FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

Lehrstuhl für Informatik 7

Rechnernetze und Kommunikationssysteme

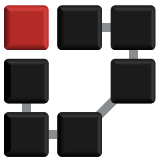
Daniel Hohner

Analyse der Möglichkeiten und Grenzen von Smart Contracts mit Hilfe einer Implementierung in Solidity

Bachelorarbeit im Fach Informatik

17. März 2019

Please cite as:
Daniel Hohner, “Analyse der Möglichkeiten und Grenzen von Smart Contracts mit Hilfe einer Implementierung in Solidity,” Bachelor Thesis (Bachelorarbeit), University of Erlangen, Dept. of Computer Science, March 2019.



Friedrich-Alexander-Universität Erlangen-Nürnberg
Department Informatik
Rechnernetze und Kommunikationssysteme

Martensstr. 3 · 91058 Erlangen · Germany

http://www7.cs.fau.de/

Analyse der Möglichkeiten und Grenzen von Smart Contracts mit Hilfe einer Implementierung in Solidity

Bachelorarbeit im Fach Informatik

vorgelegt von

Daniel Hohner

geb. am 27. Juli 1991
in Nürnberg

angefertigt am

Lehrstuhl für Informatik 7
Rechnernetze und Kommunikationssysteme

Department Informatik
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer: **Prof. Dr-Ing. Reinhard German**
M.Sc. Jonas Schlund

Abgabe der Arbeit: **17. März 2019**

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde.
Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Declaration

I declare that the work is entirely my own and was produced with no assistance from third parties.
I certify that the work has not been submitted in the same or any similar form for assessment to any other examining body and all references, direct and indirect, are indicated as such and have been cited accordingly.

(Daniel Hohner)
Erlangen, 17. März 2019

Abstract

Blockchain technologies posess a vast potential to automate, decentralize and secure other fields of activity, while also making the specific line of work more transparent. Unfortunately there have already been many negative examples in the field of blockchain, that demonstrate, how an implementation of an idea has failed. In order to prevent such mistakes, an understanding for the basics of blockchain and smart contracts shall be presented. Furthermore general patterns shall be developed, which can be applied to prevent mistakes.
The development of general patterns describes a process, which is useful to prevent known and even some unknown errors. However the application of some patterns has unwanted side effects, which results in a negative user experience, as more tasks have to be performed by the user while using the product.
Furthermore new developments, that could improve the performance of a blockchain, shall be analyzed. These developments could lead to a significant increase in performance, if first test results can be applied to real world blockchains.

Kurzfassung

Technologien, die auf Blockchain aufbauen, besitzen ein großes Potential andere Arbeitsbereiche zu automatisieren, zu dezentralisieren und sicherer und transparenter zu machen. Es gibt jedoch schon einige Negativbeispiele, die aufzeigen, wie eine Umsetzung eines Konzepts im Bereich Blockchain gescheitert ist. Zur Vermeidung von grundsätzlichen Fehlern soll ein Verständnis für die Grundlagen von Blockchain und Smart Contracts geschaffen werden, um im Anschluss allgemeine Muster zu erarbeiten, die zur Fehlervermeidung eingesetzt werden können.
Die Erarbeitung von allgemeinen Mustern stellt ein Vorgehen dar, welches dafür geeignet ist, bekannte Fehler zu vermeiden, jedoch kann es bei manchen Mustern dazu kommen, dass sich die Anwendung für den Benutzer erschwert.
Weiterhin soll ein Ausblick auf die Weiterentwicklungen gegeben werden, welche die Leistung der Blockchain steigern könnten. Diese Weiterentwicklungen sind ein positiver Fortschritt, welche einen enormen Leistungszuwachs mit sich bringen, falls sich die Kennwerte der ersten Analysen auch im Produktionsbetrieb widerspiegeln.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Durch die Entwicklung des World Wide Web im Jahr 1989 und die spätere kostenlose Bereitstellung dieses, startete ein unerwarteter Siegeszug dieser Technologie. In der heutigen Zeit ist das World Wide Web ein nicht mehr wegzudenkender Bestandteil des täglichen Lebens und viele Tätigkeiten, wie das Einkaufen, können nun auch Online durchgeführt werden. Hierbei hat man allerdings immer noch eine Verknüpfung der Onlinetätigkeit mit der physikalischen Welt. Überlegungen, wie man die Bindung der virtuellen Welt zu der physikalischen Welt aufheben kann, führten schließlich zu der Entwicklung von Cryptocurrencies, wie Bitcoin im Jahr 2009. [1]
Bitcoin zählt zu den bekanntesten virtuellen Währungen, die auf der Grundlage eines Peer-To-Peer-Netzwerkes operieren und bietet eine Möglichkeit Geldgeschäfte dezentralisiert durchzuführen. Diese Cryptocurrency basiert auf der Blockchain-Technologie und ist die erste und bekannteste Umsetzung dieser. Seit der Einführung und aufgrund des hohen Erfolges dieser virtuellen Währung hat sich in den letzten Jahren die Anzahl von digitalen Währungen (auch Coins genannt) stark vervielfacht. So gibt es heute schon 2067 verschiedene Coins, die gehandelt werden können und deren Wert sich auf über 134 Milliarden Dollar beläuft. [2]

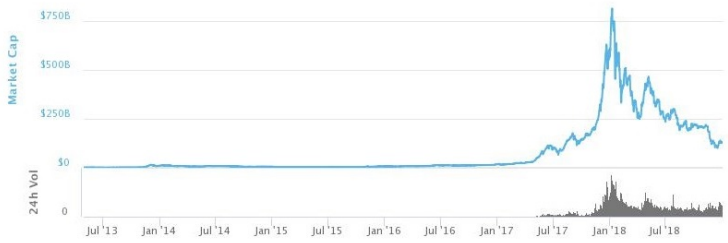


Abbildung 1.1 – Marktkapitalisierung aller Cryptowährungen[1].

[1] Global Charts | CoinMarketCap

Für diesen Anstieg kann man mehrere Gründe aufführen. Zum Einen gibt es Menschen in politisch instabilen Regionen, die ihr Eigentum in krisensichere Währungen anlegen wollen. Hierfür wurde bei den Cryptocurrencies die Menge der Coins - analog zu Gold und Silber - begrenzt, welche pro Tag „geschürft“ werden können. [3, S.1] Zum Anderen gibt es die Investoren, die aufgrund der vorherigen volatilen Kursentwicklung in die Cryptocurrency investieren und sich hohe Renditen versprechen.
Die Blockchain-Technologie kann aber auch für andere Zwecke, als als Währung, eingesetzt werden, was mit der Einführung von Ethereum im Jahr 2015 gezeigt wurde. Hierbei gibt es neben dem klassischen Anwendungsfall als Cryptowährung, welcher von Bitcoin bekannt ist, auch die sogenannten Smart Contracts. Mit der Hilfe von Smart Contracts ergibt sich eine Möglichkeit dezentrale Programme, zum Beispiel im Ethereum Netzwerk zu erstellen und zu verwenden. Diese Technologie kann unter anderem bei einem digitalen Wahlvorgang zum Einsatz kommen kann. Hierbei kann man von den Sicherheitsmechanismen der Blockchain profitieren, um im vorigen Beispiel eine Manipulation der Ergebnisse zu erschweren, wenn nicht sogar komplett zu verhindern.

1.1 Motivation

1.1.1 Unternehmerische Sicht

Die adorsys GmbH & Co. KG ist ein Softwareentwicklungs- und Consulting-Unternehmen, dessen Ziel es ist ihren Kunden im Finanz- und Versicherungssektor neue und zukunftsweisende Ideen und Strategien anzubieten. Wenn man das Kundenfeld betrachtet so kann man feststellen, dass vor allem im Finanzsektor ein Interesse an Cryptowährungen und der dahinter stehenden Blockchain-Technologie besteht. [4] Da die traditionellen Leistungen, welche die Banken anbieten, durch die Cryptowährungen in ihrem Kontext obsolet gemacht wurden, müssen die Institutionen in diesem Sektor neue Leistungen erschließen, um diejenigen Kunden, die ihre Vermögenswerte in Cryptocurrencies angelegt haben, weiterhin an das Institut binden zu können.

1.1.2 Gesellschaftliche Sicht

Das Anwendungsgebiet von Blockchainanwendungen ist weit gefächert, daher kann man davon ausgehen, dass die Benutzer dieser Anwendungen unterschiedliche technische Wissenshintergründe mit sich bringen. Die Anbieter von Anwendungen, die auf der Technologie der Blockchain beruhen, sollen also die verschiedenen Anwendungsgebiete so benutzerfreundlich und die technischen Grundlagen so transparent wie möglich gestalten. Mögliche Anwendungsgebiete sind:

- Dezentrale DNS-Server

- Firmenunabhängige Lizenzserver
- Rechteverwaltung
- Dezentrale Firmen

In diesen Anwendungsgebieten kann die Automatisierung der Prozesse und die Sicherheit, welche durch die Blockchain-Technologie gewährleistet wird, einen Nutzen für die Gesellschaft darstellen. So kann bei einem Bankrott einer Softwarefirma die Software durch die dezentrale Speicherung der Lizenzinformationen auf der Blockchain immernoch aktiviert werden. Bei einer dezentralen Lösung für die DNS-Server kann der Internetverkehr sicherer gemacht werden, da die Informationen nicht mehr zentral auf einzelnen Servern gespeichert werden. Bei dieser Lösung kann von jedem Netzwerkteilnehmer zu jeder Zeit die Korrektheit der DNS-Informationen überprüft werden.

1.2 Zielsetzung

Ziel dieser Arbeit ist es den Aufbau und die Funktionsweise der Blockchain-Technologie und Smart Contracts zu besprechen, um dann im Weiteren auf die Möglichkeiten und Grenzen von Smart Contracts im Kontext von dezentralisierten Anwendungen (DApps) und Organisationen (DAOs) einzugehen. Hierbei soll auch ein besonderer Augenmerk auf den Weiterentwicklungen der Plattform Ethereum liegen, um im Anschluss zu untersuchen, wie sich angekündigte Neuerungen auf die vorher herausgearbeiteten Chancen und Grenzen auswirken. Zur Veranschaulichung der Automatisierungsvorteile von Smart Contracts und der Sicherheitsmerkmale der Blockchain, die den Smart Contracts zugrunde liegen, wird am Ende der Arbeit eine dezentrale Firma in Solidity, der Programmiersprache von Smart Contracts, implementiert. Diese Firma soll die Prämien, von ausgeschriebenen Projekten, automatisch nach einem Review der eingereichten Lösung auszahlen. Das Review soll durch eine Abstimmung der Firmenmitglieder durchgeführt werden.

1.3 Umfeld

Die Arbeit wird von Matthias Freßdorf vom IT-Consulting-Unternehmen adorsys GmbH & Co. KG betreut. Zu den Kunden von adorsys zählen namhafte Unternehmen, wie die Teambank und die Bausparkasse Schwäbisch Hall. Für die Kunden entwickelt adorsys seit mehr als 10 Jahren individuelle Software und ist somit ein guter Partner mit Expertise bei der Entwicklung und Analyse von komplexen Systemen.

1.4 Aufbau der Arbeit

Nach der Hinführung zum Thema, soll im nächsten Kapitel die genaue Vorgehensweise zur Bestimmung der Chancen und Grenzen von Smart Contracts erarbeitet werden.

Anschließend werden die Grundlagen, wie die benötigten Fachbegriffe, für das Verständnis der Blockchain Technologie geschaffen. Hierbei wird besonders auf die grundlegenden Aspekte dieser Technologie eingegangen.

Im Anschluss werden die grundlegenden Eigenschaften und Anwendungsgebiete von Smart Contracts erklärt, um dann im Folgenden die Vorgehensweisen aufzuzeigen, welche zur Vermeidung von bekannten Fehlern eingesetzt werden können.

Nachdem näher auf Möglichkeiten zur Sicherung des eigenen Produktes zum aktuellen Stand der Technik eingegangen wurde, werden anschließend geplante Neuerungen im Bereich Blockchain, speziell bei Ethereum, vorgestellt.

Zur Veranschaulichung der vorgestellten Konzepte, wie die bekannten Fehler bei der Entwicklung im Bereich Blockchain zu vermeiden sind, soll am Ende die Anwendung der Konzepte durch die Implementierung einer dezentralen Firma aufgezeigt werden.

Kapitel 2

Vorgehensweise

Die Tätigkeiten bei der Durchführung der Bachelorarbeit lassen sich wie folgt aufteilen:

2.1 Analyse der aktuellen Situation

Zuerst wird Untersucht, wie die Rahmenbedingungen bei der Entwicklungen von Smart Contracts bestellt sind. Hierbei soll vor allem ein Verständnis für die grundlegenden Begriffe, Mechanismen und der Anwendungsgebiete der Blockchain-Technologie geschaffen werden.

2.2 Analyse der angekündigten Neuerungen

In diesem Kapitel werden die angekündigten Neuerungen von Ethereum 2.0, wie Sharding und Skalierbarkeit, auf ihren Nutzen analysiert und ihre Auswirkungen auf zukünftige Projekte beleuchtet.

2.3 Implementierung

Zur Verdeutlichung der Funktionsweise von Smart Contracts wird, unter Berücksichtigung von den vorher festgelegten Rahmenbedingungen eine dezentralisierte Firma in Solidity implementiert.

