enums. Wie in vielen anderen Sprachen können Arrays feste oder variable Länge haben sowie ein- oder mehrdimensional sein. Ihre Länge kann mit „<name>.length“ abgefragt werden. Dynamische Arrays können mit „push(..)“ vergrößert werden. Es besteht jedoch nicht die Möglichkeit, diese als Rückgabewert zurückzugeben. [vgl. 23]

Structs sind vergleichbar mit denen der Sprache C. Sie stellen Zusammenfassungen mehrerer einzelner Variablen dar. Auch Enums sind mit denen von C vergleichbar, da sie wie eine Art benannte Aufzählung zu betrachten sind. Jedes Element kann somit direkt in einen Integer (Ganzzahl) umgewandelt werden. Folgende Abbildungen zeigen beispielhaft ein Struct und eine Enum. [vgl. 23]

```
enum Directions { North, East, South, West };
```

```
struct Dog {
        string name;
        uint8 age;
        address owner; }
```

Die Mappings in Solidity können in etwa wie Hashtabellen betrachtet werden. Dabei gibt es einen Schlüssel, dem ein Wert zugeordnet wird. Dieser Schlüssel darf dabei kein Mapping, Struct, dynamischer Array oder eine Enum sein. Als Wert sind sie erlaubt. Der Zugriff erfolgt über den Schlüssel, wobei die Syntax dem Arrayzugriff entspricht. Die Schlüssel werden nicht direkt abgespeichert, vielmehr ihr Keccak256 Hashwert. Beim Erzeugen eines Mappings werden alle möglichen Schlüssel erzeugt und der Wert mit Null initialisiert. Das folgende Beispiel zeigt ein Mapping. [vgl. 23]

```
mapping (keyType => valueType) {public|private} <name>;
```

Für Referenztypen besteht zusätzlich die Möglichkeit, ihren Speicherort zu bestimmen. Dazu gibt es die „storage“ und „memory“ Keywords. Standardmäßig werden statische und

lokale Variablen dem Storage zugewiesen, wohingegen Funktionsparameter in den Memoryspeicher kommen. [vgl. 23]

Zusätzlich ermöglicht das Keyword „delete“ das Zurücksetzen von Variablen auf ihren Standardwert, zum Beispiel Null oder false. [vgl. 23]

Funktionen in Solidity sind entsprechend folgender Abbildung aufgebaut:

```
function (<parameter types>) {internal|external|public|private} [pure|constant|view|payable] [returns (<return types>)]
```

Abbildung 5: Solidity Funktionsaufbau [vgl. 23]

Nach dem Schlüsselwort „function“ folgt der Name der Funktion (Ausnahme: Fallback) und danach die Parameter. Anschließend kann sie über Keywords bezüglich Zugriffsbereich und Sichtbarkeiten modifiziert werden, dabei ist public der Standard. Optionale weitere Modifikatoren beschreiben den Einfluss auf den State des Contracts. Schließlich werden, wenn vorhanden, nach dem Schlüsselwort „returns“ die Rückgabetypen angegeben. Selbst implementierte Modifier stehen vor dem Return-Statement, in der Reihenfolge ihrer gewünschten Ausführung.

Zur Bedeutung der Schlüsselwörter:

- *Internal* bedeutet, die Funktion ist nur im aktuellen Contract, internen Bibliotheksfunktionen sowie abgeleiteten Contracts aufrufbar.
- *External* Funktionen sind nur von außen über Transaktionen ansprechbar, von innen benötigt man das „this.<function>“ Keyword, welches für den aktuellen Contract steht.
- *Public* bedeutet, die Funktion ist von außen und innen ansprechbar, wohingegen
- *private* den Zugriff auf den aktuellen Contract beschränkt.
- *Constant* und *view* bedeuten beide, dass der State des Contract nicht verändert wird, höchstens von ihm gelesen.
- *Pure* heißt, es wird weder auf die Blockchain geschrieben, noch von ihr gelesen.
- Schließlich dient das *payable* Keyword dazu, eine Funktion zu markieren, die Zahlungen empfangen darf. [vgl. 24]

Zudem gibt es noch die Konstruktorfunktion, die mit dem Keyword „constructor“ markiert wird. Sie kann Parameter besitzen, die beim Deployment in der Transaktion mit angegeben werden müssen. Sie wird nur einmal beim Deployment ausgeführt und nimmt die Initialisierung vor. Ihr Code wird nicht mit hochgeladen, daher ist sie nicht erneut aufrufbar.

Ein Sonderfall bei den Funktionen stellt auch die Fallback-Funktion dar. Sie hat keinen Namen oder Parameter. Sie wird ausgelöst, wenn der Contract aufgerufen wird und keine Funktions-ID des Contracts zum Aufruf passt oder direkt Ether an ihn gesendet werden. Der Code darin wird ausgeführt und wenn Zahlungen empfangen werden sollen, muss sie als payable markiert werden. Wird sie aktiviert, sind zur Codeausführung nur 2300 Gas

verfügbar, was für das Loggen eines Events genügt. Gibt man sie nicht selbst an, wird automatisch eine solche Funktion erzeugt.

Die Funktions-ID zur Referenzierung der Contract-Funktionen wird aus den linken vier Byte des keccak256 – Hash des Funktionskopfes gebildet.

Besonderheiten

Solidity verfügt über einige sprachliche Besonderheiten. Es werden beispielsweise Interfaces unterstützt, die nur Funktionsköpfe und Eventdeklarationen enthalten. Außerdem wird das Überladen von Funktionen ermöglicht, wobei sie gleiche Namen, aber unterschiedliche Parameter aufweisen müssen.

Zudem können Contracts abstrakt sein, indem sie nur Teile der Funktionen implementieren. Sie können dann von anderen Contracts geerbt werden. Auch Bibliotheken können realisiert werden, dabei wird anstelle des contract – Keywords das library – Keyword verwendet. Sie haben keinen State und können nicht erben oder Token empfangen. [vgl. 24]

Der Aufruf externen Contractcodes erfolgt mittels der „delegatecall“ – Funktion. Sie lädt den Code des anderen Contracts und führt dessen Funktionen im Kontext des Aufrufers aus. Das bedeutet, es wird dabei der State des Aufrufers beeinflusst.

Darüber hinaus existieren eingebaute Funktionen zur Fehlerbehandlung. Für interne Fehler, das heißt um zu prüfen, dass der Contract sich im erwarteten Zustand befindet, verwendet man „assert()“ und zum Prüfen von Eingabedaten „require()“. Die Funktion „revert()“ stoppt die Ausführung und kehrt alle Änderungen um. [vgl. 25]

Neben diesen Funktionen gibt es Hashfunktionen wie „keccak256()“ oder Funktionen wie „selfdestruct()“, die einen Contract zerstört und seine Tokens an die übergebene Adresse sendet. [vgl. 25]

Schließlich verfügt die Sprache über Ethereumspezifische Einheiten wie alle Ethereinheiten, Zeiteinheiten sowie die Möglichkeit, Block- und Transaktionsinformationen von der Blockchain abzurufen. Beispiele sind „now“ für den Zeitstempel des aktuellen Blocks, „msg.sender“ für den Absender der Nachricht an den Contract, „msg.value“ für den Wert an Ether, der mitgesendet wurde oder „blockhash(<nr.>)“ für den Hash eines Blockes. [vgl. 25] Auch Assembly kann innerhalb von Contracts eingesetzt werden.

2.4.2 Vyper

Einführung

Vyper ist eine noch sehr neue Programmiersprache, die auf Python3 basiert. Ihre Entwicklung begann Ende 2016 durch das Ethereum Team. [vgl. 26] Der Vypercompiler

kompiliert ebenfalls zu EVM Bytecode, es gibt auch hier Kommandozeilen- und Online – Compiler.

Vyper erfordert wenig Vorkenntnisse, um den Quellcode zu verstehen. Dafür werden eher weniger komplexe Contracts unterstützt. Es wurden bereits Testimplementationen für Ethereum bezüglich Casper und Sharding in Vyper durchgeführt. [vgl. 27]

Mit Vyper sollen Contracts einfach lesbar sein, sodass die Absichten der Programmierung gleich erkennbar sind und eine leichte Auditierbarkeit gewährleistet ist. Irreführender Code soll schwer zu schreiben sein. Vyper und Solidity Contracts sollen miteinander interagieren können. [vgl. 28]

Vyper verfolgt somit drei Hauptprinzipien:

1. Das Erste ist Sicherheit: es soll natürlich sein, mit Vyper sichere Smart Contracts zu entwickeln.
2. Das Zweite ist die Sprach- und Compilereinfachheit. Die Sprach- und Compilerimplementation soll einfach gestaltet sein, im Sinne der Erlernbarkeit sowie Lesbarkeit. Der Compilercode ist kompakt zu gestalten.
3. Das dritte Prinzip ist schließlich die Auditierbarkeit. Der Code soll, wie bereits erwähnt, so gut wie möglich auch für Einsteiger lesbar sein. Dabei ist es ein Ziel, das Schreiben von missverständlichem Code zu erschweren. [vgl. 29]

Daher gibt es einige Features, die Vyper bieten soll. Diese beinhalten Grenzen- und Overflowprüfung bei Arrayzugriff und Arithmetik sowie die Ermittelbarkeit einer präzisen oberen Grenze für den Gasverbrauch eines jeden Funktionsaufrufes. Weiterhin ist die Unterstützung von vorzeichenbehafteten Ganzzahlen und Festkommazahlen angestrebt. Feste Typen und Einheiten wie Zeitstempel, Wei pro Sekunde und Zeitintervalle werden angeboten. [vgl. 29]

In Vyper stehen somit Sicherheit, Eindeutigkeit, Einfachheit und der Fokus auf das unbedingt Notwendige im Vordergrund. [vgl. 28]

Aufbau

In Vyper ist es nicht möglich, Contracts dateiübergreifend zu implementieren. Daher können sie keine Imports durchführen. Alles muss innerhalb einer Datei implementiert werden. Es gibt kein „contract“ – Keyword oder Pragma. Zuerst erfolgt die Deklaration von Events, die analog zu Solidity funktionieren. Danach folgen statische Variablen, die dauerhaft im Storage gespeichert werden. Im Anschluss daran folgen die Funktionen, die entweder „private“ oder „public“ sein können und dementsprechend intern oder öffentlich aufrufbar sind. Die entsprechenden Keywords werden hier als „decorator“ mit einem @-Symbol vor die Funktion geschrieben. Die Funktionen werden hier, wie in Python, durch Einrücken anstatt Klammerung strukturiert. [vgl. 30]

Neben eigenen Funktionen gibt es auch hier vordefinierte Funktionen, wie beispielsweise „keccak256()“ und „len()“. Letztere gibt die Länge einer Bytesequenz zurück. [vgl. 31] Die folgende Abbildung zeigt ein Beispiel für den Aufbau eines Vyper Contracts.

```
1   # Beispiel Vyper
2
3   # Event
4   NewOwner: event({_new: indexed(address), _old: address})
5
6   # statische Variablen
7   owner: public(address)
8   last_change: public(timestamp)
9   fee: public(wei_value)
10
11  #Konstruktor
12  @public
13  def __init__(owner: address, fee: wei_value):
14      self.owner = owner
15      self.last_change = block.timestamp
16      self.fee = fee
17
18  @public
19  @payable   # nimmt Ether an
20  def changeOwner(_new: address):
21      assert msg.value >= fee
22      _old: address = self.owner
23      self.owner = _new
24      self.last_change = block.timestamp
25      log.NewOwner(_new, _old)
26
27  @public
28  def getOwner() -> address:
29      return self.owner
30  #.....
```

Abbildung 6: Vyper Contract Aufbau

Die Abbildung zeigt, dass der Konstruktor hier mit „__init__“ bezeichnet wird. Zudem wird anstelle des Keywords „function“ „def“ verwendet. Sichtbarkeitsmodifikatoren umschließen den Datentyp bei der Variablendeklaration (siehe Zeilen 7 bis 9).

Datentypen und Funktionen

Auch Vyper ist eine statisch typisierte Sprache. Es gibt Datentypen wie Boolean (bool), Ganzzahlen mit Vorzeichen in 128-Bit Größe (int128) und ohne Vorzeichen in 256-Bit

Größe (uint256). Zudem werden Festkommazahlen mit zehn Dezimalstellen in 128-Bit Größe unterstützt (decimal). Übliche Operationen der Arithmetik, Vergleiche sowie bitweise Operationen werden auch in Vyper angeboten, ebenso logische Operatoren. Den „address“ Datentyp findet man auch hier wieder, er ist analog zu Solidity zu betrachten. Auch hier kann die „balance“ des zugehörigen Accounts abgefragt werden, hinzu kommt hier die „codesize“. Diese ermöglicht es, die Codegröße des Contracts hinter einer Adresse abzufragen. [vgl. 32]

Vyper verfügt über Einheiten wie „timestamp“ und „timedelta“. Man kann zudem eigene Einheiten definieren, die aber auf int128 oder decimal basieren müssen. [vgl. 32] Die folgende Abbildung zeigt, wie eigene Einheiten erst definiert und dann für Variablendeklarationen angewandt werden.

```
units: {
   ml: "milliliter",
   l: "liter"
}


tasse: decimal(ml)
fass: decimal(l)
```

Abbildung 7: Benutzerdefinierte Einheiten Vyper

Weitere unterstützte Datentypen sind bytes32 als dynamische und bytes[n] als feste Bytearrays der Größe n Bytes. Diese können auch Zeichenketten enthalten. Zusätzlich können auf sie eingebaute Funktionen wie „len()“, „sha3()“ oder „concat()“ angewandt werden. Außerdem gibt es Listen, die wie Arrays betrachtet werden können und deren Zugriff ebenfalls dementsprechend erfolgt. Structs existieren ebenfalls und funktionieren wie in Solidity. [vgl. 32]

Mappings können in Vyper nur statische Variablen sein. Sie ähneln auch hier Hashtabellen. Der Schlüssel wird auch hier als Hash abgelegt und es gelten die gleichen Restriktionen wie in Solidity. Der Standard für die Sichtbarkeit statischer Variablen ist hier jedoch „private“. [vgl. 32] Im Folgenden ist ein Beispielmapping inklusive Anwendung dargestellt.

```
tokenByOwner: uint256[address]

tokenByOwner[msg.sender] = msg.value
```

Wie in Solidity kann eine Variable nicht mittels NULL dereferenziert werden, wie es in manch anderen Programmiersprachen möglich ist. Hier kann vergleichbar mit Solidity delete eine Variable mittels der Zuweisung von „None“ auf ihren Standardwert zurückgesetzt werden. Als Bezug zur aktuellen Instanz eines Contracts wird „self“ verwendet. [vgl. 33]

Den Aufbau von Funktionsköpfen verdeutlicht folgende Abbildung:

```
{@public|@private}
[{@constant|@payable}]
def <name>(<parameters>) [-> <returnTypes>]:
```

Abbildung 8: Vyper Funktionen

Anstelle eines Keywords kommt zum Verweis auf die Rückgabe ein Pfeil zum Einsatz.

Um Ether zu senden, gibt es die Funktion „send(..)“. Zudem stehen auch in Vyper Blockinformationen über die „block“ Variable oder Aufrufinformationen über „msg“ zur Verfügung und ein „selfdestruct()“ existiert analog zu Solidity. Ebenfalls vorhanden ist ein „assert <Bedingung>“, um Bedingungen zu formulieren, die zum Fortfahren erfüllt sein müssen. [vgl. 33] Fallback-Funktionen werden erst seit kurzem unterstützt, sie werden mit „__default__“ betitelt und können keine Parameter oder Rückgaben definieren. Sie müssen als public markiert sein. Wird keine solche angegeben, generiert Vyper automatisch eine, die lediglich ein Rollback durchführt. [vgl. 30]

Besonderheiten

Vyper weist, trotz Gemeinsamkeiten, einige Unterschiede zu Solidity auf. So werden keine eigenen Modifier unterstützt, um Irreführung zu vermeiden. Stattdessen soll auf „assert“ zurückgegriffen werden. Weiterhin wird Vererbung nicht unterstützt, um das Verständnis des Codes sowie die Auditierbarkeit zu verbessern. Gleiches gilt für Assembly und die Funktionsüberladung. Da Operatorüberladung besonders fehleranfällig und schwer nachvollziehbar ist, wird davon abgesehen. [vgl. 29]

Vyper verzichtet auch auf rekursive Aufrufe. Sie stehen der Ermittlung der oberen Gas-Grenze im Weg und ermöglichen Angriffe wie Denial of Service. [vgl. 29]

Externe Aufrufe sind noch ganz neu in Vyper integriert. Um sie zu ermöglichen, muss der andere Contract mit den zu nutzenden Funktionen am Anfang des Aufrufercontracts angegeben werden. Zusätzlich gibt man die Contract-Schnittstelle (Application Binary Interface) an. So können neben Variablen vom Funktions-Typ auch solche vom Typ Contract angelegt werden. [vgl. 27] Siehe dazu folgender Code:

```
contract external_contract_name():
        //Funktionen, die man nutzen will
        def beispiel() -> int128:pass

//Contract Variable:
beispiel: static(external_contract_name)
```

Die Contract-Variable muss als „static“ oder „modifiable“ angegeben werden. [vgl. 34] Die externen Aufrufe sind noch nicht gut dokumentiert.

2.4.3 LLL

Einführung

LLL steht für „Low-level Lisp-like Language“. Diese Programmiersprache war die Erste für Smart Contracts bestimmte Programmiersprache. Sie ist näher an der EVM als Solidity und Vyper, so ist direkter Speicherzugriff möglich und es können EVM Opcodes verwendet werden. LLL weist Gemeinsamkeiten mit Lisp auf, hier wird viel Klammerung zur Gliederung des Codes eingesetzt. Aufgrund der leichteren Lesbarkeit und höheren Anfängerfreundlichkeit hat Solidity LLL überflügelt, jedoch bekommt LLL mittlerweile wieder mehr Zuspruch. Die ENS Registry (Ethereum Name Service) des Ethereum Netzwerkes wurde beispielsweise in LLL implementiert.

Zur Zeit wird insbesondere durch Daniel Ellison von ConsenSys wieder auf LLL aufmerksam gemacht. In einer Neuimplementation eines Solidity-Contracts mit LLL konnte er einen um 70 % kleineren Bytecode erreichen. Dies zeigt eine wesentliche Eigenschaft von LLL: die Fähigkeit sehr kleine Bytecodes zu erzeugen und keine Redundanzen oder unübersichtlichen Sprünge zu enthalten. LLL verwendet anders als die bisherigen Sprachen Präfixnotation, das bedeutet, anstatt 1+3 schreibt man +13. [vgl. 35] [vgl. 36]

Aufbau

Der Aufbau von LLL Contracts unterscheidet sich stark von den bisher beschriebenen Sprachen. Ein solcher Contract ist immer ein einziger Ausdruck, der weitere Ausdrücke enthalten kann. Jeder Ausdruck wird in Klammern gesetzt und muss sich in einer Kontrollstruktur wie while, if, for, when befinden oder, wenn keine davon benötigt wird, von „(seq ……..)“ umschlossen sein. Dies gilt auch für den Contract im Ganzen. Der Contract muss strukturiert werden, damit der Standard für die ABI-Schnittstelle erfüllt ist. In LLL gibt es keinen Konstruktor oder Funktionen im Allgemeinen. Man unterteilt den Code aber üblicherweise in eine Init- und eine Code-Sektion. In der Init-Sektion werden Aufgaben des Konstruktors übernommen und im Code-Abschnitt die der Funktionen eines Contracts. Wie bei Solidity und Vyper wird der Code der Init-Sektion nicht mitdeployed. [vgl. 37] Das folgende Beispiel zeigt eine LLL Sequenz mit zwei inneren Ausdrücken.

```
(seq
    (mstore 0x00 13)
    (return 0x00 32))
```

In diesem Beispiel sieht man zwei innere Ausdrücke, typischerweise in Klammern. Der Erste legt die 13 im Memory an Adresse 0 ab und der Zweite gibt ab Adresse 0 32 Byte zurück, also die 13. Umschlossen werden sie von einem äußeren Ausdruck: einer Kontrollstruktur, hier „seq“.

Ein Contract wird also für gewöhnlich mit „(seq“ eingeleitet. Danach können mithilfe von „include“ Imports erfolgen, vergleichbar mit Solidity. Der angegebene Code wird dort an der Stelle eingefügt. In der Init-Sektion können beispielsweise Makros enthalten sein, das sind benannte Ausdrücke und Konstanten, wie es sie auch in C gibt. Dabei wird jedesmal der Name durch den Ausdruck ersetzt, wenn sie eingesetzt werden. [vgl. 38]

Das Kompilieren erzeugt den Deployment-Bytecode inklusive Init-Abschnitt. Beim Hochladen wird zuerst die Init-Sektion ausgeführt und dann die Größe und der Start des Code-Bereichs ermittelt. Dieser wird dann nach dem Deployment zurückgegeben. Zur Ermittlung der Codegröße gibt es den Operator „bytecodesize“. [vgl. 38]

Die Code-Sektion wird in „(returnlll ……)“ eingeschlossen, damit mithilfe dieses Operators der Code darin mit dem Deployment in den Memory-Speicher zurückgegeben wird. Danach erfolgt auch hier die Einfassung aller Ausdrücke in eine Sequenz. Die „Funktionen“ lassen sich über Makros realisieren. Beispielsweise können zu diesen Makros „Funktions-IDs“ wie in Solidity ermittelt werden, indem die Hashwerte ihrer Namen und Parameter ermittelt und abgespeichert werden. Bei einem Aufruf werden diese IDs in einer Switch-Case-Struktur durchgeprüft, bis die Richtige gefunden wird oder der Standardfall erreicht ist. Dieser ist vergleichbar mit der Fallback-Funktion. [vgl. 39]

Im Folgenden zeigt ein Beispiel den allgemeinen Aufbau eines LLL Contracts. Es enthält Kommentare mit kurzen Erklärungen. Alle Statements befinden sich in einem großen „seq“. Der Fallback besteht hier lediglich aus dem eingebauten Makro „panic“. Dieser Contract gibt einen Konstruktorparameter zurück.

```
;;;; @title Ein Beispiel der LLL Contract Struktur.
;;;; @author Daniel Ellison <daniel@syrinx.net>
;;;; Quelle: https://gist.github.com/zigguratt/ea41e2af324ac35ce657d435fd538b21

(seq

  ;; Makros für den Contract einbinden.
  (include "macros.lll")

  ;;; ---------------------------------------------------------------------------
  ;;; INIT

  ;; Der "Konstruktor soll Zahlungen ablehnen, da er nicht payable ist,
  ;; also wird abgebrochen,  wenn ETH beim deployment mitgeschickt werden.
  (when (callvalue)
    (panic))

  ;; Ein Konstruktorargument speichern im Storage.
  (codecopy *scratch* (bytecodesize) 32)
  (sstore *parameter* (mload *scratch*))

  ;;; ---------------------------------------------------------------------------
  ;;; CODE

  (returnlll
    (seq

      ;; Ein Makro aus macros.lll zum Holen der gefragten Funktions-ID.
      ;; Sie wird aus den Aufrufdaten geholt und im Memory abgelegt.
      ;; Eine Technik von Ben Edgington.

      get-function-id

      ;; ---------------------------------------------------------------------
      ;; @notice Gibt den Konstruktorparameter zurück. "function" und
      ;; "*get-constructor-parameter*" sowie "unpayable" sind Makros.
      ;; Signatur von getConstructorParameter() ist *get-constructor-parameter*
      ;; @return Konstruktorargument.

      (function *get-constructor-parameter*
        (seq unpayable

          ;; Konstruktorparameter wird im Memory abgelegt an Adresse des Makros
          ;; *scratch*, hier 0x00. Dazu aus Storage geladen.
          (mstore *scratch* (sload *parameter*))

          ;; Der 'return' Operator nimmt einen Memory Bereich und eine Länge
          ;; und gibt daten dort entsprechend angegebener Länge zurück (Byte).
          (return *scratch* 32)))

      ;; ---------------------------------------------------------------------
      ;; @notice Fallback: Keine Funktionen passten zur ID in calldata.

  (panic))))
```

Abbildung 9: Aufbau LLL Contract [40]

Datentypen und Funktionen

In LLL gibt es einige eingebaute Makros. Dazu gehören „send“, „msg“, „create“, „sha3“, „return“, „wei“ und „ether“. [vgl. 41] Aber auch Makros wie „panic“, welches invaliden As-

semblercode erzeugt und damit einen Abbruch herbeiführt, oder „returnlll“, welches bereits angesprochen wurde. [vgl. 36]

Zudem verfügt LLL über eingebaute Ausdrücke wie „callvalue“, was den mitgesendeten Wert eines Aufrufes adressiert. Weiterhin kann „codecopy“ Bytecode in den Memory-Speicher kopieren und „mload“ beziehungsweise „mstore“ dort Daten hineinladen oder speichern. Für den Storage gibt es dafür „sload“ und „sstore“. [vgl. 38]

Da, wie bereits besprochen, keine Funktionen an sich existieren, werden stattdessen einfache Anweisungsblöcke oder auch Makros eingesetzt. Ihre allgemeine Definition illustriert folgende Abbildung:

```
(def 'name (<Ausdruck>)) ;; empfohlen für Funktionsmakros

(def '*name* <Konstante>)  ;; empfohlen für Konstanten
```

Abbildung 10: LLL Makro

Es empfiehlt sich, Makros für Konstanten in Sterne zu setzen, damit sie leichter von Makros, die Funktionen entsprechen, zu unterscheiden sind. Zudem können Makros über Parameter verfügen, die dann in Klammern hinter dem Namen notiert werden. Eine Definition von gleichnamigen Makros ist bei unterschiedlicher Parameterzahl möglich. [vgl. 38] [vgl. 42]

In LLL werden alle Operationen in Präfixnotation geschrieben. Es steht zuerst der Operator und danach die zu manipulierenden Ausdrücke. Negative Zahlen können hier kein direkter Input sein, sie müssen durch Subtraktion von 0 angegeben werden. Über die Operation „lit“ können zudem Zeichenketten in den Speicher abgelegt werden. Sie werden entweder mit doppelten Anführungszeichen umfasst oder mit einem einzelnen am Anfang, was jedoch erlaubte Zeichen einschränkt. Repräsentiert werden sie als einzelnes 32 Byte-Stück. Sind die Zeichenketten größer als 32 Byte wird der Rest einfach ignoriert, wenn sie beispielsweise als Makrokonstante definiert werden. Mit dem „lit“ Operator gibt es für die Ablage im Speicher keine Beschränkung der Länge. [vgl. 42]

LLL unterstützt analog zu den anderen Sprachen Arithmetik, Logik und Vergleiche, wobei es letztere für vorzeichenlose und vorzeichenbehaftete Ausdrücke gibt. Auch bitweise Operationen sind möglich. [vgl. 42]

Variablen werden mit „set“ oder „with“ deklariert. Ihnen wird dann 32 Byte Speicher zugewiesen. Die Syntax entspricht den Makros. Speicher wird dabei byteweise adressiert, sodass die nächste Variable 32 Adressen weiter angelegt würde. Mithilfe „unset“ wird der Variablenname wieder freigegeben, jedoch nicht der Speicher. Jede neue Variable bekommt ein neues Stück. Der Wert kann mit „(get <name>)“ abgefragt werden und seine Adresse mit „(ref <name>)“. Als Kurzform für die Adresse kann nur der Name angegeben werden und für den Wert „@name“. „with“ wird zum Anlegen temporärer Variablen ge-

nutzt. Nachdem der Ausdruck, in dem sie verwendet wird, endet, wird sie dereferenziert. [vgl. 42]

Mit dem Operator „lll“ kann auch im Programmcode übergebener Code kompiliert werden und explizites Allokieren von Memory-Speicher ist byteweise möglich. Schließlich wird auch in LLL die Nutzung von Assemblercode unterstützt. [vgl. 42]

Besonderheiten

LLL verfügt nicht über Datentypen wie int, uint, bool oder structs. True und false werden mit 1 und 0 beschrieben und im Gegensatz zu den anderen Sprachen kommt die Präfixnotation zum Einsatz. Es existieren keine Funktionen und alles ist ein Ausdruck. Wie in Lisp kommen Klammern zur Strukturierung zum Einsatz.

LLL ist für Programmierer mit objektorientiertem oder funktionalem Hintergrund sicherlich schwieriger zu verstehen und zu lernen, besitzt jedoch die Möglichkeit, EVM Opcodes direkt zu nutzen und es gibt weniger Overhead, was zu schmalen Bytecodes führt. LLL könnte auch weniger fehleranfällig sein, da der eingebaute Funktionsumfang geringer ist, als in den anderen Sprachen. Die Sprache stellt somit lediglich eine schmale Schicht über der EVM dar, was ein gutes Verständnis bezüglich deren Arbeitsweise wünschenswert macht.

2.4.4 Bamboo

Einführung

Bamboo ist eine ebenfalls sehr neue Programmiersprache, die von Yoichi Hirai seit 2016 entwickelt wird. Sie funktioniert wie ein Zustandsautomat, es wird für jeden neuen Zustand ein Contract definiert, in den der alte übergeht. So soll die Anzahl aufrufbarer Funktionen auf die in einem bestimmten Zustand absolut notwendigen beschränkt werden. [vgl. 43]

Ein Ziel der Sprache ist es, sicherzustellen, dass klar ist, was in welcher Reihenfolge geschehen kann. Somit lassen sich mithilfe von Bamboo polymorphe Contracts erstellen. Dabei besitzen alle Formen dieselbe Adresse. In Bamboo werden keine Schleifen unterstützt, sodass jede Iteration ein separater Aufruf sein müsste. Auch ist nur eine Wertzuweisung bei Arrays möglich, statische Variablen können nur als Parameter für den Contract mitgegeben werden. [vgl. 44]

Da ein Contract mehrere Formen annehmen kann, wird in jeder Form explizit angegeben, welche Form als nächste anzunehmen ist, dies kann auch dieselbe sein. Der Compiler ist in OCaml geschrieben und kompiliert zu EVM Bytecode. [vgl. 45]

Aufbau

In Bamboo werden alle Formen eines Contracts in einer Datei definiert. Wenn sie statische Variablen benötigen, bekommen sie diese als Parameter übergeben. Lokale Variablen können angelegt werden. Die Contracts beginnen wie in Solidity mit dem „contract“-Keyword.

Im Körper werden wie in einer Switch-Case-Struktur Fälle für den Aufruf jeder vorhandenen Funktion definiert, ähnlich wie in LLL. Abhängig von der aufgerufenen Funktion wird in die Fälle eingetreten. Der letzte Fall ist auch hier der Standardfall, wenn keine andere Funktion zum Aufruf passt. Auch hier entspricht das in etwa der Fallback-Funktion. Die Fälle werden in „case(…){….}“ eingeschlossen, dabei stehen in den runden Klammern die Funktionsheader. Innerhalb der Fälle kann auf die Contract-Parameter zugegriffen werden. Am Ende jedes Falls steht ein return-Statement sowie der nächste anzunehmende Zustand mit dem Ausdruck „then become contractName(<Parameter>);“. Es erfolgt ein Übergang in diesen Contract, inklusive übergebener Parameter. Die folgende Abbildung veranschaulicht den Aufbau eines Bamboo Contracts:

```
1   contract BambooExample() {
2
3       case (bool calledDuringDemonstration()) {
4           //tue etwas
5           return (true) then become BambooExample();
6       }
7
8       case (bool isUnderstood()) {
9           if (Bedingung)
10              return (true) then become BambooKnowledge();
11          else if (andere Bedingung)
12              return (false) then become BambooNextExample();
13
14          abort;
15      }
16
17      default {
18          abort;
19      }
20  }
21
22  contract BambooKnowledge() {
23      case (void toBeCalledAfterLearning()) {
24          //tu etwas
25      }
26      //...
27  }
28
29  contract BambooNextExample() {
30      //...
31  }
32
```

Abbildung 11: Aufbau Bamboo Contract

Der Standardfall wird mit „default“ bezeichnet. Hier wird „abort“ aufgerufen, was die Ausführung abbricht, weil keine bekannte Funktion angesprochen wurde. Dieser Contract besteht also aus drei Formen.