Kapitel 3

Grundlagen

3.1 Entstehung

Im Jahr 2008 wurde ein Whitepaper mit dem Titel „Bitcoin: A Peer-to-Peer Electronic Cash System“ unter dem Alias Satoshi Nakamoto veröffentlicht. Satoshi Nakamoto verknüpfte vorherige Erfindungen, wie B-money und HashCash, um ein dezentrales elektronisches Geldsystem zu schaffen, welches, anders als Fiat-Währungen, unabhängig von einer zentralen Instanz ist, welche die Wertigkeit des Geldes garantiert. [3, S.4] Das Konzept von B-money wurde im November 1998 von Wei Dai veröffentlicht. Hierbei handelt es sich um zwei Vorschläge, wie man Gemeinschaften anonym modellieren kann. Im ersten Vorschlag verwaltet jeder Teilnehmer eine separate Datenbank, in der gespeichert wird, wieviel Geld jedem Account gehört. Das Handelsmedium in der Gemeinschaft wird durch einen Proof-Of-Work-Algorithmus erstellt und Nutzer tauschen signierte Nachrichten zur Aktualisierung der Datenbankzustände aus. Im zweiten Vorschlag wird der Zustand des Netzwerkes von einem Subset der Teilnehmer - Server - verwaltet und nicht mehr von jedem Teilnehmer im Netzwerk. Hierbei soll von den betroffenen Parteien einer Transaktion die Korrektheit des Netzwerkes geprüft werden, indem eine zufällige Anzahl der Server im Anschluss an die Transaktion von ihnen auf den erwarteten neuen Zustand befragt werden. [5]
In seinem Whitepaper zu Bitcoin schlägt Satoshi vor den Proof-Of-Work-Algorithmus, wie er bei B-money und HashCash verwendet wird, auch zum Einsatz zu bringen, jedoch um alle Transaktionen im Netzwerk zu bestätigen. Hierdurch wurde das sogenannte „double spend“ Problem gelöst. Es ist also nicht mehr möglich einen Geldbetrag doppelt auszugeben, bevor das Netzwerk es bemerkt. Nach der Publikation des Whitepapers 2008 startete das Bitcoin-Netwerk 2009 und wird seit 2011 nicht mehr von Nakamoto, sondern ausschließlich von Freiwilligen, gepflegt. [3, S.4]

3.2 Grundlegende Begriffe

In diesem Kapitel werden die Grundlagen für ein Verständnis der Blockchain Technologie geschaffen und zudem die benötigten Begriffe und die wesentlichen Bestandteile, auf denen Blockchains, wie Ethereum aufbauen, erklärt.

3.2.1 Fiat-Währungen

Traditionelle Währungen, wie der Euro oder der Dollar, bezeichnet man als sogenanntes Fiatgeld. Hierbei handelt es sich um ein Tauschmittel, welches durch eine zentrale Instanz ausgegeben wird und offiziell gehandelt wird. [6] Dieses Herausgeben geschieht aus dem „Nichts“, daher stammt auch der Name Fiat (lat. fiat, es werde) Geld. Eine Fiat-Währung erlangt erst seinen Wert, indem es eine Wertzuweisung durch eine staatliche Macht (meist die Regierung) erfährt. Hierdurch wird die Akzeptanz des Geldes im jeweiligen Land gesichert und kann seiner Funktion als Transaktionsmittel gerecht werden.
Im Gegensatz zu den Fiat-Währungen steht das sogenannte Warengeld, dessen Wert an einen Rohstoff - zum Beispiel ein Edelmetall - gekoppelt ist. [7] Hierbei ist die Menge des Geldes, welches man theoretisch ausgeben kann durch die Menge des Rohstoffes, die verfügbar ist, begrenzt. Dies ist bei Fiat-Währungen nicht der Fall und die Menge an Geld muss durch die zentrale Instanz kontrolliert werden, um hohe Inflationsraten zu verhindern. [8]

3.2.2 Cryptowährungen

Cryptowährungen, wie Bitcoin und Ether, können sämtliche Funktionen, wie Bezahlung, Geldanlage und Wertanzeige, der Fiat-Währungen erfüllen, liegen aber - bis auf einzelne Ausnahmen - nur im virtuellen Format vor. [3, S.1] Im Gegensatz zu den traditionellen Währungen sind Cryptowährungen, welche heutzutage meist auf der Basis der Blockchain-Technologie beruhen und dementsprechend von den Vorteilen dieser - wie Sicherheit und Schnelligkeit der Transaktionen - profitieren, nicht an ein spezifisches Land gebunden. [3, S.1 f.] Als Folge der Aufhebung der Länderbindung kann man schnell und länderübergreifend mit der gleichen Währung seine Geschäfte tätigen und man muss keine Gebühren für den Wechsel des Geldes in die benötigte Währung zahlen. Das virtuelle Format der Cryptowährungen und die ihnen inhärenten Eigenschaften - Schnelligkeit, Sicherheit und Transaktionen über Ländergrenzen hinweg - prädestinieren diese zu dem idealen Bezahlungsmittel im Internet. [3, S.3]

3.2.3 Kryptographie

Kryptographie ist eine der Basistechnologien, auf denen Blockchain aufbaut und beschäftigt sich neben der tatsächlichen Verschlüsselung auch mit Verfahren, um etwas digital zu signieren oder, um die Authentizität von einem Datum zu gewährleisten. [9, S.59] Die digitale Signatur und der digitale Fingerabdruck sind wesentliche Bestandteile, welche die Blockchain überhaupt erst agieren lassen und werden auch in viele Anwendungen, die auf einer Blockchain wie Ethereum aufbauen, eingesetzt. [9, S.59]
Obwohl die Verschlüsselung von Daten auch ein Bestandteil der Kryptographie ist, steht dies mit einer der Grundideen von Blockchain - open Ledger - in Konflikt und wird zumindest bei Bitcoin und Ethereum nicht eingesetzt. Mit Zcash gibt es jedoch einen Ansatz, bei dem Verschlüsselung in Verbindung mit Blockchain eingesetzt wird, um dem Anwender zusätzlich noch die Rechte zu geben, wer die eigenen Transaktionen mitlesen darf, und wer nicht. [10]
Im Weiteren werden die Aspekte der Kryptographie am Beispiel von Ethereum besprochen, welche bei allen Technologien, die auf Blockchain aufbauen, zum Einsatz kommen.

3.2.3.1 Wallets und Schlüssel

Im Netzwerk von Ethereum existieren zwei verschiedene Arten von Wallets, externe Wallets und Smart Contracts. Externe Wallets sind diejenigen Accounts, welche von menschlichen Nutzern verwendet werden, um mit dem Netzwerk zu interagieren und bestehen aus zwei Schlüsseln, dem **privaten** und dem **öffentlichen Schlüssel**. Hierbei identifiziert der öffentliche Schlüssel den Account im Netzwerk und kann als eine Art Kontonummer verstanden werden, während der private Schlüssel den Zugriff, ähnlich der PIN bei einem Bankkonto, auf den Account ermöglicht. [9, S.60] Der Zugriff auf den Account durch den privaten Schlüssel erfolgt nach dem Prinzip der digitalen Signatur.
Die **digitale Signatur** wird durch die Anwendung der asymmetrischen Verschlüsselung durch privaten und öffentlichen Schlüssel, unter der Annahme, dass die öffentlichen Schlüssel sicher und unveränderlich gespeichert wurden, ermöglicht. Hierbei wird das zu signierende Datum mit einem beliebigen, aber bekannten, Algorithmus, der den privaten Schlüssel verwendet, verschlüsselt und im weiteren zusammen mit dem unverschlüsselten Datum übermittelt. Der Empfänger entschlüsselt nun das verschlüsselte Datum mit dem öffentlichen Schlüssel der Person, welche das Datum signiert hat. Anschließend wird ein Vergleich des entschlüsselten Datums und des unverschlüsselt übermittelten Datums durchgeführt. Ist dieser Vergleich erfolgreich, so kann man davon ausgehen, dass die Signatur von derjenigen Person geleistet wurde, von der man es auch erwartet hat. Falls man nur an der digitalen

Signatur interessiert ist, aber nicht am Inhalt des Datums, so kann das Datum auch zuerst gehasht und anschließend signiert werden, um die Übermittlung zu beschleunigen.
Verlangt man Zugriff auf den externen Account, so signiert man die Zugriffsanfrage mit dem privaten Schlüssel, welcher dem Account zugeordnet ist und übermittelt die Anfrage. Bei einer erfolgreichen Überprüfung durch das Netzwerk erlangt man nun den Zugriff und kann weitere Aktionen mit seinem Account durchführen.
Smart Contracts besitzen im Vergleich zu externen Accounts keine privaten Schlüssel, können aber genauso wie externe Accounts auch Ether speichern, da sie einen öffentlichen Schlüssel besitzen. Die Zugriffsberechtigungen auf das gespeicherte Ether muss durch den Entwickler des Smart Contracts reguliert werden.

3.2.3.2 Hashfunktionen

Hashfunktionen sind einer der Grundbausteine der modernen Kryptographie [9, S.71] und werden dafür eingesetzt, um Eingangsdaten von beliebiger Größe auf Ausgangsdaten abzubilden, welche eine bekannte, feste Größe besitzen.
Bei Ethereum kommt eine Unterkategorie der Hashfunktionen zum Einsatz - die kryptographischen Hashfunktionen, welche folgende Eigenschaften besitzen müssen: [9, S.71]

1. Determinismus:
Die Funktion muss für den gleichen Input immer den gleichen Output generieren, der auch Hashwert genannt wird.

2. Verifizierbarkeit:
Das Hashing ist zeitlich und rechnerisch effizient.

3. Zusammenhangslos:
Schon eine kleine Änderung der Eingangsdaten führt zu einem neuen Hashwert, der so verschiedenen ist, dass keine Rückschlüsse auf den ursprünglichen Hashwert gezogen werden können.

4. Unumkehrbarkeit:
Das Berechnen der Eingangsdaten zu einem gegebenen Hashwert ist zeitlich und rechnerisch ineffizient, und zwar zu einem Grad, der einem Brute-Force-Angriff gleicht.

5. Kollisionsresistenz:
Es soll unwahrscheinlich sein, dass derselbe Hashwert anhand verschiedener Eingangsdaten berechnet werden kann.

Diese Hashfunktionen können unter anderem dafür eingesetzt werden, um einen digitalen Fingerabdruck für den Benutzer zu erstellen, oder die Korrektheit einer

Nachricht oder Datei zu überprüfen, vgl. MD5 File Validation. Weiterhin ist Kollisionsresistenz im Kontext von Blockchain sehr wichtig, da bei einer starken Kollisionsresistenz das Fälschen des digitalen Fingerabdrucks nahezu unmöglich ist und somit die Nachrichten im Netzwerk immer dem richtigen Wallet zugeordnet werden können.

3.2.4 Hash-Baum

Durch den Einsatz eines Hash-Baumes - im Englischen Merkle-Tree - kann die Integrität eines einzelnen Datums überprüft werden, ohne, dass die Gesamtdaten an sich überprüft werden müssen. Diese Methode zur Kontrolle der Integrität ist sehr performant und das Verfahren wird vor allem in Peer-to-Peer Netzwerken, wie TOR, Bitcoin und Git eingesetzt. [11]
Bei der Erstellung eines Hash-Baumes werden die Eingangsdaten in Paare aufgeteilt. Bleibt ein Eingangsdatum bei der Aufteilung übrig, dupliziert man dieses Datum und paart diese miteinander. [12] Diese Paare werden nun miteinander verknüpft und anschließend gehasht. Anschließend werden wieder Paare gebildet und der Vorgang wiederholt sich solange, bis man bei einem einzigen Wert ankommt. [12] Dieser Wert wird **Merkle-Root** genannt.

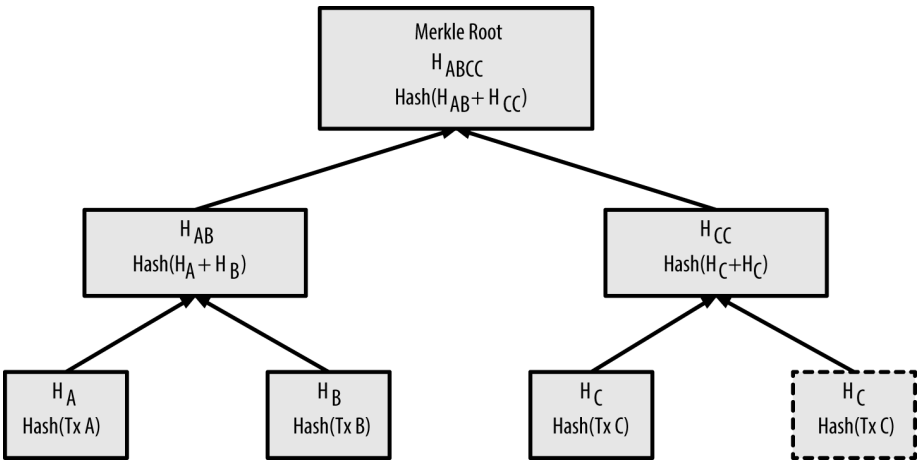


Abbildung 3.1 – Merkle-Tree mit ungerader Paaranzahl[2].

In Abbildung 3.1 [3, S.203] ist ein Hash-Baum mit drei einzigartigen Blättern zu sehen. Diese Blätter bestehen nicht aus den Rohdaten, für die man den Hash-Baum aufbauen möchte, sondern aus dem Hash der Daten (H_A, H_B, H_C). Zur Erstellung eines Merkle-Trees muss man H_C duplizieren und erstellt aus den resultierenden vier Blättern deren resultierenden Elternknoten, indem man die Kombination der

[2]Mastering Bitcoin - Chapter 9: The Blockchain

Hashwerte der Blätter wiederum hasht (Hash(H_A+H_B) = H_{AB}). Dies wird solange wiederholt, bis es nur einen einzigen resultierenden Elternknoten gibt, welcher der Merkle-Root entspricht.
Kommt beim Hashing im Merkle-Tree eine kryptographische Hashfunktion zum Einsatz, entspricht die Merkle-Root einem digitalen Fingerabdruck der Eingangsdaten, anhand dessen sicher festgestellt werden kann, ob ein Datum ein Teil des Merkle-Trees ist, oder nicht.
Bei Bitcoin und Ethereum wird der Hash-Baum eingesetzt, um zu Bestimmen, ob eine Transaktion Bestandteil eines Blocks ist, oder nicht. Hierbei sind die einzelnen Transaktionen die Blätter im Hash-Baum, welche zu der Merkle-Root zusammengeführt werden. Möchte ein Benutzer jetzt eine Kontrolle durchführen, ob eine Transaktion im Block enthalten ist, so müssen nur Hashwerte übertragen und überprüft werden, was weniger Verkehr als die Übertragung der Transaktionsdaten verursacht und weniger Rechenleistung in Anspruch nimmt. [3, S.203]

3.2.5 Transaktion

Transaktionen sind signierte Nachrichten, welche von einem externen Wallet stammen. Diese werden im Netzwerk übermittelt und in der Blockchain gespeichert. Transaktionen verändern den Zustand der Blockchain oder können dazu eingesetzt werden einen Smart Contract in der Virtuellen Maschine zur Ausführung zu bringen. Möchte nun Alice zwei Ether an Bob überweisen, so erstellt Alice eine Transaktion mit folgenden Bestandteilen: [9, S.100]

- Nonce
- Gaspreis
- Gaslimit
- Empfänger
- Wert
- Daten

Hierbei wird die Nonce eingesetzt, um ein Replay der Transaktion zu verhindern, also um sicherzustellen, dass die Transaktion auch wirklich vom Sender kommt und nicht abgefangen, von einem Angreifer manipuliert und weitergesendet wurde. Gaspreis und Gaslimit definieren, wie viel die Ausführung der Transaktion maximal kosten wird, während Empfänger, Wert und das Datenfeld festlegen, an wen die Überweisung des mitgeschickten Ethers geht. Im Beispiel ist der Empfänger Bob, der Wert der Transaktion ist 2 Ether und im Datenfeld steht die Binärkodierung des Transferbefehls.