Datentypen und Funktionen

Die Syntax der Sprache weist viele Gemeinsamkeiten mit Solidity auf. Es gibt identische Datentypen wie address, uint8, uint256, bool, bytes32 sowie zusätzlich void. Arithmetik und Vergleiche werden ebenfalls unterstützt. Auch Mappings werden angeboten, sie werden ähnlich wie in Solidity deklariert: key => value <name>; . Der Zugriff erfolgt über <name>[key].

Die einzelnen Fälle enthalten im Kopf die Funktionsheader mit ihrem Rückgabetyp. Die Logik im Körper kann zudem if-Statements enthalten. Am Ende eines Falls steht immer ein „return“ mit optionalem Rückgabewert in Klammern und es ist immer eine Angabe des nächsten Zustands (Contracts) notwendig.

Es kann auch innerhalb eines Contracts ein neuer erzeugt werden. Die Anweisung zeigt folgendes Beispiel:

```
contractName instanzName = deploy contractName(<parameter>) with value(msg) reentrance {abort;};
```

Dabei wird analog zu Soliditys msg.value hier mittels „value(msg)“ der mitgesendete Betrag an Wei referenziert. Zusätzlich wird während des Aufrufs auf Reentranz geprüft und in diesem Fall die Ausführung abgebrochen. Ein Contract kann auch als Parameter mitgegeben werden.

Neben „value(msg)“ gibt es „sender(msg)“ oder auch „now(block)“ für den Aufrufer beziehungsweise den aktuellen Zeitstempel. Die aktuelle Instanz eines Contracts wird auch in Bamboo mit „this“ referenziert. Zudem kann ein „Kontostand“ mit „balance(<address>)“ abgefragt werden. Bei externen Aufrufen sollte man die Reentranzprüfung immer mitverwenden.

Bamboo verfügt bereits über einige native Funktionen. Dazu gehören beispielsweise „keccak256()“, „selfdestruct()“ und „pre_ecdsarecover()“. Letzteres gewinnt aus einer Signatur die zugehörige Adresse. Die Funktion „iszero()“ führt eine Prüfung auf 0 durch. Es gibt in Bamboo jedoch keine Konstruktoren. Stattdessen kann man einen Einstiegscontract nutzen, der übergebene Parameter auf Anfangswerte setzt. [vgl. 89]

Ruft man darüber hinaus eine Funktion ohne Rückgabewert innerhalb eines Falls auf, so wird ihr Ergebnis „void“ zugewiesen. Dies kann beim expliziten Aufruf der default-Funktion eines externen Contracts der Fall sein. [vgl. 89]

Besonderheiten

Bamboo bietet keine Sichtbarkeits- und Zugriffsmodifikatoren. Bisher gibt es zudem keine Strings oder payable-Markierung. Imports anderer Dateien sind ebenfalls nicht möglich. Folglich können keine Bibliotheken oder Interfaces genutzt werden. Innerhalb eines Contracts können keine statischen Variablen angelegt werden. Jedoch wird nativ eine Prüfung auf Reentranz unterstützt, was einer sehr häufigen Sicherheitslücke vorbeugen kann. Bamboo ist noch in einer frühen Entwicklungsphase und daher noch nahezu gar nicht dokumentiert. Laut Entwickler können daher noch einige kleine Bugs im Bytecode vorhanden sein. Der Ansatz von Contracts als Zustandsautomat ist jedoch interessant und birgt auch Potenzial für Anwendung in Spielen. [vgl. 45]

2.4.5 eWASM

Bei eWASM handelt es sich um einen neuen Ansatz, der eine Alternative zur Ethereum Virtual Machine darstellen soll. eWASM steht dabei für „Ethereum flavoured Web Assembly“ und basiert auf dem Web Assembly (WASM), welches für Webanwendungen eingesetzt wird. Bei Web Assembly handelt es sich um einen Bytecode für eine stackbasierte virtuelle Maschine, der aus Quellcode, geschrieben in Programmiersprachen wie C, C++ und Rust, kompiliert wird. Dieser Code soll in Browsern besonders geringe Lade- und Ausführungszeiten benötigen und wird daher bereits von vielen Browseranwendungen unterstützt. Web Assembly wird seit 2017 in Browser integriert und vom „World Wide Web Consortium“ weiterentwickelt. [vgl. 65] Es soll auf hardwareseitige Fähigkeiten zurückgreifen und dadurch größere Effizienz in der Ausführung erreichen und zudem auch für Nicht-Browseranwendungen geeignet und mit JavaScript kompatibel sein. [vgl. 66] Die höhere Effizienz wird durch kleinere Datentypen erreicht, die an CPUs angepasst sind, sodass Berechnungen unkomplizierter und schneller möglich sind. [vgl. 67, Abschnitt Design Rationale]

Es ist möglich, in vielen bereits etablierten Programmiersprachen den Quellcode zu verfassen und in das WASM-Format zu kompilieren. Neben dem Bytecode kann auch ein Textformat erzeugt werden, welches lesbar das Binärformat darstellt. Dies ist hilfreich für Debugging, da so Opcode für Opcode nachvollziehbar ist. [vgl. 67, Abschnitt Text Format]

eWASM ist ein eingeschränkter Teil von WASM, entwickelt für Ethereum Contracts. So gibt es keine Fließkommazahlen. Imports sowie der Aufruf externer Funktionen sind jedoch möglich. eWASM soll das Entwickeln von Smart Contracts in etablierten Programmiersprachen wie C, C++ und Rust ermöglichen und bessere Performance und Hardwarekompatibilität bieten. [vgl. 68] Dies soll den Vorteil mitbringen, dass bereits vorhandene Entwicklungsumgebungen und Tools sowie Bibliotheken eingesetzt werden können. Zudem besteht im Umgang mit diesen Sprachen mehr Erfahrung und Expertise, als in den neuen EVM-Sprachen. Für die Ethereum Clients ist daher geplant, dass sie sowohl e-

WASM als auch die EVM unterstützen können, wobei beim Laden eines Contracts geprüft wird, ob eine eWASM-Signatur vorhanden ist. Wenn ja wird er als entsprechender Contract ausgeführt, sonst in der EVM. Wird letztere nicht unterstützt, kann ein Transcompiler den Bytecode zu eWASM umwandeln. Darüber hinaus soll das Deployment eines eWASM Contract eine Verifizierung durch einen „Sentinel Contract“ erfordern. [vgl. 68, vm_semantics.md] Eine virtuelle Maschine für eWASM Code ist „Hera“.

Die WASM Funktionalitäten werden auf Ethereum angepasst, sodass sie deterministisch sind und die Ethereum API wird über das „Ethereum Environment Interface“ bereitgestellt. So werden Funktionen wie „getAddress“, „getBalance“, „call“, „log“, „create“ und „revert“ unterstützt. Zudem gibt es Datentypen wie bytes, address, u128 und u256 für den Einsatz im Memory-Speicher. Von Web Assembly werden i32 (32 Bit Integer), i64 und i32ptr (32 Bit Zeiger) mitgeliefert. Alle Parameter und Rückgabewerte in Contracts müssen einem Web Assembly-Datentyp entsprechen. [vgl. 68, eth_interface.md]

Auch bei eWASM soll es Contracts beziehungsweise Interfaces mit festen Adressen geben, die nativ unterstützt werden. Dazu gehören der Sentinel Contract, der EVM zu eWASM Transcompiler und EVM-native Contracts, zum Beispiel „keccak256“ und „ecrecover“. [vgl. 68, system_contracts.md]

Vorteilhaft ist somit, dass eWASM, im Gegensatz zur EVM mit ihren 256 Bit Datentypen, Operationen in sehr viel weniger Schritten durchführen kann. Es sind weniger Optimierungen des Codes notwendig, so wird in einem Lauf kompiliert. Weiterhin können etablierte Sprachen mit ihren Tools und Debuggern eingesetzt werden, es sind keine neuen Sprachen erforderlich. Die Performanz soll das Einbinden von Light Clients in Browser ermöglichen und weniger High-Level Instruktionen sollen die Gasabschätzung verbessern. [vgl. 69, 70] Die Gaskosten für Operationen sind noch nicht festgelegt, jedoch wird angestrebt, dass sie geringer als für ihr EVM-Äquivalent sein sollen. [vgl. 68, backwards_compatibility.md] Für eWASM sind bislang kaum konkrete Contractbeispiele vorhanden. Parity arbeitet jedoch aktiv an der Thematik mit Rust als Programmiersprache.

3 Vergleich anhand eines ERC20 Contracts

Dieses Kapitel betrachtet ERC20-Contracts in den jeweiligen Sprachen, anhand denen die Lese- und Programmierfreundlichkeit, der Funktionsumfang, die Fehleranfälligkeit in der Programmierung und die Community-Unterstützung eingeschätzt und verglichen werden sollen. Zudem werden die Kosten für Deployment und den Aufruf der Transferfunktion ermittelt und gegenübergestellt. Zunächst jedoch soll der Standard vorgestellt werden.

3.1 Der ERC20 Tokenstandard

Der ERC20-Standard ist ein Tokenstandard im Ethereum-Netzwerk, dabei steht ERC für „Ethereum Request for Comments“. Token stellen hier handelbare Objekte beziehungsweise Werte dar, dazu gehört auch Ether.

Der Standard wurde im November 2015 von Fabian Vogelsteller und Vitalik Buterin veröffentlicht. Darin werden Token über Smart Contracts verwaltet und dort intern zwischen Nutzern transferiert sowie deren Guthaben protokolliert. Es gibt dabei festgelegte Funktionen in Form eines Interface, die implementiert werden müssen, wenn ein Token den Standard erfüllen will. Es ist prinzipiell jedem möglich, einen solchen Tokencontract zu erstellen. [vgl. 46]

Neben dem ERC20-Standard gibt es viele weitere Standards, die teilweise ebenfalls Token betreffen. Der Einsatz dieses Standards ist beliebt für Crowdfundingprojekte und „Initial Coin Offerings“ (ICOs), dem Vorabverkauf von Token, beispielsweise durch Startups, um Geld zu sammeln. [vgl. 47]

Ein ERC20-Contract protokolliert alle Tokenbesitzer mit ihrem Guthaben und definiert den vorhandenen Gesamtbetrag. Die Token basieren auf der Ethereum Blockchain, da Bitcoin durch lediglich einfache Skripte eingeschränkt ist. Der Transfer erfolgt, wie bereits erwähnt, über den Contract und nicht direkt an die Empfängeradresse. Der Smart Contract passt dann entsprechend die Guthaben an. [vgl. 48]

Ein solches Token kann selbst als Zahlungsmittel dienen oder für etwas anderes stehen, zum Beispiel Aktien oder Zertifikate. Der Standard kann in Solidity beispielsweise als Interface formuliert und in den Contract eingebunden werden. [vgl. 49] Vorteilhaft ist, dass der Standard den Austausch beziehungsweise Handel vereinheitlicht und implementierende Contracts so miteinander kompatibel macht. Die folgende Abbildung zeigt die Funktionen des Standards als Interface für Solidity.

```
contract ERC20 {

    function totalSupply() public view returns (uint256);
    function balanceOf(address who) public view returns (uint256);
    function transfer(address to, uint256 value) public returns (bool);
    function allowance(address owner, address spender) public view returns (uint256);
    function transferFrom(address from, address to, uint256 value) public returns (bool);
    function approve(address spender, uint256 value) public returns (bool);
    event Approval(address indexed owner, address indexed spender, uint256 value);
    event Transfer(address indexed from, address indexed to, uint256 value);
}
```

Abbildung 12: ERC20 Interface [51]

- Die Funktion „totalSupply“ gibt den Gesamtvorrat an Token zurück.
- „balanceOf“ fragt das Guthaben einer Adresse ab.
- „transfer“ und „transferFrom“ erlauben den Tokentransfer.
- „approve“ genehmigt einer Adresse, Token einer anderen Adresse auszugeben.
- Schließlich kann der dafür freigegebene Betrag mit „allowance“ abgefragt werden.
- Zusätzlich gibt es zwei Events, die in den Transfer- oder Approvefunktionen ausgelöst werden.

ERC20 Tokencontracts bergen jedoch auch Gefahren, dies zeigt folgendes Beispiel der „ERC20 Short Address Attack“:

Diese entsteht durch eine fehlende Prüfung der Adresslänge im Aufrufercontract bei einer Transferfunktion. Ein Angreifer kann sich eine Adresse mit zum Beispiel zwei Nullen am Ende generieren lassen. Dafür benötigt er circa $16^2 = 256$ Versuche. Nun kann er sich einen Exchange, also eine Tausch- oder Handelsplattform, suchen, die über genügend Token verfügt, beispielsweise 256 000. Hinterlegt er 1000 Token und fordert sie mit der generierten Adresse ohne Angabe der Nullen zurück, kann er sich nun 256 000 auszahlen lassen. Dies hängt damit zusammen, dass durch das Fehlen einer Prüfung die Argumente der Funktion einfach gepackt werden und das fehlende Byte hinten an die Aufrufdaten gepaddet wird. Das erste Argument dabei ist die Funktions-ID, dann folgt die Adresse und der Betrag als 256 Bit Integer. Da dieser über viele führende Nullen verfügt, werden einfach zwei davon zur Adresse genommen, die so wieder vollständig ist. Das nun beim Betrag fehlende Byte wird hinten wieder als Nullen angehängt, was zu einem Linksshift um zwei Stellen und damit zu einer Multiplikation des Betrags um 256 führt. Somit erhält der Angreifer anstelle von 1000 nun 256 000 Token. Dies geschieht unbemerkt erst nach der Prüfung des Guthabens. Eine Lösung wäre es, vor einem Transfer aufruferseitig die Größe der Aufrufdaten zu überprüfen. [vgl. 50]

Weiterhin ist es möglich, dass Token verloren gehen, wenn der Empfänger ein nicht ERC20-kompatibler Contract ist. Dieser erkennt die Transaktion nicht, wenn ihm in Form jener Token Ether zugeschickt werden. Dies soll ein anderer Standard, der ERC223 lösen, indem die Transaktion bei Inkompatibilität abgewiesen wird. [vgl. 52]

Neben dem ERC223 gibt es weitere Tokenstandards wie den ERC721. Dieser verfolgt einen ganz anderen Ansatz, bezieht er sich auf einzigartige Token, von denen es somit keine großen Mengen gibt. Dies ist mit Sammelkarten vergleichbar. Ihren Wert erhalten sie durch ihre Einzigartigkeit. Dieser Standard wurde 2017 von Dieter Shirley vorgeschlagen. Er enthält über die ERC20 Funktionalität hinaus weitere Funktionen, die den Besitz einzelner Token mithilfe von IDs Adressen zuordnen. Zuerst wurden sie im Zuge des „Cryptokitties“-Projektes von Axiom Zen eingesetzt. Sie können neben Häusern, Kunstwerken, digitalen Werten wie den Cryptokitties auch negative Werte wie Schulden oder Kredite darstellen. Durch sie können alle möglichen Arten von Informationen Wert erhalten. Es geht hierbei nicht wie bei ERC20 um die montäre Bedeutung der Token, sondern um das Token selbst, beispielsweise als Sammelwert. [vgl. 53, 54]

In den nächsten Abschnitten soll nun die Implementation des ERC20-Standards in den verschiedenen Sprachen betrachtet werden. Dabei werden im Bereich der Lesbarkeit und Programmierfreundlichkeit zum einen der Aufbau und die Modularisierungsmöglichkeiten sowie zum anderen die Quellcodelängen der Sprachen verglichen. In Bezug auf den Funktionsumfang werden Events, besondere Funktionen und vorhandene Datentypen inklusive Variablen zur Blockchaininteraktion betrachtet und es wird neben der Community-Unterstützung auf einige Kategorien eingegangen, die die Fehleranfälligkeit in der Programmierung betreffen. Dazu gehören die Standardsichtbarkeiten, native Prüffunktionen und die Namensgebung von Variablen und Funktionen.

3.2 ERC20 in Solidity

Die vollständige Implemementation [vgl. 55, 56] befindet sich in den Anlagen zu Kapitel drei.

Lese- und Programmierfreundlichkeit

1. Aufbau und Modularisierung

Der Solidity Contract besitzt durch den funktionalen Stil eine klare Struktur. Es sind immer ein Versionspragma und das Contract-Objekt erforderlich. Aufgrund der Syntaxähnlichkeit mit Typescript und Java sind die Lesbarkeit und Verständlichkeit sehr hoch. Imports ermöglichen eine Modularisierung des Quellcodes, so wird in diesem Beispiel das Interface des ERC20-Standard importiert. Man kann dadurch kleinere Codedateien erreichen und einzelne Module in anderen Kontexten wiederverwenden.

```
1  pragma solidity ^0.4.22;
2
3  import "./ERC20.sol";
4
5  /**
6   * @title ERC20 Contract mit fester Tokenzahl
7   */
8  contract ERC20Token is ERC20 {
```

Durch eine größere Anzahl an Imports kann die Lesbarkeit jedoch auch verringert werden, da mit hohem Grad der Modularisierung auch ein vermehrtes Hin- und Herspringen im Code erforderlich wird, um ihn in Gänze nachvollziehen zu können. Dies gilt ebenso für Modifier, die Solidity bietet, um beispielsweise zusätzliche Prüfungen an Funktionen anzufügen. Einerseits ist dies sehr nützlich, da keine Implementation in jeder einzelnen Funktion nötig ist. Andererseits werden Modifier im Allgemeinen an anderer Stelle definiert als eingesetzt und dementsprechend ist auch in diesem Fall ein Springen im Code erforderlich, was bei größerer Anzahl das Nachvollziehen erschwert. In vorliegendem Contract besteht dieses Problem jedoch nicht, da die Komplexität nicht sehr hoch ist. Insgesamt ist die Verständlichkeit mit Kenntnissen in der funktionalen Programmierung oder Objektorientierung hoch.

```
18      modifier onlyOwner {
19          require(msg.sender == owner);
20          _;
21      }
```

Es besteht zudem Unterstützung von Vererbung, die zusätzlich zu den Interfaces besonders die Erstellung komplexer Anwendungen erleichtert.

2. Quellcodelänge

Der Solidity Quellcode umfasst in etwa 80 Zeilen, abzüglich der Kommentare. Mit sieben Funktionen, einer Fallback-Funktion, einem Konstruktor und drei Events ist der Contract übersichtlich. Da ein Import des ERC20-Interface möglich ist, müssen die zugehörigen Events nicht neu deklariert werden, was den Code, wenn auch geringfügig, verkürzt.

Funktionsumfang

3. Events

Solidity erlaubt den Einsatz von Events. Diese können Parameter besitzen, von denen bis zu drei aus der Transaktion herausfilterbar sind. Dies ermöglicht Clients die Kenntnisnahme von beispielsweise Transfers, die sie betreffen. Besonders vorteilhaft is dies für Light Clients, die nicht die gesamte Blockchain speichern wollen, sondern nur an sie betreffenden Informationen interessiert sind. Events kommen auch in diesem Contract zum Einsatz. In Solidity können sie zudem von der Nulladresse ausgehen.

```
event Approval(address indexed owner, address indexed spender, uint256 value);
event Transfer(address indexed from, address indexed to, uint256 value);
```

4. Datentypen und Blockchainvariablen

Solidity verfügt über viele Datentypen, was für Bequemlichkeit in der Programmierung sorgt. Es gibt eine breite Auswahl an Größen für beispielsweise Ganzzahlen. Allerdings werden Aliase für Datentypen unterstützt, was zu Missverständnissen führen kann. So entspricht uint beispielsweise auch uint256. Weiterhin werden Mappings unterstützt, welche ebenfalls die Programmierung erleichtern. Sie können verschachtelt sein, was wiederum verwirren kann. Folgende Mappings kommen im ERC20-Contract zum Einsatz:

```
mapping(address => uint256) balances;
mapping (address => mapping (address => uint256)) internal allowed;
```

Typcasting ist in Solidity ebenfalls möglich, birgt aber auch Fehlerpotenzial, wenn zum Beispiel versucht wird, einen größeren in einen kleineren Typen umzuwandeln.

Schließlich ermöglichen spezielle Variablen wie „msg.sender“ oder „now“ Interaktion mit beziehungsweise den Abruf von Blockchain- und Transaktionsdaten. Dies erleichtert die Entwicklung von Smart Contracts in hohem Maße, da die Daten nicht selbst aus den Aufrufdaten extrahiert werden müssen.

5. Besondere Funktionen

Solidity verfügt über die Möglichkeit, einen Konstruktor zur Initialisierung zu definieren. Dieser kann, wie auch in diesem Contract Parameter besitzen. Er ist optional und wird beim Hochladen auf die Blockchain nicht mitdeployt. Er ist eindeutig als Konstruktor benannt, so ist keine Verwechslung mit einer gewöhnlichen Funktion möglich:

```
33      // Konstruktor: bekommt maximale Tokenanzahl übergeben und Dezimalstellen.
34      constructor(uint256 _initialTokenAmount, uint8 _decimals) public {
35          owner = msg.sender;
36          totalToken = _initialTokenAmount * 10 ** uint(_decimals);
37          balances[owner] = totalToken;
38          decimals = _decimals;
39          emit Transfer(address(0), owner, totalToken);
40      }
```

Zudem kann eine Fallback-Funktion definiert werden, die als Standardfunktion aufgerufen wird, wenn beispielsweise Ether an den Contract gesendet werden oder keine Funktion mit der aus dem Aufruf übereinstimmt. Sie kann als zahlungsempfangsfähig markiert werden und akzeptiert nur dann erhaltene Ether. Die Fallback-Funktion hat keinen Namen oder Parameter und kann nichts zurückgeben. Auch sie ist optional, bei Fehlen werden Ether abgewiesen.

```
113     // Fallback
114     function() public payable {
115         revert();
116     }
```

Fehleranfälligkeit in der Programmierung

6. Standardsichtbarkeiten

Die Standardsichtbarkeiten sind in Solidity für globale Variablen internal und für Funktionen public. Problematisch ist, dass keine Pflicht besteht, einen entsprechenden Modifikator anzugeben, um gewünschte Sichtbarkeiten explizit zu machen. Dies kann zu unerwünschten Zugriffen auf Funktionen führen, da es bei public möglich ist, von innerhalb sowie außerhalb des Contracts derartige Funktionen aufzurufen und dementsprechende Stateänderungen vorzunehmen. Somit können in dem Zusammenhang Sicherheitslücken in Contracts entstehen.

Für die Fallback-Funktion gilt als Standard external, was bedeutet, dass sie nur von außerhalb ansprechbar ist. Da sie auch die Aufgabe hat, den Erhalt von Ether zu behandeln und auf Contractaufrufe zu reagieren, ist dies in diesem Fall unproblematisch. Insgesamt ist die Auswahl an Sichtbarkeitsmodifikatoren groß, was feine Einstufungen der Sichtbarkeiten erlaubt. Jedoch birgt ein derartiger Umfang auch das Risiko, ungeeignete Modifikatoren auszuwählen, wenn die teilweise nicht allzu großen Unterschiede zwischen ihnen nicht erfasst werden.

7. Prüffunktionen

Solidity verfügt über Funktionen wie „require“ und „assert“ zur Prüfung von Bedingungen. Dabei wird im Fehlerfall ein Rollback ausgelöst. So kann sichergestellt werden, dass bei unerwünschtem Verhalten oder Eingaben nicht fortgefahren und alles rückgängig gemacht wird.

```
54    function transfer(address _to, uint256 _value) public returns (bool) {
55        require(_to != address(0));
56        require(_value <= balances[msg.sender]);
57        require(balances[_to] <= balances[_to]+_value);
```

Zudem kann ein Rollback auch explizit ausgelöst werden, wie in diesem Contract in der Fallback-Funktion, um erhaltene Ether abzuweisen. Auch Modifier lassen sich für Prüfzwecke einsetzen und können die Funktionsausführung an Bedingungen knüpfen. Dies ist besonders hilfreich, um Zugriffe auf public Funktionen zu begrenzen. Für die meisten Prüfungen, wie hier die Konsistenz von Tokenguthaben beziehungsweise Over- und Underflows, müssen in entsprechenden Funktionen selbst Maßnahmen ergriffen und gegebenenfalls durch Tests überprüft werden.

8. Namensgebung

In Solidity können Funktionen und Variablen nicht gleich benannt werden, was daher nicht zu Missverständnissen führen kann. Allerdings ist es möglich Funktionen und Operatoren zu überladen, was wiederum problematisch sein kann und Verwechslungen möglich macht. Events und Funktionen können jedoch nicht gleich benannt werden, zumindest müssen sie sich, wie in diesem Contract, um Groß- und Kleinschrei-

bung unterscheiden. Deshalb werden Events, anders als Funktionen, in der Regel mit einem Großbuchstaben begonnen.

Community – Unterstützung

Solidity verfügt über eine sehr große Community. Man findet in Foren und sozialen Netzwerken viele Hilfestellungen und Informationen. Zudem gibt es eine ausführliche Dokumentation. Die Sprache besitzt durch den hohen Funktionsumfang und die damit einhergehende Bequemlichkeit in der Entwicklung viele Anhänger. So gibt es für Solidity bereits einige Tools und Entwicklungsumgebungen wie Remix, Truffle und mehrere Testtools.

3.3 ERC20 in Vyper

Der ERC20-Contract in Vyper ist ebenfalls in den Anlagen Kapitel 3 zu finden. [vgl. 57]

Lese- und Programmierfreundlichkeit

1. Aufbau und Modularisierung

Vyper basiert auf Python und weist eine dementsprechend klare Struktur auf. Es handelt sich ebenfalls um eine funktionale Sprache, die trotz fehlender Klammern sehr gut lesbar und verständlich ist. Der ERC20 – Contract ist auch ohne Vorkenntnisse gut zu überblicken. Besonders die Funktions-Decorator heben sich ab. Vyper ermöglicht kein Einbinden von Interfaces oder Vererbung, Imports sind bislang nicht möglich. Daher muss der gesamte Code in einer Datei implementiert werden. Dies kann bei komplexeren Anwendungen zu längeren Quellcodes führen, was in diesem Contract jedoch nicht der Fall ist. Die Events des Standards müssen hier dementsprechend selbst deklariert werden. Dies erfolgt vor der Deklaration globaler Variablen:

```
3   # Events
4   Transfer: event({_from: indexed(address), _to: indexed(address), _value: uint256})
5   Approval: event({_owner: indexed(address), _spender: indexed(address), _value: uint256})
6   NewOwner: event({_newOwner: indexed(address)})
7
8   # Variablen
9   total: public(uint256)
10  decimals: public(int128)
11  balances: int128[address]
12  allowed: int128[address][address]
13  owner: public(address)
14
```

Durch die fehlende Modularisierung kann auch nicht von wiederverwendbaren Teilen profitiert werden. Auch Modifier werden nicht unterstützt, sodass jede Prüfung in ent-

sprechende Funktionen erneut eingefügt werden muss. Auch dies kann den Quellcode vergrößern, allerdings erfordert das Lesen keine Sprünge im Code.

Zudem macht die fehlende Vererbung komplexe Anwendungen schwerer realisierbar, es kann demgegenüber jedoch auch Missverständnissen vorbeugen, die mit der Zugehörigkeit geerbter Funktionen zu Oberklassen zusammenhängen. Durch den geringeren Funktionsumfang werden einerseits risikoträchtige Funktionen wie „call“ in Solidity vermieden, andererseits kann es erhöhtes Fehlerrisiko bedeuten, wenn dies mit Eigenimplementationen ausgeglichen werden soll.

2. Quellcodelänge

Der Quellcode des Vyper Contracts umfasst ebenfalls in etwa 80 Zeilen ohne Kommentare. Es gibt auch hier sieben Funktionen, drei Events und einen Konstruktor. Die Fallback-Funktion ist nicht implementiert, wird jedoch bei Fehlen automatisch generiert. Auch dieser Quellcode ist somit kurz und übersichtlich, die fehlende Unterstützung von Imports hat in diesem Fall keine besonderen Auswirkungen, lediglich die Eventdeklarationen müssen selbst zusätzlich eingefügt werden.

Funktionsumfang

3. Events

Auch Vyper erlaubt die Deklaration von Events, ebenfalls mit bis zu drei filterbaren Parametern. Sie können hier jedoch nicht von der Nulladresse ausgelöst werden und müssen vor den globalen Variablen und Funktionen deklariert werden.

4. Datentypen und Blockchainvariablen

Vyper verfügt über viele Datentypen und bietet zusätzlich Einheiten, die auch selbst definierbar sind. Es gibt weniger Ganzzahltypen, dafür werden aber auch Festkommazahlen angeboten. Die Einschränkung an Datentypen gegenüber Solidity hat den Vorteil, dass weniger Fehlerpotenzial durch eine ungeeignete Wahl besteht. Auch auf Aliase wird in diesem Zusammenhang verzichtet, sodass hierbei Missverständnisse vermieden werden. Mappings werden auch in Vyper unterstützt, die Syntax ist jedoch etwas komplizierter als in Solidity. Auch hier können sie verschachtelt sein:

```
11    balances: int128[address]
12    allowed: int128[address][address]
```

Typcasting wird in Vyper ebenfalls zwischen bestimmten Datentypen unterstützt, so kann von uint128 zu uint256 oder zu einer Festkommazahl gecastet werden. Die Sichtbarkeitsmodifikatoren umschließen hier den Datentyp, was in Bezug auf die Lesbarkeit gewöhnungsbedürftig erscheint. Dafür schaffen die Decorators an den Funktionen eine gute Übersicht über die Sichtbarkeiten:

```
30  # Fragt Guthaben eines Account ab.
31  @public
32  @constant
33  def balanceOf(_owner: address) -> uint256:
34
35      return convert(self.balances[_owner], 'uint256')
```

Analog zu Solidity sind Variablen für die Blockchaininteraktion vorhanden, die die Programmierung und insbesondere den Abruf von Block- und Transaktionsinformationen erleichtern.

5. Besondere Funktionen

Vyper bietet ebenfalls einen Konstruktor, der Parameter besitzen kann. Auch hier wird er nicht mitdeployt. Er ist ebenfalls optional und dient der Initialisierung.

```
15  #Konstruktor
16  @public
17  def __init__(_decimals: uint256, _initialSupply: uint256):
18      self.owner = msg.sender
19      self.decimals = _decimals
20      self.total =_initialSupply * convert(10, 'uint256') ** _decimals
21      self.balances[msg.sender] = convert(self.total, 'int128')
22
```

Die Fallback-Funktion ist erst seit kurzem integriert, auch sie ist optional und wird bei Fehlen generiert. Dann löst sie jedoch lediglich ein Rollback aus. Sie besitzt keine Parameter oder Rückgabe und hat einen festgelegten Namen. Auch sie kann empfangsfähig für Ether markiert werden. Sie muss zusätzlich immer als public gekennzeichnet werden. In diesem Contract wird sie automatisch generiert.