3.2.6 Peer-to-Peer-Netzwerk

In einem Peer-to-Peer-Netzwerk (P2P-Netzwerk) benutzt und stellt ein Nutzer (Peer) die grundlegenden Ressourcen des Netzwerkes zur Verfügung. Die bereitgestellten Ressourcen sind ein Teil der eigenen verfügbaren Ressourcen und umfassen unter anderem Speicherplatz, Rechenleistung und Bandbreite. [13] Im Vergleich zu einer zentralisierten Netzwerkstruktur kommt es bei einem P2P-Netzwerk also nicht zu einem Leistungseinbruch, falls viele Nutzer dem Netzwerk beitreten, sondern zu einem Leistungszuwachs, da diese nach dem Beitreten ihre Ressourcen dem Netzwerk zur Verfügung stellen.
Jeder Peer besitzt in einem P2P-Netzwerk die gleichen Berechtigungen und wird allgemein als Knoten (Node) im Netzwerk bezeichnet. [13] Zudem besitzt ein P2P-Netzwerk keinen zentralen Server, auf dem die Daten des Netzwerks gespeichert werden, stattdessen speichern alle Peers die Daten und tauschen diese untereinander aus. Es gibt also keinen zentralen Schwachpunkt im Netzwerk, bei dem die Daten manipuliert werden können und es gibt keine zentrale Instanz, welche bestimmen kann, für was die Ressourcen im Netzwerk eingesetzt werden. [13] Weiterhin ist das P2P-Netzwerk voll funktionsfähig, falls alle Nodes, welche den kompletten Datensatz speichern, bis auf eine ausfallen, da anhand dieser das Netzwerk wieder aufgebaut werden kann.
Nicht jede Node muss im Kontext der Blockchain den kompletten Datensatz speichern, dies übernehmen die sogenannten **Full-Nodes**. Neben den Full-Nodes gibt es noch andere Arten von Nodes, unter ihnen sind Mining- und Staking-Nodes vertreten. Hierbei sind Mining-Nodes für Proof of Work (PoW) und Staking-Nodes für Proof of Stake (PoS) von Bedeutung.

3.3 Konsensfindung

Sobald man eine gemeinsame Wahrheit dezentral speichern will, ist es essenziell, dass dies nach einem Schema abläuft, das für alle Teilnehmer nachvollziehbar ist. Dieser deterministische Weg zu einer gemeinsamen Wahrheit zu kommen, wird Konsensfindung genannt. Das Problem der Konsensfindung existierte vor der Entwicklung der Blockchain und kam schon bei dezentralen P2P-Netzwerken vor, als man den Zustand über alle Peers im Netzwerk synchronisieren wollte. [9, S.319]

3.3.1 Gründe für Konsensfindung im Netzwerk

Agiert man in einem dezentralen Netzwerk, so kommt zwangsläufig die Frage auf, wie man die gemeinsame Wahrheit findet und sich über sie verständigt, sodass jeder Teilnehmer im Netzwerk auf der Grundlage des gleichen Zustands arbeiten kann.

Hierbei ist eine der zentralen Schwierigkeiten, dass es nach Definition keine zentrale Instanz geben soll, welche man nach dem aktuellen Zustand fragen kann und welche diesen auch festsetzt. Genau diese Eigenschaft bietet aber auch einen herausstehenden Vorteil und Anreiz für die Blockchain - das Netzwerk ist transparent und man benötigt keine Erlaubnis, um dem Netzwerk beitreten zu dürfen. [9, S.319]
Allgemein möchte man in einem anonymen, dezentralen Netzwerk keiner Benutzergruppe vertrauen müssen, welche Wahrheit im Netzwerk gerade die Richtige ist. Eine gemeinsame Wahrheit muss aber auch in Fällen findbar sein, bei denen sich Teilnehmer im Netzwerk falsch verhalten und diese manipulieren wollen. Dieses Problem ist in der Informatik als byzantinischer Fehler bekannt und dieses Problem wird im Weiteren näher erläutert.

3.3.2 Byzantinischer Fehler

Der byzantinische Fehler beruht auf dem fiktiven historischen Problem der byzantinischen Generäle. Bei diesem Problem belagert die byzantinische Armee eine Stadt, ist in verschiedene Divisionen aufgeteilt, welche von einzelnen Generälen befehligt werden und um die Stadtgrenzen herum stationiert sind. Die Generäle müssen sich auf einen von zwei Plänen einigen und diesen dann auch gemeinsam durchführen. [14]
Zur Kommunikation zwischen den Generälen werden Nachrichtenboten eingesetzt und folgende Pläne stehen zur Auswahl:

- Gemeinsamer Angriff zu einem festgelegten Zeitpunkt
- Kollektiver Rückzug

Die Lösung dieses Problems stellt sich als sehr einfach heraus, wären alle Nachrichtenboten und Generäle vertrauenswürdig. Jedoch wurden einige der Akteure von den Belagerten bestochen und es besteht die Chance, dass sie die Befehle nicht befolgen oder korrekt weitergeben. [14] Dies hat zur Folge, dass sich mehrere mögliche Szenarien ergeben, bei denen Teile der Armee den falschen Befehlen folgen. Ein mögliches Szenario des fehlerhaften Nachrichtentransports ist in Abbildung 3.2 [14] dargestellt.

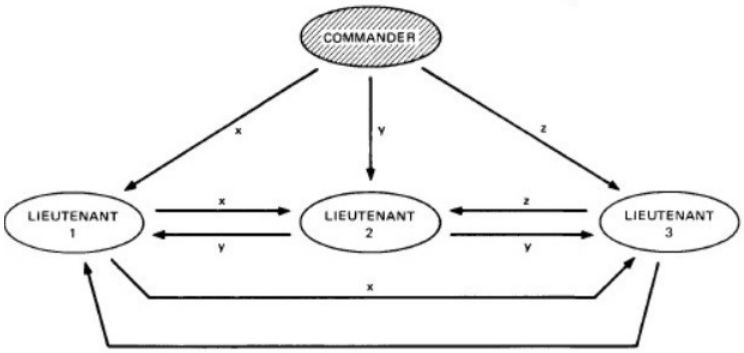


Abbildung 3.2 – Nachrichtentransport mit korrumpierten Kommandant[3].

In diesem Beispiel ist der Kommandant die kompromittierte Partei im System und er gibt den Heerführern unterschiedliche Befehle, welche diese wiederum untereinander austauschen. Geht man davon aus, dass der Kommandant nur einem der Heerführer (Lieutenant 2) den richtigen Befehl (Y) gegeben hat, aber den beiden anderen (Lieutenant 1 & 3) einen falschen, aber gleichen, Befehl (X = Z), so kommt es dazu dass sich die Heerführer darauf einigen, den falschen Befehl (X = Z) zu befolgen. Die Komplexität dieses Problems nimmt mit der Anzahl der Teilnehmer zu. Befinden sich neben dem Kommandanten nur zwei Heerführer im Netzwerk, so müssen 2 Nachrichten zwischen den Heerführern ausgetauscht werden, um einen Konsens im Netzwerk der Generäle herzustellen, während es bei drei Heerführern schon 6 Nachrichten und bei vier Heerführern 12 Nachrichten sind. Die Anzahl der benötigten Nachrichten wächst also rapide mit der Anzahl der Heerführer n und lässt sich durch $(n-1)*n$ ausdrücken.
Das Problem der byzantinischen Generäle kann man auf das Thema Blockchain überführen und hierbei feststellen, dass nach dem gleichen Prinzip ein Konsens der Teilnehmer, ohne benötigtes Vertrauen in eine zentrale Instanz, gebildet werden kann, während es gleichzeitig möglich ist, dass einzelne Teilnehmer korrumpierbar sind und eine falsche Wahrheit verbreiten möchten, dies jedoch keine Auswirkungen auf die tatsächliche Wahrheit hat.

3.3.3 Proof of Work - PoW

3.3.3.1 Was ist Proof of Work?

PoW ist der bekannteste Algorithmus, welcher als erstes bei einer Blockchain eingesetzt wurde, [15] um das Problem der Konsensfindung zu lösen, und kommt sowohl bei Bitcoin und als auch noch bei Ethereum zum Einsatz.

[3]The Byzantine Generals' Problem - All Things Ledger - Medium

Die Grundidee hinter PoW ist es, dass die Teilnehmer im Netzwerk, welche die Transaktionen bestätigen, einen Nachweis erbringen müssen, dass sie eine gewisse Menge an Arbeit für die Berechnung des nächsten Blocks aufgebracht haben. Hierbei gilt beim PoW-Algorithmus, dass die längste Blockchain automatisch die korrekte ist, [15] d.h. solange die Hälfte der Arbeit, welche die Miner im Netzwerk erbringen, ehrlich erfolgt, ist die Sicherheit im Netzwerk garantiert. Ein Nebenprodukt bei der anfallenden Rechenarbeit - auch Mining genannt - ist die Wertschöpfung in Form von Coins, welche den Minern als Anreiz für ihre Arbeit ausgezahlt werden.

3.3.3.2 Wie funktioniert Proof of Work?

Die Grundlage der bekanntesten Implementierung des PoW-Algorithmus bei Bitcoin ist die Berechnung des Hashes eines Hashes, durch doppelte Anwendung des SHA256 Algorithmus. [16] Die grundlegende Idee hierbei ist es die Eigenschaft des kryptographischen Hashes, dass er einfach verifiziert werden kann, aber es schwer genug ist den Ausgangswert zu einem vorgegebenen Hash zu finden, auszunutzen. Dies hat zur Folge, dass die Berechnungen beim Schürfen einen gewissen Aufwand benötigen, aber das Überprüfen des Ergebnisses eines Miners durch die Community schnell und einfach erfolgen kann.
Grundsätzlich müssen folgende Schritte beim Schürfen eines neuen Blocks im Netzwerk durchgeführt werden:

- Neuen Block mit den verfügbaren Transaktionen erstellen
- Zusammengestellten Block mit NONCE kombinieren und hashen
- NONCE anpassen bis gewünschtes Muster des Hashes erreicht wurde

Nachdem die Transaktionen zu einem Block zusammengefasst wurden, welchen man bestätigen möchte, wird an diesen eine NONCE angehängt und das Ergebnis anschließend mit der vorgegebenen Hash-Funktion gehasht. Nun überprüft man, ob der berechnete Hashwert dem vorgegebenen Format für einen gültigen Block entspricht. Dieses Format wird bei Bitcoin durch die Anzahl der führenden Nullen vorgegeben und ist ein Indikator für die Schwierigkeit für das Finden eines neuen Blocks. [17]
Den Anstieg der Schwierigkeit durch die Erhöhung der benötigten Nullen kann man sich analog zum mehrmaligen Würfeln vorstellen. Zum Werfen von drei Sechsen hintereinander benötigt man im statistischen Mittel grundsätzlich mehr Versuche, als zwei Sechsen hintereinander zu werfen - genauso ist es unwahrscheinlicher einen Hashwert mit drei führenden Nullen zu erzeugen, als einen Wert mit zwei führenden Nullen.
Liegt der Hashwert im korrekten Format vor, so hat man den nächsten gültigen

Block gefunden und kann ihn an die Blockchain anhängen, falls nicht passt man die NONCE an und wiederholt den Prozess solange bis der nächste Block entweder von einem selbst oder von jemanden anderen gefunden wurde. Danach wiederholt sich der Prozess für die nächsten Transaktionen, welche im nächsten Block bestätigt werden sollen.
Andere Implementierungen, wie Ethash bei Ethereum, des Proof of Work Algorithmus agieren nach einem ähnlichen Prinzip und suchen auch nach einem korrekten Hashwert, um den nächsten Block zu bestätigen, es kommen aber andere Hash-Funktionen und andere Ausgangsdaten für die Berechnung zum Einsatz, [18] um den Einsatz von dedizierter Hardware sogenannten application specific integrated circuits (ASICs) zur Beschleunigung des Schürfen zu erschweren. [9, S.321]

3.3.3.3 Fazit

Im Prinzip wird bei PoW Rechenleistung - effektiv Strom - in Sicherheit für das Netzwerk umgewandelt. Während der Algorithmus den Vorteil hat, dass die korrekte Funktionsweise erprobt ist, und er unter der Annahme, dass die Mehrzahl der Nutzer im Netzwerk ehrlich sind, auch sicher ist, besitzt er leider auch ein fundamentales Problem. Die Nutzung von PoW verbraucht bei vielen Anwendern große Mengen an Strom [15] und ist dementsprechend, im Vergleich zu anderen Algorithmen, bei einer langfristigen oder neuen Anwendung, nicht mehr effizient genug, um eine Nutzung zu empfehlen. Weiterhin kommt es durch den Zusammenschluss von Minern zu sogenannten Miningpools dazu, dass die Miner nicht mehr so zufällig verteilt sind, wie man es sich eigentlich wünscht. In der folgenden Abbildung 3.3 [19] ist zu erkennen, dass ein Großteil der Blöcke bei Ethereum von vier Miningpools (Ethermine, Sparkpool, Nanopool und f2pool2) bestätigt wurden.

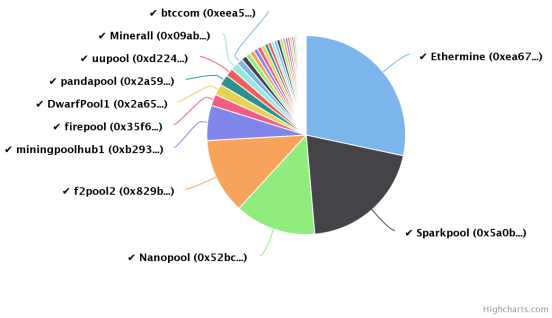


Abbildung 3.3 – Ethereum Top Miners[4].

[4]Top Miners over the last 24h - etherchain.org

Die Konsolidierung der Miner in die Miningpools birgt nun die Gefahr, dass sich die großen Pools absprechen könnten und somit einen 51% Angriff auf das Netzwerk durchführen könnten.

3.3.4 Proof Of Stake - PoS

3.3.4.1 Was ist Proof of Stake

PoS ist ein weiterer bekannter Algorithmus, der das Problem der Konsensfindung löst und wurde zum ersten Mal bei Peercoin in 2012 eingesetzt. [20] Hierbei ist die Grundidee, dass derjenige Benutzer, der den nächsten Block erstellt, durch ein Zufallssystem ausgelost wird, welches unter anderem die Menge der Cryptowährung, die der Benutzer besitzt, berücksichtigt.

3.3.4.2 Wie funktioniert Proof of Stake?

Bei PoS nehmen alle Benutzer des Netzwerkes an der Erstellung eines neuen Blocks teil, sobald sie einen Stake besitzen. Dieser Stake kann entweder das alleinige Besitzen der Cryptowährung der Blockchain sein, oder eine fest eingefrorene Menge der Cryptowährung, welche der Nutzer hinterlegen muss. [20] Ein neuer gefundener Block muss folgende Ungleichung erfüllen, um gültig zu sein: [21]

$$hash(hash(B_{\text{prev}}), A, t) \leqslant bal(A)M/D$$

wobei:

- B_{prev} der gültige Block ist, an dem der neue Block angehängt werden soll
- A die Addresse des Nutzers ist
- t die aktuelle Zeit ist
- $bal(A)$ der Stake des Nutzers A ist
- M die Anzahl der Argumente der Hashfunktion ist
- D die vorgegebene Schwierigkeit ist

Angenommen es gibt nur fünf Stakeholder in einem Netzwerk, das insgesammt 100 Token seiner Cryptowährung bereitstellt. Alice besitzt 30 Token, Bob 20 Token, Charlie 15 Token, David 10 Token und Eric die restlichen 25 Token, so lässt sich bei diesem Beispiel die Wahrscheinlichkeit, welcher Stakeholder der Ersteller des nächsten Blocks ist, einfach bestimmen. Die Chance ist proportional zu der Menge an Token, die der Nutzer hält - Alice hätte zum Beispiel eine 30% Chance den nächsten Block zu erstellen.

3.3.4.3 Fazit

Vergleicht man PoW mit PoS, so stellt man fest, dass PoS weniger Strom benötigt, um Blöcke zu bestätigen, trotzdem besitzt PoS andere Schwachstellen. Eine dieser Schwachstellen ist das „Nothing at Stake“-Problem. [20] Sollte es zu einer Situation kommen, bei der Abgestimmt werden muss, welche Chain, von zwei parallel erstellten Subchains, diejenige ist, welche man als Mainchain weiterführen möchte, so gibt es keinen Mechanismus, der die Stakeholder bestraft, falls sie jeweils die Hälfte ihres Stakes auf eine der beiden Subchains setzen. Dies kann dazu führen, dass es zu keiner eindeutigen Entscheidung kommt, da alle Stakeholder ihren Stake auf die beiden Subchains gleichmäßig aufteilen. Dieses Problem wird bei Casper, der PoS Umsetzung von Ethereum, durch eine der Slashing-Conditions gelöst, welche definieren, in welchen Situationen der Stakeholder automatisch für sein Fehlverhalten bestraft wird.
Weiterhin ist die Auswahl des Users, der den nächsten Block bestätigt, anhand der Menge der Cryptocurrency, die er besitzt, problematisch, da hierbei die reichsten User im Netzwerk beim Bestätigungsvorgang bevorzugt werden und sich somit der Bestätigungsprozess um diese User zentralisiert. Dieses Problem wird bei den verschiedenen PoS Blockchains unterschiedlich gelöst. Bei Peercoin wird die Chance einen neuen Block zu bestätigen mit dem Alter der Coins des Users gewichtet, die er im Moment besitzt, [21] während bei NXT die Chance mit der Menge der Coins, die er besitzt, und einem weiteren Zeitfaktor gewichtet wird. [21] So haben im System von NXT alle User am Anfang eine niedrige Chance den Block zu bestätigen, welche mit der Zeit wächst, aber wieder die reichsten User im Netzwerk bevorzugt.

3.3.5 Delegated Proof of Stake

3.3.5.1 Was ist Delegated Proof of Stake

Delegated Proof of Stake (DPoS) ist eine Weiterentwicklung von PoS und es findet zum Beispiel Anwendung bei BitShares oder Lisk. [21] Hierbei werden die neuen Blöcke von einer vorher festgelegten Gruppe (Delegates) erstellt, die für ihre Teilnahme am Netzwerk belohnt werden, jedoch bestraft werden, falls sie sich entgegen den Interessen des Netzwerkes verhalten.

3.3.5.2 Wie funktioniert Delegated Proof of Stake

Bei DPoS ist das Verfahren, wie neue Blöcke erstellt werden, einfach gehalten und erfolgt nach dem folgenden Schema:

1. Delagates fassen Transaktionen zu Blöcken zusammen
2. Delegates bestätigen die erstellte Blöcke, indem sie diese signieren
3. Der Block wird an die Blockchain angehängt, sobald eine vorher festgelegte Anzahl von Delegates den Block bestätigt haben

Jedoch ist das Verfahren, wie die Delegates bestimmt werden, von Umsetzung zu Umsetzung verschieden. Bei Tendermint können die erstellten Blöcke von allen Benutzern im Netzwerk signiert werden. [21] Weiterhin ist es möglich, dass die Delegates durch einen Wahlprozess im Netzwerk von allen Benutzern bestimmt werden. [21] Dies ermöglicht eine Kontrolle der Delegates durch die Nutzer des Netzwerkes.
Bei BitShares werden neue Blöcke durch sogenannte „Zeugen“(Witnesses) erstellt, welche den vorher genannten Delegates entsprechen. Jeder Stakeholder besitzt, anteilig an der Menge von BitShares, die er besitzt, Stimmrechte, welche er nach eigenem Ermessen auf die Zeugen aufteilen kann. Anschließend werden die Stimmen gezählt und die N Zeugen ausgewählt, welche die meisten Stimmen erhalten haben und deren Summe an Stimmen mindestens 50% der Gesamtstimmanzahl ausmacht. [22] Die Wahl wird in regelmäßigen Abständen wiederholt.
Weiterhin wird die Reihenfolge, in der die Zeugen die Blöcke nun bestätigen, zufällig neu bestimmt, nachdem alle Zeugen jeweils einen Block bestätigt haben. [21]

3.3.5.3 Fazit

DPoS adressiert einige Probleme, welche PoS hat, wie zum Beispiel das „Nothing at Stake“-Problem, besitzt jedoch auch negative Aspekte, die PoS nicht besitzt. Abhängig davon, wie die Delegates ausgewählt werden, besitzt DPoS einen unterschiedlichen Grad an Zentralität, da die Anzahl der Delegates entweder von Anfang an beschränkt ist, oder der Status gekauft werden muss, indem ein Geldbetrag als Sicherheit hinterlegt wird. Die Umsetzung bei BitShares, dass die Delegates durch die Nutzer des Netzwerkes gewählt werden, ist die demokratischste Lösung, birgt aber auch die Gefahr, dass die Nutzer ihre Wahl automatisieren und grundsätzlich ihre Stimme an denselben Zeugen vergeben, was dazu führt, dass sich die Auswahl der Zeugen nicht mehr verändert und sich ein unveränderliches zentrales Gremium bildet, das das Netzwerk kontrolliert.

3.3.6 Vergleich der Konsensalgorithmen

Vergleicht man die vorgestellten Konsensalgorithmen miteinander, so stellt man fest, dass es hierbei keinen klaren Favoriten gibt, der alle wichtigen Aspekte für eine Blockchain risikofrei behandelt.
PoW ist sicher und erprobt, lässt sich aber schwer skalieren und schränkt somit die Leistungsfähigkeit des Netzwerkes ein. Gegen PoW spricht vor allem der hohe Energieverbrauch, welcher auf die Dauer nicht tragbar sein wird. Weiterhin hat sich gezeigt, dass es zu einer Zentralisierung der Miningkraft durch die Bildung von Miningpools kommt. Somit sind die Miner im Netzwerk nicht mehr zufällig verteilt und es kann zu Problemen führen, falls es zu einer Absprache zwischen den Miningpools kommt.
PoS ist im Vergleich zu PoW energieeffizienter und skaliert auch besser mit der Anzahl von Nutzern im System, da jeder Nutzer theoretisch einen Block bestätigen kann. Jedoch müssen Probleme wie das „Nothing at Stake“-Problem und die verschiedenen Arten, wie die Machtverhältnisse konsolidiert werden können, gelöst werden, um PoS ohne Vorbehalte empfehlen zu können.
DPoS baut auf PoS auf und ist dementsprechend auch energieeffizienter und skalierbarer als PoW. DPoS löst aber auch Probleme von PoS, indem Delegates bei einem Fehlverhalten, wie bei „Nothing at Stake“, bestraft werden können und somit dazu animiert werden sich korrekt zu verhalten. Weiterhin können alle Nutzer im Netzwerk durch ein Wahlsystem, wie es bei BitShares zum Einsatz kommt, zu einem Delegate bestimmt werden. Jedoch müssen die User aktiv am Netzwerk teilnehmen, sodass sich der Pool an Delegates regelmäßig verändern kann und es nicht zu einer Zentralisierung der Macht kommt.
Stellt man sicher, dass der Pool an Delegates veränderlich bleibt und für jeden zugänglich ist, stellt DPoS eine gute Weiterentwicklung für eine skalierbare Blockchain dar. PoW ist in den Punkten Sicherheit und Einsatzerfahrung gegenüber PoS und DPoS zu präferieren, jedoch ist die Skalierbarkeit der resultierenden Blockchain limitiert und es sollte ein Konzept ausgearbeitet werden, wie eine Umstellung auf einen anderen Konsensalgorithmus zu erfolgen hat, sobald die Leistung zu stark begrenzt ist. Beim Einsatz von PoS müssen genaue Überlegungen getroffen werden, wie die Konsolidierung der Machtverhältnisse auf einzelne User im Netzwerk verhindert werden kann und, ob es überhaupt erforderlich ist, dass prinzipiell jeder Nutzer im Netzwerk für die Erstellung von Blöcken ausgewählt werden kann, oder ob eine demokratische Wahl von Vertretern, die diese Aufgabe übernehmen, sinnvoller ist.

3.4 Anwendungen für die Blockchain

Es gibt eine große Anzahl von möglichen Anwendungsgebieten, welche durch den Einsatz einer Blockchain dezentralisiert und auch in Aspekten wie Sicherheit und Verfügbarkeit verbessert werden können. Im Weiteren werden mögliche Einsatzbereiche besprochen.

3.4.1 Klassisch: Cryptowährung

Der klassische und auch bekannteste Anwendungsfall ist es, eine Cryptowährung auf der Grundlage einer Blockchain zu erstellen. Cryptowährungen können zu den Fiat-Währungen gezählt werden, da sich die Benutzer entweder auf einen freien Wert verständigen, oder der Wert festgelegt wird, indem die Cryptowährung an den Kurs einer etablierten Währung, wie Dollar oder Euro, gekoppelt wird. Es gibt verschiedene Arten, wie die Währungsmenge vergrößert wird, in den meisten Fällen geschieht dies aber, indem die Miner bei PoW oder die Validatoren bei PoS eine Belohnung für ihre Arbeit und ihr Risiko erhalten.

3.4.2 DNS-Server

Ein weiteres Anwendungsgebiet für die Blockchain ist es, mit ihr einen dezentralen DNS-Server aufzubauen. Das aktuelle DNS-System ist schon ein verteiltes Netzwerk, jedoch wird es im Moment von einer zentralen Instanz - der ICANN (Internet Corporation for Assigned Names and Numbers) - reguliert und ist in einer hierarchischen Struktur aufgebaut. [23] Durch diese Struktur wird der Hauptserver ein attraktives Angriffsziel, da ein erfolgreicher Angriff die Auflösung der Adressnamen zu den zugehörigen IP-Adressen erschwert, falls nicht sogar verhindert.
Stellt man nun dieses System schrittweise auf eine funktionierende Lösung um, die auf einer Blockchain aufbaut, so verändert sich die hierarchische Struktur zu einer flachen Struktur, da alle Teilnehmer alle Einträge speichern und validieren. Zur Umsetzung ist aber ein klares Konzept nötig, wer Einträge neu erstellen und alte Einträge ändern darf. Weiterhin muss das neue System mindestens die gleiche Leistung erbringen, wie das alte System, und es dürfen auf den normalen Anwender keine zusätzlichen Schritte oder Kosten zukommen.

3.4.3 Smart Contracts

Ein interessantes Feature, welches manche Blockchains anbieten, sind die sogenannten Smart Contracts. Hierbei handelt es sich um Programme, die dezentral auf der Blockchain, meist in einer Virtuellen Maschine, ausgeführt werden können. Ihre Grundlagen und Funktionsweise wird im nächsten Kapitel genauer erläutert.

Kapitel 4

Smart Contracts

4.1 Grundlagen

Smart Contracts verfügen im Gegensatz zu den externen Accounts nur einen öffentlichen Schlüssel. Dieser öffentliche Schlüssel dient, wie eine Kontonummer dazu den Smart Contract zu identifizieren und ermöglicht es dem Smart Contract weiterhin Vermögenswerte zu verwalten.
Der Begriff Smart Contract ist irreführend, da es sich hierbei nur um Programme handelt, aber nicht um legal bindende Verträge. Diese Programme sind, sobald sie ausgeführt werden, nicht mehr veränderlich, was bedeutet, dass der Smart Contract bei einem Update erneut in der neuen Version erstellt werden muss. Weiterhin sollte die alte Version für die Anwender gesperrt werden.
Smart Contracts kontrollieren sich im Gegensatz zu externen Accounts selbst, indem sie die ihnen einprogrammierten Regeln befolgen und werden von einer Virtuellen Maschine ausgeführt. Im Weiteren wird die nähere Funktionsweise der Smart Contracts im Kontext von Ethereum behandelt.