Fehleranfälligkeit in der Programmierung

6. Standardsichtbarkeiten

Vyper schreibt für jede Funktion vor, dass sie mit einem Sichtbarkeitsmodifikator versehen werden müssen, um Sichtbarkeiten explizit zu machen und Unklarheiten zu vermeiden. Es gibt hier deutlich weniger Auswahl als in Solidity, was aber die Unterscheidung erleichtert, da die Abstufungen weniger fein sind. Der Standard für die Sichtbarkeit von globalen Variablen ist hier, anders als in Solidity, private. So werden Zugriffe automatisch eingeschränkt. Die Fallback-Funktion ist immer public. Ein guter Stil wird hier somit erzwungen.

7. Prüffunktionen

Es wird in Vyper nur eine Funktion zur Prüfung von Bedingungen angeboten, die bei Fehlschlag ein Rollback auslöst. Sie wird auch in diesem Contract dafür eingesetzt:

```
26      assert msg.sender == self.owner
27      self.owner = _new
28      log.NewOwner(_new)
```

Hier besteht das Problem, dass bei komplexeren Bedingungen die Lesbarkeit ohne Klammerung verringert wird. Dies gilt ebenso für if-Statements, die so schnell unübersichtlich wirken:

```
62  @public
63  def transferFrom(_from: address, _to: address, _value: int128(uint256)) -> bool:
64
65      if _value <= self.allowed[_from][msg.sender] and \
66         (_value <= self.balances[_from] and self.balances[_to] + _value >= self.balances[_to]):
67
```

Da keine Modifier unterstützt werden, können sie nicht für Prüfzwecke eingesetzt werden, sodass Prüfungen in jede Funktion einzeln implementiert werden müssen. Auch die Zugriffe auf public Funktionen müssen derart limitiert werden. Weiterhin gibt es keine explizite Funktion für ein Rollback. Vorteilhaft ist jedoch, dass Vyper eine native Over- und Underflowprüfung bei Arrayzugriff und Arithmetik bietet, was einen häufigen Programmierfehler darstellt.

8. Namensgebung

Es ist ebenfalls in Vyper nicht gestattet, Funktionen und Variablen gleich zu benennen, zudem ist keine Überladung möglich. Dies soll Missverständnissen vorbeugen. Auch Events müssen sich in ihrer Benennung zumindest durch Groß- und Kleinschreibung von Funktionen unterscheiden. Diese Restriktionen tragen zum guten Verständnis des Quellcodes bei.

Community-Unterstützung

Vyper hat noch keine so breite Unterstützung wie Solidity, obgleich die Community wächst. Man findet bereits Informationen und Hilfestellungen in Foren und auf Blogs, wenn auch geringeren Umfangs als für Solidity. Auch eine Dokumentation ist bereits vorhanden, jedoch ist sie noch nicht vollständig. Für Vyper existieren zudem ein Onlinecompiler und Kompatibilität mit den Truffle-Entwicklungstools.

3.4 ERC20 in LLL

Der Quellcode in LLL ist vollständig in den Anlagen zu Kapitel drei zu finden. [58]

Lese- und Programmierfreundlichkeit

1. Aufbau und Modularisierung

LLL ist keine funktionale Sprache und besitzt daher keine Funktionen. Sie ähnelt Lisp und nutzt Klammern zur Strukturierung. Weitere Maßnahmen, um die Lesbarkeit zu erleichtern, muss der Entwickler beispielsweise durch Einrücken und Kommentare selbst umsetzen. Die Lesbarkeit ist geringer als in Vyper und Solidity, auch die Präfixnotation erschwert dies. So wirkt auch der ERC20-Contract komplex und eine ausführliche Kommentierung und Gliederung sind maßgeblich für das Verständnis. Anstelle von Funktionen werden alle Ausdrücke in Kontrollstrukturen wie zum Beispiel Sequenzen zusammengefasst. Ein Import von Dateien ist in LLL möglich, sodass der Code modularisierbar und einzelne Teile gegebenenfalls wiederverwendbar sein können. Die Aufgabe von Funktionen kann durch Makros übernommen werden. Dabei müssen viele Funktionalitäten, die andere Sprachen nativ bieten, selbst implementiert werden, zum Beispiel der Abgleich von Funktions-IDs mit der ID aus den Aufrufdaten:

```
100  ;; Prüft, ob gespeicherte Funktions - ID einem bekannten Hash entspricht und
101  ;; führt dann <code-body> aus.
102  ;; @param function-hash Der 4-byte Hash einer bekannten Fkt.signatur.
103  ;; @param code-body Der auszuführende Code im Falle eines Treffers.
104
105  (def 'function (function-hash code-body)
106    (when (= (mload mem-func) function-hash)
107      code-body))
108
```

Dabei besteht die Gefahr, dass durch entsprechende Eigenimplementationen jener Funktionalitäten Fehler gemacht werden, die bei nativer Unterstützung nicht auftreten können.

Makros können vergleichbar mit Modifiern eingesetzt werden, so müssen sie nur einmalig definiert werden und können anschließend beliebig oft im Contract benutzt werden. Dies kann, wie in Solidity, durch Sprünge im Code die Lesbarkeit verringern.

LLL unterstützt wie Vyper keine Vererbung und ist für Entwickler aus dem Bereich der Objektorientierung insgesamt schwerer zu verstehen. Dies hängt auch mit dem direkten Speicherzugriff und der Präfixnotation zusammen. So ist viel Einarbeitung notwendig.

Der Übersicht halber teilt man den Quellcode in einen INIT und einen CODE Abschnitt, um die Initialisierung von den eigentlichen Funktionen abzugrenzen. Letztere werden dann in einer Art Switch-Case-Struktur durchlaufen und anhand obigem Abgleich der Funktions-IDs die aufgerufene Funktionalität ermittelt.

```
199     ;; Gibt Guthaben eines Account zurück.
200     ;; @abi balanceOf(address) constant returns (uint256)
201     ;; @param owner Die Adresse des Accountbesitzers.
202     ;; @return Guthaben.
203
204     (function get-balance-of
205       (seq
206
207        (def 'owner arg1)
208
209        (return (sload (balance owner)))))
210
```

Alle Funktionalitäten werden über eine gespeicherte ID, hier die ID von get-balance-of, mit derjenigen aus den Aufrufdaten abgeglichen und bei Übereinstimmung wird die zugehörige Sequenz ausgeführt. Somit muss das funktionale Verhalten wie hier im ERC20-Contract selbst mithilfe von Makros realisiert werden.

2. Quellcodelänge

Der Quellcode des Contracts umfasst in LLL ohne Kommentare circa 210 Zeilen. Dies ist fast das Dreifache von Solidity und Vyper. Der Contract ist deutlich komplexer, da es weitaus weniger vordefinierte Funktionen gibt. Es müssen beispielsweise die Argumente eines Aufrufs selbst aus den Aufrufdaten extrahiert werden:

```
65      ;; Makros, die Funktionsargumente aus calldata laden
66
67      (def 'arg1 (calldataload 0x04))
68      (def 'arg2 (calldataload 0x24))
69      (def 'arg3 (calldataload 0x44))
```

Dementsprechend ist eine große Anzahl an Makrodefinitionen notwendig, um die gewünschten Funktionen nachzubilden. Gleiches gilt für Datentypen, die nicht nativ angeboten werden. Aus diesem Grund beginnt im ERC20-Contract das eigentliche Programm erst in Zeile 154. Dies ist jedoch notwendig für die Lesbarkeit und ermöglicht es, den Quellcode insgesamt funktionaler aussehen zu lassen. Die Makros ließen sich alternativ in eine andere Datei auslagern und importieren.

Funktionsumfang

3. Events

LLL unterstützt ebenfalls das Auslösen von Events. Dabei können bis zu drei Parameter herausgefiltert werden. Die Events sind hier sehr nah an der EVM, sodass beim Auslösen die Event-ID selbst mit übergeben werden muss. In Solidity und Vyper ist dies nicht notwendig.

```
;; Event - Transfer(address,address,uint256)
(event3 transfer-event-id (caller) to value)))
```

4. Datentypen und Blockchainvariablen

LLL verfügt über kaum Datentypen, alle werden als 32 Byte Word angesprochen, gegebenenfalls mit Padding. Neben Ganzzahlen (auch Hexadezimal) können dies auch Bytesequenzen oder Literale sein. Die Sprache bietet somit diesbezüglich weniger Bequemlichkeiten. So gibt es beispielsweise keine Wahrheitswerte, diese müssen selbst über Makros umgesetzt werden:

```
18      ;; Booleans
19      (def 'false 0)
20      (def 'true  1)
```

LLL ermöglicht, anders als die übrigen Sprachen, den Einsatz von Operatoren zur Speicherverwaltung, zum Beispiel „sstore“ und „mstore“. Mit diesen werden Parameter oder globale Werte, aber auch temporäre lokale Daten im Speicher abgelegt oder geladen:

```
;; Das Aufruferguthaben. In Memory speichern.
(mstore scratch0 (sload (balance (caller))))
```

EVM Opcodes sind zudem direkt einsetzbar, auch für den Abruf von Blockchain- und Transaktionsdaten, wie beispielsweise „callvalue“:

```
148     (def 'not-payable
149       (when (callvalue) (revert)))
150
```

Der direkte Speicherzugriff und Einsatz von EVM Opcodes bergen jedoch auch Gefahren, da besonders bei Entwicklern mit wenig Erfahrung in Low-Level Sprachen größeres Fehlerpotenzial besteht. Viele Mechanismen müssen selbst implementiert werden, so gibt es beispielsweise auch keine Mappings. In diesem Contract wird dies gelöst, indem an die Speicheradresse im Storage, die der Adresse des Tokenbesitzers entspricht, das Guthaben abgelegt wird. Es kann mithilfe des sload Operators aus dem Speicher abgerufen werden.

```
55      ;; Guthaben sind gespeichert bei s[owner_addr].
56      (def 'balance (address) address)
```

LLL bietet kein Casten zwischen Datentypen an, ebenso gibt es keine Aliase. Somit können damit zusammenhängende Missverständnisse ausgeschlossen werden.

5. Besondere Funktionen

Da es keine Funktionen in LLL gibt, sind auch Konstruktor und Fallback-Funktion im engeren Sinne nicht vorhanden. Für Initialisierungen nutzt man den Contractbeginn, der üblicherweise als INIT-Abschnitt vom Hauptteil abgegrenzt wird. Der Hauptteil ist in die „(returnlll …)“ Anweisung eingefasst. Auch in LLL wird der davor liegende Initialisierungsabschnitt nicht mit deployt. Er ist ebenso wenig Pflicht wie in Solidity und Vy-

per. In diesem Contract werden im genannten Abschnitt dem Contractbesitzer zu Beginn alle Token zugewiesen:

```
153   ;; ------------------------------------------------------------------------
154   ;; INITIALISIERUNG
155   ;;
156   ;; Alle Token werden anfangs dem Contractbesitzer zugeschrieben.
157
158   (sstore (balance (caller)) token-supply)
```

Da der Contract in Cases durchlaufen wird, nutzt man als Fallback den letzten Case, der nur erreicht wird, wenn die Aufruf-ID mit keiner Funktion übereinstimmt. Im ERC20-Contract wird in diesem Fall die Ausführung abgebrochen:

```
327       ;; ------------------------------------------------------------------------
328       ;; Fallback: Keine Funktionen passten zur gegebenen ID.
329
330       (revert)))
331   )
```

Standardmäßig ist in LLL alles „payable“, was bedeutet, man muss entsprechend kennzeichnen, wo dies nicht gewünscht ist. Dies ist mithilfe eines Makros möglich und kann bereits zu Beginn des CODE-Abschnittes für den gesamten Contract angegeben werden.

```
160   ;; ------------------------------------------------------------------------
161   ;; CONTRACT CODE
162
163   (returnlll
164     (seq not-payable uses-functions
165
```

Fehleranfälligkeit in der Programmierung

6. Standardsichtbarkeiten

LLL besitzt keine Sichtbarkeitsmodifikatoren und somit keine Möglichkeiten, die Sichtbarkeiten beziehungsweise Zugriffe einzuschränken. Dementsprechend muss alles als öffentlich betrachtet werden, was Sicherheitsrisiken bedeuten kann. Funktionalitäten, die privat beziehungsweise nur Contractintern genutzt werden sollen, sollten nicht in die ABI aufgenommen werden beziehungsweise in den Abgleich der Funktions-IDs in den Cases.

7. Prüffunktionen

Nativ besitzt LLL keine Makros zur Bedingungsprüfung, diese müssen selbst implementiert werden. Man hat dafür lediglich Kontrollstrukturen wie beispielsweise if, when, unless oder while zur Verfügung. So müssen auch zur Zugriffsprüfung entsprechend Makros implementiert und an benötigter Stelle eingesetzt werden. Prüfungen

erfordern in LLL mehr „Nachdenken“, da selbst einfache Strukturen, wie die Prüfung der Parameteranzahl eigens umgesetzt werden müssen. Dies gilt ebenfalls für das Extrahieren der Parameter aus den Aufrufdaten. Hierbei besteht die Gefahr, solche Prüfmechanismen zu vergessen, wenn man aus anderen Sprachen kein selbstständiges Testen dieses Umfangs gewohnt ist.

```
120  ;; ------------------------------------------------------------------------
121  ;; Stellt richtige Argumentzahl bei Aufrufen sicher. Schutz vor "ERC20
122  ;; short address attack". Für nicht-constant Funktionen.
123
124  (def 'has-one-arg    (unless (= 0x24 (calldatasize)) (revert)))
125  (def 'has-two-args   (unless (= 0x44 (calldatasize)) (revert)))
126  (def 'has-three-args (unless (= 0x64 (calldatasize)) (revert)))
```

Es ist jedoch möglich explizit einen Abbruch der Ausführung auszulösen. Dafür besitzt LLL nativ das Makro „panic“, welches demnächst durch „revert“ abgelöst wird. In diesem Contract erzeugt das revert-Makro einen invaliden Opcode:

```
79   (def 'revert () (revert 0 0))
80
```

8. Namensgebung

Es ist möglich, in LLL Makros zu überladen. Sie müssen sich lediglich in ihren Parametern unterscheiden. Auch native Operatoren und Makros können überladen werden. Dies kann die Lesbarkeit und Verständlichkeit jedoch stark verringern und zu Fehlern durch Verwechslungen führen. Makrokonstanten, die einen Wert repräsentieren, können jedoch nicht gleich benannt werden. Events werden wie Funktionen über ihre ID referenziert. Da diese als konstante Makros definiert werden, können sie nicht gleich benannt werden. Die IDs unterscheiden sich, da sie aus den Hashwerten der Funktonsköpfe beziehungsweise Eventdeklarationen gebildet werden:

```
29   ;; Vorberechnete Funktions - IDs.
30   (def 'get-name         0x06fdde03) ; name()
31   (def 'get-symbol       0x95d89b41) ; symbol()
32   (def 'get-decimals     0x313ce567) ; decimals()
33   (def 'get-total-supply 0x18160ddd) ; totalSupply()
34   (def 'get-balance-of   0x70a08231) ; balanceOf(address)
35   (def 'transfer         0xa9059cbb) ; transfer(address,uint256)
36   (def 'transfer-from    0x23b872dd) ; transferFrom(address,address,uint256)
37   (def 'approve          0x095ea7b3) ; approve(address,uint256)
38   (def 'get-allowance    0xdd62ed3e) ; allowance(address,address)
39
40   ;; Event IDs
41   (def 'transfer-event-id ; Transfer(address,address,uint256)
42     0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef)
43
44   (def 'approval-event-id ; Approval(address,address,uint256)
45     0x8c5be1e5ebec7d5bd14f71427d1e84f3dd0314c0f7b2291e5b200ac8c7c3b925)
```

Community-Unterstützung

Die Community in LLL ist deutlich kleiner als für Solidity. Die Sprache ist in den vergangenen Jahren zunehmend in den Hintergrund gerückt, da sie wenig Bequemlichkeiten und viel Fehlerpotenzial bietet. Man findet nur sehr vereinzelt Informationen in Foren und auf Blogs, wobei Daniel Ellison von ConsenSys die meisten Beiträge und Anleitungen liefert. Auch Tools sind kaum vorhanden, es gibt keine spezielle Entwicklungsumgebung. Eine Dokumentation existiert, ist jedoch bislang unvollständig.

3.5 ERC20 in Bamboo

Der Bamboo Contract [vgl. 59] ist in den Anlagen zum Kapitel drei in Gänze zu finden.

Lese- und Programmierfreundlichkeit

1. Aufbau und Modularisierung

Bamboo ist eine funktionale Sprache, die demzufolge über Funktionen klar strukturiert ist. Es gibt wie in Solidity Contract-Objekte und die Syntax ähnelt ebenso Solidity. Die Lesbarkeit ist somit hoch und erleichert den Einstieg. Bisher werden in Bamboo keine Imports oder Vererbung unterstützt, sodass keine Modularisierung beziehungsweise Aufteilung auf mehrere Dateien möglich ist. Dadurch können die Quellcodes bei komplexeren Anwendungen sehr groß werden. Üblicherweise werden innerhalb einer Datei mehrere Contracts definiert, die Zustände repräsentieren und ineinander übergehen können. Sie belegen dabei alle dieselbe Adresse. Die Funktionen werden vergleichbar mit LLL in Cases definiert und können in jeder Contractform unterschiedlich sein. Es gibt keine globalen Variablen, sie können ledigleich als Contractparameter übergeben werden.

```
18  contract Token
19  (uint256 totalSupply
20  ,address => uint256 balances
21  ,address => address => uint256 allowances
22  ,address owner
23  )
24  {
```

Modifier werden nicht unterstützt, sodass wie in Vyper die Funktionalität in jede Funktion implementiert werden muss. Dies erleichtert das Lesen des Codes, da nicht im Quelltext gesprungen werden muss. Bei komplexen Anwendungen kann die Lesbarkeit durch die Contractübergänge erschwert werden, wenn beispielsweise ein Übergang von einem Contract weit unten im Quellcode in einen Contract weit oben erfolgt. Durch die geringe Komplexität ist die ERC20-Implementation jedoch nicht betroffen.

2. Quellcodelänge

Mit etwa 90 Zeilen ohne Kommentare ist der Quellcode in Bamboo nur geringfügig länger als in Solidity und Vyper. Damit ist er gut überschaubar. Der Contract besteht aus zwei Formen, einer zur Initialisierung und einer für die Hauptfunktionalität. Trotz fehlender Modularisierung ist der Code nicht deutlich umfangreicher. Neben den sieben Funktionen gibt es drei Events und in jeder Contractform einen Fallback. Als Konstruktor dient der Initialisierungscontract.

Funktionsumfang

3. Events

Events werden in Bamboo analog zu Solidity deklariert. Man hat ebenso die Möglichkeit, bis zu drei Parameter herauszufiltern, bei Angabe des entsprechenden Keywords. Sie können derzeit nicht von der Nulladresse ausgehen.

```
14   event Transfer(address indexed _from, address indexed _to, uint256 _amount);
15   event Approval(address indexed _owner, address indexed _spender, uint256 _value);
16   event NewOwner(address indexed _new);
```

4. Datentypen und Blockchainvariablen

Bamboo besitzt einige Datentypen analog zu Solidity, insgesamt jedoch etwas weniger. So gibt es bislang keine Strings und weniger Ganzzahltypen. Damit besteht allerdings weniger Risiko einer ungeeigneten Auswahl, denn die Abstufungen zwischen ihnen sind größer als in Solidity. Mappings sind ebenfalls vorhanden und können zudem verschachtelt werden. Ihre Syntax ist leicht verständlich.

```
,address => uint256 balances
,address => address => uint256 allowances
```

Die Blockchaininteraktion ist mit Variablen wie „sender(msg)“ ebenfalls möglich.

Bisher werden keine Aliase für Datentypen unterstützt, sodass damit einhergehende Missverständnisse nicht auftreten können. Gleiches gilt für Typcasting.

5. Besondere Funktionen

In Bamboo findet man keinen Konstruktor an sich. Stattdessen definiert man dafür einen Initialisierungscontract, der auch Parameter besitzen kann. Dieser kann dann nach seiner Ausführung in den Hauptcontract übergehen und ist nicht mehr aufrufbar. Hier wird er somit als Contract mitdeployt und ändert erst nach erstem Aufruf seine Form. Dieser Contract heißt hier „PreToken“. Ein solcher Contract ist optional.

```
contract PreToken
(uint256 totalSupply
,address => uint256 balances
,address => address => uint256 allowances
)
{
    default
    {
        balances[sender(msg)] = totalSupply;
        return then become Token(totalSupply, balances, allowances, sender(msg));
    }
}
```

Da die Funktionen in Bamboo in Cases definiert werden, gibt es für den Fallback einen „default“-Case. Er wird üblicherweise als letzter Fall definiert und kommt bei explizitem Aufruf oder bei nicht vorhandener Übereinstimmung des Aufrufes mit einer anderen Funktion zum Einsatz. Dieser Case besitzt zudem keine Parameter. Fehlt er, wird bei Erreichen des Contractendes eine Ausnahme ausgelöst. Eine Markierung vergleichbar mit payable in Solidity und Vyper gibt es wie in LLL nicht. Hier ist unklar, wie sich eine Funktion bei Aufruf mit Ether verhält. Bei direktem Aufruf des Contracts wird zumindest der Fallback ausgeführt.

In obigem Codeauszug des ERC20-Contracts ist die Initialisierung im Fallback-Case definiert. Sie wird bei erstem Aufruf des Contracts durchgeführt und anschließend wird in den Hauptcontract übergegangen.

Fehleranfälligkeit in der Programmierung

6. Standardsichtbarkeiten

Es sind analog zu LLL keine Sichtbarkeitsmodifikatoren vorhanden. Deshalb muss davon ausgegangen werden, dass alle Funktionen öffentlich ansprechbar sind. Dies kann problematisch sein, wenn Funktionen enthalten sind, die nicht jeder Aufrufer ausführen können soll. Dementsprechend müssen Zugriffe durch Prüfungen limitiert werden.

7. Prüffunktionen

Für Prüfzwecke gibt es in Bamboo keine Modifier, um die Funktionen ergänzt werden können. Prüfungen müssen deshalb in jede Funktion neu implementiert werden. Es gibt zudem keine Funktionen zur Bedingungsprüfung, lediglich if-Statements können dafür eingesetzt werden. Schleifen werden nicht unterstützt.

```
    case(bool transferFrom(address _from, address _to, uint256 _amount))
    {
        if (balances[_from] < _amount) abort;
        if (allowances[_from][sender(msg)] < _amount) abort;
```

Das explizite Auslösen von Abbrüchen ist in Bamboo möglich, zusätzlich kann eine native Reentranzprüfung bei externen Aufrufen eingesetzt werden, was in diesem Contract jedoch nicht notwendig ist.

8. Namensgebung

In Bamboo ist es möglich, Funktionen und Variablen gleich zu benennen, so ist dies im Tokencontract beispielsweise bei der Funktion und der gleichnamigen Variable „totalSupply“ der Fall. Bei nicht so leicht überschaubaren Contracts kann dies die Verständlichkeit erschweren, besonders, falls zukünftig Funktionsvariablen unterstützt würden.

```
25      case(uint256 totalSupply())
26      {
27          return totalSupply then become Token(totalSupply, balances, allowances, owner);
28      }
```

Für Events gilt auch hier, dass sie sich zumindest in Groß- und Kleinschreibung von Funktionen unterscheiden müssen. Zudem werden sie nur innerhalb von Cases ausgelöst und nicht wie Funktionen als Case definiert, sodass Verwechslungen unwahrscheinlich sind. Auch Variablen können keine identischen Namen haben, selbst wenn sie von unterschiedlichem Typ sind.

Überladung von Funktionen wird nicht unterstützt, gleichnamige Funktionen können nur in unterschiedlichen Contractformen definiert werden. Dies beugt Missverständnissen vor.

Community-Unterstützung

Bamboo besitzt unter den betrachteten Sprachen die geringste Community-Unterstützung. Man findet in Foren und auf Blogs nahezu keine Hilfestellungen oder Anleitungen, lediglich über den Entwickler selbst. Dieser stellt jedoch einige Beispiele zur Verfügung, die das Verständnis erleichtern. Zudem ist bislang keine Dokumentation vorhanden, was wahrscheinlich mit der Neuheit der Sprache zusammenhängt. Dementsprechend ist Bamboo auch noch nicht sehr weit verbreitet. Es gibt bisher noch keine Entwicklungsumgebungen oder Tools speziell für die Erstellung von Bamboo Contracts.Trotz geringer Unterstützung ist das Prinzip der Sprache gut verständlich und der Quellcode ist zum Großteil selbsterklärend.

3.6 Deployment: Codegrößen und Kosten im Vergleich

Für das Deployment von Smart Contracts gibt es im Allgemeinen verschiedene Möglichkeiten. Im Folgenden soll dies in den jeweiligen Sprachen vorgestellt und die Bytecodegrößen sowie Deploymentkosten ermittelt werden.

Solidity

Eine sehr nutzerfreundliche Methode zum deployen eines Contracts besteht in Solidity in der Verwendung der Remix IDE. Sie ermöglicht das Kompilieren, Debuggen und Deployment von Contracts und liefert viele nützliche Informationen. Sie kann mit einer lokalen Blockchain oder beispielsweise Metamask, als Brücke zum Ethereum-Netzwerk, verbunden werden. Die folgende Abbildung zeigt die Remix IDE:



Abbildung 13: Remix DIE

Links kann man alle geöffneten Dateien sehen, in der Mitte hat man die Quellcodedateien und rechts weitere Optionen zum kompilieren, ausführen oder auch debuggen von Contracts. In diesem Fall wurde die Remix IDE in der „Environment“ Einstellung mit dem Localhost und dem Ganache Client verbunden. Dieser läuft auf Port 8545 und simuliert eine Blockchain. Ganache ist konfigurierbar und ein großer Vorteil ist, dass Transaktionen sofort gemined werden und es daher keine Wartezeiten gibt. Daher eignet es sich für erste Integrationstests eines Smart Contracts. Es werden nun beim Start alle verfügbaren Accounts angezeigt. Dies umfasst zusätzlich die privaten Schlüssel und den „Mnemonic“-Satz, der die Wiederherstellung der Accounts in einer Wallet ermöglicht. Dies zeigt folgende Abbildung:

```
$ ganache-cli
Ganache CLI v6.0.3 (ganache-core: 2.0.2)

Available Accounts
==================
(0) 0x168210fffd6473602edb81b76cc105914cee5eb0
(1) 0x7f8c16e077156617ec91f2c36e8619b634fb57a2
(2) 0x821f82dfc55bb477ad0b59dc2d907eb138a3546d
(3) 0x04eacf1afd67b17861f17fbf1d5768d8ba1a6ef8
(4) 0x59bf2c4018de745f62cd216f2af8e44e6fdf65bd
(5) 0x093dbbe970e66d6e01eae5199d860097ad432edc
(6) 0x0cd9b8bb4bc00c452662d8de699f13764cd6b4a8
(7) 0xb1e165605bee6bd74424cca7fc53938104b9af9c
(8) 0x512ccfb9e232d1ba2a3048006650143fb7ac0040
(9) 0x1ed171844883cb7cff938ee5846689a7cefb0dfa

Private Keys
==================
(0) 81b6e740bbfe0a1c7d812f6bb118e52f3dfe3c7bdc46add420a3b8037438f4cc
(1) 79f84aea9221b34d0715be9abb9f873b2aff80df2d551f5e5d95967bb1b2da6a
(2) 6d7a943dff14a89d193148f5a54a0187e0dd2f948dc73764089dfe69ddc34720
(3) 048d4b4e502e75083e93c54b48d85a23dbfd0e63c72b44e6ca8a74cd2f6b28da
(4) febef4f44090a107c06b53d16e1c2587fc4511f2a59bf67b88703b89b6e151aa
(5) 37b793e613f845a7bcdfc9049656eb7c39f6f71e9ad38a8b7df4bb785c66d039
(6) 73952d1646b70d841272b762a793fbd6923bda3690545a5777993744e8284a89
(7) 113184f9fb0c68166ff50cd56948e9d51f67050516ba082ba6e324f4a46dd62f
(8) 0a5689bf81d5c7f20a0a4c086ae63cc6191cd3a732afa22fc263735410c57640
(9) bd04ba77e94ad282302bd9a8cab418260e9019279708e92ee541cb91cb0aa01d

HD Wallet
==================
Mnemonic:      sunset battle bulb distance nuclear say prize elevator mask still
 notable grant
Base HD Path:  m/44'/60'/0'/0/{account_index}

Listening on localhost:8545
```

Abbildung 14: Ganache

Im „Compile“-Tab der Remix IDE kann der Contract kompiliert werden, wobei auch Warnungen oder Fehler angezeigt werden. Die Option „Details“ zeigt unter anderem den Deploymentbytecode, die ABI, die Funktionshashes und die Gaskostenabschätzung an.

Die ABI ist hierbei eine Schnittstelle, die ein Standard zur Interaktion mit Smart Contracts auf der Blockchain ist. Sie enthält im JSON-Format kodiert alle Funktionen und Events, inklusive Sichtbarkeiten sowie Parametern. Auch die Fallback und der Konstruktor sind vorhanden. In einem Array sind zu jedem Event und jeder Funktion, die nicht private ist, Typ, Name, Parameter mit deren Typ und Name sowie Bestandteilen bei Arrays, Ausgabeparameter sowie eine Angabe ob sie payable oder constant ist und wie sie den State beeinflusst zu finden. Events enthalten zudem Informationen über „indexed“-Argumente, aber keine Ausgabeparameter oder payable-Markierung. [vgl. 60] Abbildung 15 zeigt einen Auszug der ABI des Solidity Contracts.

```
{
	"constant": false,
	"inputs": [
		{
			"name": "_to",
			"type": "address"
		},
		{
			"name": "_value",
			"type": "uint256"
		}
	],
	"name": "transfer",
	"outputs": [
		{
			"name": "",
			"type": "bool"
		}
	],
	"payable": false,
	"stateMutability": "nonpayable",
	"type": "function"
},
```

Abbildung 15: ABI

In der Remix Entwicklungsumgebung kann der Contract nach erfolgreichem Kompilieren im „Run“-Tab mit „Create“ deployed werden. Die Transaktion wird dann im Terminal und der Remix IDE angezeigt. Nach dem Deployment sind zudem die Funktionen im rechten Teil sichtbar.

```
Transaction: 0x85a3e950431adac08fe9f104cd32ad215ed9197d5bd7aec93a4c2f6d8cdad186
Contract created: 0xc2de56870a7edcce529873c96f20ac62ff21715b
Gas usage: 1052085
Block Number: 1
Block Time: Fri Jun 08 2018 08:05:56 GMT+0200 (Mitteleuropäische Sommerzeit)
```

Abbildung 16: Ganache Ausgabe

Beim Deployment wurden, laut Remix und Ganache, 1 052 085 Gas verbraucht, ein Blick auf die Schätzung im „Details“-Bereich zeigt 763 955 Gas, wobei 85 955 für die Transaktion und 678 000 für die Codespeicherung auf der Blockchain erwartet wurden. Es sind somit 288 130 Gaseinheiten zu wenig geschätzt worden.