4.1.1 Virtuelle Maschine

Die Virtuelle Maschine von Ethereum (EVM) ist der Kern des Ethereumnetzwerks. Hierbei handelt es sich um eine quasi Turing-vollständige Automaten (state machine), welcher die Funktionalität von Ethereum bereitstellt. Die EVM ist nur quasi Turing-vollständig, da alle Prozesse nur eine limitierte Anzahl von Berechnungen durchführen können. [9, S.297] Diese Limitierung wird Gaslimit genannt, wobei Gas ein Maß für die Rechenleistung ist, die eine Berechnung benötigt. Durch die Limitierung der Gasmenge, welche ein Prozess verbrauchen darf, wurde das Halteproblem gelöst - entweder der Prozess führt die Instruktionen erfolgreich aus, oder wird automatisch abgebrochen, sobald das Gaslimit erreicht wurde. [9, S.297]

4.1.2 Funktionsweise

Smart Contracts werden in einer speziellen Programmiersprache geschrieben und beinhalten genau die Funktionalität, welche der Autor implementiert. Bei der Ausführung wird der Programmcode in Bytecode übersetzt, der für die EVM verständlich ist und die weitere Ausführung des Smart Contracts erfolgt nun durch die EVM. Bei der Erstellung erhält der Autor des Smart Contracts nur spezielle Rechte, falls er diese auch einprogrammiert hat.
Smart Contracts können nicht automatisiert eigene Funktionen ausführen. Hierzu bedarf es einem Aufruf einer Funktion des Smart Contracts durch einen externen Account. Smart Contracts können aber Funktionen eines anderen, auf der EVM laufenden, Smart Contracts erfolgreich aufrufen. Hierbei ist es aber notwendig, dass der ursprüngliche Aufruf in der Kette von Aufrufen von einem externen Account stammt. [9, S.129]

4.2 Anwendungen von Smart Contracts

4.2.1 DAOs

Dezentrale Autonome Organisationen - kurz DAOs - sind wie der Name schon sagt dezentralisierte Organisationen, die unabhängig Entscheidungen treffen können und nicht von einer zentralen Instanz, wie zum Beispiel dem Vorstand in einer Aktiengesellschaft, kontrolliert werden. Im Kontext Blockchain sind DAOs Firmen, die nur aus Programmcode bestehen und dementsprechend nur nach den vorher festgelegten und programmierten Regeln agieren. Viele dieser dezentralen autonomen Organisationen bauen auf dem Ökosystem Ethereum auf. Ein bekanntes Beispiel für eine DAO, die leider gescheitert ist, wird im folgenden Kapitel näher besprochen.

4.2.1.1 The DAO

Die Idee hinter „The DAO“ war es ein dezentrales Unternehmen ohne Firmensitz zu gründen, wobei die Investoren Stimmrechte im Verhältnis zu ihrer Investition bekamen. [24] Die Dezentrale Autonome Organisation sollte wie ein Investmentfond funktionieren und das eingesammelte Kapital in Startups und andere Produkte investieren, um einen Gewinn für die Investoren zu erzielen. [25]
Die Entscheidungen, in welche Bereiche und Unternehmen investiert werden sollte, wurde nicht von einer zentralen Instanz, wie einem Geschäftsführer, getroffen, sondern diese Entscheidungen sollten von den Investoren über Abstimmungen getroffen werden. Dieses Vorgehen bricht mit der klassischen Struktur, die von den allgemein bekannten Unternehmen bekannt ist, indem die Ausrichtung des Unternehmens von der Mehrheit der Teilhaber vorgegeben wird.

Die Investoren selbst können die Vorschläge, wie sich die Unternehmung weiterentwickeln oder in welche Produkte sie investieren soll, einreichen, und somit eine neue Abstimmung anstoßen. Zur Vereinfachung dieses Prozesses stellten die Entwickler hinter „The DAO“ Vorlagen zur Verfügung, um auch Laien die Möglichkeit zu bieten sich an dem Geschäftsgeschehen zu beteiligen. [25]
In der folgenden Abbildung 4.1 [25] ist der allgemeine Ablauf einer Investition der DAO zu erkennen, welcher im Folgenden weiter erläutert wird.

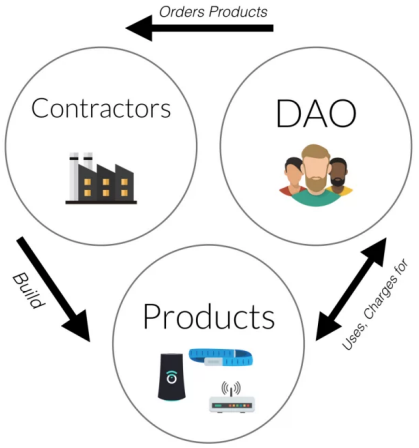


Abbildung 4.1 – DAO-Flow[5].

Eine Investition der Organisation sollte wie folgt ablaufen:

1. Erfolgreiche Abstimmung über die Investition
2. Erstellen eines Smart Contracts für die Investition
3. Automatische Auszahlung bei Leistungserbringung durch den Smart Contract

Nach der erfolgreichen Abstimmung über den von einem Investor eingebrachten Investitionsvorschlag wird ein Smart Contract aufgesetzt, welcher die Rechte und Pflichten der beiden Parteien - Investor und Investitionsempfänger - festlegt. Hierbei hinterlegt der Investor den ausgehandelten Investitionsbetrag auf dem Smart Contract. Sobald der Investitionsempfänger den Auftrag erledigt hat und dies auch nachweislich bei dem Smart Contract bestätigt hat, zahlt dieser automatisch den Investitionsbetrag an den Auftragsnehmer aus, während der Investor Zugriff auf das Auftragsergebnis erhält.

Aufbau und Funktionsweise der DAO verkörpern zwei grundsätzliche Vorteile, welche normale Investitionsorganisationen nicht besitzen: alle Aktionen sind transparent

[5]The DAO bricht Crowdfunding-Rekorde und sammelt fast 160 Millionen ein | Gründerszene

und von allen einsehbar und werden zudem demokratisch beschlossen. Aber genau diese grundlegende Eigenschaft wurde schließlich ausgenutzt, um die DAO zum Fall zu bringen.
Im Falle, dass ein Investor die Entscheidung der Mehrheit ablehnt, steht es ihm frei sein Kapital aus der DAO abzuziehen und somit wieder zurückbekommen. [24] Dieser Prozess sollte, als zusätzliches Zugeständnis den Investoren gegenüber, zu mehr Vertrauen in die DAO führen, da ein Investor durch dieses Feature nicht an die Entscheidung der Mehrheit gefesselt ist. Genau durch dieses Feature wurde aber ein Angriff möglich, welcher weitreichende Auswirkungen auf das gesamte Ethereumnetzwerk hatte - der sogenannte Reentrance-Angriff.
Der hier durchgeführte Reentrance-Angriff machte sich eine grundsätzliche Funktionsweise der Smart Contracts und eine Unachtsamkeit der Programmierer der DAO zu nutze, um mehr als die eigenen hinterlegten Vermögenswerte aus der DAO abzuziehen. Im Falle, dass ein Investor seine Vermögenswerte aus der DAO abziehen möchte, wird das hinterlegte Investment in eine Child-DAO überwiesen, auf die der Investor dann nach einer Wartefrist Zugriff erhält. Bei diesem Prozess wurde aber die interne Buchführung, wie hoch das Investment des einzelnen Investors ist, erst nach Abschluss des Transfers angepasst. Dies machte sich der Angreifer zu Nutze und rief in der sogenannten Fallback-Methode seines Angriffscontracts die Funktion, um sein Investment aus der DAO abzuziehen, rekursiv auf. Hierbei muss man wissen, dass der Transfer von Ether standardmäßig über die Fallback-Methode erfolgt und diese somit zwangsläufig von der DAO aufgerufen werden musste.
Der Angriff führte zu einem Hardfork von Ethereum, da letztendlich der Verlust der Geldmenge, welche in der DAO investiert war, zu einem Vertrauensverlust in die Sicherheit von Ethereum und der Tokens, welche auch von der DAO verwendet wurden, geführt und somit die Existenz des Netzwerkes gefährdet hätte. Bei diesem Hardfork wurde der Zustand der DAO auf den Zustand vor dem Hack zurückgesetzt, sämtliche hinterlegten Investitionen eingefroren und den Investoren eine Möglichkeit geboten, ihre Investition zu einem fixen Verhältnis wieder in Ethereum umzuwandeln. [26] Ein Hardfork ist normalerweise eine nicht abwärtskompatible Weiterentwicklung der Blockchain, welche zu einer Spaltung des Netzwerkes führt. [27] In dieser Situation wurde aber ein Hardfork dafür verwendet, um in das Ökosystem von Etherum einzugreifen. Generell ist es nicht schwer alle Mitglieder des Netzwerkes dazu zu bringen nach dem Fork in das neue Netzwerk zu wechseln, im Falle des DAO-Hacks kam es jedoch zu Meinungsverschiedenheiten der Nutzer, was dazu führte dass es seit diesem Zeitpunkt zwei separat voneinander agierende Versionen von Ethereum gibt - Ethereum und Ethereum-Classic. Die Nutzer von Ethereum-Classic vertreten die Meinung, dass der Code Gesetz ist, und dementsprechend auch das Ausnutzen einer Sicherheitslücke nicht zu einer Korrektur der „Betreiber“ des Netzwerkes führen sollte, während die Nutzer von Ethereum dem Eingriff, um den Angriff rückgängig

zu machen, zustimmten. [26]
Eine Möglichkeit den Reentrace-Angriff zu verhindern ist im Kapitel 4.3.1 Checks Effects Interaction - Wechselwirkung nachzulesen. In diesem Kapitel werden die angreifbaren Codeausschnitte gezeigt und besprochen wie der Code überarbeitet werden kann. Ein vollständiges Beispiel für einen Reentrance-Angriff ist im Anhang zu finden.

4.2.1.2 MakerDAO

MakerDAO ist eine aktuell florierende dezentrale autonome Organisation. Die Organisation stellt zwei verschiedene Token zur Verfügung - DAI und Maker (MKR). [28] Bei DAI handelt es sich um eine sogenannte Stable Coin, und ihr Wert ist an den Kurs des Dollars gekoppelt. Im Gegensatz zu DAI existiert von MKR nur eine begrenzte Menge. MKR kann bei MakerDAO entweder dafür eingesetzt werden, um eine Stimme bei Wahlen abzugeben, bei denen zum Beispiel entschieden wird, wie hoch die Sicherheitsleistung bei Krediten im System sein muss, oder um Stabilitätsgebühr bei der Rückzahlung eines DAI Kredites zu entrichten. [28] Die entrichtete Menge an MKR-Tokens, die verwendet wird, um die Stabilitätsgebühr zu bezahlen, wird hierbei vernichtet. [28]
Möchte man DAI erstellen, so erstellt man eine „collaterized debt position“ (CDP) anhand der folgenden Schritte: [28]

1. Erstelle CDP Smart Contract
2. Überweise die Sicherheitsleistung (im Moment: Ether) an CDP
3. DAI können abgehoben werden
4. Sicherheitsleistung auszahlen lassen:
    - Schulden zurückzahlen
    - Stabilitätsgebühr in MKR entrichten
    - Sicherheitszahlung abheben

Allen CDPs wird ein Risikofaktor zugeordnet, welcher sich mit der Zeit anpasst. Sollte der Risikofaktor einen Schwellenwert überschreiten, so wird der CDP automatisch liquidiert.
Der Wert von 1 DAI ist fest an den Wert von 1 Dollar gekoppelt. Dies führt dazu, dass DAI verglichen mit Cryptowährungen, die an keine bekannte Währung gekoppelt sind, besser dazu geeignet ist, um Geschäfte in der realen Welt durchzuführen. Zur Stabilisierung des DAI-Werts, werden die Gebühren angepasst, die bei der Erstellung von neuen DAI anfallen. [28] Sinkt der Kurs des DAI unter 1 Dollar, so steigen die Gebühren bei der Erstellung von DAI. So muss für die gleiche Menge an DAI mehr

Ether hinterlegt werden und der Preis des DAI steigt normalerweise wieder. Analog dazu sinken die Gebühren, falls der Wert des DAI die 1 Dollar Marke überschreitet. Sollte es zu dem Fall kommen, dass der Wert des DAI unkontrolliert verfällt, so kann die Maker Plattform durch einen Wahlprozess aufgelöst werden. Kommt es zu diesem Prozess, so wird die Erstellung von neuen CDPs eingefroren. Wurde bei der Wahl beschlossen die Maker Plattform aufzulösen, so können DAI wieder zu einem festen Kurs in Ether umgewandelt werden. [28]
In der Zukunft sollen bei der Erstellung von CDPs auch andere Sicherheitsleistungen als Ether möglich sein. Die Modularität von MakerDAO und die Möglichkeit andere Sicherheitsleistungen zu hinterlegen, könnte dazu führen, dass sich aus diesem System eine Art Weltwährung entwickelt. [28]

4.2.2 DApps

Dezentralisierte Anwendungen (DApps) werden, analog zu den dezentralen autonomen Organisationen, nicht von einer zentralen Instanz betrieben und weiterentwickelt, sondern idealerweise von einer Vielzahl von unabhängigen Entwicklern. Dies hat zur Folge, dass die Vorstellungen, wie die Entwicklung der Anwendung voranschreiten soll auch nicht von einem zentralen Gremium entschlossen wird, sondern von der Gemeinschaft.
Hierbei müssen Anwendungen vier Grundeigenschaften besitzen, um sie als dezentralisierte Anwendungen klassifizieren zu können. [29]

1. Eigenschaft - Open Source:

Eine dezentralisierte Anwendung muss über einen öffentlich zugänglichen Quelltext verfügen, der von jedem - auch denjenigen Personen, die nicht an der Entwicklung beteiligt sind - eingesehen, umprogrammiert und frei verwendet werden kann und darf.
Weiterhin muss die Anwendung autonom operieren - d.h. die Anwendung soll sich idealerweise selbstständig den Marktreaktionen, die für sie wichtig sind, anpassen. Wie schon erwähnt, gibt es keine zentrale Instanz, die entscheidet, wie sich die weitere Entwicklung der Anwendung gestaltet. Solche Entscheidungen müssen durch eine Mehrheit aller beteiligten Entwickler entschieden werden.

2. Eigenschaft - Blockchain:

Alle dezentralisierten Anwendungen müssen ihre Daten auf einer Blockchain speichern und profitieren somit von den Sicherheitsaspekten der Blockchain an sich. Dies hat zur Folge, dass die Daten einer Anwendung, welche an sich keine Sicherheitslücken aufweise, nur gehackt werden können, wenn die Blockchain an sich gehackt wurde.

3. Eigenschaft - Kryptografisch verschlüsselte Token:
Kryptografische Token sind Bestandteil der kryptografisch verschlüsselten Blockchain. Sie stellen eine Kopie eines sensiblen Datensatzes - zum Beispiel eine Überweisung von einer Währungseinheit - einer Blockchain dar. Diese Token werden erst zu dem Ledger hinzugefügt, wenn mehrere Miner im Netzwerk die Transaktion bestätigt haben.

4. Eigenschaft - Erzeugung von Token:
Dezentralisierte Anwendungen müssen einen Mechanismus anbieten, um die Token zu generieren, sodass die Miner einen Anreiz haben Transaktion zu bestätigen. Hierfür gibt es die schon besprochenen Konsens-Algorithmen, wie unter anderen *Proof-of-Work* und *Proof-of-Stake*.

Weiterhin kann man dezentralisierte Anwendungen in drei Unterkategorien unterteilen, welche logisch aufeinander aufbauen: [29]

- Typ 1: DApps, die eine eigene Blockchain anbieten
- Typ 2: DApps, die eine Blockchain des Typ 1 verwenden
- Typ 3: DApps, die eine Blockchain des Typ 2 verwenden

Ethereum erfüllt alle vier Eigenschaften einer dezentralisierten Anwendung und kann, da es eine eigene Blockchain betreibt, dem Typ 1 zugeordnet werden. Aragon, bietet einen Service an, um Anwendungen auf der Grundlage von Smart Contracts in Ethereum zu schreiben, [30] und kann somit Typ 2 zugeordnet werden. Jegliche Anwendungen, die von Entwicklern mit der Hilfe von Aragon entwickelt werden, können dem Typ 3 zuordnen.

4.3 Design Muster für Smart Contracts

Aufgrund des hohen Wertes von Ethereum [31] und anderen Cryptowährungen, die Smart Contracts anbieten, sind Sicherheitslücken und Bugs in Smart Contracts meist mit hohen Wertverlusten gekoppelt. Eines der bekanntesten Beispiele hierfür ist der DAO Hack aus dem letzten Kapitel. Zur Vermeidung dieser Verluste gibt es Design Muster, welche Richtlinien vorgeben und die Sicherheit dadurch verbessern sollen. Im Folgenden werden einige dieser Design Muster vorgestellt.

4.3.1 Checks Effects Interaction - Wechselwirkung

In diesem Muster wird beschrieben, wie der Code strukturiert werden soll, um Seiteneffekte und unerwünschtes Verhalten zur Laufzeit zu verhindern. Es kann verwendet werden, um das Reentrance Problem, welches im DAO-Hack ausgenutzt

wurde, zu verhindern. Hierfür soll eine vorgegebene Struktur befolgt werden, die wie folgt definiert ist.

4.3.1.1 Funktionsweise

Zuerst sind alle Überprüfungen durchzuführen, die ein unerwünschtes Verhalten zur Folge haben können. Zu diesen Überprüfungen zählen zum Beispiel die Kontrolle, ob bei einer Überweisung genügend Mittel zur Verfügung stehen oder ob der Funktionsaufruf von der erwarteten Person kommt. [32] Nach den Kontrollen sollen alle Zustandsvariablen angepasst werden, bevor im Anschluss die Interaktion - hierzu zählt auch das Senden von Ether - mit einem anderen Smart Contract erfolgt. Dieser Ablauf sollte eingehalten werden, da die Interaktion mit einem anderen Smart Contract die Kontrolle über den Programmfluss an diesen übergibt, was es diesem ermöglicht schadhaften Code zur Ausführung zu bringen. Ein Beispiel hierfür ist der schon genannte Reentrance Angriff. Dieser ermöglicht es dem Angreifer Code zur Ausführung zu bringen, bevor der erste „normale“ Funktionsaufruf abgeschlossen wurde. Hierbei ist vor allem der low-level Funktionsaufruf *address.call()* als problematisch anzusehen, da hierbei das restliche verfügbare GAS mit übergeben wird und für weitere Funktionsaufrufe auf Seite des Angreifers verwendet werden kann. Bei Überweisungen von Ether sollten die Funktionen *address.transfer()* und *address.send()* bevorzugt werden, da diese das mitgesendete GAS auf 2300 und damit soweit limitieren, dass nur noch ein Event auf Seiten des Empfängers geloggt werden kann. [33]

4.3.1.2 Anwendungsbeispiel

Das vollständige Beispiel für einen verwundbaren Smart Contract ist im Anhang einsehbar, während die Schwachstelle in Abbildung 4.2 und die Behebung der Schwachstelle in Abbildung 4.3, durch die Anwendung des beschriebenen Sicherheitsmusters, einsehbar ist.

```
function get() public
{
  // transfer funds - caller's code is executed can be reentered
  if (!msg.sender.call.value(balances[msg.sender])()) {
    throw;
  }
  // modifiy callers balance - Modification comes too late ↘
      therefore transfer can be executed multiple times
  balances[msg.sender] = 0;
}
```

Abbildung 4.2 – Verwundbarer Auszug eines Smart Contracts

```
function get() public
{
  // Get balance of caller (msg.sender)
  uint256 amount = balances[msg.sender];
  // check if enough funds are available for caller
  if(amount > 0) {
    // set balance of caller to 0
    balances[msg.sender] = 0;
    // transfer the balance to the caller
    msg.sender.transfer(amount);
  }
}
```

Abbildung 4.3 – Anwendung des Checks-Effects-Pattern

4.3.2 Withdrawal Pattern - Abhebemuster

Das Abhebemuster schlägt eine Codestruktur vor, welche es Angreifern erschweren soll, den ausführenden Smart Contract zu blockieren. Hierfür soll statt der intuitiven Art und Weise eine Überweisung abzubilden eine abstraktere Art und Weise angewendet werden, welche im Folgenden näher besprochen wird. [34]

4.3.2.1 Funktionsweise

Die intuitive Art und Weise eine Überweisung mit Hilfe des Wechselwirkungsmusters abzubilden, besteht aus folgenden Schritten:

1. Überprüfen, ob die Überweisung durchgeführt werden kann
2. Kontostand anpassen

3. Vermögenswerte überweisen
4. Rückkehr zum „normalen“ Programmfluss

Hierbei kann ein Angreifer die Anpassung des neuen Zustands verhindern und somit sein Vermögen grundsätzlich auf dem Smart Contract einfrieren und weitere Änderungen verhindern. Dies ist für diesen besonders interessant, falls dem Angreifer ein Nutzen entsteht, falls er zum Beispiel als derjenige eingetragen ist, der die größte Geldmenge im aktuellen Contract hat. Dieser Nutzen könnte zum Beispiel ein Vetorecht bei weiteren Entscheidungen sein.
Das Abhebemuster beugt dieses unerwünschte Verhalten vor, indem die Schritte der intuitiven Lösung wie folgt angepasst werden:

1. Überprüfen, ob die Überweisung durchgeführt werden kann
2. Vermögenswerte für Überweisung vormerken
3. Kontostand anpassen
4. Rückkehr zum „normalen“ Programmfluss
5. Benutzer muss getrennte Funktion zum Überweisen aufrufen

Die wichtigste Änderung des Programmablaufs ist hierbei, dass die Vermögenswerte nur zur Überweisung freigegeben werden und die tatsächliche Überweisung dieser getrennt von den Anpassungen von dem jeweiligen User aufgerufen werden muss. Dies hat zur Folge, dass ein Angreifer nur noch die Abhebefunktion für sich selbst blockieren kann und somit die Funktionsweise des Smart Contracts nicht mehr komplett blockiert werden kann und wichtige, vorher blockierbare, Änderungen nicht mehr verhindert werden können.

4.3.2.2 Anwendungsbeispiel

Im folgenden Beispiel [34] in Abbildung 4.4 wird die angreifbare, intuitive Lösung für eine Überweisung dargestellt. Hierbei kann der Angreifer die Änderungen blockieren, indem er einen Smart Contract verwendet, um diejenige Person im angreifbaren Contract zu werden, welche am meisten Ether besitzt. Die Anpassung derjenigen Person, welche am meisten Ether besitzt, kann relativ einfach blockiert werden. Es muss nur verhindert werden, dass der Smart Contract Ether über die sogenannte Fallback-Funktion empfangen kann, was durch die Anweisung *revert* in der Fallback-Funktion realisiert werden kann.
Dieser Angriff kann durch die Anwendung des Abhebemusters relativ einfach abgefangen werden, was in der folgenden Abbildung 4.5 einzusehen ist.
Ein vollständiges Beispiel kann in der offiziellen Soliditydokumentation [34] nachgelesen werden.

```
1  // Vulnerable Contract
2  contract KingOfTheHill {
3    ...;
4    function becomeRichest() public payable returns(bool) {
5      if(msg.value > mostSent) {
6        // changes can be blocked
7        richest.transfer(mostSent);
8        ...;
9      }
10   }
11 }
12 // Attack Contract
13 contract BlockKingOfTheHill {
14   // fallback-function to block transfer of funds and undo all ↘
         changes
15   function() external payable {
16     revert();
17   }
18   ...;
19 }
```

Abbildung 4.4 – Angreifbare intuitive Umsetzung einer Überweisung[6].

```
1  // Secured Contract
2  contract KingOfTheHill {
3    ...;
4    function becomeRichest() public payable returns(bool) {
5      if(msg.value > mostSent) {
6        // mark funds for withdrawal
7        pendingWithdrawals[richest] += mostSent;
8        ...;
9      }
10   }
11   function withdraw() public {
12     uint256 amount = pendingWithdrawals[msg.sender];
13     pendingWithdrawals[msg.sender] = 0;
14     msg.sender.transfer(amount);
15   }
16 }
```

Abbildung 4.5 – Anwendung des Abhebemusters[7].

[6] Common Patterns - Solidity 0.5.3 documentation
[7] Common Patterns - Solidity 0.5.3 documentation

4.3.3 Mutex

Beschäftigt man sich mit dem Reentrance Problem, so kann man erkennen, dass der rekursive Funktionsaufruf der Abheb-Funktion verhindert werden kann, indem die Funktion grundsätzlich nicht mehrfach und gleichzeitig ausgeführt werden kann. Eine Möglichkeit dies zu verhindern sind die sogenannten Sperren (Mutex Locks).

4.3.3.1 Funktionsweise

Mutex Locks (mutual exclusion locks) sind genau solche Sperren, die, wenn sie richtig angewendet werden, eine vollständige Ausführung des Abschnittes garantieren, bevor die Sperre wieder aufgehoben wird. Trotzdem birgt die Verwendung von Sperren, um unerwünschte Interaktionen zwischen zwei oder mehreren Smart Contracts zu verhindern, natürlich auch die Gefahren, die von der parallelen Programmierung bekannt sind. Bei der Anwendung muss drauf geachtet werden, dass es nicht zu diesen Gefahren (u.a. Deadlocks, Livelocks) kommen kann. [35]

4.3.3.2 Anwendungsbeispiel

In Abbildung 4.6 ist ein Ausschnitt eines Smart Contracts zu sehen, in dem ein Mutex zur Sicherung eines Abschnittes verwendet wird. Ein vollständiges Beispiel befindet sich im Anhang dieser Arbeit.

```
1  // mutex
2  bool private mutex = false;
3  function deposit() payable public
4  {
5    // check if mutex is unlocked
6    require(!mutex);
7    // lock -> execute -> unlock critical code
8    mutex = true;
9    balances[msg.sender] += msg.value;
10   mutex = false;
11 }
```

Abbildung 4.6 – Anwendung eines Mutex zur Sicherung eines Abschnitts

4.3.4 Circuit Breaker - Sicherung

Die elektrische Sicherung unterbricht bei Überspannung oder einem Kurzschluss den Stromfluss im Netzwerk. Das Verhalten dieses Verhaltensmusters wurde analog zu der Funktionsweise der elektrischen Sicherung modelliert, um ähnliche positive Auswirkungen im Programmfluss zu erreichen.

4.3.4.1 Funktionsweise

Die Sicherung blockiert, sobald eine vorgegebene Bedingung eintrifft, die Ausführung der gesicherten Teile des Codes. [36] Dies hat zur Folge, dass im Falle eines Auftretens unerwünschter Programmfehler die Ausführung unterbrochen werden kann und somit die Fehler nicht dazu verwendet werden können, um die eventuell hinterlegten Werte zu entwenden.
Zur Sicherung dieser Werte müssen sich die Anteilseigner auf eine Strategie einigen, wie die Werte aus dem Smart Contract wieder abgezogen und verteilt werden. Idealerweise erfolgt dies automatisch, aber da nicht alle eventuellen Schwachstellen im Voraus abgedeckt werden können, sollte man sich darauf einigen, wie man bzw. wer diesen Ausführungstop manuell auslösen darf.

4.3.4.2 Anwendungsbeispiel

Eine Möglichkeit wäre, alle Funktionen des Smart Contracts bis auf die Abheb-Funktion einzufrieren und diese überweist die hinterlegten Werte auf ein zeitlich eingefrorenes Konto, sodass man sich im Anschluss ohne Zeitdruck weitere Gedanken zu der Verteilung des Geldes machen kann. Dieses Beispiel ist in Abbildung 4.7 auszugsweise und im Anhang vollständig dargestellt.

```
1  contract CircuitBreaker {
2    bool public isStopped = false;
3    // modifier to check if code execution is frozen
4    modifier frozen {
5          require(!isStopped, "execution was frozen");
6          _;
7      }
8      ...;
9    // function gated by freeze modifier
10   function transfer() public payable frozen {
11     ...;
12   }
13   // function gated by enableIfFrozen analog to previous function
14 }
```

Abbildung 4.7 – Anwendung der Sicherung

4.3.5 Speed Bump - Verzögerung

Sobald eine größere Anzahl an Anteilseignern sich dazu entschließen gleichzeitig ihre Vermögenswerte aus einem Smart Contract zu entfernen kann es zu weitreichenden Problemen für die Anteilseigner kommen, die ihre Vermögenswerte nicht abziehen.