GASESTIMATES

```
{
    "Creation": {
        "codeDepositCost": "678000",
        "executionCost": "85955",
        "totalCost": "763955"
    },
    "External": {
        "": "245",
        "allowance(address,address)": "838",
        "approve(address,uint256)": "22331",
        "balanceOf(address)": "625",
        "changeOwner(address)": "22149",
        "owner()": "580",
        "totalSupply()": "424",
        "transfer(address,uint256)": "44051",
        "transferFrom(address,address,uint256)": "64991"
    }
}
```

Abbildung 17: Gasschätzung Remix

Die Aufrufe werden im Vergleich dazu deutlich günstiger geschätzt. Führt man nun in Remix die Transferfunktion zwischen zwei Ganache-Accounts aus, so zeigt die Ausgabe einen Gasverbrauch von 51 923 Gas, geschätzt wurden von Remix 44 051 Gas. Auch hier ist die Differenz mit über 7 000 Gas recht groß.

```
transact to ERC20Token.transfer pending ...

▾[block:2 txIndex:0]  from:0x168...e5eb0 to:ERC20Token.transfer(address,uint256) 0xc2d...1715b value:0 wei
data:0xa90...00003 logs:1 hash:0x3a3...4b900
```

status	1
from	0x168210fffd6473602edb81b76cc105914cee5eb0
to	ERC20Token.transfer(address,uint256) 0xc2de56870a7edcce529873c96f20ac62ff21715b
gas	51923 gas
transaction cost	51923 gas
input	0xa9059cbb0000000000000000000000007f8c16e077156617ec91f2c36e8619b634fb57a20000000000000000000000000000000000000000000000000000000000000003

Abbildung 18: Transferdetails Solidity

Dies zeigt aber auch, dass das Deployment des Contract die teuerste Interaktion ist, zumal für jede benötigte Gaseinheit bezahlt werden muss. Der Aufruf von Funktionen ist kostengünstiger und es verursachen nur diejenigen Kosten, die den State verändern, beispielsweise totalSupply und balanceOf lesen nur aus dem Contract, daher haben sie keine teuren Stateänderungen zur Folge.

Zum Zeitpunkt der Ermittlung liegt der sichere, möglichst geringe Gaspreis mit einer Aufnahme in einen Block in 30 Minuten bei 4 GWei und der Durchschnitt mit Aufnahme in weniger als fünf Minuten bei 5GWei. Der empfohlene Betrag für eine Aufnahme in weniger als zwei Minuten liegt bei 12GWei. [vgl. 10]

Dies bedeutet für das Deployment Kosten von zurzeit 2,15€ als sicherer Mindestbetrag, 2,69€ als durchschnittlicher Betrag und 6,46€ als höchster empfohlener Wert. Für eine Transaktion sind es somit 0,11€ mit 4Gwei oder 0,13€ mit 5GWei beziehungsweise 0,32€ als Kosten mit 12 GWei. [62]

Es lässt sich mithilfe von Remix die Zusammensetzung des verbrauchten Gas sehr gut ermitteln. Die EVM zählt von der Obergrenze beim deployen runter, um den genauen noch verfügbaren Wert zu jeder Zeit vorliegen zu haben. Bei der Contracterzeugung gibt es Gas, was immer verbraucht wird. Dazu gehören 21 000 Gas für die Transaktion an sich sowie 32 000 für die Contracterzeugung, zusammen 53 000 Gas. Dazu kommt Gas für die Eingabedaten aus der Transaktion: der Input kann in Remix eingesehen werden. Hier sind dies 7 763 Hexadezimalzeichen, was 3880 Byte entspricht. Nun müssen Null- und Nicht-Nullbytes unterschiedlich gewertet werden, was die Ermittlung ihrer Anzahl erfordert. Dazu kommt ein Python-Skript zum Einsatz, zu finden in den Anlagen Kapitel drei, welches die Bytes entsprechend zählt. [63] Es ergibt für diesen Contract 190 Nullbytes und 3690, die nicht Null sind, insgesamt also 3 880. Ein Nullbyte kostet 4 Gas und ein anderes 68 Gas, so ergeben sich für die Eingabedaten Kosten von 190 * 4 + 3690 * 68 = 251 680 Gas. [vgl. 61] Inklusive der 53 000 zuvor beschriebenen Gas sind es 304 680, die bereits ohne Konstruktorausführung verbraucht werden. Von den 1 052 085 Gas sind somit noch 747 045 übrig. Neben dem Konstruktor muss auch noch für die Codespeicherung auf der Blockchain bezahlt werden. Dies lässt sich über 200 Gas/Byte * |Laufzeitbytecode| berechnen. Der Code lässt sich aus den Details in Remix entnehmen und zum Zählen der Bytes in das Skript einfügen. Das Ergebnis sind insgesamt 3 390 Bytes, somit nimmt der Konstruktor 490 Bytes ein. Also kostet die Codespeicherung 200 * 3 390 = 678 000 Gas. Dies macht den größten Teil aus. Zieht man dies nun von den verbliebenen 747 045 Gas ab, bleiben 69 405, die für den Konstruktor verbraucht werden. [vgl. 61] Insgesamt ergibt sich also folgendes:

1 052 085 Verbrauch – 53 000 Transaktions- und Contracterzeugungskosten – 251 680 Inputs – 678 000 Codespeicherung – 69 405 Konstruktor. Dies zeigt, dass Speicher auf der Blockchain mit Abstand am teuersten ist.

Vyper

In Vyper gibt es bisher keine Entwicklungsumgebung wie Remix, jedoch einen Online-Compiler, der Bytecode und ABI ausgeben kann. Dieser kann zum Kompilieren verwendet werden, das Deployment erfolgt über das Terminal mithilfe von „Node“ und der „web3“-Bibliothek.

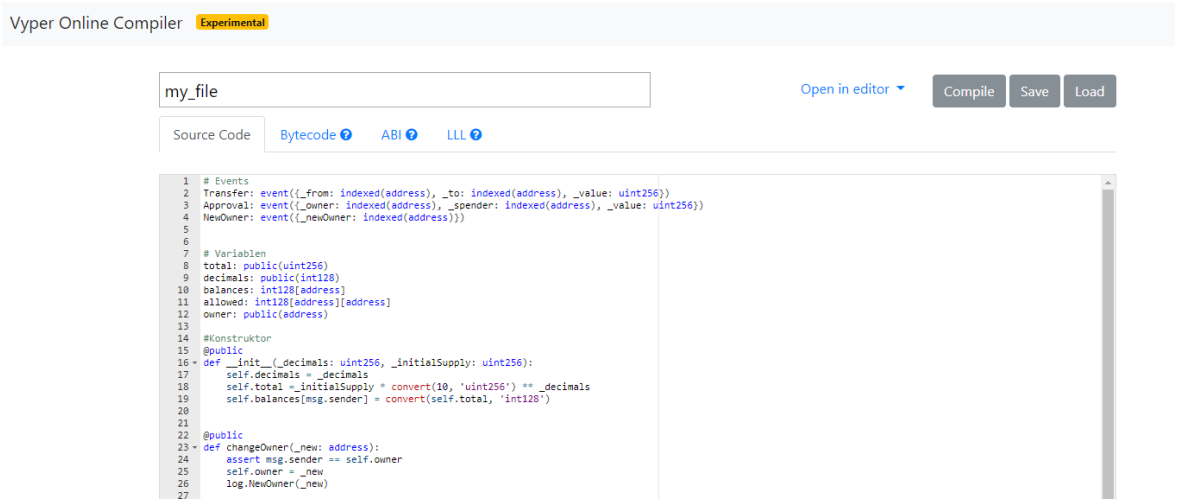


Abbildung 19: Vyper Onlinecompiler

Folgende Abbildung zeigt den Befehlsablauf. Das erste Terminalfenster dient hierbei dem Start des Ganache-Clients und muss dann im Hintergrund geöffnet bleiben. Alle anderen Befehle werden, in Javascript, im zweiten Fenster ausgeführt. Der Bytecode und die ABI können aus dem Onlinecompiler kopiert und eingefügt werden. Das Speichern der Ganache-Accounts in der Variable erfolgt asynchron, da die Blockchain Zeit zum Antworten benötigt. Dies gilt gleichermaßen für Transaktionen.

1. **Terminalfenster:**

- ganache-cli

2. **Terminalfenster:**

- node
- var Web3 = require('web3');
- var web3 = new Web3('http://localhost:8545');
- var ERC = new web3.eth.Contract(abi);
- var bin = '*Bytecode*';
- var abi = *ABI*;
- var accounts;

Accounts speichern:

- web3.eth.getAccounts().then(accts => {accounts = accts});

Deployment und Gasschätzung:

- var erc = ERC.deploy({data: bin, arguments: [18, 1000]}).send({from: accounts[0], gas: 3000000}).then(console.log);
- ERC.deploy({data: bin, arguments: [18, 1000]}).estimateGas(function(err, gas){console.log(gas);});

Setzen der Adresse und Funktionsaufrufe:

- ERC.options.address = '0xb0b7f3342ae5f11714265464be3ef92685f5497f';
- ERC.methods.totalSupply().call().then(console.log);
- ERC.methods.transfer(accounts[1],3).send({from: accounts[0]}).then(console.log);
- ERC.methods.transfer(accounts[1],3).estimateGas(function(err, gas){console.log(gas);});

Abbildung 20: Ablauf Deployment Vyper

Die web3-Funktion „estimateGas“ ermöglicht eine Abschätzung des Gasverbrauchs von Funktionen. Dabei entspricht die Schätzung für das Deployment dem Verbrauch: 605 047 Gas, das sind in etwa 400 000 weniger als Solidity. Ein Transfer wird auf 36 673 Gas geschätzt, tatsächlich werden jedoch 51 673 beim Aufruf durch Ganache angegeben, fast genauso viel wie in Solidity. Die Schätzung liegt hier um circa 15 000 Gas daneben. Die Besonderheit in Vyper ist, dass die ABI nach dem Kompilieren ebenfalls für jede Funktion einen geschätzen Gasverbrauch enthält. Im vorigen Kapitel wurde bereits angesprochen, dass Vyper für alle Funktionen eine Gasobergrenze ermittelbar machen will. Dort wird für einen Transfer als obere Grenze 74 586 geschätzt, was über dem tatsächlichen Verbrauch liegt. Diese Schätzung ist der ersteren jedoch vorzuziehen, da ein höherer Wert sicherer ist. So treten weniger „Out-Of-Gas“-Probleme auf.

Das Pythonskript, zu finden in den Anlagen Teil 3, ergibt für den Contract hier 2 198 Bytes, was deutlich weniger ist als in Solidity. Der kleinere Bytecode kann mit den höheren Restriktionen der Sprache zusammenhängen. Für das Deployment fallen daher mit 4 GWei 1,24€ an, mit 5 GWei 1,55€ und mit 12 GWei 3,71€, was fast die Hälfte von Solidity ist. Dabei muss beachtet werden, dass diese Preise starken Schwankungen unterliegen können, da der Etherpreis immer wieder variiert. Die Transferkosten lägen hier bei 0,11€, 0,13€ oder 0,32€. Sie entsprechen den Kosten in Solidity. Da Vyper ebenfalls EVM-

basiert ist, ist die Zusammensetzung der Deploymentkosten gleich: die Transaktion an sich, die Contracterzeugung, die Konstruktorausführung, Eingabedaten und die Speicherung des Laufzeitbytecodes. Das Deployment ist auch hier am teuersten, in etwa elfmal die Transferkosten. Eine genaue Aufschlüsselung des Gasverbrauchs ist durch fehlende Tools wie Remix hier nicht so gut möglich.

LLL

In LLL erfolgt das Deployment und die Interaktion analog zu Vyper, da auch hier keine entsprechende Entwicklungsumgebung vorhanden ist. Der Befehl „lllc –hex ERC20.lll“ liefert den Bytecode. Um die ABI zu erhalten gibt es hier keinen Schalter, man kann stattdessen die ABI des Solidity-Interface verwenden, da die Funktionen die gleichen sind. Dazu kann erneut Remix genutzt werden. Kopiert man den Bytecode in das Pythonskript, ergibt sich eine Anzahl von 913 Bytes, was sehr stark unter den bisherigen Werten liegt, obwohl der Quellcode deutlich länger war. Vypers Bytecode ist in etwa doppelt so groß und der von Solidity mehr als viermal so groß. Das Deployment verbraucht hier 306 463 Gas, dieser stimmt auch hier mit der „estimateGas“-Funktion überein. Für den Transfer werden 50 546 Gas benötigt, obwohl nur 35 546 geschätzt wurden. Somit liegt die Schätzung bei erneut 15 000 zu wenig.

Das Deployment kostet hier unter den gleichen Voraussetzungen wie bisher im Minimum 0,41€, durchschnittlich 0,69€ und im Maximum 0,83€. Die Kosten sind signifikant geringer als in Vyper und Solidity. Der Transfer kostet kaum weniger als in den anderen Sprachen. So liegt er bei 0,07€, 0,11€ oder 0,14€. Durch den viel geringeren Umfang ist somit deutlich weniger Speicherplatz auf der Blockchain erforderlich und der Overhead der anderen Sprachen macht sich bemerkbar.

In LLL gilt ebenfalls die gleiche Kostenzusammensetzung wie in den zuvor besprochenen Sprachen. Die Verteilung und Höhe des verbrauchten Gas variiert natürlich bezüglich Inputs, Codespeicherung und Konstruktor.

Bamboo

Auch für Bamboo existiert noch keine dedizierte Entwicklungsumgebung. Insbesondere, da diese Sprache von allen betrachteten die neuste ist. Einen Onlinecompiler gibt es ebenfalls nicht. Das Deployment wird somit wie in Vyper und LLL ausgeführt. Kompiliert wird über „bamboo < ERC20.bbo“. Mit dem Schalter „–abi“ kann auch die ABI ausgegeben werden. Der Gasverbrauch, den Ganache angibt, beträgt ganze 2 903 711 Gas. Das ist mit Abstand der höchste Verbrauch bisher. Er ist dreimal so groß wie in Solidity und sogar in etwa zehnmal so groß wie in LLL. Dies könnte damit zusammenhängen, dass ein Contract hier aus mehreren „Teilcontracts“ besteht. Ein Transfer verbraucht auch deutlich mehr Gas: 79 555. Geschätzt wurden dabei 64 555, erneut 15 000 zu wenig. Trotzdem ist der Verbrauch um fast 30 000 Gas höher als bei den übrigen Sprachen. Dies könnte durch die Contract-Übergänge verursacht werden. Zudem besitzt Bamboo noch keine

Optimierungsmöglichkeiten, welche Redundanzen im Bytecode verringern könnten. Es gibt hier ebenso keine Möglichkeit die genaue Kostenzusammensetzung zu verfolgen. Der Unterschied zu den anderen Sprachen ist groß.

Eine Besonderheit ist, dass der Contract nach dem deployen zunächst mit ausreichend Gas aufgerufen werden muss, damit er von „PreToken“ zu „Token“ wird. [vgl. 64] Infolgedessen wird der „default“-Case ausgeführt und dies führt dazu, dass der Aufrufer zum Besitzer wird und zunächst die Token erhält. Will man dies vermeiden, kann man den Contract selbst zum Besitzer der Token machen, indem man stattdessen „balances[this]“ den Gesamtvorrat zuweist.

Die Kosten für das Deployment sind entsprechend hoch. Die untere Grenze ist 3,93€, der Mittelwert 6,55€ und der obere empfohlene Grenzwert 7,86€. Der Transfer ist jedoch ähnlich wie in den bisherigen Sprachen. Hier fallen Kosten von 0,11€, 0,18€ oder 0,21€ an.

Die Größe des Bytecodes liegt hier bei 13 194 Bytes. Das ist mehr als das Dreifache von Solidity und sogar etwa 14-mal so viel wie LLL. Somit ist das Deployment um einiges teurer als in den anderen Sprachen.

3.7 Zusammenfassung und Einschätzung der Unterschiede

Lese- und Programmierfreundlichkeit

In Bezug auf die Lesbarkeit der Contracts fallen besonders Solidity und Vyper positiv auf. Der Code lässt sich gut strukturieren und die Syntax ist verständlich. Auch Bamboo ist sehr übersichtlich, erfordert jedoch etwas mehr Eingewöhnung in eine Denkweise Richtung Zustandsautomaten. Durch viele sprachliche Parallelen zu Solidity wird dies jedoch erleichtert. LLL dagegen benötigt viel Eingewöhnungszeit verglichen mit den übrigen Sprachen, da die EVM-Nähe und Präfixnotation sowie die Klammerung das Verständnis erschweren. Es gibt durch fehlende Funktionen weniger Gliederungsmöglichkeiten, Makrodefinitionen im Kopf der Datei oder bei Auslagerung erfordern Sprünge im Code beziehungsweise zwischen Dateien, was die Übersichtlichkeit verringert. Dies macht eine selbstständige Strukturierung durch Einrücken und Kommentierung nahezu notwendig.

Auch in Solidity kann eine Modularisierung die Lesbarkeit erschweren, allerdings besteht so die Möglichkeit, Teile des Codes wiederzuverwenden und große Quellcodedateien zu vermeiden. Bei Vyper und Bamboo kann es durch fehlende Modularisierung zu sehr umfangreichen Quellcodes kommen, auch dies kann nachteilig für den Überblick sein. Die ERC20-Contracts sind durch ihre geringe Komplexität jedoch in Solidiy, Vyper und Bamboo mit 80 bis 90 Zeilen in etwa gleich lang. Lediglich LLL ist mit 210 Zeilen durch die vielen benötigten Makros deutlich umfangreicher.

Die Programmierfreundlichkeit ist besonders in Solidity und Vyper sehr groß. Solidity hat den größten Funktionsumfang, unterstützt als einzige der Sprachen Vererbung und besitzt eine gut verständliche Syntax. Die große Anzahl an Funktionen und Datentypen sorgt für Bequemlichkeit. Vyper verfügt zwar über mehr Restriktionen, ist jedoch ebenso leicht verständlich, sodass der Einstieg leichtfällt. Die Sichtbarkeiten von Funktionen sind durch ihre Notation besonders hervorgehoben. Bamboo ist durch die Ähnlichkeit mit Solidity ebenso gut zu verstehen und umzusetzen, jedoch in den Funktionen bisher eingeschränkt. Es ist ein Eindenken in die Contract-Übergänge erforderlich und es gibt bislang weniger „bequeme“ Funktionen. Dabei besteht erhöhtes Fehlerpotenzial durch eigene Implementation noch nicht unterstützter Funktionalitäten. Die Syntax ist nichtsdestotrotz schnell erlernbar. Für LLL sind bezüglich der Programmierung Kenntnisse in Lisp oder ähnlichen Sprachen sehr von Vorteil. Mit Erfahrung in funktionalen Sprachen ist ein Umdenken erforderlich und der Einstieg deutlich schwerer. Es gibt kaum Datentypen und keine Funktionen, zudem erfolgt direkte Arbeit mit dem Speicher. Es sind viele eigene Implementationen erforderlich, die andere Sprachen von sich aus bieten oder durch einfache Funktionsaufrufe umgesetzt werden können. Dies muss bei der Programmierung berücksichtigt werden. Fehleranfällig sind auch die Präfixnotation und die EVM-Nähe. Alle Sprachen sind jedoch kompatibel mit dem ABI-Standard, sodass diese Contracts auf der Blockchain auch miteinander kompatibel sind.

Funktionsumfang

Der Funktionsumfang ist am größten in Solidity. Es wird eine Vielzahl von Funktionen, Datentypen und Modifikatoren angeboten. Jedoch sind die teilweise sehr geringen Unterschiede zwischen den Datentypen problematisch, da sie für einen Einsteiger möglicherweise nicht klar ersichtlich sind und ein größeres Risiko für eine ungeeignete Auswahl besteht. Vyper besitzt ebenfalls schon viel Funktionalität und Datentypen, die aber gegenüber Solidity bewusst eingeschränkt sind und dementsprechend klarer unterscheidbar. Die Einschränkung soll Fehler durch Missverständnisse vermeiden. Auch Bamboo limitiert die Datentypen und besitzt einen geringeren Funktionsumfang als Solidity, aber auch als Vyper. Alle drei Sprachen bieten jedoch Mappings an, die in der Smart Contract Entwicklung besonders hilfreich sind. Lediglich LLL besitzt diese nicht, sodass ein Workaround nötig ist. Zudem verfügt LLL über kaum Datentypen, selbst Wahrheitswerte müssen über Makros umgesetzt werden. Es gibt daher kaum Bequemlichkeiten und die Fehlergefahr durch eigene Umsetzung ist präsent.

Alle der betrachteten Sprachen bieten Events an, allerdings kann nur Solidity bislang die Nulladresse als Parameter übergeben. LLL erfordert zusätzlich die manuelle Übergabe der Event-ID, was in den anderen Sprachen nicht notwendig ist. Variablen für den Zugriff auf Blockchain- und Transaktionsdaten sind in allen Sprachen vorhanden.

Ein Konstruktor kann in Solidity, Vyper und LLL definiert werden, obgleich dies in LLL nur in Form einer Sequenz möglich ist. Er wird in diesen Sprachen zudem nicht mit deployt. Bamboo bietet keinen Konstruktor an sich, so muss dafür eine Contractform definiert wer-

den, die mit deployt wird. Erst beim erstmaligen Aufruf wird die Initialisierung durchgeführt und in einen Hauptcontract übergegangen.

Schließlich besteht in allen betrachteten Sprachen die Möglichkeit, einen Fallback zu definieren, wobei nur Vyper und Solidity eine payable-Markierung unterstützen. Während sie in Bamboo gar nicht eingesetzt werden kann, ist in LLL ein Makro erforderlich, um das Gegenteil umzusetzen. Standardmäßig gilt in LLL alles als payable, sodass angegeben werden muss, wo dies nicht erwünscht ist. Hier besteht die Gefahr, dies bei der Entwicklung zu vergessen.

Fehleranfälligkeit in der Programmierung

Die Fehleranfälligkeit in der Programmierung ist insgesamt am geringsten in Vyper, da diese Sprache auf die Vermeidung häufiger Fehler konzipiert ist. Einfachheit und Eindeutigkeit spielen eine große Rolle. Demgegenüber besitzt LLL das größte Fehlerpotenzial, insbesondere durch die Syntax, die wenigen nativen Funktionalitäten und den direkten Speicherzugriff. Auch die Übersichtlichkeit ist in LLL am geringsten.

Bezüglich Standardsichtbarkeiten schneidet dementsprechend Vyper am besten ab, da globale Variablen generell „private“ sind. Bei Funktionen besteht Kennzeichnungspflicht, was Klarheit schafft. Diese Vorschrift besteht in Solidity nicht, auch ist die Standardsichtbarkeit für Funktionen „public“. Somit besteht ein Sicherheitsrisiko, wenn dies bei der Implementierung nicht berücksichtigt wird und Funktionen unerwünscht für jeden zugänglich sind. Noch problematischer ist, dass Bamboo und LLL keine Sichtbarkeitsmodifikatoren besitzen. Demzufolge muss alles als öffentlich behandelt werden.

Für Prüfungen besitzen nur Solidity und Vyper Funktionen, die bei Fehlschlag ein Rollback auslösen. Eine Funktion für ein explizites Rollback bietet Vyper dafür als einzige der Sprachen nicht. In Solidity können für Prüfzwecke zusätzlich Modifier und in LLL Makros eingesetzt werden, bei den übrigen beiden Sprachen ist hier eine Implementation der entsprechenden Prüfung in jede einzelne Funktion erforderlich, was zu längeren Quellcodes führen kann.

Besonders sind in Vyper die nativen Over- und Underflowprüfungen sowie die Reentranzprüfung im Zuge externer Aufrufe in Bamboo. Damit begegnen die Sprachen häufigen Gefahren in Smart Contracts. Ihr restriktives Design soll ebenso zur Vermeidung von Fehlern beitragen, insbesondere indem Missverständnisse und Uneindeutigkeiten verringert werden.

Missverständnisse und Verwechslungen sind allerdings in Solidity und LLL mögliche Gefahrenquellen, da Überladung von Funktionen beziehungsweise Makros sowie Operatoren möglich ist. Ebenso können die Aliase für Datentypen in Solidity dies begünstigen.

Community-Unterstützung

Bezüglich der Community-Unterstützung bietet keine andere Sprache so viele Hilfestellungen wie Solidity. Eine Vielzahl an Foren, Blogs, Videos und eine ausführliche Dokumentation ermöglichen es, Hilfe und Anleitungen zu erhalten und einen Einstieg zu finden. Es sind bereits einige Tools für die Entwicklung und Tests von Contracts vorhanden.

Auch Vyper und LLL verfügen über eine Dokumentation, die jedoch bisher nicht vollständig ist. Es gibt für sie noch wenig Unterstützung in Foren und Blogs, sie nimmt jedoch insbesondere für Vyper zu. Die Sprache verfügt bereits über einen Onlinecompiler. In LLL findet man hauptsächlich Anleitungen von Daniel Ellison, die sehr zum Verständnis der Sprache beitragen.

Für Bamboo gibt es bislang die geringste Unterstützung durch die Community. Man findet bisher nur über den Entwickler selbst Material, der jedoch gute Beispiele zur Verfügung stellt. Dies hängt besonders mit der Neuheit der Sprache zusammen.

Zusammenfassend soll folgende Tabelle einige Spracheigenschaften zum Vergleich gegenüberstellen.

Tabelle 1: Spracheigenschaften im Vergleich

Kriterium	**Solidity**	**Vyper**	**LLL**	**Bamboo**
Imports	Ja	Nein	Ja	Nein
Interfaces	Ja	Nein	Nein	Nein
Vererbung	Ja	Nein	Nein	Nein
Events	Ja	Ja	Ja	Ja
Mappings	Ja	Ja	Nein	Ja
Blockchainvariablen	Ja	Ja	Ja	Ja
Funktionen	Ja	Ja	Nein	Ja
Konstruktor	Ja	Ja	Ja	Nein

Fallback	Ja	Ja	Ja	Ja
Sichtbarkeitsmodifikatoren	Ja	Ja	Nein	Nein
Rollbackfunktion	Ja	Nein	Ja	Ja
Prüffunktionen	Ja	Ja	Nein	Nein
Überladung	Ja	Nein	Ja	Nein
Dokumentation	Ja	Ja	Ja	Nein

Deployment

Das Deployment ist in Solidity am einfachsten umzusetzen, da Remix eine vielseitige grafische Oberfläche besitzt, die sich leicht mit Ethereum-Netzwerken oder einer lokalen Blockchain verbinden lässt. Über die Kommandozeile erfordert es dagegen zumindest ein wenig Kenntnisse von JavaScript und bietet optisch weniger Übersichtlichkeit. Es zeigte sich, dass LLL mit Abstand den kürzesten Bytecode erzeugt und dementsprechend auch am günstigsten zu deployen ist, obwohl der Quellcode der längste ist. So umfasst der Bytecode lediglich 913 Bytes. Vyper liegt hier mit 2 198 Bytes auf dem zweiten Platz und Solidity folgt mit 3 880 Bytes auf Platz drei. Den Schluss bildet Bamboo mit 13 194 Bytes, dem dreifachen von Solidity.

Der Gasverbrauch eines Transfers ist in Solidity, Vyper und LLL in etwa gleich: sie verbrauchen zwischen 50 000 und 52 000 Gas. Bamboo verbraucht fast 30 000 mehr, vermutlich durch die Contract-Übergänge. Schätzungen durch web3 liegen hier immer zwischen 7 000 und 15 000 Gas zu tief. Beim Deployen verbraucht LLL mit 306 463 Gaseinheiten am wenigsten. Danach folgt Vyper mit 605 047 Gas und Solidity mit 1 052 085 Gas. Bamboo verbraucht fast das Dreifache von Solidity: 2 903 711 Gas. Dies zeigen noch einmal folgende Abbildungen:

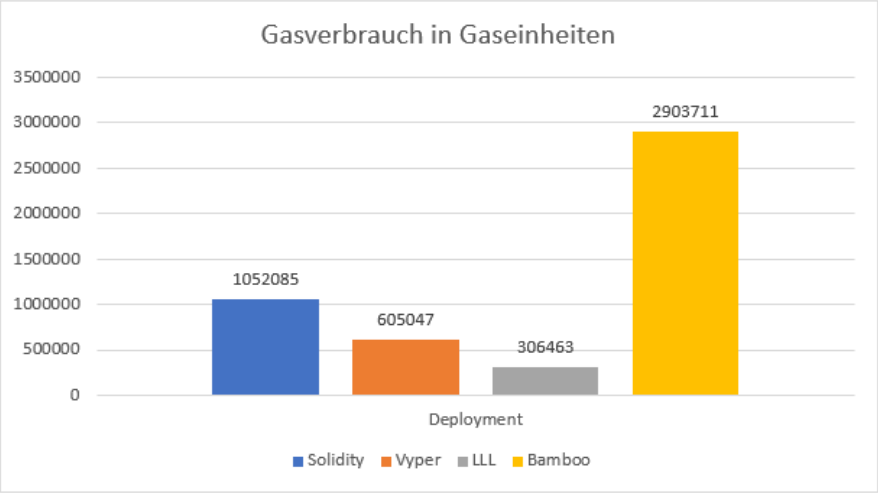


Abbildung 21: Gaskosten Deployment

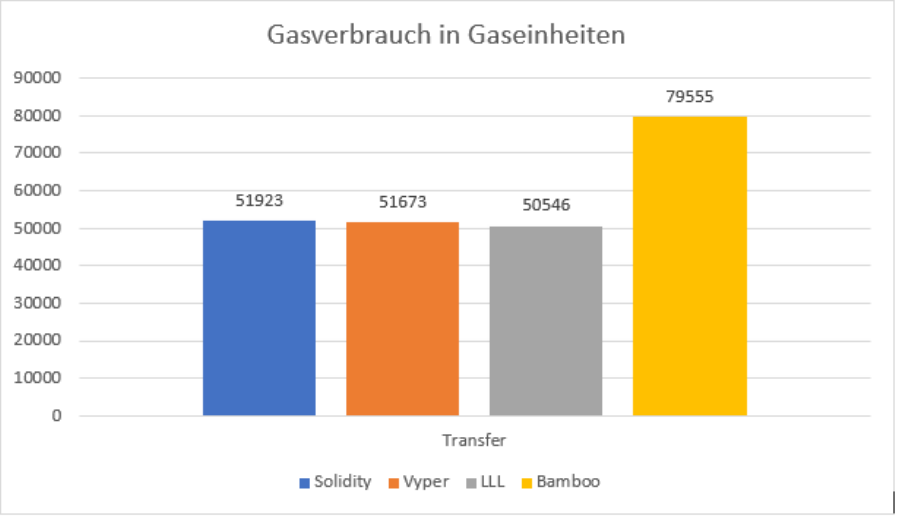


Abbildung 22: Gaskosten Transfer

Im Durchschnitt (mit 5 GWei pro Gas) kostet das Deployment zur Zeit des Schreibens in Solidity 2,69€, 1,55€ in Vyper und 0,69€ in LLL. Der Bamboo Contract kostet 6,55€ und ist daher der teuerste. Wie anhand des Gasverbrauchs erwartet, ist LLL am billigsten. Die Unterschiede sind hier groß. Bereits Vyper ist mehr als einen Euro billiger als Solidity.

Der Transfer kostet mit derzeit durchschnittlichem Gaspreis (5 GWei) bei Vyper und Solidity 0,13€. In LLL ist er nur geringfügig billiger, so kostet er hier 0,11€. In Bamboo ist die Abweichung ebenfalls nicht sehr groß, die Kosten belaufen sich auf 0,18€. Dies könnte damit zusammenhängen, dass für den Aufruf einer Funktion mit gleicher Funktionalität und Parametern auch in etwa gleiche Kosten anfallen. Lediglich die Umsetzung der entsprechenden Funktion weicht etwas voneinander ab, was auch den Preis beeinflussen kann. Auch dazu folgt zusammenfassend eine Abbildung:

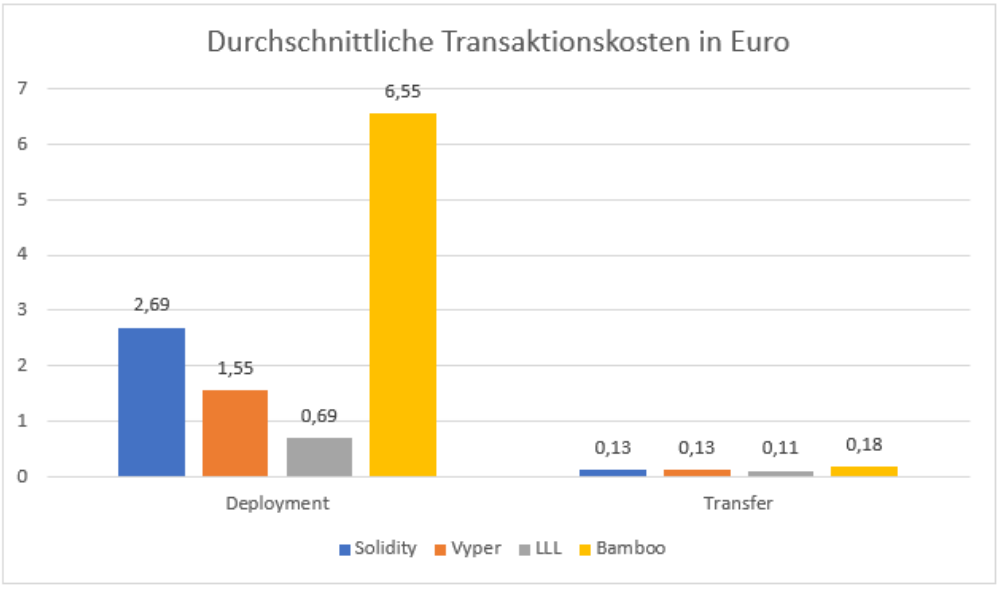


Abbildung 23: Transaktionskosten in Euro

Insgesamt ist LLL mit Abstand am kostengünstigsten und speichersparendsten beim Deployment. So sind nur 30% der Gaseinheiten von Solidity notwendig. Bamboo dagegen ist am kostenintensivsten: das Deployen kostet mehr als das Neunfache von LLL. Gegenüber Solidity ist auch Vyper schon um 40%-Punkte sparsamer. LLL ist also sehr empfehlenswert, wenn man sich in die Syntax und Programmierung eingearbeitet hat oder entsprechend Vorkenntnisse besitzt. Da dies in Vyper vermutlich deutlich schneller möglich ist, ist auch diese Sprache sehr zu empfehlen, da sie bereits Ersparnisse ermöglicht. In Bamboo sind diesbezüglich noch Optimierungen erforderlich, trotzdem besitzt die Sprache Potenzial.

4 Sicherheitsaspekte und häufige Gefahren

In diesem Kapitel sollen die häufigsten Gefahren in der Smart Contract Entwicklung vorgestellt und anschließend die Sprachen auf Sicherheitsmechanismen und Sicherheitsmuster bezüglich dieser untersucht werden. Zudem werden sprachspezifische Unsicherheiten zusammengetragen und die Vor- und Nachteile der EVM als Laufzeitumgebung und Basis der Sprachen betrachtet. Abschließend erfolgt eine Zusammenfassung gewonnener Erkenntnisse.

4.1 Typische Gefahren in der Smart Contract Entwicklung

Für die Entwicklung von Smart Contracts haben sich im Laufe der Zeit einige häufig auftretende Sicherheitsprobleme hervorgetan. Neben klassischen Denkfehlern, wie sie auch in gewöhnlichen Anwendungen auftreten, kommt es auch immer wieder zu Fehlern, die mit den Eigenheiten der Blockchain einhergehen. Eine Auswahl der typischen Gefahren soll im Folgenden vorgestellt werden.

Reentranz

Eines der am häufigsten vorkommenden Sicherheitsprobleme entsteht durch externe Aufrufe anderer Contracts. Diese können bösartig sein und im Zuge des Aufrufes in den eigenen Contract zurückrufen. Bezeichnet wird dies als „Reentranz“. Das Zurückrufen eines Angreifers kann unerwünschte Änderungen des Zustandes des Aufrufers bedeuten sowie zum Verlust von Token führen, wenn beispielsweise eine Funktion zur Auszahlung vom Zahlungsempfänger erneut aufgerufen wird, bevor der ursprüngliche Aufruf zurückkehrt. Einen hohen Verlust erlitt auf diese Weise die „DAO“ (Decentralized Autonomous Organization), von der ein Angreifer 2016 Token im Wert von 60 Millionen Dollar stiehlt. [vgl. 71, 72 S.6]

Das Problem entsteht meist dadurch, dass in Contracts externe Aufrufe stattfinden, bevor alle lokalen Änderungen durchgeführt wurden:

```
function sendFunds() public {
        msg.sender.call.value(balances[msg.sender])();
        balances[msg.sender] = 0;
}
```

In obigem Solidity Beispiel wird das Guthaben des Aufrufers gesendet, bevor es intern auf Null gesetzt wurde, sodass er im Zuge des call() als Teil seiner Contractausführung die Funktion erneut aufrufen und wieder den Betrag erhalten kann. Da der ursprüngliche Aufruf noch nicht zurückgekehrt ist, konnte der Betrag noch nicht zurückgesetzt werden. Zu-

dem schickt call() hier alles verfügbare Gas mit, sodass der Angreifer dies fortführen kann, bis alles aufgebraucht ist. Um dem entgegenzuwirken wird empfohlen, erst alle lokalen Änderungen vorzunehmen, bevor extern interagiert wird. Dies bezeichnet man als „Checks-Effects-Modell“.

```
function sendFunds() public {
        uint amount = balances[msg.sender];
        balances[msg.sender] = 0;
        msg.sender.transfer(amount);
}
```

In diesem Beispiel wird zuerst der Betrag zwischengespeichert und das Guthaben auf Null gesetzt. Danach erfolgt der Transfer, hier auch mit der sichereren transfer()-Funktion, die nur 2 300 Gas mitliefert und im Fehlerfall ein revert() auslöst.

Das Problem der Reentranz kann zudem über mehrere Funktionen übergreifend bestehen, wenn sie einen State teilen. Wird ein Contract aufgerufen, dem nicht vertraut wird, empfiehlt sich eine entsprechende Markierung als „untrusted“, um explizit zu machen, was dadurch beeinflusst werden kann. [vgl. 73]

Transaction Ordering Dependence

Eine weitere häufige Gefahr besteht in der „Transaction Ordering Dependence“. Von dieser Schwachstelle sind 2016 15,8% der Contracts betroffen [vgl. 74]. Dabei geht es um Manipulation der Transaktionsreihenfolge bei der Aufnahme in einen Block. Wenn beispielsweise ein Nutzer eine Antwort auf ein Rätsel eines Smart Contracts einreicht, um eine Belohnung zu erhalten, kann der Besitzer des Contract auf diese Interaktion lauschen, da jeder alle Transaktionen einsehen kann. So kann er in einer eigenen Transaktion mit höherem Gaspreis die Belohnung auf null setzen. Da eine Chance besteht, dass die Transaktion des Besitzers dann zuerst in den Block gelangt, erhält der Nutzer in diesem Fall keine Belohnung. Auch ein anderer Nutzer hätte zum Beispiel seine Antwort stehlen können. [vgl. 72, S.4] Diese Schwachstelle fand sich beispielsweise im Lotteriecontract „The Run“ wieder [vgl.71] und ist sehr schwer zu vermeiden.

Unsichere Zufallsgeneratoren

In Smart Contracts findet sich zudem immer wieder das Problem der unsicheren Zufallsgeneratoren. Oft werden Blockdaten eingesetzt, um Zufall zu erzeugen. Diese sind jedoch öffentlich sichtbar, zum Beispiel Blocknummern, oder durch den Miner beeinflussbar wie der Zeitstempel. Daher sind diese Zufallswerte oft voraussehbar oder können vorberechnet werden. Selbst Seeds, die als „private“ markiert sind, können gelesen werden. Auch dieses Problem lag im „The Run“ Contract vor. [vgl. 71] Es gibt jedoch auch Möglichkeiten, wie den Contract RANDAO, um bessere Zufallswerte zu erzeugen.

```
function play() public payable {
        require(msg.value >= 1 ether);
        if (block.blockhash(blockNumber) % 2 == 0) {
                msg.sender.transfer(this.balance);
        }
}
```

Hier wird die Blocknummer als Zufallsseed eingesetzt. Das Problem ist, dass die Blocknummer eines älteren Blocks als die 256 vorherigen nicht erreichbar ist und somit auf Null gesetzt wird. Gleiches gilt für den aktuellen Block, der noch nicht bekannt ist. Dies kann sich ein Angreifer zunutze machen. [71]

Abhängigkeit von Zeitstempeln

Da zudem die Abhängigkeit von Zeitstempeln eine Sicherheitslücke darstellt, eignen sich diese ebenso wenig zur Erzeugung von Zufall. Aber auch der Ablauf von Auktionen und Lotterien kann durch Zeitstempel beeinflusst werden, was problematisch ist, da Miner diesen manipulieren können. Dieses Problem trat im „GovernMental“ Contract auf. [vgl. 71] Normalerweise ist der Zeitstempel die lokale Zeit des Systems des Miners. Sie kann jedoch einige Sekunden abweichen und der Block wird trotzdem akzeptiert. So besteht die Möglichkeit, dass zum Beispiel eine eigentlich vor wenigen Sekunden beendete Auktion noch ein Gebot des Miners erhält, da er seine Transaktion in einen Block mined, der einen Zeitstempel von wenigen Sekunden zuvor enthält. Auch bei Zufallswerten durch Zeitstempel kann er versuchen, diese vorzuberechnen. Sich auf diese zu verlassen ist daher sehr riskant. [vgl. 72, S.4f.]

Ungeprüfte Rückgabewerte

Sehr problematisch kann auch das Nichtprüfen von Rückgabewerten in Smart Contracts sein, besonders bei Transferfunktionen. Gleiches gilt für unbehandelte Exceptions. Betroffen sind insbesondere die Solidity Funktionen send(), call() und delegatecall(). Sie lösen bei Fehlschlag keine Exception aus, sondern liefern false zurück. Bei Nichtprüfung können Fehler so unbemerkt bleiben und es wird mit dem Code fortgefahren in der Annahme, alles wäre in Ordnung. Dies kann Inkonsistenzen zur Folge haben. So wäre das beispielsweise auch in Abbildung 29 möglich, da die Rückgabe von call() nicht geprüft wird und trotz Fehler dann das Guthaben auf null gesetzt würde. Ein verwundbarer Contract ist hier „King of the Ether“. [vgl. 71]

Schlecht umgesetzte Zugriffskontrolle

Eine schlechte Zugriffskontrolle auf bestimmte Funktionen eines Contracts birgt ebenfalls Gefahren eines Missbrauchs. Dieses Problem besteht nicht nur in Smart Contracts, auch klassische Anwendungen sind, laut OWASP, oft davon betroffen. Wenn eine Contractfunktion zum Setzen des Besitzers beispielsweise von jedem zugänglich ist, kann somit jeder die Kontrolle über diesen und eventuell hinterlegte Token erhalten. Dies war zum

Beispiel ein Problem der Parity Multi-Sig Wallet. Deren Initialisierungsfunktion wurde in eine Bibliothek ausgelagert, welche aber als „contract“ implementiert war und so selbst von jedem Aufrufer in Besitz genommen werden konnte. [vgl. 71] Deshalb ist es notwendig, kritische Funktionen auf bestimmte Adressen zu beschränken und Prüfungen durchzuführen. Zudem sollten Besitzer eines Contracts im besten Fall nur einmalig im Konstruktor gesetzt werden.

```
function setOwner(address _new) public {
        owner = _new;
}
```

Diese Abbildung zeigt ein Beispiel für eine fehlende Kontrolle des Aufrufers einer öffentlich zugänglichen Funktion. Sie sollte auf den aktuellen Besitzer beschränkt werden.

Denial of Service

Schließlich besteht eine der häufigsten Gefahren im „Denial of Service“. In Smart Contracts entsteht dieses Problem beispielsweise durch Out-Of-Gas-Exceptions oder ständigem revert bei externen Aufrufen. So ist es einem Empfänger von Token möglich, Transfers zu ihm absichtlich fehlschlagen zu lassen, indem er bei Zahlungsempfang jedes Mal ein Rollback auslöst. Auch kann ein Angreifer im Zusammenhang mit fehlerhafter Zugriffskontrolle Rechte erlangen und Smart Contracts blockieren oder durch entsprechende Eingabeparameter komplexe Berechnungen auslösen und so den Gasverbrauch in die Höhe treiben, bis das Limit erreicht ist. Contracts, die diesen Problemen ausgesetzt wurden, sind „GovernMental“ und die Parity Multi-Sig Wallet. [vgl. 71]

In einer Auktion, die entsprechend verwundbar ist, ist es einem Angreifer zum Beispiel möglich, zu verhindern, dass ein neuer Höchstbietender gesetzt werden kann, indem er die Rückzahlung seines hinterlegten Gebotes verhindert. Dazu führt er in der Fallbackfunktion seines Contracts einfach nur ein Rollback durch. Dies löst im Auktionscontract ebenfalls ein Rollback aus, sodass der neue Höchstbietende niemals gesetzt wird [vgl. 73]:

```
contract Auction {
    address currentLeader;
    uint highestBid;

    function bid() payable {
        require(msg.value > highestBid);

        require(currentLeader.send(highestBid));

        currentLeader = msg.sender; // bei Fehlschlag nicht gesetzt
        highestBid = msg.value;
    }
}
```

[73]. Als Lösung sollte man anstelle des Sendens eines Gebotes eine Funktion zum Einziehen dessen durch die berechtigte Adresse implementieren („Pull“-Mechanismus). Um zu verhindern, dass es durch Schleifen zu einem zu hohen Gasverbrauch kommt, ist zudem das Iterieren über große Arrays zu vermeiden. Besonders, wenn darin Auszahlungen erfolgen sollen, besteht die Gefahr, dass der Fehlschlag einer einzelnen Zahlung dazu führt, dass alle rückgängig gemacht werden oder dies den Verbrauch von zu viel Gas bedeutet. So kann die Ausführung dauerhaft fehlschlagen. Dies ist besonders gefährlich, wenn Nutzer durch beispielsweise Vergrößern eines Arrays die Anzahl an Iterationen erhöhen können. [vgl. 73]

Over- und Underflows

Neben diesen Gefahren existieren natürlich viele weitere. Diese entstehen zum Teil auch durch Programmier- und Denkfehler, wie sie in gewöhnlichen Anwendungen vorkommen. So treten Over- und Underflows immer wieder auf, wenn Prüfungen fehlen oder Denkfehler enthalten. So kann ein Überlauf von zum Beispiel einer Tokenanzahl zu plötzlich sehr niedrigem Guthaben führen oder umgekehrt. Begünstigt wird dies durch eine ungeeignete Auswahl an Datentypen.

Da sich deployte Contracts unveränderlich auf der Blockchain befinden, ist grundsätzlich ein ausgiebiges Testen notwendig, bevor das Deployment erfolgen sollte. So empfiehlt es sich, zusätzlich zu Codeanalysen die Funktion in einem Testnetzwerk oder mit Ganache und beispielsweise den Truffle-Entwicklungstools zu testen. Auch Mechanismen einzubauen, die im Falle des Auffindens von Bugs nach dem Deployment den Contract stoppen, können das Ausnutzen dieser verhindern. Upgrades sind in Form von Weiterleitungen der Aufrufe auf eine neue Adresse möglich.

Im Folgenden werden die Smart Contract Sprachen auf Sicherheitsmechanismen bezüglich dieser Gefahren untersucht sowie sprachspezifische Sicherheitsrisiken gesammelt.

4.2 Sicherheitsmechanismen und Gefahren der Sprachen

Solidity

Reentranz

Mit Solidity als Programmiersprache gibt es in Contracts keine nativen Schutzmechanismen gegen Reentranz. Besonderes Gefahrenpotenzial bergen hier Funktionen wie „call()“. Diese ermöglicht den Aufruf eines externen Contracts mit der gesamten Menge an noch verfügbarem Gas. Dies kann dazu führen, dass genügend vorhanden ist, um bei der Ausführung der Fallback-Funktion des aufgerufenen Contracts in den Aufrufer zurückzurufen und Manipulationen vorzunehmen. Bei Solidity hat sich hier ein sogenanntes „Antipattern“ hervorgetan. So werden oft nach externen Aufrufen noch lokale Änderungen vorgenommen, was im Falle eines Wiedereintritts im Zuge der externen Interaktion jedoch

noch nicht erfolgt ist und somit Manipulationen begünstigt. Daher hat sich als Security Pattern hier das bereits angesprochene „Checks-Effects-Modell“ etabliert, in welchem erst alle lokalen Prüfungen und Operationen erfolgen und zum Schluss die externen Aufrufe, zu sehen auf Seite 64. Um Reentranz zu verhindern besteht zudem die Möglichkeit Mutexe einzusetzen, die die Funktionen abschließen und erst bei Austritt wieder freigeben. Dies eignet sich auch zur Vermeidung funktionsübergreifender Reentranz. Darüber hinaus gilt es als empfehlenswert, „call()“ zu vermeiden und stattdessen „transfer()“ oder „send()“ zu nutzen, da diese Funktionen nur 2 300 Gas mitgeliefert bekommen, was für einen Wiedereintritt nicht ausreicht.

Transaction Ordering Dependence

Auch für die Transaction Ordering Dependence gibt es keine nativen Sicherheitsmechanismen, da dieser Gefahr im Allgemeinen schwer zu begegnen ist. So muss dafür abhängig vom Contract eine Lösung gefunden werden. Bei Auktionen empfehlen sich der Einsatz von Prä-Commitments, das heißt eine vorherige Ankündigung des Gebots, und die Hinterlegung von Kautionen.

Unsichere Zufallsgeneratoren

Solidity Contracts sind zudem von unsicheren Zufallsgeneratoren betroffen. Es gibt keinen eigenen Algorithmus oder nativen Contract. Daten wie „now“ und „block.blockhash()“ sind hier aus genannten Gründen unsicher. Es gilt als allgemeine Empfehlung Zufall außerhalb der Blockchain zu erzeugen oder über BTCRelay Bitcoin-Blockheader zu nutzen. Auch der RANDAO-Contract hat sich als gute Alternative erwiesen. Dieser nutzt den SHA3-Algorithmus und Kautionen zur Erzeugung eines Zufallswertes. So muss, wer sich beteiligen will, erst den Hash eines Wertes mit einer Kaution hinterlegen. Dann muss der Wert innerhalb eines bestimmten Zeitraums eingereicht werden. Stimmt dieser mit dem gehashten Wert überein, wird er als Seed für die Zufallszahl genutzt, sonst oder bei Nichteinreichung wird die Kaution einbehalten. [vgl. 75]

Ein Problem besteht zudem, wenn es Entwicklern nicht bewusst ist, dass alles auf der Blockchain öffentlich ist. So sind private Variablen durchaus lesbar, besonders, wenn sie bei der Contracterzeugung in einer Transaktion gesetzt wurden. Somit bedeutet eine „private“-Markierung nicht, dass entsprechende Daten versteckt werden. Definiert man daher statische Seeds auf diese Weise, können sie trotzdem von möglichen Angreifern gesehen und für Vorberechnungen eingesetzt werden.

Abhängigkeit von Zeitstempeln

Auch Zeitstempel, die über „now“ beziehungsweise „block.timestamp“ abgerufen werden können, sind in Solidity nicht zuverlässig. Man sollte sie möglichst nicht für zeitkritische Aktionen nutzen. Im Zusammenhang mit Lotterien und Zufall oder Auktionen ist von ihnen abzuraten. Als Richtlinie gibt es eine 30-Sekunden-Regel: so ist der Einsatz des Zeitstempels in Ordnung, wenn eine Zeitabweichung von 30 Sekunden unproblematisch für

die Funktion beziehungsweise den Contract ist. Eine Alternative ist der Einsatz von Blocknummer oder Blockzeit, um die Zeit darzustellen. Jedoch gilt es zu beachten, dass die Blockzeit variieren kann. [vgl. 76]

```
modifier auction_complete {
    require(auctionEndBlock <= block.number)
    _;
}
```

Der obige Modifier [vgl. 76] kann eingesetzt werden, um vor dem Zugriff auf bestimmte Funktionen einer Auktion zu prüfen, ob diese beendet ist. Die Blocknummer des Endes muss bei der Initialisierung angegeben werden.

Ungeprüfte Rückgabewerte

Einige Solidity Funktionen bergen zudem die Gefahr ungeprüfter Rückgabewerte. So liefern, wie bereits erwähnt, „call()“, „send()“ und auch „delegatecall()“ bei Fehler nur false zurück, sodass eine fehlende Prüfung dieser Inkonsistenzen bedeuten kann. Dazu kann man sie in ein „require()“ setzen, das im Fehlerfall dann ein revert auslöst. Eine weitere Möglichkeit besteht in der Verwendung von „transfer()“, was selbst bei Fehlschlag eine Exception auslöst und ein Rollback durchführt. Einige Entwicklungsumgebungen liefern bereits beim Kompilieren Hinweise auf ungeprüfte Rückgabewerte. [vgl. 76] Folgende Abbildung zeigt beispielhaft die Prüfung von Rückgabewerten externer Aufrufe:

```
if(!user.send(100)){
// Fehlerbehandlung
}

// oder:

require(user.send(100));

// oder:

user.transfer(100);
```

Diese Unsicherheit kommt sehr häufig vor, so prüfen 27,9% der Contracts, die eine Forschungsgruppe mithilfe des Oyente-Testtools betrachtete, die Rückgabe von „send()“ nicht, sodass Exceptions unbemerkt bleiben [vgl. 72, S.6].

Bei Funktionen wie „call()“ gilt es jedoch zu beachten, dass das Warten auf Rückgabewerte auch riskant sein kann, so kann der aufgerufene Contract die Rückkehr bei genügend Gas verzögern. Wenn es nicht möglich ist, „transfer()“ stattdessen einzusetzen, sollte man zumindest die mitgelieferte Menge an Gas beschränken.

Schlecht umgesetzte Zugriffskontrollen

Native Zugriffskontrollen für Funktionen sind ebenfalls nicht vorhanden. Wenn sie nicht öffentlich aufrufbar sein sollen, können sie als „private“ markiert werden. Will man öffentliche Aufrufe lediglich beschränken, können Modifier eingesetzt werden. So kann zum Beispiel der Aufrufer geprüft werden. Allgemein gilt es als guter Stil alle Sichtbarkeiten explizit zu markieren, damit bezüglich Zugriffen Klarheit besteht.

Denial of Service

Schließlich kann auch Denial of Service nicht nativ abgewehrt werden. Risiken können jedoch minimiert werden. So sollten Schleifen vermieden werden, insbesondere Iteration über Arrays ohne feste Länge. Zudem gilt die Auszahlung von Token in einer Schleife als Antipattern, da ein fehlschlagender Aufruf die Umkehr aller bedeutet und jegliche Auszahlung verhindert. Auch besteht dabei die Gefahr zu hohen Gasverbrauchs. Kann man eine derartige Vorgehensweise nicht vermeiden, wird im Allgemeinen eine Beschränkung von Transfers pro Transaktion angeraten, wobei sich der Fortschritt gemerkt werden muss. Am besten geeignet, um ein DoS aufgrund eines Rollbacks beim Empfänger zu vermeiden oder ein Out-of-Gas durch Schleifen, ist der Einsatz eines Pull-Mechanismus, der es berechtigten Empfängern erlaubt, ihre Token einzuziehen. Allgemein sollte man Fehler bei externen Aufrufen erwarten und darauf achten, dass diese nicht den ganzen Contract blockieren. Fallback-Funktionen sollten nicht zu komplex gestaltet werden, um dort Out-of-Gas zu verhindern. Eigene Contracts sind im besten Fall einer Gasschätzung zu unterziehen, um sicherzustellen, dass es nicht durch bestimmte Eingaben oder Arraygrößen zu solchen Fehlern kommt. Daher kann auch die Prüfung von Eingabewerten hilfreich sein, damit zu komplexe Berechnungen, die zu viel Gas verbrauchen, verhindert werden können.

Over- und Underflows

Bei Arithmetik und Zugriffen auf Arrays muss in Solidity manuell auf Over- und Underflows geprüft werden. Alternativ kann die von OpenZeppelin zur Verfügung gestellte Bibliothek „SafeMath“ für sichere Arithmetik eingesetzt werden. Gerade bei Tokenguthaben ist eine sichere Arithmetik wichtig, um Inkonsistenzen und Überlauf dessen zu verhindern. Auch die Auswahl richtiger Datentypen spielt eine Rolle. Solidity bietet viele Typen für Ganzzahlen an, aus denen entsprechend der gewünschten Funktion gewählt werden muss. Ein uint8 ist beispielsweise für ein Tokenguthaben gemessen in Wei nicht ausreichend. Dies betrifft ebenso den Index in beispielsweise for-Schleifen. Ist der Typ zu klein gewählt, kann ein Überlauf eine Endlosschleife auslösen, die alles Gas aufbraucht. Auch Denkfehler kommen hier häufig vor, so macht eine Prüfung, ob ein uint kleiner ist als Null, zum Beispiel keinen Sinn.

Sprachspezifische Gefahren

In Solidity gibt es weitere sprachspezifische Gefahren zu beachten. Beispielsweise gibt es einen Unterschied zwischen Interfaces und abstrakten Contracts, den es zu beachten gilt. Erstere können keine implementierten Funktionen enthalten und nicht erben. Zudem erfordert das Erben von abstrakten Contracts immer die Implementation aller Funktionen, sonst ist der erbende Contract selbst abstrakt. Bei Interfaces müssen auch alle Funktionen umgesetzt werden, jedoch kann der es implementierende Contract nicht zum Interface werden, wenn er Implementationen enthält. Gefährlich ist zudem die Unterstützung der Überladung von Funktionen sowie Variablen wie „msg“. Dies verringert die Lesbarkeit und kann zu Missverständnissen führen. So kann ein bösartiger Contract unerwünschte Funktionalitäten hinter scheinbar harmlosen Bezeichnungen verbergen. Darüber hinaus können von der Verwendung veralteter Strukturen Sicherheitsprobleme ausgehen. So wird vom Einsatz von „tx.origin“ abgeraten und es gilt „selfdestruct“ gegenüber „suicide“ und „keccak256“ gegenüber „sha3“ zu bevorzugen. Es muss weiterhin beachtet werden, dass bei Vererbung und der Angabe von Modifiern deren Reihenfolge eine Rolle spielt. Bei der Vererbung können Funktionen mehrfach überladen werden. Auch, dass theoretisch Rekursion möglich ist, birgt besonders das Risiko von zu viel Gasverbrauch. Schließlich besitzt die Sprache keine Fließ- oder Festkommazahlen, was bei Division von Integern zu Ungenauigkeiten durch Abrunden führen kann.

Insgesamt besitzt Solidity gegen die häufigsten Gefahren keine nativen Sicherheitsmechanismen, was logisch erscheint, da die meisten Contracts in Solidity geschrieben wurden und die Sammlungen an Schwachstellen auf dort gefundenen Problemen basieren. Man kann jedoch durch Einhaltung von Security Patterns und Gestaltungsempfehlungen für Smart Contracts Risiken minimieren. Auch Bibliotheken wie SafeMath und Funktionen wie „require()“ und „assert()“ können, richtig angewendet, hilfreich sein, um die häufigsten Probleme zu vermeiden.

Vyper

Reentranz

Die Reentranz betreffend gibt es auch in Vyper keine nativen Schutzmechanismen. Allerdings kann auch hier das Checks-Effects-Modell eingesetzt werden. Es findet sich zur Funktion zum Senden von Ether, „send()“, keine Information bezüglich weitergeleitetem Gas, sodass grundsätzlich Vorsicht geboten sein sollte. Bei „raw_call()“ dagegen, die dem direkten Aufruf einer Adresse dient, muss ein fester Betrag an Gas übergeben werden, sodass hier Klarheit über den mitgelieferten Betrag herrscht, der zur Verfügung steht und eine genaue Regulierung möglich ist. Als weitere Möglichkeit zur Absicherung sind hier ebenfalls Mutexe möglich.

```
balances: uint256[address]
locked: bool

@public
@payable
def deposit() -> bool:
     assert not locked
     locked = true
     balances[msg.sender] += msg.value
     locked = false
     return true
```

Dieses Beispiel zeigt hier die Umsetzung eines Mutexes mithilfe eines Booleans und eines assert-Statements.

Transaction Ordering Dependence

Auch eine Transaction Ordering Dependence kann mit Vyper nicht ausgeschlossen werden. Es muss, analog zu Solidity, contractabhängig eine Lösung gefunden werden. Das bedeutet, es können auch in Vypercontracts Prä-Commitments und Kautionen für Marktplätze und Auktionen eingesetzt werden.

Unsichere Zufallsgeneratoren

Weiterhin verfügt auch Vyper nicht über native Mechanismen zur Erzeugung von Zufall. Dieses Thema ist im Allgemeinen zurzeit noch Teil aktiver Forschung. Da es ebenso in Vyper Zugriff auf verschiedene Blockdaten gibt, sollte damit auch hier vorsichtig umgegangen werden und vom Einsatz für die Erzeugung von Zufall abgesehen werden. Es gilt wie in Solidity: besser Zufall außerhalb der Blockchain erzeugen. Wenn man RANDAO nutzen will, muss in Vyper beachtet werden, dass externe Aufrufe bislang die Definition einer Contractvariable und die Angabe der zu nutzenden Funktionen im Kopf des Contract erfordern. Zudem sind solche Interaktionen noch schlecht dokumentiert. Die „private“-Markierung kann auch in diesem Fall nicht vor Einsicht schützen.

Abhängigkeit von Zeitstempeln

Die Nicht-Vertrauenswürdigkeit der Blockdaten schließt auch die Zeitstempel mit ein, die wie in Solidity natürlich vom Miner abhängen. So kann die 30-Sekunden-Regel eine gute Richtlinie sein oder es sollte auf Blocknummern zurückgegriffen werden.

Ungeprüfte Rückgabewerte

Bezüglich ungeprüfter Rückgabewerte ist Vyper nicht eindeutig. So hat „send()“ keinen Rückgabewert, was vermuten lässt, dass ein Fehlschlag eine Exception auslöst. Zur Sicherheit sollte es mit „assert“ verwendet werden, um ein Rollback zu garantieren. „raw_call“ dagegen liefert entsprechend der Rückgabe des aufgerufenen Contracts Bytes zurück und die Länge dieser Daten wird beim Aufruf festgelegt, sodass die Rückgabe

geprüft werden kann. Hier besteht die Dokumentation betreffend Verbesserungspotenzial, damit Fehler durch Missverständnisse verhindert werden können.