Zum Schutz der verbleibenden Anteilseigner und, um die Handlungsfähigkeit des Smart Contracts weiterhin sicherzustellen, müssen Schritte eingeleitet werden, die eine gewisse Zeit benötigen, um voll funktionsfähig zu sein. Hierbei hilft das Speed Bump Muster.

4.3.5.1 Funktionsweise

Funktionen, welche Einschränkungen für die Handlungsfähigkeit des Smart Contracts mit sich bringen, werden absichtlich mit einem Zeitpuffer versehen. Dies kann zum einen erfolgen, um bösartige Angriffe abzufangen, [37] oder, um Schritte einzuleiten, die für ein Fortbestehen des Smart Contracts nötig sind. Hierfür müssen die kritischen Funktionen erkannt und mit einer Zeitverzögerung versehen werden. Zur weiteren Absicherung sollte das Speed Bump Pattern in Verbund mit dem Sicherung Muster kombiniert werden, welches einem die Möglichkeit bieten sollte die bösartigen Aktionen rückgängig zu machen. [37] Das Vorgehen, dass gewisse Aktionen mit einem Zeitpuffer versehen werden, ist aus der Bankenwelt schon bekannt. Möchte man sein Bankkonto aufkündigen und Gebühren vermeiden, so muss man die Kündigungsfrist einhalten - es ist also nicht möglich sein Konto sofort kostenfrei aufzukündigen, man muss eine Zeit lang warten bis man Zugriff auf das gesamte Vermögen hat.

4.3.5.2 Anwendungsbeispiel

Ein Anwendungsbeispiel des Speed Bump Musters ist in Abbildung 4.8 auszugsweise dargestellt. Das vollständige Beispiel ist im Anhang einzusehen.

```
1  contract SpeedBump {
2    ...;
3    // announce msg.sender wants to withdraw money
4      function requestWithdrawal() public {
5          requestedWithdrawalAt[msg.sender] = now;
6      }
7
8    function withdraw() public {
9          // check if msg.sender has waited long enough to withdraw
10         require(requestedWithdrawalAt[msg.sender] >= now + ↘
               waitTime, "did not wait long enough");
11         // get balance of msg.sender and transfer
12     }
13 }
```

Abbildung 4.8 – Anwendung des Speed Bumps

4.3.5.3 Verallgemeinerung - Rate Limit

Das Speed Bump Muster kann in seiner Funktionsweise verallgemeinert werden, sodass der Funktionsaufruf nicht nur für den jeweiligen Anwender gesperrt wird, sondern allgemein für alle Nutzer nicht aufrufbar ist, bevor die vorgegebene Zeitperiode abgelaufen ist. Hierbei muss aber genau überlegt werden, ob eine Zeitsperre sinnvoll ist, da zumindest bei Proof-of-Work Blockchains der Zeitstempel in gewissen Maße von der Person abhängig ist, die den nächsten Block schürft und somit niedrige Zeitspannen nicht sicher - ohne Vertrauen - durchsetzbar sind. Weiterhin könnte zur Vermeidung von vielen gleichzeitigen Funktionsaufrufen ein MUTEX stattdessen eingesetzt werden, ohne dass man von ungenauen Zeitgebern abhängig ist.
Eine Anwendung des Rate Limits kann in Abbildung 4.9 eingesehen werden. In diesem Beispiel wird davon ausgegangen, dass die Zeitgeber genau und zuverlässig arbeiten und nicht manipuliert werden können.

```
1  contract RateLimit {
2    uint256 currentTime = now;
3    // limit frequency of function call
4    modifier isLimited {
5      require(now >= currentTime + 1 minutes);
6      currentTime = now;
7      _;
8    }
9    function withdraw() public isLimited {
10     ...;
11   }
12 }
```

Abbildung 4.9 – Anwendung des Rate Limits

4.3.6 Balance Limit - Saldolimit

Alle Speicherorte, an denen eine hohe Summe Geld gelagert wird, sind attraktive Ziele für Personen, die etwaige Sicherheitslücken ausnutzen und sich dadurch unbefugt Zugang zu den Vermögenswerten verschaffen könnten. Ein einfacher und primitiver Lösungsansatz, um das eigene Projekt unattraktiver für Angreifer zu machen, ist es die maximale Menge an Vermögenswerten zu begrenzen, die verwaltet werden kann.

4.3.6.1 Funktionsweise

Zur Begrenzung der gespeicherten Vermögenswerte legt man intern ein Limit fest, welches nicht überschritten werden darf. Sobald dieses Limit erreicht wurde, lehnen

alle Methoden, mit denen man Geld hinterlegen kann, den Aufruf ab. Hierbei muss darauf geachtet werden, dass dies in einer Art und Weise erfolgt, welche die Vermögenswerte, die beim Methodenaufruf mitgesendet wurden, beim Ablehnen wieder dem ursprünglichen Aufrufer zurücküberweist.

4.3.6.2 Anwendungsbeispiel

Im der folgenden Abbildung - Abbildung 4.10 - ist ein Smart Contract dargestellt, welcher dieses Designmuster implementiert. Es werden hier nur die relevanten Teile des Codes gezeigt, während im Anhang wieder das vollständige Beispiel einzusehen ist.

```
1  contract BalanceLimit {
2      uint256 public limit;
3      ...;
4
5      constructor(uint256 _limit) public {
6          limit = _limit;
7      }
8
9      // deny all transfers over limit
10     function() external payable {
11         require(address(this).balance + msg.value <= limit, ↘
               "contract holds too much ETH");
12         balances[msg.sender] += msg.value;
13     }
14     ...;
15 }
```

Abbildung 4.10 – Anwendung des Balance Limits

4.3.6.3 Verallgemeinerung

Es gibt ein besseres Verfahren, um die maximale Vermögensmenge, welche an einer Stelle gespeichert wird, zu begrenzen, ohne dass man ein internes hartes Limit festlegt und, sobald dieses erreicht wurde, sämtliche Überweisungen ablehnt. Hierfür erstellt man zur Speicherung des Kapitals Subcontracts, welche jeweils nur einen begrenzten Anteil des Gesamtkapitals verwalten. Zur Umsetzung dieses Prinzips eignet sich das Entwurfsmuster Fabrikmethode, welches im Weiteren genauer beschrieben wird.

4.3.7 Factory Method - Fabrikmethode

Die Fabrikmethode ist ein Entwurfsmuster, das ein Interface beschreibt, wie man Unterklassen erstellt, aber die Instanziierung der gewünschten Klasse komplett von der Unterklasse übernommen wird. [38, S.107]

4.3.7.1 Funktionsweise

Bei dem Entwurfsmuster Fabrikmethode gibt es eine genau definierte Rollenverteilung. Zum Einen gibt es den Erzeuger und den konkreten Erzeuger, zum Anderen Produkt und konkretes Produkt. Hierbei implementiert jeweils die konkrete Version das Interface der abstrakten, zu ihr passenden Rolle. Es implementiert also der konkrete Erzeuger das Interface, welches die Erzeugerklasse bereitstellt und dies in einer Art und Weise, sodass der konkrete Erzeuger die konkreten Produkte erstellen kann, welche das Interface des abstrakten Produktes implementieren. Im UML-Diagramm, [39] welches in Abbildung 4.11 zu sehen ist, ist Creator der konkrete Erzeuger, welcher die beiden konkreten Produkte ProductA und ProductB erzeugt, die wiederum das Produktinterface IProduct implementieren. Sobald der Client eines dieser Produkte benötigt, ruft er die Fabrikmethode des Creators (*creator.FactoryMethod()*) auf und die Erstellung des benötigten Produktes wird komplett von der Fabrikmethode abgearbeitet.

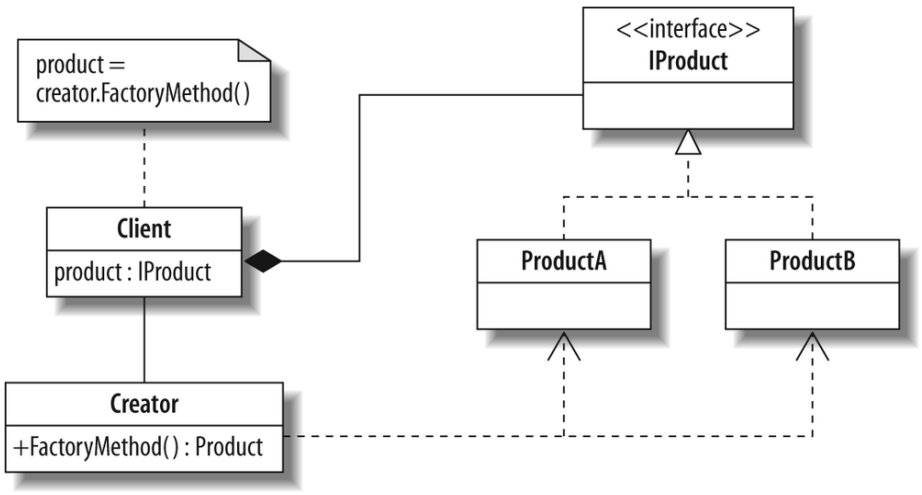


Abbildung 4.11 – Fabrikmethode als UML-Diagramm[8].

[8] C# 3.0 Design Patterns - Judith Bishop

4.3.7.2 Gründe für die Anwendung

Bei der Anwendung dieses Design Musters ergeben sich mehrere Vorteile, welche für den Anwender von besonderer Bedeutung seien können:

- Entkopplung von Aufrufer und Implementierung
- Sicherheitsvorteile durch die Entkopplung
- Verbesserung der Lesbarkeit des Programms

Besonders die Entkopplung von Aufrufer und Implementierung kann bei der Anwendung im Bereich der Smart Contracts einen Vorteil bedeuten, da hierdurch, bei korrekter Umsetzung, die Sicherheit der verwalteten Inhalte gesteigert werden kann. Hierbei können vor allem auch unterschiedliche Inhalte verwaltet werden, ohne, dass der Anwender unterschiedliche Methoden aufrufen muss. Die Fabrik entscheidet anhand eines Parameters, welche Klasse erstellt werden muss. Dies kann bei einer Blockchain-Anwendung zum Beispiel durch die Art des mitgesendeten Zahlungsmittels passieren. Die Fabrik erstellt unterschiedliche Wallets für Ether und für Coins, welche die Anwendung akzeptiert und speichert diese dann auch separat voneinander.
Die Erstellung der separaten Wallets hat zur Wirkung, dass eines der Hauptprobleme, die große Anwendungen, die auf Blockchain beruhen, angegangen werden kann. Große Blockchain-Anwendungen, die für einen Angriff attraktiv sind, speichern meißt hohe Summen an der jeweiligen Währung, auf dem das jeweilige Netzwerk aufbaut. Dies war auch bei „The DAO“ der Fall. Der besprochene Angriff hätte abgeschwächt werden können, wäre die hinterlegte Geldsumme auf mehrere „Subcontracts“ aufgeteilt worden. Dies hätte zur Folge gehabt, dass pro Angriffsvektor nur ein Subcontract angegriffen hätte werden können und somit auch nur das Vermögen aus diesem entwendet werden können. Es wären also mehrere Reentrance-Angriffe nötig gewesen, um bei einer geschickten Aufteilung der Vermögenswerte, die gleiche Geldmenge zu stehlen. Weiterhin ergibt sich durch das Aufteilen in mehrere Unterverträge die Möglichkeit den Zugang zu diesen besser modular aufzubauen. Eine Möglichkeit hierfür ist es, dass jeder Stakeholder nur Zugriff auf den Subcontract hat, auf dem sich auch sein Stake befindet, mit dem er sich in die Organisation eingekauft hat. Hierbei sollte man aber beachten, dass es nicht vorkommen sollte, dass die Stakes auf mehrere Unterverträge aufgeteilt werden.
Ein weiterer positiver Gesichtspunkt, der für die Anwendung der Fabrikmethode spricht, ist die Möglichkeit weitere Funktionalität hinzuzufügen, ohne, dass die bisher erstellten Subcontracts angepasst werden müssen. Im Factory-Contract wird weiterhin nur die Schnittstelle zur Erstellung der Subcontracts angeboten und in Folge dessen auch die erstellten Objekte verwaltet. Die Verwaltung der Objekte erfolgt, indem die Referenz des Objektes gespeichert wird - im Falle von Smart

Contracts die Adresse des erstellten Subcontracts im Netzwerk. Möchte man nun ein weiteres Produkt hinzufügen muss man nur die Fabrik anpassen und das gewünschte Produkt implementieren. An der Implementierung des Clients ändert sich nichts. Dieser kann sich eine Instanz des neuen Produktes von der Fabrikmethode erstellen lassen, indem die Aufrufparameter angepasst werden - im Falle der Verwaltung von unterschiedlichen Tokens im Blockchain-Netzwerk sendet der Benutzer einfach statt der alten Tokens die neuen Tokens bei der Erstellung eines Wallets mit. Die Fabrikmethode entscheidet anhand der Art von Tokens, welche mitgeschickt wurden, welche Art von Wallet zu erstellen ist und gibt die Adresse des Wallets zurück.
Weiterhin verbessert sich die Lesbarkeit des Quellcodes. Die Methoden, welche aufgerufen werden, können vom Anwender des Designmusters so benannt werden, dass genau ersichtlich ist, was an diesem Punkt im Programmfluss vorgeht, da auf die Nutzung des Konstruktors verzichtet werden soll.

4.3.7.3 Anwendungsbeispiel

Im der folgenden Abbildung 4.12 ist die Anwendung der Fabrikmethode zu sehen. Ein vollständiges Beispiel ist bei der Implementierung, welche dieser Arbeit beiliegt einzusehen.

```
1  contract Wallet {
2    // concrete Wallet
3    constructor(address _creator, address _owner,
4          uint256 _unlockDate)
5          public payable {
6      ...;
7    }
8    ...;
9  }
10 // wallet factory
11 contract WalletFactory {
12   ...;
13   function createWallet(address _owner, uint256 _unlockDate)
14     public payable returns (address wallet) {
15     // create new Wallet
16     wallet = new Wallet(msg.sender, _owner, _unlockDate);
17     ...;
18 }
```

Abbildung 4.12 – Anwendung der Fabrikmethode

4.3.8 State - Zustand

Der Zustand ist ein Verhaltensmuster, welches es einem Objekt erlaubt sein Verhalten zu ändern, sobald der interne Zustand wechselt. [38, S.305] Anstatt das gesamte Verhalten effektiv in einem großen Block zu modellieren, trennt man diesen in mehrere Unterklassen auf und bei einer Zustandsänderung wird die verwendete Unterklasse im Smart Contract ausgetauscht.