Schlecht umgesetzte Zugriffskontrollen

Zugriffskontrollen werden nicht nativ angeboten, zudem gibt es keine Modifier, die Funktionen ergänzen können. Daher muss, um Unsicherheiten durch fehlende Kontrollen zu vermeiden, in jeder Funktion eine manuelle Prüfung beispielsweise mittels „assert“ erfolgen, sofern eine Einschränkung erforderlich ist. Zugriffe von außerhalb auf Funktionen können auch hier über das „private“ Keyword verhindert werden. Vorteilhaft in Vyper ist es, dass die Sichtbarkeitsmarkierung von Funktionen Pflicht ist, sodass immer Klarheit bezüglich Zugriffen besteht.

Denial of Service

Denial of Service kann in Vyper ebenso auftreten wie in Solidity. So kann es beispielsweise durch Out-of-Gas oder Rollback beim Empfänger auftreten. Daher empfiehlt sich die Vermeidung von Schleifen für Auszahlungen. Da grundsätzlich keine Endlosschleifen unterstützt werden, können diese nicht zu Out-of-Gas führen. Die eingebaute Gasabschätzung von Funktionen kann hilfreich sein, zu hohen Gasverbrauch zu erkennen. Trotzdem werden hier „Pull“-Mechanismen für Auszahlungen empfohlen. Wenn eine Schleife verwendet werden soll, so müssen die Iterationen unbedingt pro Transaktion limitiert werden.

```
@public
def refund():
    ind: int128 = self.refundIndex
    for i in range(ind, ind + 30):
        if i >= self.nextFunderIndex:
            self.refundIndex = self.nextFunderIndex
            return
        send(self.funders[i].sender, self.funders[i].value)
        self.funders[i] = None
    self.refundIndex = ind + 30
```

Im obigen Codebeispiel wird die Anzahl an Iterationen auf 30 pro Transaktion beschränkt. [33, Crowdfund] Dazu muss sich der Index des als nächsten auszuzahlenden Betrags gemerkt werden. Zu beachten gilt hier jedoch, dass durch das Setzen der Mapping-Einträge auf „None“ nach dem Senden eine Reentranzgefahr bestehen könnte. So müsste man „sender“ und „value“ zwischenspeichern und den Eintrag vorher auf null setzen.

Da in Vyper noch keine Fallback-Funktionen vorhanden sind, muss nicht auf zu komplexen Code dort geachtet werden, dies könnte sich allerdings zukünftig ändern. Zudem wird ein Contract bei direktem Etherempfang eine Exception auslösen. Da auch Eingabedaten in Funktionen zu komplexe Berechnungen auslösen könnten, sollten in Vyper eine Prü-

fung dieser und gegebenenfalls eine Beschränkung zulässiger Werte umgesetzt werden. Im Gegensatz zu Solidity wird keine Rekursion erlaubt, sodass dies keine Gefahr darstellt.

Over- und Underflows

Eine Besonderheit ist hier, dass es native Grenz- und Überlaufprüfungen bei Arrayzugriff und Arithmetik gibt. So kommt es diesbezüglich nicht zu Inkonsistenzen bei Tokenguthaben oder zu durch Überlauf entstehenden Dauerschleifen. Trotzdem sind adäquate Datentypen zu wählen, sodass Grenzen nicht zu schnell erreicht sind.

Sprachspezifische Gefahren

Vyper ist eine Sprache, die eine höhere Sicherheit als Solidity zum Ziel hat und dementsprechend über mehr Restriktionen verfügt. Das Fehlen von Vererbung, Assemblercode, Überladung von Funktionen und Operatoren sowie Rekursion soll Gefahren durch Missbrauch und Falschbenutzung verhindern, da es sich um sehr fehleranfällige Features handelt. Wie in Solidity ist eine gleiche Benennung von Variablen und Funktionen nicht gestattet, um Missverständnissen vorzubeugen. Vorteilhaft ist die Referenz von statischen Variablen mittels „self“. So sind sie als solche klar erkennbar und die Modifizierung des States ist an entsprechenden Stellen offensichtlich. Es werden Festkommazahlen unterstützt, sodass bei Division nicht so stark gerundet werden muss wie in Solidity. Ungünstig sind allerdings die Unklarheiten bezüglich des Verhaltens der „send()“-Funktion im Fehlerfall und ihrer mitgelieferten Menge an Gas. Durch die fehlende Modularisierung können zudem bei komplexeren Quellcodes Fehler in der Implementierung im Zusammenhang mit schlechterer Übersichtlichkeit schlechter gefunden werden. Der geringere Funktionsumfang als Solidity birgt das Risiko von Fehlern bei der Nachbildung von dort unterstützten Funktionen.

Insgesamt besitzt Vyper gegen die meisten der häufigsten Gefahren ebenfalls nur Schutz durch Security Patterns. Allerdings gibt es mehr Restriktionen, wie die obligatorische Sichtbarkeitsmarkierung der Funktionen und der Ausschluss von Imports und Vererbung, was die Sicherheit verbessern soll. Auch die Grenzprüfungen sind sehr vorteilhaft und verringern die Gefahr von Over- und Underflows maßgeblich. Schließlich kann auch die Gasschätzung hilfreich sein, um zu hohen Verbrauch verhindern zu können.

LLL

Reentranz

Da, wie bereits beschrieben, in LLL sehr viele Strukturen und Prüfungen selbst implementiert werden müssen, gibt es keine nativen Sicherheitsfunktionen für die häufigsten Schwachstellen. Es besteht die Gefahr des Vergessens von Prüfungen, die in anderen Sprachen nicht manuell umgesetzt werden müssen. Security Patterns können jedoch auch hier eingesetzt werden. Zusätzlich empfiehlt sich eine gute Codestrukturierung.

Gegen Reentranz sollten daher auch in LLL das Checks-Effects-Modell oder Mutexe eingesetzt werden.

Transaction Ordering Dependence

Analog zu den anderen betrachteten Sprachen hängen Schutzmaßnahmen gegen die Transaction Ordering Dependence vom konkreten Contract ab. So wird im ERC20 Beispiel in der approve-Sequenz beispielsweise ein Nullsetzen des autorisierten Betrags gefordert, bevor er neu gesetzt werden kann. Dadurch kann die autorisierte Adresse nicht vor dem Mining der Änderung in einen Block noch schnell in einer Transaktion mit höherem Gaspreis die Token ausgeben und danach nochmal den neuen Betrag.

Unsichere Zufallsgeneratoren

Auch die unsicheren Zufallsgeneratoren stellen in LLL ein Risiko dar. Durch EVM Opcodes wie „Timestamp“ und „Blockhash“ besteht die Gefahr einer Verwendung dieser unter der Annahme, sie seien zuverlässig.

Abhängigkeit von Zeitstempeln

Dies bedeutet auch, dass der Zeitstempel nicht vertrauenswürdig ist und seine Verwendung mit Vorsicht erfolgen sollte. Deshalb gelten gleiche Empfehlungen wie in Solidity und Vyper.

Ungeprüfte Rückgabewerte

Ungeprüfte Rückgabewerte sind in einigen Makros ebenfalls Sicherheitsrisiken. Dies gilt beispielsweise für die „send“ Makros, sie liefern 1 bei Erfolg und 0 bei Fehler zurück. Auch bei der Contracterzeugung mittels „create“ sollte die Rückgabe geprüft werden, hier wird bei Fehler 0 und sonst die Adresse zurückgeliefert. Ein Abbruch muss in diesen Fällen mit „revert“ oder „panic“ selbst vorgenommen werden.

Schlecht umgesetzte Zugriffskontrollen

Um der Gefahr fehlender Zugriffskontrollen zu begegnen, lassen sich Makros, vergleichbar mit Solidity's Modifiern einsetzen, diese werden dann in die entsprechende Sequenz eingefügt:

```
(def 'only-owner (when (!= (caller) @@owner)
                       (revert)))
```

Hier ein Beispiel für ein Makro, welches als Aufrufer nur den Besitzer des Contracts zulässt.

In LLL gibt es keine Sichtbarkeitsmodifikatoren. Will man Zugriffe auf Sequenzen von außen unterbinden, sollte man sie nicht in die ABI aufnehmen und nicht mit in den Abgleich der Funktions-IDs. Besser ist es jedoch als Aufrufer nur den Contract selbst zuzulassen.

Denial of Service

Betrachtet man das Risiko von Denial of Service, so zeigt sich, dass keine Gefahr durch Rekursion innerhalb von Makros besteht, da dies nicht erlaubt ist. Der Compiler würde den Code des Makros immer wieder in dasselbige einsetzen, bis es zum Coredump kommt. Allerdings kann es durch Schleifen zu Out-of-Gas kommen sowie zur Blockierung durch Rollback bei aufgerufenen Contracts. Folglich gilt auch hier: Auszahlungen in Schleifen sind zu vermeiden und „Pull“-Mechanismen sind zu bevorzugen.

```
(seq
        (def 'value arg1)
        (def 'success 0x00)

        (if (< (balance (caller)) value)
                (revert)
                (seq
                   (sstore (balance (caller)) (- (sload (balance (caller))) value))
                   (mstore success (send(21000 (caller) value))))
        (if (!= (mload success) 1)
                (revert)
                (return true))
```

Das Codebeispiel zeigt hier eine Sequenz zum Einzug eines Tokenbetrags. Dabei werden Parameter des Aufrufes, hier „arg1“, welches selbst ein Makro ist, zunächst für die Verwendung vorbereitet. „arg1“ lädt das erste Argument der aufgerufenen Funktion aus den Aufrufdaten. Ein Memoryslot wird für die Speicherung der Rückgabe vordefiniert. Es wird geprüft, ob genug Guthaben vorhanden ist, was eingezogen werden kann und das interne Guthaben entsprechend verringert. Schließlich wird es dem Aufrufer zugesandt und der Erfolg geprüft. Zu beachten ist, dass vor dieser Sequenz die Funktions-ID ausgewertet werden muss.

Obwohl es keine Fallback-Funktion an sich gibt und es vielmehr der Standardfall im Case-Statement der Funktionsabgleiche ist, sollte man diese Sequenz möglichst geringhalten, was den Rechenaufwand betrifft. So können Gasprobleme vermieden werden. Natürlich ist in LLL ebenso eine Abschätzung der Gaskosten in Verbindung mit Testdeployments vor dem offiziellen Deployment sehr zu empfehlen.

Prüfungen der Eingabedaten können bei Aufrufen einem Überschreiben des Speichers durch zu lange Daten oder Gasproblemen durch komplexe Berechnungen vorbeugen. Zum Beispiel lässt sich die Größe der Aufrufdaten mit der erwünschten Größe entsprechend erwarteter Parameter abgleichen.

Over- und Underflows

Schließlich muss Over- und Underflows manuell begegnet werden. Es gibt zwar keine Datentypen wie uint8 in Solidity, die schnell überlaufen können, aber wenn es maximale Werte wie einen Tokenvorrat gibt, muss im Zuge von Transfers auf Konsistenz geprüft werden.

```
(when (|| (> value (mload balanceFrom))
          (> value allTokens)
          (> (+ value (mload balanceTo)) allTokens)
          (revert))
```

Obiges Beispiel zeigt die Prüfung eines übergebenen Wertes. So wird sichergestellt, dass er nicht größer als das Guthaben einer Adresse ist, nicht größer als der gesamte vorhandene Betrag und dass der Empfänger nach Erhalt nicht mehr als den Gesamtvorrat besitzt.

Sprachspezifische Gefahren

In LLL gibt es zudem nur Ganzzahldivision, was bedeutet, dass Vorsicht im Hinblick auf gerundete Ergebnisse geboten ist. Durch die geringe Abstraktionsschicht ist LLL als Sprache insgesamt fehleranfälliger, besonders für Einsteiger. Da viele manuelle Prüfungen erforderlich sind, ist das Gefahrenpotenzial durch Vergessen dieser nicht von der Hand zu weisen. Zugriffe und Sichtbarkeiten sind deutlich weniger erkennbar als in Vyper. Es schützen auch fehlende Funktionen nicht vor Fehlerquellen, da Opcodes wie „Call“ und „Callcode“ trotzdem verfügbar und direkt einsetzbar sind. Viele Funktionalitäten können nachgebildet werden, wobei wiederum eine Gefahr durch fehlerhafte Umsetzung besteht. Auch die Syntax und der schlechtere Überblick über den Quellcode können Sicherheitsrisiken bedeuten. Dies erschwert darüber hinaus den Audit. Eine Modularisierung kann hier hilfreich sein. So können wiederverwendbare Makros in eine separate Datei ausgelagert und bei Bedarf importiert werden. Ein Problem stellt allerdings die mögliche Gleichbenennung von Makros dar. So müssen sie sich lediglich in ihren Parametern unterscheiden. Missverständlicher Code ist somit nicht auszuschließen.

Insgesamt bestehen in LLL gegen die häufigen Unsicherheiten keine nativen Schutzmechanismen. Es lassen sich einige Security Patterns von Solidity ableiten, trotz alledem sind deutlich mehr manuelle Prüfungen erforderlich. Fehler sind aufgrund der Syntax schwerer auffindbar.

Bamboo

Reentranz

Im Gegensatz zu den anderen Sprachen findet man in Bamboo eine native Reentranzprüfung. So wird bei externen Aufrufen nach dem Keyword „reentrance“ angegeben, was in einem solchen Fall zu tun wäre, zum Beispiel ein Abbruch mittels „abort“:

```
bool success = market.buyToken() reentrance { abort; }
```

Somit ist dies die einzige der betrachteten Sprachen, die eine derartige native Prüfung bietet.

Transaction Ordering Dependence

Gegen die Transaction Ordering Dependence muss jedoch, analog zu den anderen Sprachen, eine eigene Lösung implementiert werden. So können auch hier Kautionen hinterlegt oder Prä-Commitments eingesetzt werden.

Unsichere Zufallsgeneratoren

Auch ist Bamboo durch unsichere Zufallsgeneratoren gefährdet. So finden sich keine Funktionen zur Erzeugung eines verlässlichen Zufallswertes. Beispielsweise ist „now(block)“ als Zeitstempel nicht zu empfehlen. Darüber hinaus ist unklar, ob der Zugriff auf RANDAO bereits möglich ist, da bisher keine Interfaces unterstützt werden. Folglich können die zu nutzenden Funktionen nicht auf diese Weise definiert werden. Man müsste sie eventuell neu als zusätzlichen Contract in die Datei implementieren, denn in jedem Fall würde für eine Interaktion die ABI benötigt.

Abhängigkeit von Zeitstempeln

Da hier ein Zeitstempel vergleichbar mit denen der anderen Sprachen vorhanden ist, besteht eine Unsicherheit bei Abhängigkeit von diesem. Daher kann die 30-Sekunden-Regel genutzt werden, um einzuschätzen, inwieweit ein Contract für Angriffe, die auf Manipulation der Zeit basieren, anfällig ist. Gegebenenfalls sollte, wie bereits in den anderen Sprachen beschrieben, zum Beispiel die Blocknummer als Alternative verwendet werden.

Ungeprüfte Rückgabewerte

Um Gefahren bezüglich ungeprüfter Rückgabewerte zu vermeiden, sind auch in Bamboo bislang Funktionen manuell auf ihren Wert zu prüfen und im Fehlerfall geeignet zu behandeln. Lediglich ein fehlschlagendes Deployment löst eine Ausnahme aus.

Schlecht umgesetzte Zugriffskontrolle

Bamboo verfügt weder über Sichtbarkeitsmodifikatoren noch über native Funktionen zur Zugriffskontrolle. Da jedoch in jeder Contractform nur Funktionen zur Verfügung stehen, die dort aufrufbar sein sollen, besteht eine geringere Gefahr dafür, dass eine Funktion in unerwünschtem Kontext aufgerufen wird. Will man Zugriffe auf bestimmte Umstände beschränken, so müssen, vergleichbar mit Vyper, diese in jeder Funktion entsprechend implementiert werden:

```
if (sender(msg) != _owner)
        abort;
```

Grundsätzlich sind in Bamboo alle Funktionen öffentlich aufrufbar.

Denial of Service

Die Gefahr eines Denial of Service Angriffes besteht auch mit dieser Sprache, jedoch mit Einschränkungen. So werden keine Schleifen unterstützt, damit es nicht so leicht zu Out-of-Gas kommt beziehungsweise das Blockgaslimit erreicht wird. [vgl. 44] So müsste jede Iteration in einer separaten Transaktion erfolgen. Auszahlungen in Schleifen sind dementsprechend keine Gefahr was die Verhinderung von (Rück-) Zahlungen bei mehreren Empfängern betrifft. Man sollte jedoch trotzdem mit Fehlschlägen externer Aufrufe durch Rollbacks beim Empfänger rechnen und für Zahlungen Pull-Mechanismen einsetzen, damit eine fehlschlagende Zahlung nicht den Contract blockiert, sondern nur die Transaktion dessen betrifft, der die Funktion zum Einzug der Token aufruft. Auch besteht die Gefahr, dass der „default“-Fall eines Contracts zu komplex gestaltet ist, sodass zu viel Gas verbraucht wird, wenn er aufgerufen wird. Bisher ist es noch nicht dokumentiert, ob ein justieren des bei direktem Aufruf mitzusendenden Gas überhaupt möglich ist. Auch bei Berechnungen, die Eingabedaten von Nutzern verwenden, sollte eine Prüfung der Argumente auf Zulässigkeit erfolgen, um unerwünschter Komplexität und Fehlern vorzubeugen.

Over- und Underflows

Anders als Vyper, aber analog zu Solidity und LLL, findet man hier keine eingebauten Grenzprüfungen. So sollte man zum einen Datentypen in der richtigen Größe wählen und zum anderen manuell auf Über- oder Unterlauf bei Arithmetik prüfen sowie die Größe von Arrays verfolgen.

```
if (balances[sender(msg)] < _amount) abort;
if (balances[_to] + _amount < balances[_to]) abort;
if (totalSupply < _amount) abort;
```

Dieses Beispiel zeigt Prüfungen von Tokenguthaben auf Überlauf und unerwünschte Werte. Zu beachten gilt es auch hier, dass Denkfehler in Prüfungen ebenfalls Unsicherheiten zur Folge haben können.

Sprachspezifische Gefahren

Da bisher weder Fließ- noch Festkommazahlen unterstützt werden, sollte bei Berechnungen hier auf Ungenauigkeiten durch Runden geachtet werden. Auch ist es eine potenzielle Gefahr, dass Variablen und Funktionen gleich benannt werden können, da dies zu Missverständnissen und Verwechslungen führen kann, besonders, falls zukünftig Funkti-

onsvariablen unterstützt würden. Bamboos noch eingeschränkte Funktionalität erfordert, ähnlich wie in LLL, teilweise eigenständige Implementationen, zum Beispiel von sicherer Arithmetik. Bibliotheken können derzeit nicht eingebunden werden. Dadurch kann es, wie in vorigen Kapiteln bereits angesprochen, zu Fehlern durch eigene Umsetzung kommen, zumal diese in der Regel weniger Prüfungen unterzogen werden, als nativ eingebaute Funktionen. Die Syntax der Sprache ist von weitaus geringerer Fehleranfälligkeit als in LLL. Da keine Sichtbarkeiten definierbar sind, hat man einerseits nicht die Hürde der richtigen Auswahl, andererseits jedoch auch keine Möglichkeit, private Funktionen zu definieren. Somit sind grundsätzlich alle Funktionen öffentlich, was man nicht außer Acht lassen darf. Die fehlende Unterstützung von Assemblercode schließt eine Fehlerquelle, die in Solidity und LLL besteht, aus. Auch Vererbung ist nicht möglich, so können weitere Funktionen stattdessen in eine neue Contractform integriert werden. Ein Hindernis kann in Bamboo jedoch die bislang fehlende „payable“-Markierung sein, was es gegebenenfalls erforderlich macht, bei jeder Funktion eine Prüfung auf Erhalt von Ether zu integrieren, in der dies eine Rolle spielen könnte. So kann ein Rollback durchgeführt werden, wenn unerwünscht „value(msg)“ nicht Null ist. Ungünstig ist zudem, dass es bisher keine Dokumentation über eine Funktion zum Etherversand gibt, die zum Beispiel mit „send()“ in Vyper oder „transfer()“ in Solidity vergleichbar ist. Da es sich dabei um eine wichtige Funktion handelt, die auch in Bezug auf eine sichere Benutzung in Contracts gut dokumentiert werden muss, ist dies für eine zukünftige Nutzung der Sprache unbedingt notwendig.

Insgesamt kann Bamboo teilweise Schutzmechanismen gegen häufige Gefahren aufweisen, was ein Vorteil gegenüber Solidity und LLL ist. So kann Reentranz vermieden und einige Risiken Denial of Service betreffend, insbesondere jene im Zusammenhang mit Schleifen, minimiert werden. In anderen Fällen ist eine manuelle Absicherung nötig, die teilweise durch Sicherheitsmuster, die von Solidity ableitbar sind, möglich ist. Einige Mechanismen, wie der Einsatz geprüfter Bibliotheken, sind noch nicht möglich. Dafür ist die Begrenzung von Funktionen auf bestimmte Contractformen ein guter Weg, um aufrufbare Funktionen einzuschränken. Somit weist Bamboo, die Sicherheit betrachtend, bereits viele gute Ansätze auf.

4.3 Vor- und Nachteile des Einsatzes der EVM als Laufzeitumgebung

Die Ethereum Virtual Machine stellt die Laufzeitumgebung der in dieser Arbeit beschriebenen Sprachen dar und hat sich im Allgemeinen als virtuelle Maschine für Smart Contracts durchgesetzt. Durch das neue eWASM Projekt, welches eine Alternative zur EVM bieten und sie möglicherweise sogar zukünftig ersetzen soll, stellt sich die Frage nach den Vor- und Nachteilen, die die klassische Laufzeitumgebung aufweist und welche Eigenschaften eine Ersetzung dieser rechtfertigen beziehungsweise dem entgegensprechen könnten.

Da es sich bei der EVM um eine Turing-vollständige virtuelle Maschine handelt, die nur durch das Gaskonzept darin beschränkt wird, erlaubt sie die Entwicklung komplexer Anwendungen und ermöglicht daher die Realisierung einer Vielzahl unterschiedlicher Programme auf Blockchainbasis. So können neben klassischem Tokenhandel auch Lizenz- und Nutzerverwaltungen sowie beispielsweise Tracking- und Zertifizierungsanwendungen im Bereich Supply Chain umgesetzt werden. Zudem ist sie speziell für die Ethereum Blockchain und Interaktionen mit dieser entwickelt, sodass viele blockchainspezifische Features vorhanden sind. Es gibt 32 Byte große Speicherslots, um Kryptografie, die wesentlich für die sichere Eigenschaft der Blockchain ist, zu erleichtern sowie um Adressen in einen Slot ablegen zu können und somit leichter zu handhaben. Darüber hinaus kann der Speicher nicht nur stackartig wie bei Bitcoin angesprochen werden, denn beispielsweise bei Arrays in Storage und Memory ist ein zufälliger Zugriff auf Elemente möglich. Schließlich sind die Ergebnisse der Codeausführung vollständig deterministisch, da das Gaskonzept verhindert, dass es zu endlosen Schleifen kommt. So kann im Allgemeinen davon ausgegangen werden, dass jede Anwendung zum Ende kommt, zumindest, wenn alles Gas aufgebraucht wurde. [vgl. 77, 78]

Die Ethereum Virtual Machine weist jedoch auch einige Probleme und Schwerfälligkeiten auf, die teilweise mit ihrer Anpassung an die Blockchain zusammenhängen. So sind vor allem die 32 Byte großen Speicherslots und Integer ineffizient für Berechnungen und verringern die Performanz, da Prozessoren auf 16, 32 oder 64 Bit Arithmetik ausgelegt sind. Dies macht 256 Bit Operationen entsprechend langsam und aufwändig. So benötigt eine 32 Bit Maschine zur Addition zweier 32 Bit Integer nur zwei Operationen, dagegen sind bei zwei 256 Bit Integern 16 Schritte erforderlich [vgl. 79]:

```
mov eax, dword [number1]
add eax, dword [number2]
```

Abbildung 24: Addition 2er 32 Bit Integer [79]

```
mov eax, dword [number]
add dword [number2], eax
mov eax, dword [number1+4]
adc dword [number2+4], eax
mov eax, dword [number1+8]
adc dword [number2+8], eax
mov eax, dword [number1+12]
adc dword [number2+12], eax
mov eax, dword [number1+16]
adc dword [number2+16], eax
mov eax, dword [number1+20]
adc dword [number2+20], eax
mov eax, dword [number1+24]
adc dword [number2+24], eax
mov eax, dword [number1+28]
adc dword [number2+28], eax
```

Abbildung 25: Addition 2er 256 Bit Integer [79]

Als Grund für die Größe der Speicherslots werden, wie bereits angesprochen, die Adressen und Kryptografie angegeben. Allerdings wird in Contracts selbst nahezu keine Kryptografie implementiert, da dies sehr rechenintensiv und dementsprechend teuer werden kann. Daher werden dafür in der Regel native Contracts wie zum Beispiel der keccak256 in Solidity genutzt, was somit dieser Begründung widerspricht. Auch sind Speicheradressen 256 Bit groß, obwohl so viel gar nicht zu belegen ist und derartig große Datentypen sind selbst für das Zählen von Wei ungeeignet, da so eine große Menge nicht existieren kann. Allerdings ist es nicht von der Hand zu weisen, dass sie sich für die Ablage von Adressen und zur Speicherung von Hashwerten besonders eignen. [vgl. 79]

Ebenfalls problematisch ist die Fragmentierung des Speichers, denn es gibt keine Möglichkeit zu dessen Freigabe bei der Programmausführung. So wird für jedes benötigte Stück Speicher dieser neu allokiert und man müsste selbst die Speicheradressen überwachen, um nicht mehr benötigte Bereiche neu zu überschreiben. Dies birgt allerdings das Risiko der Korruption des Speichers, wenn dieser doch noch benötigt wird. Zudem ist der Überblick über den Speicher schlecht, denn es genügt das Schreiben an eine ungenutzte Stelle, um diese neu zu allokieren. Operatoren für das Speichermanagement wie beispielsweise in C existieren nicht. Deshalb können die Speicherkosten sehr hoch werden, so steigen sie quadratisch mit dem Speicherverbrauch. Es gibt auch keine Möglichkeit, auf beliebige Elemente des Stack zuzugreifen, denn im Gegensatz zum Memory und Storage lassen sich nur die oberen 16 Elemente ansprechen. Aus diesem Grund können nicht mehr lokale Variablen angelegt werden. Leider ist jedoch der Stack der einzige kostenfreie Speicher. [vgl. 79]

Weiterhin steht der einfache Satz an Opcodes der Kompaktheit im Weg, zumal für komplexere Operationen so mehr Schritte erforderlich sind. Auch kann in einen Smart Contract nicht einfach eine externe Funktion im Zuge eines Aufrufes einzeln geladen werden. So muss immer der gesamte Contract in den Memoryspeicher geholt werden, was es billiger macht, kleinere Funktionen selbst neu in den eigenen Contract einzufügen. Schließlich ist auch das Debugging schwierig, da es nur eine Ausnahme, nämlich Out-of-Gas, gibt und kein Logging. Angesichts der Tatsache, dass zudem generell kaum Debugger für die EVM und ihre Programmiersprachen existieren, wird das Auffinden von Fehlern nicht gerade erleichtert. [vgl. 79]

Oft kritisiert werden zudem die hohen Kosten, die mit der Blockchain-Interaktion einhergehen. Ein einfacher Ethertransfer kostet bereits 21 000 Gas, was bei aktuell durchschnittlichem Gaspreis in etwa 65 Cent (0.76$) entspricht. Wie teuer der Speicher ist, zeigt ein Preis von 20 000 Gas für das Ablegen einer 32 Byte Zahl. Das ist fast so viel wie eben genannter Transfer. Ein Gigabyte zu speichern kostet demnach momentan über 20 Millionen Dollar bei durchschnittlichem Gaspreis, was in etwa 54 000 Ether entspricht. Vergleicht man dies mit Amazon AWS, so zahlt man dort in etwa 0.02$. Der Unterschied ist somit gravierend. [vgl. 77, 10, 62]

Einigen dieser Probleme soll eWASM begegnen: Datentypen sind kleiner und an übliche Prozessoren angepasst (32 und 64 Bit). Dies soll die Effizienz und Geschwindigkeit in der Ausführung steigern. Auch die Gaskosten für Operationen sind geringer als ihr EVM Äquivalent geplant [vgl. 68, backwards_compatibility.md]. Darüber hinaus können so etablierte Programmiersprachen wie C, C++ und Rust genutzt werden, für die bereits viele Entwicklungsumgebungen und Debugger existieren und die eine Vielzahl von Bibliotheken bieten. So soll für die Clients ein Transcompiler die Umwandlung von EVM-Bytecode zu eWASM-Code ermöglichen oder beide Laufzeitumgebungen unterstützt werden. Ob eWASM langfristig tatsächlich in der Lage ist, die EVM abzulösen, bleibt jedoch abzuwarten.

4.4 Zusammenfassung und Einschätzung

Betrachtet man die in Smart Contracts am häufigsten vorkommenden Sicherheitslücken zeigt sich, dass sie zum Großteil in allen EVM-basierten Sprachen relevant sind. Fast alle Dokumentationen dieser beziehen sich auf Solidity Contracts, obgleich sich einige der Sicherheitsmuster auch in die anderen Sprachen übernehmen lassen, wie beispielsweise das „Checks-Effects-Modell“ oder der „Pull“-Mechanismus.

Abgesehen von Bamboo sind alle betrachteten Sprachen durch Reentranz gefährdet, besonders, da nicht nur die Funktionen der Sprachen und der Aufbau des Codes, sondern auch die EVM Opcodes selbst (wie „CALL“) betroffen sind. Solidity wurde daher um Funktionen wie „transfer()“ und „send()“ ergänzt, deren Gas streng limitiert ist.

Auch durch die Eigenschaften der Blockchain sind manche Gefahren in jeder Sprache auffindbar. Dies gilt für die Transaction Ordering Dependence, da die Reihenfolge der Transaktionen in einem Block nicht von vornherein klar ist. So könnte sogar eine Transaktion mit niedrigerem Gaspreis theoretisch vor einer mit höherem Betrag in den Block gelangen. Dies kann nicht von der Allgemeinheit vorhergesehen werden und hängt vom Miner des Blockes ab. Deshalb ist ein Schutz vor dieser Gefahr schwierig.

Eine weitere Gefahr, die aus den Eigenschaften der Blockchain resultiert und in allen Sprachen eine Rolle spielt, ist die Öffentlichkeit der Daten: vermeintlich sichere Zufallsgeneratoren, die Blockdaten verwenden sowie der Zeitstempel sind unzuverlässig und teilweise manipulierbar. Dies zeigt auch, dass die Sicherheit in hohem Maße von den Minern und ihrer Aufrichtigkeit abhängt, denn in vielen Fällen könnten sie sich oder anderen Vorteile verschaffen. Jener Art von Gefahren wird mithilfe von Sicherheitsmustern begegnet, die sich zum Großteil auf alle Sprachen anwenden lassen.

Viele Funktionen der Sprachen sind darüber hinaus von Fehlern im Zusammenhang mit ungeprüften Rückgabewerten betroffen. Lediglich „transfer()“ in Solidity und fehlschlagende Deployments in den jeweiligen Sprachen lösen von sich aus Exceptions aus. Demzufolge sind auch hier entsprechend allgemeiner Empfehlungen manuell Prüfungen umzusetzen. Gleiches gilt für Zugriffskontrollen, hier hat Vyper den Vorteil, dass eine Sichtbar-

keitsmarkierung von Funktionen obligatorisch ist. Bamboo erlaubt dafür die Eingrenzung erreichbarer Funktionen durch deren Beschränkung auf bestimmte Contractformen.

Allerdings ist es in Vyper sowie Bamboo problematisch, dass durch die noch unvollständige Dokumentation oder noch fehlende Umsetzung bei manchen Funktionen das Verhalten im Fehlerfall unklar ist, wovon beispielsweise Vypers „send()“ und eine noch nicht dokumentierte entsprechende Funktion in Bamboo betroffen sind. Dies kann unerwünschtes Verhalten bedeuten, wenn dieses von der Erwartung abweicht. Das gilt ebenso für die fehlende „payable“-Markierung in Bamboo.

Gegen DoS können in den jeweiligen Sprachen bezüglich eines unerwarteten Rollbacks Security Patterns zum Einsatz kommen, die verhindern, dass der gesamte Contract durch diesen Fehlschlag aufgehalten wird. Dazu gehört der „Pull“-Mechanismus bei Auszahlungen. In Bezug auf Out-of-Gas, zum Beispiel durch Schleifen, besitzt nur Bamboo durch Verzicht auf solche im Allgemeinen weniger Risiko. Vyper erlaubt zwar Schleifen, jedoch keine endlosen. Da dies die einzige Sprache mit integrierter Grenz- und Überlaufprüfung ist, kann auch durch Over- oder Underflows hier keine Endlosschleife entstehen und alles Gas konsumieren. Die eingebaute Gasschätzung von Vyper ist ein weiterer Vorteil, der dabei hilft, Gasprobleme von Funktionen frühzeitig zu erkennen. Solidity bietet keine weiteren Sicherheitsfeatures, lediglich „require()“ ist bei externer Interaktion beziehungsweise zur Eingabeprüfung durch ein automatisches Rollback im Fehlerfall ein sehr nützliches Hilfsmittel.

Auffällig ist, dass LLL keinerlei Sicherheitsmechanismen bietet. Da diese Sprache sehr EVM-nah ist, muss aus vorhandenen Makros und Opcodes vieles an Funktionalität selbst implementiert werden, so wie beispielsweise das not-payable-Makro im ERC20 Contract. Ein Problem dabei ist, dass vorhandene Opcodes wie „CALL“ und „DELEGATECALL“ selbst Gefahrenpotenzial besitzen. Durch die fehleranfällige Syntax und direkte Speicherzugriffe ist die Sprache besonders Anfängern nicht zu empfehlen. So gibt es zu viele mögliche Fehlerquellen. Dazu gehört auch die Makroüberladung. Zudem sind die Sicherheitsmuster hier schwieriger zu übertragen als in die anderen Sprachen. Dies lässt sich zum einen mit der Syntax und zum anderen mit fehlenden nativen Makros begründen. Insgesamt bestehen in LLL die meisten Sicherheitsrisiken, besonders, wenn die Sprache ohne tiefergehende Erfahrung mit der Blockchaintechnologie benutzt wird.

Vyper und Bamboo bieten somit nativ mehr Sicherheit als Solidity und LLL. Solidity hat zwar größeren Support und Funktionsumfang, aber auch viele spezielle Gefahren wie Überladung, Rekursion, Assembly sowie fehlende Sichtbarkeitsvorschriften. Mittlerweile gibt es viel Aufklärung zu Unsicherheiten im Bereich von Smart Contracts und die meisten Sicherheitsmuster sind für Solidity entworfen. Auch Bibliotheken wie SafeMath von Open Zeppelin sollen Gefahren minimieren. Für Einsteiger sind die vielen Funktionen und Hilfestellungen sehr hilfreich. Da Vyper allerdings mehr Sicherheit bietet, ist es für einige Contracts, die nicht auf Vererbung angewiesen sind, eine gute Alternative. So kann es zum Beispiel für den ERC20 Contract eine bessere Wahl sein. Auch Bamboo kann, die

Sicherheit betreffend, für bestimmte Anwendungsfälle eine sicherere Alternative zu Solidity darstellen. In folgender Tabelle sind die Sicherheitsaspekte der Sprachen noch einmal kurz zusammengefasst:

Tabelle 2: Sicherheitsaspekte der Sprachen

Sprache	**Native Sicherheitsmechanismen**	**Spezifische Gefahren**	**Sicherheitsmuster vorhanden?**
Solidity	- send() und transfer() erlauben keine Reentranz - transfer() führt Rollback bei Fehler durch - assert und require für Prüfungen	Überladung, keine Sichtbarkeitsvorschriften, Vererbung, sehr große Menge an Funktionen, Assembly, Rekursion, Abrunden bei Integerdivision	Ja
Vyper	- Over- /Underflowprüfung - obligatorische Sichtbarkeit von Funktionen - kein Überladen, Vererbung, Assembly, Rekursion, Endlosschleifen, Import - Festkommazahlen - native Gasschätzung	Unklares Verhalten bei send()-Fehlschlag und unklare Menge an mitgeliefertem Gas, unvollständige Dokumentation	Ja, ableitbar von Solidity
LLL	------	Syntax, Speicherzugriff, viele manuelle Prüfungen, Makroüberladung, keine Sichtbarkeiten, sicherheitskritische Opcodes, Assembly	Ja, teilweise von Solidity ableitbar, aber schwieriger
Bamboo	- Reentranzprüfung - kein DoS oder Out-of-Gas durch Schleifen - Limitierung von Funktionen	Keine Sichtbarkeiten oder payable-Markierung, gleiche Variablen- und Funktionsbenennung möglich,	Ja, leicht von Solidity ableitbar, sofern unterstützt

	auf bestimmte Contractform - kein Assembly, Vererbung, Import	unvollständige Dokumentation, keine Hinweise auf Funktion zum Etherversand bisher	

Da alle der betrachteten Sprachen auf der EVM basieren, sind sie auch von deren Vor- und Nachteilen beeinflusst. Sie profitieren von der Anpassung auf die Blockchain, mussten allerdings dafür neu entwickelt werden, sodass keine Erfahrungen, Tools und Compiler, die schon länger existent sind, eingesetzt werden können. Daher gab und gibt es viele unbekannte Gefahren, die erst im Zusammenhang mit Vorfällen erkannt werden.

Die 32 Byte großen Speicherslots hemmen die Sprachen in ihrer Effizienz und verursachen zu große Bytecodes, sodass entsprechend hohe Kosten daraus resultieren können. Der EVM und damit auch den Sprachen fehlt es an Debuggingmöglichkeiten, obgleich es bereits Tools zum Test von Solidity Contracts gibt. Zudem erlaubt die Turing-Vollständigkeit zwar theoretisch komplexe Anwendungen, jedoch sind solche durch die entstehenden Kosten praktisch nicht sinnvoll.

Daher sind neue Ansätze wie eWASM für Smart Contracts sehr interessant. Eine Verbesserung der Effizienz und kleinere Datentypen könnten auch die Kosten verringern. Zudem kann der Einsatz etablierter Programmiersprachen mit ihren Tools und Bibliotheken durch viel Erfahrung und Expertise von Vorteil sein. Die allgemeinen Gefahren dieser Sprachen sind bekannter, jedoch nicht in Verbindung mit der Blockchain. Deshalb ist es fraglich, inwieweit sie für die Anwendung in Smart Contracts nützlich sind. Gefahren bezüglich Blockchaineigenschaften bestünden auch mit eWASM. Ebenso kann der sehr große Umfang von Sprachen wie C oder C++ neue Risiken bedeuten, beispielsweise durch Threads oder Pointer. Momentan sind eWASM Bytecodes noch sehr groß, so umfasst der Bytecode eines ERC20 Beispiels von Parity 26 897 Bytes, was signifikant größer als die der betrachteten EVM-Sprachen ist [vgl. 80]. Somit ist auch bei eWASM einiges an Optimierungen nötig, wenn es in Bezug auf Effizienz und Kosten die EVM ersetzen soll.

5 Fazit und Ausblick

Die vergleichende Betrachtung von Solidity, Vyper, LLL und Bamboo hat gezeigt, dass es bezüglich der Bytecodegrößen und Deploymentkosten unter den Sprachen einige gibt, die in bestimmten Punkten besser abschneiden als Solidity. So ist LLL mit Abstand die günstigste und speichersparendste Sprache, Vyper nimmt dabei den zweiten Platz ein und Bamboo schneidet hier, zumindest noch aktuell, am schlechtesten ab. Dafür weist diese Sprache bessere Sicherheitsmechanismen und einige schützende Restriktionen auf, die Solidity nicht bietet. In Bezug auf die Sicherheit schneidet auch Vyper gut ab, außerdem kann diese Sprache geringfügig besser als Bamboo eingeschätzt werden, da sie weiter entwickelt und mit Sichtbarkeiten sowie payable-Markierung im Vorteil ist. LLL kann als gefahrenträchtigste Sprache eingeordnet werden, denn sie erfordert speziellere Kenntnisse durch die Syntax. Die vielen manuell umzusetzenden Mechanismen sind fehleranfällig. Am leichtesten zu entwickeln sind Solidity Contracts, da bereits einige Tools wie Remix vorhanden sind und Bequemlichkeiten bieten. Auch besitzt Vyper einen Onlinecompiler, das Deployment muss jedoch, analog zu LLL und Bamboo, über die Kommandozeile erfolgen. Zudem erfordern die letzteren beiden Sprachen ein Kompilieren über die Konsole. Aus diesem Grund ist wohl Solidity durch die umfangreichste Unterstützung am besten für Einsteiger geeignet. Wer jedoch mit der Kommandozeile umgehen kann, wird auch mit Vyper einen leichten Einstieg finden. Bamboo und LLL sind dagegen für Anfänger nicht unbedingt zu empfehlen.

Betrachtet man die Sicherheit und Kosten zusammen, liegt Solidity mit durchschnittlicher Wertung im Mittelfeld. LLL weist zwar deutlich geringere Kosten auf, bietet aber auch weniger Sicherheit und schneidet daher insgesamt schlechter ab. Bamboo kann mit guter Sicherheit punkten, jedoch sind die sehr hohen Kosten von Nachteil. Demzufolge ist sie nur geringfügig besser als LLL einzuordnen. Langfristig kann sich dies jedoch durch die weitere Entwicklung noch verbessern. Zudem ist es durchaus denkbar, dass bei bestimmten Anwendungen eine gute Sicherheit auch höhere Kosten rechtfertigt. Schließlich ist Vyper mit guter Sicherheit und durchschnittlichen Kosten gegenüber Solidity etwas besser einzuordnen. So waren die Kosten und die Bytecodegröße etwas geringer und die Verständlichkeit gut. Insgesamt stellt sie daher die beste Alternative der betrachteten Sprachen zu Solidity dar.

Natürlich kann nicht allgemein von der sichersten und günstigsten Sprache gesprochen werden, da andere in einzelnen Kategorien durchaus besser abschneiden. So hat jede Programmiersprache ihre Stärken und Schwächen und die Wahl hängt auch vom konkreten Anwendungsfall ab. Wird Vererbung benötigt, bietet sich vielleicht Solidity an, wohingegen bei beschränkteren Contracts Vyper eine geeignetere Sprache sein kann. So haben auch die übrigen Sprachen ihre Anwendungsfälle. Folgende Abbildung soll noch ein-

mal die Einordnung der Sprachen zum Abschluss grob zusammenfassen. Die Zahlen der Achsen stehen für „keine“, „durchschnittlich“, „hoch“ und „sehr hoch“. Je heller der Hintergrund, desto besser ist eine Sprache bezüglich Sicherheit und Kosten zur Contractentwicklung geeignet:

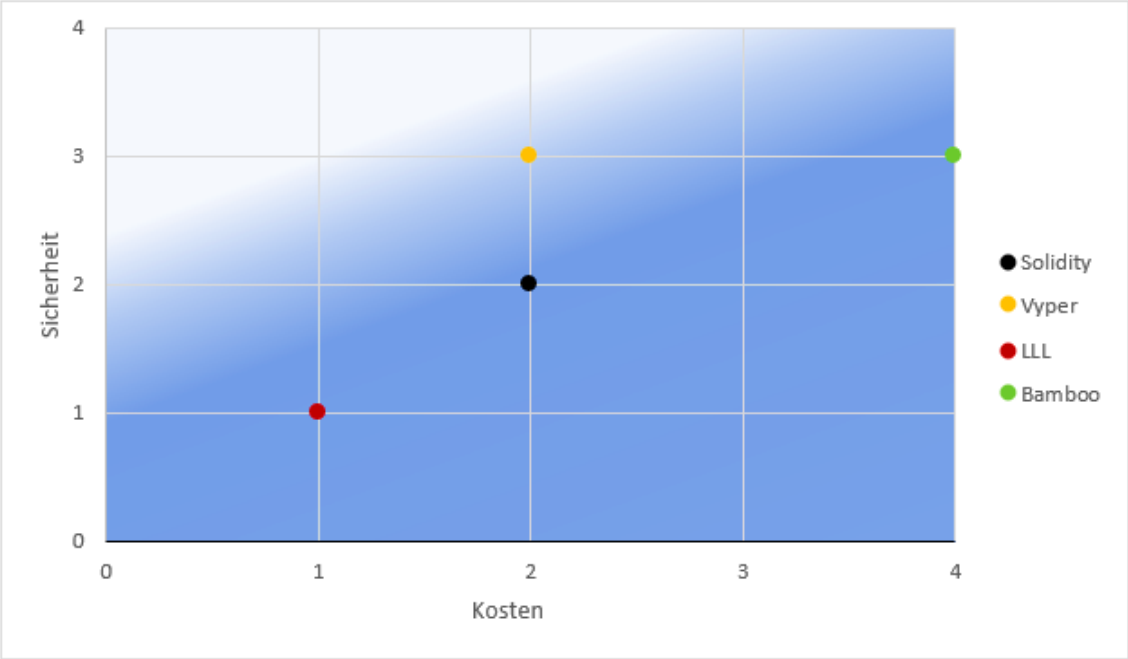


Abbildung 26: Einordnung der Sprachen

Natürlich sind die betrachteten Sprachen nicht die einzigen, die für die Entwicklung von Smart Contracts existieren. Jedoch sind es die aktuell bekanntesten. Einen weiteren interessanten Ansatz bietet beispielsweise „Babbage“, eine Sprache, die es möglich machen soll, leicht verständliche Contracts zu entwickeln. Dabei werden diese als Maschine dargestellt, die den Ablauf der Funktionen visualisiert. Dies kann man sich ähnlich wie bei der Darstellung von Schaltungen vorstellen. So soll leichter zu sehen sein, wann welche Funktion erreichbar ist und wie sie einander beeinflussen. Aktuell ist Babbage jedoch noch lediglich ein Konzept. [vgl. 81] So zeigt auch dieser Ansatz, dass es immer wieder neue Sprachen und Denkansätze für die Contractentwicklung gibt und sich das Ethereum-Netzwerk in ständiger Weiterentwicklung befindet.

Es wird beispielsweise ein Wechsel vom Proof of Work zu Proof of Stake geplant, bei dem es Validatoren geben soll, die abwechselnd neue Blöcke vorschlagen und validieren beziehungsweise über einen neuen abstimmen. Dabei hängt das Gewicht der Stimme eines Validators von dessen hinterlegter Kaution ab. Proof of Stake soll deutlich energieeffizienter als das bisherige Proof of Work sein. Bei Fehlverhalten eines Validators, zum Beispiel bei Abstimmung für Blöcke auf mehreren Zweigen der Hauptkette, führt dies zum Verlust der Kaution. [vgl. 82] Das Protokoll dafür wird als „Casper“ bezeichnet. Es befindet sich in der Entwicklung durch das Ethereum Development Team.

Ein weiteres vielversprechendes Projekt ist das „Sharding“, welches die Skalierbarkeit und den Transaktionsdurchsatz verbessern soll. Momentan ist dies mit sieben bis 15 Transaktionen pro Sekunde im Vergleich zu Visa-Zahlungen mit teilweise 50 000 deutlich langsamer [84]. Beim Sharding wird der State und die Historie in Teile geteilt, die als „Shards“ bezeichnet werden. Die Nodes in einem Shard müssen nur die Transaktionen innerhalb dessen validieren und diesen Teil der Kette speichern. Trotzdem gibt es eine Hauptkette, die die Header der Shards verwaltet. So sollen mehrere 1000 Transaktionen pro Sekunde möglich werden. Die Integration von Casper und Sharding soll gemeinsam erfolgen. [vgl. 83, 85]

Es wird kontinuierlich an Verbesserungen des Ethereum-Netzwerkes gearbeitet, daher ist es durchaus vorstellbar, dass es sich mit seiner Währung als globales Zahlungsmittel und Handelsplattform etabliert. Besonders die Smart Contracts bieten Potenzial für eine Anwendung in vielen Bereichen. Es wird sich zeigen, ob sich langfristig andere Programmiersprachen gegenüber Solidity durchsetzen können und vielleicht noch neue entstehen, die durch bessere Sicherheit und Kostengünstigkeit bestehende Sprachen überflügeln. Wünschenswert wäre, dass sie zur Verringerung der Angreifbarkeit von Contracts beitragen, um das Netzwerk besser gegen Missbrauch zu härten.

6 Literatur

[1] Wikipedia: Ethereum, (online), 2017. URL: https://de.wikipedia.org/wiki/Ethereum, verfügbar am 30.05.18

[2] ethdocs: Ethereum Homestead documentation: Introduction – What is Ethereum?, (online), 2016. URL: http://ethdocs.org/en/latest/introduction/what-is-ethereum.html, verfügbar am 30.05.18

[3] Wikipedia: Kryptowährung, (online), 2018. URL: https://de.wikipedia.org/wiki/Kryptow%C3%A4hrung, verfügbar am 30.05.18

[4] ethdocs: Ethereum Homestead documentation: Mining, (online), 2016. URL: http://ethdocs.org/en/latest/mining.html, verfügbar am 30.05.18

[5] GitHub: White Paper – ethereum/wiki, (online), 2018. URL: https://github.com/ethereum/wiki/wiki/White-Paper, verfügbar am 30.05.18

[6] ethdocs: Ethereum Homestead documentation: Account Management, (online), 2016. URL: http://ethdocs.org/en/latest/account-management.html, verfügbar am 30.05.18

[7] Readthedocs-Solidity: Introduction to Smart Contracts: The Ethereum Virtual Machine – Accounts, (online), 2018. URL: http://solidity.readthedocs.io/en/develop/introduction-to-smart-contracts.html#accounts, verfügbar am 30.05.18

[8] ethdocs: Ethereum Homestead documentation: Ether, (online), 2016. URL: http://ethdocs.org/en/latest/ether.html, verfügbar am 30.05.18

[9] Etherscan: Ethereum Average GasLimit Chart, (online), 2018. URL: https://etherscan.io/chart/gaslimit, verfügbar am 30.05.18

[10] ETH Gas Station: - , (online), 2018. URL: https://ethgasstation.info/index.php, verfügbar am 30.05.18

[11] Slock.it: The USN: Developing the Universal Sharing Network, (online), 2018. URL: https://slock.it/usn.html, verfügbar am 30.05.18

[12] Weipprecht, A.: Krypto Magazin: Was ist eine 51%-Attacke?, (online), -. URL: https://www.krypto-magazin.de/was-ist-eine-51-attacke/, verfügbar am 31.05.18

[13] Hecking, C.; de Vries, A.: Spiegel Online: Das Bitcoin-System verbraucht mehr Strom als die Schweizer Volkswirtschaft, (online), 2018. URL: http://www.spiegel.de/wirtschaft/unternehmen/bitcoin-energieverbrauch-fuer-produktion-und-verwaltung-a-1208635.html, verfügbar am 31.05.18

[14] Malmo, C.: Motherboard: Warum der aktuelle Ethereum-Hype auch eine Schattenseite hat, (online), 2017. URL: https://motherboard.vice.com/de/article/7x9zwx/warum-der-aktuelle-ethereum-hype-auch-eine-schattenseite-hat, verfügbar am 31.05.18

[15] Readthedocs-Solidity: Introduction to Smart Contracts: The Ethere-um Virtual Machine – Storage, Memory and the Stack, (online), 2018. URL: http://solidity.readthedocs.io/en/develop/introduction-to-smart-contracts.html#index-6, verfügbar am 31.05.18

[16] Howard: Qtum Blog: Diving Into The Ethereum VM, (online), 2017. URL: https://blog.qtum.org/diving-into-the-ethereum-vm-6e8d5d2f3c30, verfügbar am 31.05.18

[17] Wikipedia: Smart Contract, (online), 2018. URL: https://de.wikipedia.org/wiki/Smart_Contract, verfügbar am 31.05.18

[18] Eatough, J.: BTC-Echo: Rechtliche Unsicherheiten bei Blockchains und Smart Contracts, (online), 2017. URL: https://www.btc-echo.de/rechtliche-unsicherheiten-bei-blockchains-und-smart-contracts/, verfügbar am 31.05.18

[19] Diedrich, H.: Ethereum: Blockchains, digital assets, smart contracts, decentralized autonomous organizations, -1. Auflage - Leipzig: CreateSpace Independent Publishing Platform (u.a.), 2016

[20] Wikipedia: Solidity, (online), 2018. URL: https://de.wikipedia.org/wiki/Solidity, verfügbar am 31.05.18

[21] Readthedocs-Solidity: Solidity, (online), 2018. URL: http://solidity.readthedocs.io/en/develop/index.html, verfügbar am 31.05.18

[22] Crypto51: PoW 51% Attack Cost, (online), 2018. URL: https://www.crypto51.app/, verfügbar am 01.06.18

[23] Readthedocs-Solidity: Solidity in Depth – Types, (online), 2018. URL: http://solidity.readthedocs.io/en/develop/types.html, verfügbar am 01.06.18

[24] Readthedocs-Solidity: Solidity in Depth – Contracts, (online),

2018. URL:
http://solidity.readthedocs.io/en/develop/contracts.html, verfügbar am 01.06.18

[25] Readthedocs-Solidity: Solidity in Depth – Units and Globally Available Variables, (online), 2018. URL: http://solidity.readthedocs.io/en/develop/ units-and-global-variables.html, verfügbar am 01.06.18

[26] Buntix, JP: themerkle: What Is Ethereum's Viper Coding Language?, (online), 2017. URL: https://themerkle.com/what-is-ethereums-viper-coding-language/, verfügbar am 01.06.18

[27] CoinPupil: Ethereum's Vyper and Cardano's Plutus: Two Smart Contract Paradigms, (online), 2018. URL: https://coinpupil.com/ethereum/ethereum-vyper-cardano-plutus/, verfügbar am 01.06.18

[28] Banov, M.: Coinbureau: Beginners Guide to the Vyper Smart Contract Language, (online), 2018. URL: https://www.coinbureau.com/smart-contracts/beginners-guide-vyper-language/, verfügbar am 01.06.18

[29] Buterin, V.: Readthedocs-Vyper: Vyper, (online), 2017. URL: http://viper.readthedocs.io/en/latest/index.html, verfügbar am 01.06.18

[30] Buterin, V.: Readthedocs-Vyper: Structure of a Contract, (online), 2017. URL: http://viper.readthedocs.io/en/latest/structure-of-a-contract.html, verfügbar am 01.06.18

[31] Buterin, V.: Readthedocs-Vyper: Built in Functions, (online), 2017. URL: http://viper.readthedocs.io/en/latest/built-in-functions.html, verfügbar am 01.06.18

[32] Buterin, V.: Readthedocs-Vyper: Types, (online), 2017. URL: http://viper.readthedocs.io/en/latest/types.html, verfügbar am 01.06.18

[33] Buterin, V.: Readthedocs-Vyper: Vyper by Example, (online), 2017. URL: http://viper.readthedocs.io/en/latest/vyper-by-example.html#, verfügbar am 01.06.18

[34] fubuloubu: Reddit Community: -, (online), 2018. URL: https://www.reddit.com/r/ethdev/comments/86e948/beginners_guide_to_the_vyper_smart_contract/, verfügbar am 01.06.18

[35] Edington, B.: Readthedocs-LLL Compiler Documentation: LLL Introduction, (online), 2017. URL: http://lll-docs.readthedocs.io/en/latest/lll_introduction.html, verfügbar am 04.06.18

[36] Ellison, D.: media.consensys: An Introduction to LLL for Ethereum Smart Contract Development, (online), 2017. URL: https://media.consensys.net/an-introduction-to-lll-for-ethereum-smart-contract-development-e26e38ea6c23, verfügbar am 04.06.18

[37] Ellison, D.: media.consensys: The Structure of an LLL Contract – Part 1, (online), 2017. URL: https://media.consensys.net/the-structure-of-an-lll-contract-5adfd322be2, verfügbar am 04.06.18

[38] Ellison, D.: media.consensys: The Structure of an LLL Contract – Part 2, (online), 2017. URL: https://media.consensys.net/the-structure-of-an-lll-contract-part-2-bf57a5a91829, verfügbar am 04.06.18

[39] Ellison, D.: media.consensys: The Structure of an LLL Contract – Part 3, (online), 2017. URL: https://media.consensys.net/the-

structure-of-an-lll-contract-part-3-4c5caf7bef64, verfügbar am 04.06.18

[40] Ellison, D. <daniel@syrinx.net>: GitHubGist: A contract that illustrates LLL contract structure, (online), 2017. URL: https://gist.github.com/zigguratt/ea41e2af324ac35ce657d435fd538b21, verfügbar am 04.06.18

[41] Edington, B.: Readthedocs-LLL Compiler Documentation: Language Reference: Built-in Macros, (online), 2017. URL: http://lll-docs.readthedocs.io/en/latest/lll_reference.html#built-in-macros, verfügbar am 04.06.18

[42] Edington, B.: Readthedocs-LLL Compiler Documentation: Language Reference: LLL Syntax, (online), 2017. URL: http://lll-docs.readthedocs.io/en/latest/lll_reference.html#lll-syntax, verfügbar am 04.06.18

[43] Hirai, Y.: Medium: Bamboo compiler started producing EVM bytecode, (online), 2017. URL: https://medium.com/@pirapira/bamboo-compiler-started-producing-evm-bytecode-6a55e4633de9, verfügbar am 04.06.18

[44] Github: pirapira: Bamboo Manifest – pirapira/bamboo, (online), 2018. URL: https://github.com/pirapira/bamboo/blob/master/doc/manifest.md, verfügbar am 04.06.18

[45] Hirai, Y.: Medium: Lengthier Notes on Bamboo 0.0.03, (online), 2018. URL: https://medium.com/@yoichihirai/lengthier-notes-on-bamboo-0-0-03-ca463ab1b9bf, verfügbar am 04.06.18

[46] BTC-ECHO: Was ist ein ERC-20 Token?, (online), o.A. URL: https://www.btc-echo.de/tutorial/was-ist-ein-erc-20-token/,

verfügbar am 12.06.18

[47] Wikipedia: ERC20, (online), 2018. URL: https://en.wikipedia.org/wiki/ERC20, verfügbar am 12.06.18

[48] Bergmann, C.: BitcoinBlog: ERC20: Die Basis der meisten ICO, (online), 2017. URL: https://bitcoinblog.de/2017/10/03/erc20-die-basis-der-meisten-ico/, verfügbar am 12.06.18

[49] KryptoVergleich: Ethereum Blockchain: Wie funktionieren ERC20 Tokens, (online), o.A. URL: https://krypto-vergleich.de/erc20-tokens, verfügbar am 12.06.18

[50] Vessenes, P.: The ERC20 Short Address Attack Explained, (online), 2017. URL: https://vessenes.com/the-erc20-short-address-attack-explained/, verfügbar am 12.06.18

[51] Alt Street: Hackernoon: Create your own Ethereum Token., (online), 2018. URL: https://hackernoon.com/create-your-own-ethereum-token-bfa6302084da, verfügbar am 12.06.18

[52] Ray, S.: Hackernoon: ERC20 Tokens, (online), 2018. URL: https://hackernoon.com/erc20-tokens-b3b50c95ad08, verfügbar am 12.06.18

[53] Nash, G.: Medium: The Anatomy of ERC721, (online), 2017. URL: https://medium.com/crypto-currently/the-anatomy-of-erc721-e9db77abfc24, verfügbar am 12.06.18

[54] Horne, J.: Medium: The Playful Paradigm Shift, (online), 2018. URL: https://medium.com/@jacobscott/the-playful-paradigm-shift-4bf35d9d1d11, verfügbar am 12.06.18

[55] OpenZeppelin: Github: BasicToken.sol, (online), 2018. URL: https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/token/ERC20/BasicToken.sol, verfügbar am 12.06.18

[56] OpenZeppelin: Github: StandardToken.sol, (online), 2018. URL: https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/token/ERC20/StandardToken.sol, verfügbar am 12.06.18

[57] GitHub: ethereum/vyper: ERC20.v.py, (online), 2018. URL: https://github.com/ethereum/vyper/blob/master/examples/tokens/ERC20.v.py, verfügbar am 13.06.18

[58] Edginton, B.: GitHub: benjaminion/LLL_erc20: erc20.lll, (online), 2017. URL: https://github.com/benjaminion/LLL_erc20/blob/master/erc20.lll, verfügbar am 13.06.18

[59] GitHub: pirapira: pirapira/bamboo: 01b_erc20better.bbo, (online), 2017. URL: https://github.com/pirapira/bamboo/blob/master/src/parse/examples/01b_erc20better.bbo, verfügbar am 14.06.18

[60] Readthedocs-Solidity: Appilcation Binary Interface Specification, (online), 2018. URL: http://solidity.readthedocs.io/en/develop/abi-spec.html, verfügbar am 15.06.18

[61] Ryan, D.: Hackernoon: Costs of a Real World Ethereum Contract, (online), 2017. URL: https://hackernoon.com/costs-of-a-real-world-ethereum-contract-2033511b3214, verfügbar am 15.06.18

[62] Bankenverband: Währungsrechner, (online), 2018. URL: https://bankenverband.de/service/waehrungsrechner/, verfügbar

am 15.06.18

[63] Ryan, D.: GitHubGist: count_bytes.py, (online), 2017. URL: https://gist.github.com/djrtwo/4b80f6dfebc5575e66508edd95a3b0c2, verfügbar am 15.06.18

[64] GitHub: pirapira: Bamboo 0.0.01 Tutorial, (online), 2018. URL: https://github.com/pirapira/bamboo/blob/master/doc/tutorial.md, verfügbar am 15.06.18

[65] Wikipedia: WebAssembly, (online), 2018. URL: https://de.wikipedia.org/wiki/WebAssembly, verfügbar am 03.07.18

[66] WebAssembly.org: WebAssembly, (online), 2018. URL: https://webassembly.org/, verfügbar am 03.07.18

[67] WebAssembly.org: WebAssembly – Documentation, (online), 2018. URL: https://webassembly.org/docs/high-level-goals/, verfügbar am 03.07.18

[68] GitHub: ewasm/design: eWASM Design Overview and Specification, (online), 2018. URL: https://github.com/ewasm/design, verfügbar am: 03.07.18

[69] ZeroKnowledge.fm: Harryson, F., Rettig, L., Beregszaszi, A.: What and why of eWASM with Lane and Alex, (online), 2018. URL: http://www.zeroknowledge.fm/26, verfügbar am 03.07.18

[70] YouTube: Asseth Association Ethereum: J1 SAY 07 Alex Beregszaszi Why eWASM, (online), 2018. URL: https://www.youtube.com/watch?v=gJLVTG7uI5o, verfügbar am 03.07.18

[71] NCC Group: Decentralized Application Security Project (or DASP) Top 10 of 2018, (online), 2018. URL: http://dasp.co/, verfügbar am 04.07.18

[72] Luu, L.; Chu, D.-H.; Olickel, H.; Saxena, P.; Hobor, A.: Making Smart Contracts Smarter, (online), 2016. URL: https://eprint.iacr.org/2016/633.pdf, verfügbar am 04.07.18

[73] ConsenSys Diligence: Ethereum Smart Contract Best Practices: Known Attacks, (online), 2018. URL: https://consensys.github.io/smart-contract-best-practices/known_attacks/, verfügbar am 04.07.18

[74] Olickel, H.: Medium: Why Smart Contracts Fail: Undiscovered Bugs and what we can do about them, (online), 2016. URL: https://medium.com/@hrishiolickel/why-smart-contracts-fail-undiscovered-bugs-and-what-we-can-do-about-them-119aa2843007, verfügbar am 04.07.18

[75] GitHub: randao/randao: RANDAO: A DAO working as RNG of Ethereum, (online), 2018. URL: https://github.com/randao/randao, verfügbar am 05.07.18

[76] ConsenSys Diligence: Ethereum Smart Contract Best Practices: Recommendations for Smart Contract Security in Solidity, (online), 2018. URL: https://consensys.github.io/smart-contract-best-practices/recommendations/, verfügbar am 05.07.18

[77] crypto.bi: The Ethereum EVM computing cost is absurd, (online),-. URL: https://crypto.bi/tape/blog/gas/, verfügbar am 10.07.18

[78] Wilcke, J.: Medium: Optimising the Ethereum Virtual Machine, (online), 2016. URL: https://medium.com/@jeff.ethereum/optimising-the-ethereum-

virtual-machine-58457e61ca15, verfügbar am 10.07.18

[79] Earls, J.: Medium: The Faults and Shortcomings of the EVM, (online), 2017. URL: https://medium.com/@earlz/the-faults-and-shortcomings-of-the-evm-bde4d09b8b6a, verfügbar am 10.07.18

[80] GitHub: paritytech/pwasm-token-example: pwasm-token-example/compiled/token.wasm, (online), 2018. URL: https://github.com/paritytech/pwasm-token-example/tree/master/compiled, verfügbar am 11.07.18

[81] YouTube: Ethereum Foundation: Babbage: A mechanical smart contract language, (online), 2017. URL: https://www.youtube.com/watch?v=8ejHUqMHTcA, verfügbar am 16.07.18

[82] GitHub: ethereum/wiki: Proof of Stake FAQs, (online), 2018. URL: https://github.com/ethereum/wiki/wiki/Proof-of-Stake-FAQs#what-is-proof-of-stake, verfügbar am 16.07.18

[83] GitHub: ethereum/wiki: Sharding FAQs, (online), 2018. URL: https://github.com/ethereum/wiki/wiki/Sharding-FAQs, verfügbar am 16.07.18

[84] Lange, G: Kryptoszene: Sharding kommt zu Ethereum, (online), 2018. URL: https://kryptoszene.de/sharding-kommt-zu-ethereum/, verfügbar am 16.07.18

[85] Knobloch, M.: Coin Hero: Ethereum: Vitalik Buterin kündigt Entwicklungsfortschritte bei Sharding an, (online), 2018. URL: https://coin-hero.de/ethereum-vitalik-buterin-kuendigt-sharding-an/, verfügbar am 16.07.18

[86] Slock.it: Internes Paper, 2018.

[87] Kryptocoinr: Kryptocoins.net: Ethereum: Entwicklung Anzahl der Ether-Einheiten, (online), 2016. URL: http://kryptocoins.net/2016/02/ethereum-entwicklung-anzahl-der-ether-einheiten/, verfügbar am 17.07.18

[88] GitHub: ewasm/evm2ewasm: Transcompiles EVM code to e-WASM, (online), 2018. URL: https://github.com/ewasm/evm2wasm, verfügbar am 17.07.18

[89] GitHub: pirapira/bamboo: examples, (online), 2017. URL: https://github.com/pirapira/bamboo/blob/master/src/parse/examples/, verfügbar am 18.07.18

Tools

Remix - Solidity IDE:

https://remix.ethereum.org/#optimize=false&version=soljson-v0.4.24+commit.e67f0147.js, verfügbar am 16.08.18

Ganache:

https://github.com/trufflesuite/ganache-cli, verfügbar am 16.08.18

Vyper Online Compiler:

https://vyper.online/, verfügbar am 16.08.18

Anlagen

Kapitel 2 ..... A-I

Kapitel 3 ..... A-V

Anlagen, Kapitel 2

Beispielcontract Solidity:

```
pragma solidity ^0.4.22;  //gibt Compilerversion an

contract Example {

    address public owner;     // statische Variablen
    uint public changeTime;
    uint public fee;

    // Event
    event OwnerChanged(address indexed new, address indexed old);

    // Konstruktor, wird seit Kompilerversion 0.4.22 explizit so
    // bezeichnet. Vorher hätte man Example() geschrieben.
    constructor (uint _fee) public {
        owner = msg.sender;
        fee = _fee;
        changeTime = now;
    }

    // Fallback-Fkt., akzeptiert keine Ether. Kehrt bei Aufruf alles um.
    function() {
        revert();
    }

    // Akzeptiert Ether, von außen aufrufbar.
    function changeOwner(address _new) external payable {
        require (msg.value >= fee);
        address old = owner;
        owner = _new;
        changeTime = now;
        emit OwnerChanged(_new, old);
    }

    // Getter für den Besitzer, wird auch automatisch für public
    // Variablen generiert.
    function getOwner() public view returns (address) {
        return owner;
    }
    //.....
}
```

Beispielcontract Vyper:

```
# Beispiel Vyper

# Event
NewOwner: event({_new: indexed(address), _old: address})

# statische Variablen
owner: public(address)
last_change: public(timestamp)
fee: public(wei_value)

#Konstruktor
@public
def __init__(owner: address, fee: wei_value):
    self.owner = owner
    self.last_change = block.timestamp
    self.fee = fee

@public
@payable   # nimmt Ether an
def changeOwner(_new: address):
    assert msg.value >= fee
    _old: address = self.owner
    self.owner = _new
    self.last_change = block.timestamp
    log.NewOwner(_new, _old)

@public
def getOwner() -> address:
    return self.owner
#.....
```

Beispielcontract LLL:

```
;;;; @title Ein Beispiel der LLL Contract Struktur.
;;;; @author Daniel Ellison <daniel@syrinx.net>
;;;; Quelle: https://gist.github.com/zigguratt/ea41e2af324ac35ce657d435fd538b21

(seq

  ;; Makros für den Contract einbinden.
  (include "macros.lll")

  ;;; ---------------------------------------------------------------------------
  ;;; INIT

  ;; Der "Konstruktor soll Zahlungen ablehnen, da er nicht payable ist,
  ;; also wird abgebrochen,  wenn ETH beim deployment mitgeschickt werden.
  (when (callvalue)
    (panic))

  ;; Ein Konstruktorargument speichern im Storage.
  (codecopy *scratch* (bytecodesize) 32)
  (sstore *parameter* (mload *scratch*))

  ;;; ---------------------------------------------------------------------------
  ;;; CODE

  (returnlll
    (seq

      ;; Ein Makro aus macros.lll zum Holen der gefragten Funktions-ID.
      ;; Sie wird aus den Aufrufdaten geholt und im Memory abgelegt.
      ;; Eine Technik von Ben Edgington.

      get-function-id

      ;; ---------------------------------------------------------------------------
      ;; @notice Gibt den Konstruktorparameter zurück. "function" und
      ;; "*get-constructor-parameter*" sowie "unpayable" sind Makros.
      ;; Signatur von getConstructorParameter() ist *get-constructor-parameter*
      ;; @return Konstruktorargument.

      (function *get-constructor-parameter*
        (seq unpayable

          ;; Konstruktorparameter wird im Memory abgelegt an Adresse des Makros
          ;; *scratch*, hier 0x00. Dazu aus Storage geladen.
          (mstore *scratch* (sload *parameter*))

          ;; Der 'return' Operator nimmt einen Memory Bereich und eine Länge
          ;; und gibt daten dort entsprechend angegebener Länge zurück (Byte).
          (return *scratch* 32)))

      ;; ---------------------------------------------------------------------------
      ;; @notice Fallback: Keine Funktionen passten zur ID in calldata.

  (panic))))
```

Beispielcontract Bamboo:

```
contract BambooExample() {

    case (bool calledDuringDemonstration()) {
        //tue etwas
        return (true) then become BambooExample();
    }

    case (bool isUnderstood()) {
        if (Bedingung)
            return (true) then become BambooKnowledge();
        else if (andere Bedingung)
            return (false) then become BambooNextExample();

        abort;
    }

    default {
        abort;
    }
}

contract BambooKnowledge() {
    case (void toBeCalledAfterLearning()) {
        //tu etwas
    }
    //...
}

contract BambooNextExample() {
    //...
}

```

Anlagen, Kapitel 3

Tokencontract Solidity:

```
pragma solidity ^0.4.22;

import "./ERC20.sol";

/**
 * @title ERC20 Contract mit fester Tokenzahl
 */
contract ERC20Token is ERC20 {

    mapping(address => uint256) balances;
    mapping (address => mapping (address => uint256)) internal allowed;
    uint256 totalToken;
    uint8 decimals; // meist 18
    address public owner;

    event OwnerChanged(address indexed _address);

    modifier onlyOwner {
        require(msg.sender == owner);
        _;
    }

    /**
     * Erlaubt dem Contractbesitzer das Ändern desselbigen.
     * @param _address Neuer Besitzer.
     */
    function changeOwner(address _address) public onlyOwner {
        require(_address != 0x0);
        owner = _address;
        emit OwnerChanged(_address);
    }

    // Konstruktor: bekommt maximale Tokenanzahl übergeben und Dezimalstellen.
    constructor(uint256 _initialTokenAmount, uint8 _decimals) public {
        owner = msg.sender;
        totalToken = _initialTokenAmount * 10 ** uint(_decimals);
        balances[owner] = totalToken;
        decimals = _decimals;
        emit Transfer(address(0), owner, totalToken);
    }

    /**
     * @dev Der gesamte existierende Betrag an Token.
    */
    function totalSupply() public view returns (uint256) {
        return totalToken;
    }
```

```
48
49      /**
50      * @dev Token an eine Adresse senden.
51      * @param _to Die Empfängeradresse.
52      * @param _value Der zu sendende Betrag.
53      */
54      function transfer(address _to, uint256 _value) public returns (bool) {
55          require(_to != address(0));
56          require(_value <= balances[msg.sender]);
57          require(balances[_to] <= balances[_to]+_value);
58          balances[msg.sender] = balances[msg.sender] - _value;
59          balances[_to] = balances[_to] + _value;
60          emit Transfer(msg.sender, _to, _value);
61          return true;
62      }
63
64      /**
65       * @dev Transfer von Token von einer Adresse auf eine andere.
66       * @param _from Die Absenderadresse.
67       * @param _to Die Empfängeradresse.
68       * @param _value Die Menge an Token.
69      */
70      function transferFrom(address _from, address _to, uint256 _value) public returns (bool) {
71          require(_to != address(0));
72          require(_value <= balances[_from]);
73          require(_value <= allowed[_from][msg.sender]);
74          require(balances[_to] <= balances[_to]+_value);
75
76          balances[_from] = balances[_from] - _value;
77          balances[_to] = balances[_to] + _value;
78          allowed[_from][msg.sender] = allowed[_from][msg.sender] - _value;
79          emit Transfer(_from, _to, _value);
80          return true;
81      }
82
83      /**
84       * @dev Erlaube der angegebenen Adresse, Token auszugeben, die msg.sender gehören.
85       * @param _spender Die Adresse, die Guthaben ausgeben soll.
86       * @param _value Die auszugebende Menge an Token.
87      */
88      function approve(address _spender, uint256 _value) public returns (bool) {
89          allowed[msg.sender][_spender] = _value;
90          emit Approval(msg.sender, _spender, _value);
91          return true;
92      }
93
94      /**
95       * @dev Funktion zur Prüfung des Guthabens, welches eine Adresse einer anderen freigegeben hat.
96       * @param _owner Der Besitzer der Token.
97       * @param _spender Die Adresse, die Token ausgeben will.
98       * @return Ein uint256, gibt noch für den spender verfügbare Menge Token an.
99      */
100     function allowance(address _owner, address _spender) public view returns (uint256) {
101         return allowed[_owner][_spender];
102     }
103
104     /**
105      * @dev Liefert das Guthaben der angegebenen Adresse.
106      * @param _owner Die abzufragende Adresse.
107      * @return Ein uint256, beschreibt Guthaben der Adresse.
108     */
109     function balanceOf(address _owner) public view returns (uint256) {
110         return balances[_owner];
111     }
112
113     // Fallback
114     function() public payable {
115         revert();
116     }
117
118 }
```

Tokencontract Vyper:

```
# Vyper Port of MyToken

# Events
Transfer: event({_from: indexed(address), _to: indexed(address), _value: uint256})
Approval: event({_owner: indexed(address), _spender: indexed(address), _value: uint256})
NewOwner: event({_newOwner: indexed(address)})

# Variablen
total: public(uint256)
decimals: public(int128)
balances: int128[address]
allowed: int128[address][address]
owner: public(address)

#Konstruktor
@public
def __init__(_decimals: uint256, _initialSupply: uint256):
    self.owner = msg.sender
    self.decimals = _decimals
    self.total =_initialSupply * convert(10, 'uint256') ** _decimals
    self.balances[msg.sender] = convert(self.total, 'int128')


@public
def changeOwner(_new: address):
    assert msg.sender == self.owner
    self.owner = _new
    log.NewOwner(_new)

# Fragt Guthaben eines Account ab.
@public
@constant
def balanceOf(_owner: address) -> uint256:

    return convert(self.balances[_owner], 'uint256')

# Gibt Gesamtanzahl an Token zurück.
@public
@constant
def totalSupply() -> uint256:

    return self.total


```

```
# Sendet `_amount` Token an `_to` vom eigenen Account.
@public
def transfer(_to: address, _amount: int128(uint256)) -> bool:

    if self.balances[msg.sender] >= _amount and \
       self.balances[_to] + _amount >= self.balances[_to]:

        self.balances[msg.sender] -= _amount
        self.balances[_to] += _amount
        log.Transfer(msg.sender, _to, convert(_amount, 'uint256'))

        return True
    else:
        return False


# Transferiert Token von einem angegebenen Account zu einem anderen.
@public
def transferFrom(_from: address, _to: address, _value: int128(uint256)) -> bool:

    if _value <= self.allowed[_from][msg.sender] and \
       (_value <= self.balances[_from] and self.balances[_to] + _value >= self.balances[_to]):

        self.balances[_from] -= _value
        self.allowed[_from][msg.sender] -= _value
        self.balances[_to] += _value
        log.Transfer(_from, _to, convert(_value, 'uint256'))

        return True
    else:
        return False


# Erlaubt _spender vom eigenen Account Token auszugeben, bis _amount erreicht ist.
# Bei erneutem Aufruf wird der aktuelle Wert überschrieben.
@public
def approve(_spender: address, _amount: int128(uint256)) -> bool:

    self.allowed[msg.sender][_spender] = _amount
    log.Approval(msg.sender, _spender, convert(_amount, 'uint256'))

    return True


 # Liefert erlaubten Betrag, der im Namen eines anderen ausgegeben werden kann.
 @public
 @constant
 def allowance(_owner: address, _spender: address) -> uint256:

     return convert(self.allowed[_owner][_spender], 'uint256')
```

Tokencontract LLL:

```
;;; ------------------------------------------------------------------------
;;; Implementation eines ERC20 Tokencontract in LLL
;;;
;;; Ben Edgington - ben@benjaminion.xyz

(seq

  ;; ------------------------------------------------------------------------
  ;; KONSTANTEN

  ;; Token Parameter.
  ;; 0x40: Text eines String hier gespeichert bei Rückgabe an Aufrufer
  (def 'token-name-string   (lit 0x40 "LLL Coin - love to code in LLL."))
  (def 'token-symbol-string (lit 0x40 "LLL"))
  (def 'token-decimals 0)
  (def 'token-supply 100) ; 100 Token gesamt

  ;; Booleans
  (def 'false 0)
  (def 'true  1)

  ;; Memory layout.
  (def 'mem-ret    0x00) ; Fest wegen compiler Makro für return.
  (def 'mem-func   0x00) ; Kein Konflikt mit mem-ret, also wiederverwenden.
  (def 'mem-keccak 0x00) ; Wie oben.
  (def 'scratch0   0x20)
  (def 'scratch1   0x40)

  ;; Vorberechnete Funktions - IDs.
  (def 'get-name         0x06fdde03) ; name()
  (def 'get-symbol       0x95d89b41) ; symbol()
  (def 'get-decimals     0x313ce567) ; decimals()
  (def 'get-total-supply 0x18160ddd) ; totalSupply()
  (def 'get-balance-of   0x70a08231) ; balanceOf(address)
  (def 'transfer         0xa9059cbb) ; transfer(address,uint256)
  (def 'transfer-from    0x23b872dd) ; transferFrom(address,address,uint256)
  (def 'approve          0x095ea7b3) ; approve(address,uint256)
  (def 'get-allowance    0xdd62ed3e) ; allowance(address,address)

  ;; Event IDs
  (def 'transfer-event-id ; Transfer(address,address,uint256)
    0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef)

  (def 'approval-event-id ; Approval(address,address,uint256)
    0x8c5be1e5ebec7d5bd14f71427d1e84f3dd0314c0f7b2291e5b200ac8c7c3b925)

  ;; ------------------------------------------------------------------------
```

```
;; HILFSMITTEL

;; ------------------------------------------------------------------------
;; Folgendes definiert die Key-data-Strukturen:
;;     - balance(addr) => value
;;     - allowance(addr,addr) => value

;; Guthaben sind gespeichert bei s[owner_addr].
(def 'balance (address) address)

;; Allowances sind gespeichert bei s[owner_addr + keccak256(spender_addr)]
(def 'allowance (owner spender)
  (seq
    (mstore mem-keccak spender)
    (add owner (keccak256 mem-keccak 0x20))))

;; ------------------------------------------------------------------------
;; Makros, die Funktionsargumente aus calldata laden

(def 'arg1 (calldataload 0x04))
(def 'arg2 (calldataload 0x24))
(def 'arg3 (calldataload 0x44))

;; ------------------------------------------------------------------------
;; Revert ist ein return, dass kein restliches Gas verbraucht.
;; Bei invalidem Input eingesetzt, neu für 'panic'.
;;
;; Note: REVERT opcode wird implementiert in Metropolis (EIP 140).
;; Bis dahin verursacht es eine invalid instruction exception (vgl.
;; "throw" in Solidity).

(def 'revert () (revert 0 0))

;; ------------------------------------------------------------------------
;; Makro zur Rückgabe von strings.

(def 'return-string (string-literal)
  (seq
    (mstore 0x00 0x20)           ; Zeigt auf string: memory Speicheradresse
    (mstore 0x20 string-literal) ; Länge. String selbst wird zu 0x40 kopiert.
    (return 0x00 (& (+ (mload 0x20) 0x5f) (~ 0x1f)))))
                                 ; Runde return auf 32 byte Stück auf.

;; ------------------------------------------------------------------------
;; Makro zum Auslösen von Events

```

```
(def 'event3 (id addr1 addr2 value)
  (seq
    (mstore scratch0 value)
    (log3 scratch0 0x20 id addr1 addr2)))

;; -------------------------------------------------------------------------
;; Prüft, ob gespeicherte Funktions - ID einem bekannten Hash entspricht und
;; führt dann <code-body> aus.
;; @param function-hash Der 4-byte Hash einer bekannten Fkt.signatur.
;; @param code-body Der auszuführende Code im Falle eines Treffers.

(def 'function (function-hash code-body)
  (when (= (mload mem-func) function-hash)
    code-body))

;; -------------------------------------------------------------------------
;; Holt die Funktions - ID und speichert sie in memory.

(def 'uses-functions
  (seq
    (mstore mem-func 0)
    (calldatacopy (+ mem-func 28) 0x00 4)))

;; -------------------------------------------------------------------------
;; GUARDS

;; -------------------------------------------------------------------------
;; Stellt richtige Argumentzahl bei Aufrufen sicher. Schutz vor "ERC20
;; short address attack". Für nicht-constant Funktionen.

(def 'has-one-arg    (unless (= 0x24 (calldatasize)) (revert)))
(def 'has-two-args   (unless (= 0x44 (calldatasize)) (revert)))
(def 'has-three-args (unless (= 0x64 (calldatasize)) (revert)))

;; -------------------------------------------------------------------------
;; Prüfen, dass Adressen nur 160 bit haben & revert wenn nicht.
;; Wird bei nicht-constant Funktionen eingesetzt.

(def 'is-address (addr)
  (when
    (shr addr 160)
    (revert)))

;; -------------------------------------------------------------------------
;; Prüfen, dass Transferwerte kleiner als gesamter Vorrat sind &
;; revert wenn nicht. So keine negativen Werte.

```

```
(def 'is-value (value)
  (when (> value token-supply) (revert)))

;; ------------------------------------------------------------------------
;; Revert beim Empfang von Ether. Sofortige Nutzung, damit schon beim Deployment
;; wirksam.

(def 'not-payable
  (when (callvalue) (revert)))

not-payable

;; ------------------------------------------------------------------------
;; INITIALISIERUNG
;;
;; Alle Token werden anfangs dem Contractbesitzer zugeschrieben.

(sstore (balance (caller)) token-supply)

;; ------------------------------------------------------------------------
;; CONTRACT CODE

(returnlll
  (seq not-payable uses-functions

    ;; ------------------------------------------------------------------------
    ;; Getter für Name des Token.
    ;; @abi name() constant returns (string)
    ;; @return Tokenname als string.

    (function get-name
      (return-string token-name-string))

    ;; ------------------------------------------------------------------------
    ;; Getter für das Tokensymbol.
    ;; @abi symbol() constant returns (string)
    ;; @return Das Tokensymbol.

    (function get-symbol
      (return-string token-symbol-string))

    ;; ------------------------------------------------------------------------
    ;; Getter für Dezimalstellen des Token.
    ;; @abi decimals() constant returns (uint256)
    ;; @return Die Dezimalstellen.
```

```
187     (function get-decimals
188       (return token-decimals))
189
190     ;; -------------------------------------------------------------------------
191     ;; Getter für den Gesamtvorrat an Token.
192     ;; @abi totalSupply() constant returns (uint256)
193     ;; @return Gesamte Tokenzahl.
194
195     (function get-total-supply
196       (return token-supply))
197
198     ;; -------------------------------------------------------------------------
199     ;; Gibt Guthaben eines Account zurück.
200     ;; @abi balanceOf(address) constant returns (uint256)
201     ;; @param owner Die Adresse des Accountbesitzers.
202     ;; @return Guthaben.
203
204     (function get-balance-of
205       (seq
206
207         (def 'owner arg1)
208
209         (return (sload (balance owner)))))
210
211     ;; -------------------------------------------------------------------------
212     ;; Transferiert _value an Token zu _to.
213     ;; @abi transfer(address, uint256) returns (bool)
214     ;; @param to Empfängeraccount.
215     ;; @param value Tokenanzahl.
216     ;; @return Erfolg (true). Andere Ergebnisse führen zu Revert.
217
218     (function transfer
219       (seq has-two-args (is-address arg1) (is-value arg2)
220
221         (def 'to    arg1)
222         (def 'value arg2)
223
224         (when value ;; value == 0 ist  no-op
225           (seq
226
227             ;; Das Aufruferguthaben. In Memory speichern.
228             (mstore scratch0 (sload (balance (caller))))
229
230             ;; Revert wenn Aufruferguthaben zu wenig.
231             (when (> value (mload scratch0))
232               (revert))
```

```
233
234             ;; Transfer
235             (sstore (balance (caller)) (- (mload scratch0) value))
236             (sstore (balance to) (+ (sload (balance to)) value))
237
238             ;; Event - Transfer(address,address,uint256)
239             (event3 transfer-event-id (caller) to value)))
240
241         (return true)))
242
243     ;; ------------------------------------------------------------------------
244     ;; Sende _value an Token von Adresse _from an Adresse _to
245     ;; @abi transferFrom(address,address,uint256) returns (bool)
246     ;; @param from Absenderaccount.
247     ;; @param to Empfängeraccount.
248     ;; @param value Tokenanzahl.
249     ;; @return Erfolg (true). Alles andere führt zu Revert.
250
251     (function transfer-from
252       (seq has-three-args (is-address arg1) (is-address arg2) (is-value arg3)
253
254         (def 'from  arg1)
255         (def 'to    arg2)
256         (def 'value arg3)
257
258         (when value ; value == 0 is a no-op
259
260           (seq
261
262             ;; Daten in memory für Effizienz.
263             (mstore scratch0 (sload (balance from)))
264             (mstore scratch1 (sload (allowance from (caller))))
265
266             ;; Revert wenn zu wenig Guthaben oder nicht genug approved.
267             (when
268               (||
269                 (> value (mload scratch0))
270                 (> value (mload scratch1)))
271               (revert))
272
273             ;; Transfer durchführen & Update allowance.
274             (sstore (balance from) (- (mload scratch0) value))
275             (sstore (balance to) (+ (sload (balance to)) value))
276             (sstore (allowance from (caller)) (- (mload scratch1) value))
277
278             ;; Event - Transfer(address,address,uint256)
279             (event3 transfer-event-id from to value)))
280
```

```
      (return true)))

    ;; -------------------------------------------------------------------------
    ;; Erlaubt _spender vom Account des Aufrufers Guthaben auszugeben,
    ;; bis zu _value. Ein neuer Aufruf überschreibt die vorherige
    ;; allowance mit _value.
    ;; @abi approve(address,uint256) returns (bool)
    ;; @param spender Der Account, der value ausgeben darf.
    ;; @param value Maximaler erlaubter Wert.
    ;; @return Erfolg (true). Andere Ergebnisse führen zu Revert.

    (function approve
      (seq has-two-args (is-address arg1) (is-value arg2)

        (def 'spender arg1)
        (def 'value   arg2)

        ;; Zwinge Nutzer erst die allowance auf 0 zu setzen, bevor
        ;; auf einen anderen Wert.
        (when
          (&& value (sload (allowance (caller) spender)))
          (revert))

        (sstore (allowance (caller) spender) value)

        ;; Event - Approval(address,address,uint256)
        (event3 approval-event-id (caller) spender value)

        (return true)))

    ;; -------------------------------------------------------------------------
    ;; Gibt die Menge zurück, die spender noch ausgeben darf von owner
    ;; @abi allowance(address,address) constant returns (uint256)
    ;; @param owner Der Besitzeraccount.
    ;; @param spender Der ausgebende Account.
    ;; @return Das verbleibende auszugebende Guthaben.

    (function get-allowance
      (seq

        (def 'owner   arg1)
        (def 'spender arg2)

        (return (sload (allowance owner spender)))))

    ;; -------------------------------------------------------------------------
    ;; Fallback: Keine Funktionen passten zur gegebenen ID.

    (revert)))
)
```

Tokencontract Bamboo:

```
contract PreToken
(uint256 totalSupply
,address => uint256 balances
,address => address => uint256 allowances
)
{
    default
    {
        balances[sender(msg)] = totalSupply;
        return then become Token(totalSupply, balances, allowances, sender(msg));
    }
}

event Transfer(address indexed _from, address indexed _to, uint256 _amount);
event Approval(address indexed _owner, address indexed _spender, uint256 _value);
event NewOwner(address indexed _new);

contract Token
(uint256 totalSupply
,address => uint256 balances
,address => address => uint256 allowances
,address owner
)
{
    case(uint256 totalSupply())
    {
        return totalSupply then become Token(totalSupply, balances, allowances, owner);
    }

    case(uint256 balanceOf(address a))
    {
        return balances[a] then become Token(totalSupply, balances, allowances, owner);
    }

    case(bool transfer(address _to, uint256 _amount))
    {
        if (balances[sender(msg)] < _amount) abort;
        if (sender(msg) == _to)
        {
            log Transfer(sender(msg), sender(msg), _amount);
            return true then become Token(totalSupply, balances, allowances, owner);
        }
        balances[sender(msg)] = balances[sender(msg)] - _amount;
        if (balances[_to] + _amount < balances[_to]) abort;
        balances[_to] = balances[_to] + _amount;
        log Transfer(sender(msg), _to, _amount);
        return true then become Token(totalSupply, balances, allowances, owner);
```

```
48      }
49
50      case(bool approve(address _spender, uint256 _amount))
51      {
52          if (balances[sender(msg)] < _amount) abort;
53          if (sender(msg) == _spender) abort;
54          allowances[sender(msg)][_spender] = _amount;
55          log Approval(sender(msg), _spender, _amount);
56          return true then become Token(totalSupply, balances, allowances, owner);
57      }
58
59      case(uint256 allowance(address _owner, address _spender))
60      {
61          return allowances[_owner][_spender] then become Token(totalSupply, balances, allowances, owner);
62      }
63
64      case(bool transferFrom(address _from, address _to, uint256 _amount))
65      {
66          if (balances[_from] < _amount) abort;
67          if (allowances[_from][sender(msg)] < _amount) abort;
68          if (_from == _to)
69          {
70              log Transfer(_from, _to, _amount);
71              return true then become Token(totalSupply, balances, allowances, owner);
72          }
73
74          balances[_from] = balances[_from] - _amount;
75          allowances[_from][sender(msg)] = allowances[_from][sender(msg)] - _amount;
76          balances[_to] = balances[_to] + _amount;
77          log Transfer(_from, _to, _amount);
78          return true then become Token(totalSupply, balances, allowances, owner);
79      }
80
81      case(bool changeOwner(address _new))
82      {
83          if (sender(msg) != owner) abort;
84          if (iszero(_new)) abort;
85          log NewOwner(_new);
86          return true then become Token(totalSupply, balances, allowances, _new);
87      }
88
89      default
90      {
91          abort;
92      }
93
94  }
```

Zählskript Python:

```
1   txdata="bytecode"
2
3   def count_zero_bytes(data):
4     count = 0
5     for i in xrange(0, len(data), 2):
6       byte = data[i:i+2]
7       if byte == "00":
8         count += 1
9     return count
10
11  def count_non_zero_bytes(data):
12    return (len(data) / 2) - count_zero_bytes(data)
13
14  print("zero-bytes: {0}".format(count_zero_bytes(txdata)))
15  print("non-zero-bytes: {0}".format(count_non_zero_bytes(txdata)))
```

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht.

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

Mittweida, den 04.10.2018

Laura Moritz