4.3.8.1 Funktionsweise

Beim Verhaltensmuster Zustand gibt es mindestens drei verschiedene Akteure [38, S.306], die in einer fest vorgeschriebenen Art und Weise miteinander interagieren. Diese Akteure können wie folgt unterteilt werden:

- Kontext
- Zustand
- Konkreter Zustand

Diese Akteure sind auch im folgenden Diagramm [40] in Abbildung 4.13 zu erkennen.

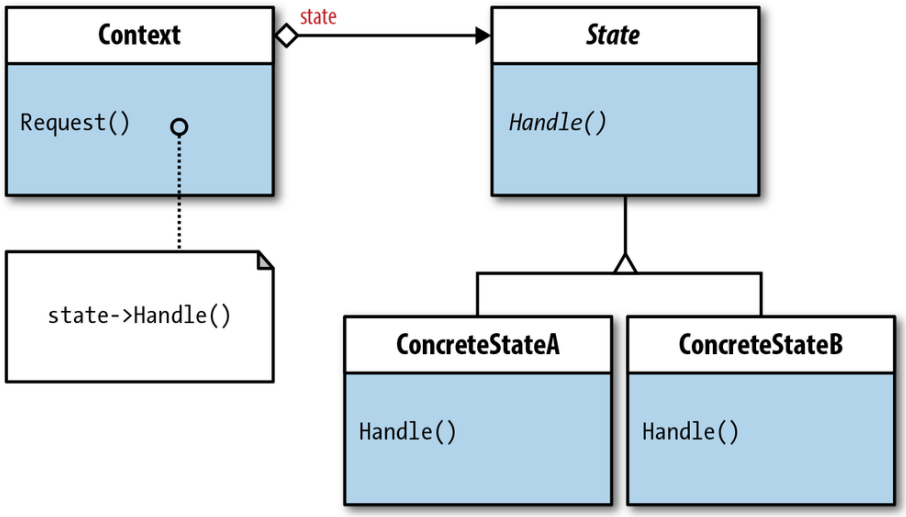


Abbildung 4.13 – Zustand als UML-Diagramm[9].

Hierbei definiert der Kontext das Interface, welches den Zugriff durch die Clients ermöglicht. Zudem wird vom Kontext auch die Instanz des aktuellen konkreten Zustands verwaltet. Diese Instanz ändert sich, sobald es zu einem Zustandsübergang im Programmfluss kommt.

[9]Learning PHP Design Patterns - William Sanders

Der zweite Akteur - der Zustand - definiert ein allgemeines Interface, welches das generelle Verhalten aller erzeugten Zustandsobjekte festlegt. Zudem kann die Schnittstelle ein Standardverhalten anbieten, welches zum Beispiel alle Funktionsaufrufe, welche vom Kontext kommen und nicht einen konkreten Zustand initialisieren, ablehnt. Sobald es beim Programmfluss dazu kommt, dass der Zustand in einen der vorher definierten konkreten Zustande wechselt, wird das Standardverhalten durch die konkrete Implementierung dieses Zustandes überschrieben.
Der dritte Akteur des Verhaltensmusters ist der konkrete Zustand. Der konkrete Zustand muss, falls ein Standardverhalten für die Funktionsaufrufe in der Schnittstelle existiert, nur die Teile des Interfaces implementieren, welche auch tatsächlich für den konkreten Zustand von Bedeutung sind. Existiert kein Standardverhalten im Interface, so muss für alle Funktionen eine Implementierung angeboten werden.
Die allgemeine Funktionsweise des Designmusters kann wie folgt beschrieben werden. Der Client interagiert nur mit dem Kontext, welcher intern den aktuellen Zustand des Programms gespeichert hat. Der Funktionsaufruf führt, abhängig von dem gespeicherten Zustand, zu unterschiedlichen Ergebnissen, welche auch eine Zustandsänderung auslösen können. Hierbei ist wichtig zu erwähnen, dass sich die Aufrufe des Clients nicht aufgrund des internen Zustandes ändern müssen.

4.3.8.2 Gründe für die Anwendung

Die Anwendung dieses Designmusters hat die folgenden zwei Vorteile [38, S.308] zur Folge:

- Trennung des Verhaltens
- Explizite Zustandsübergänge

Bei der Anwendung des Verhaltensmusters Zustand, im klassischen Sinne, wird das gesamte Verhalten eines konkreten Zustandes in einem Objekt - den Zustandsunterklassen - gekapselt, was ein Hinzufügen eines neuen Zustandes vereinfacht, da hierbei nur eine neue Zustandsunterklasse erstellt werden muss. Behandelt man die Zustände des Programms intern über Kontrollstrukturen, so müssen bei einer Änderung alle Bedingungen dieser Kontrollstruktur überprüft und angepasst werden, während beim Zustandsmuster nur die Zustandsübergänge neu definiert werden müssen. Dieser Vorteil lässt sich bei Smart Contracts aber nur schwer bis gar nicht nachbilden.
Weiterhin werden die Zustandsübergange durch die Anwendung des Zustandsmusters explizit. Dies bedeutet, dass die Zustandsübergänge durch verschiedene Objekte, anstatt von internen Variablen, im Kontext repräsentiert werden. Dies hat zur Folge, dass der Zustandsübergang für den Kontext zu einer atomaren Operation wird, da

hierbei nur ein Zustandsobjekt durch ein anderes Zustandsobjekt ausgetauscht werden muss, während bei einer Modellierung des internen Zustandes durch interne Variablen meist mehrere von diesen angepasst werden müssen.

4.3.8.3 Anwendungsbeispiel

Im Folgenden Beispiel [34] in Abbildung 4.14 ist die allgemeine Funktionsweise des Designmusters dargestellt. Hierbei musste die folgende Einschränkungen bezüglich der Funktionalität und Anpassungen bei der Umsetzung gemacht werden. In diesem einfachen Beispiel sind nur lineare Zustandsübergänge möglich - jeder Zustand kann also nur in **einen** anderen Zustand übergehen. Dies ist aber für einfache und kurzweilig bestehende Smart Contracts ausreichend. Bei der folgenden Umsetzung führt aus Kostengründen nicht der konkrete Zustand den gewünschten Codeabschnitt aus, sondern der konkrete Zustand wirkt als eine Art Sperre, welche die Ausführung der konkreten Funktion kontrolliert. Eine vollständige Implementierung ist in der Quelle zu finden und weiterhin kann die Anwendung bei der Implementierung, welche dieser Arbeit beiliegt, eingesehen werden.

```
1  contract StateMachine {
2    enum States { State1, State2 }
3    // current state
4    States public state = States.State1;
5    // state lock
6    modifier atState(States _state) {
7      require(state == _state, "wrong state");
8      _;
9    }
10   // function only called in certain state
11   function bid()
12     public
13     payable
14     atState(States.State1) {
15     ...;
16   }
17 }
```

Abbildung 4.14 – Anwendung des Zustand Musters[10].

4.3.9 Fazit

Allgemein kann von den aufgeführten Design Mustern gesagt werden, dass sie durchaus ihren Nutzen für die sichere Entwicklung von Smart Contracts haben. Bei einigen

[10] Common Patterns - Solidity 0.5.3 documentation

dieser Muster sind jedoch genaue Überlegungen notwendig, wann und ob eine Anwendung sinnvoll ist. Weiterhin stellte sich bei der Recherche zu den klassischen Designmustern, vor allem zu den Designmustern der Gang-Of-Four, heraus, dass der allgemeine Vorteil - die Erweiterbarkeit zur Laufzeit - bei der Umsetzung der Muster in Smart Contracts verloren geht. Im Weiteren wird näher auf die Vor- und Nachteile der vorgestellten Designmuster eingegangen.
Das Wechselwirkungs- und das Abhebe-Muster sind so essentiell für die Sicherheit eines Smart Contracts, dass beide es in die offizielle Dokumentation der verbreiteten Programmiersprache für Smart Contracts - Solidity - geschafft haben und sollten dementsprechend grundsätzlich angewendet werden.
Die Anwendung von Mutex birgt viele Gefahren in sich, welche aus der parallelen Programmierung bekannt sind. Diese Gefahren sind hauptsächlich Deadlocks oder Livelocks. Bei Deadlocks blockiert im schlimmsten Fall die Ausführung des gesamten Smart Contracts, was auch dazu führt, dass auf dem Contract gespeicherte Vermögenswerte nicht mehr abgezogen werden können und hierdurch verloren gehen. Nicht ganz so schlimm sind Livelocks, da hierbei durch ablaufenden Zustandswechsel mit der Zeit das verfügbare Gas verbraucht ist und der Ablauf somit von der Virtual Machine unterbrochen wird. Eine korrekte Ausführung von dem Code, welcher den Livelock ausgelöst hat, ist aber auch meist nicht möglich, was wiederum zu einem Einfrieren der Vermögenswerte führen könnte. Deshalb ist von einer Anwendung von eigenen Mutex Sperren abzuraten und es sollten andere Zugriffskontrollen verwendet werden.
Das Design Muster Sicherung ist besonders zur Absicherung von Codeabschnitten geeignet, welche ein Angreifer ausnutzen könnte, kann aber auch generell als „Panic-Button“ implementiert werden. Die Sicherung sollte aber generell nur in Kombination mit anderen Mustern, welche eine Zugriffskontrolle umsetzen, verwendet werden, da es wenig hilfreich ist den Smart Contract bei einem Angriff oder einem Fehlverhalten nur einzufrieren, ohne etwas gegen das unerwünschte Verhalten ausrichten zu können. Dies ist vor allem wichtig, falls das Einfrieren durch die Sicherung zeitlich begrenzt ist.
Das Muster Verzögerung und die Verallgemeinerung Rate Limit sollten genauso wie die Sperre durch Mutex nur mit Vorsicht angewendet werden, da es hierdurch zum Einen zu einer Einschränkung des Anwenders kommt, welche dieser als unangenehm empfinden kann, und zum Anderen kann genau diese Einschränkung auch dazu führen, dass etwaige Aktionen, welche als Reaktion gegen einen Angriff durchgeführt werden müssen, eventuell auch von dieser zeitlichen Verzögerung betroffen sind. Grundsätzlich sind für eine korrekte Anwendung in einer komplexen Anwendung viele genaue Überlegungen notwendig, bei denen auch die Angriffsvektoren berücksichtigt werden müssen. Um bei der Anwendung der beiden Muster zu Verhindern, dass die Komplexität weiter ansteigt, sollten die zeitlich begrenzten Abschnitte des

Smart Contracts so kurz wie möglich gehalten werden.
Das Muster Saldolimit ist eine einfache Möglichkeit die gespeicherte Vermögensmenge zu limitieren, indem man ab einem bestimmten Betrag die Überweisungen von weiteren Mitteln einfach ablehnt. Diese Limitierung hat aber auch zur Folge, dass die maximale Benutzeranzahl unvorhersehbar limitiert und die Benutzerzahl von Anwendung zu Anwendung verschieden ist. Eine bessere Anwendung dieses Designmusters ist es, dieses mit der Fabrikmethode zu kombinieren und die erstellten Objekte mit dem Saldolimit zu versehen.
Die Fabrikmethode ist besonders gut für Einsatzbereiche geeignet, bei denen man den Zugriff auf die Einzelobjekte beschränken oder das maximale, an einer Stelle gespeicherte, Vermögen limitieren möchte. Der große Vorteil dieses Musters ist es, dass die Verwaltung der Strukturen und der Inhalte voneinander getrennt sind und somit auch dem Verwaltungsvertrag der Zugriff auf diese Inhalte verwehrt werden kann. Weiterhin ermöglicht die Fabrikmethode die Erstellung von mehreren Objekten durch den gleichen Benutzer - er kann also zusätzlich selbst darüber entscheiden, ob er alles an einem Ort oder an verschiedenen Orten speichern möchte. Zusätzlich kann durch die Fabrikmethode eine Möglichkeit geschaffen werden, unterschiedliche Werte zu speichern, ohne dass der Anwender Einblicke über die interne Funktionsweise der Fabrik benötigt - die Fabrik entscheidet, welches Objekt zu erstellen ist und gibt dem Anwender in jedem Fall die gleiche Antwort zurück, über die er auf dieses Objekt zugreifen kann.
Das Muster Zustand sollte nur in Anwendungsfällen, in welchen der Ablauf in diskrete unabhängige Zustände aufgeteilt werden kann, angewendet werden. Bei den Anwendungsfällen ist eine genaue Planung erforderlich, was in den jeweiligen Zuständen ausgeführt werden darf, und was nicht. Weiterhin muss sichergestellt werden, dass es zu keinen endlosen Zyklen bei der Ausführung kommen kann, da hierdurch, wie bei einem Livelock, nicht mehr auf die gespeicherten Werte zugegriffen werden kann. Dieses Designmuster hat aber trotz der, im Vorfeld benötigten, Planung den gravierenden Vorteil, dass man Sicherstellen kann, dass die abgesicherten Funktionen auch nur in dem Ablauf aufgerufen werden können, welcher auch ursprünglich geplant war. Dies schränkt die Angriffsfläche, welche sich einen etwaigen Angreifer bietet, stark ein, während es keine negativen Auswirkungen auf die tägliche Anwendung mit sich zieht.

Kapitel 5

Neuerungen

Die Beliebtheit und die vielseitige Anwendbarkeit von Blockchains führen dazu, dass es zwangsläufig zu Weiterentwicklungen kommt, welche die Performance, Sicherheit und Skalierbarkeit der spezifischen Blockchains verbessern sollen. Im Folgenden werden die geplanten Neuerungen bei Ethereum analysiert.

5.1 Casper

Unter dem Begriff „Casper“ sind die verschiedenen Forschungsausrichtungen und Implementierungen zu PoS bei Ethereum zusammengefasst. Zum Zeitpunkt der Erstellung dieser Arbeit gibt es zwei Forschungsrichtungen: [41]

- Casper the Friendly Finality Gadget (Casper-FFG)
- Casper the Friendly GHOST: Correct-by-Construction (Casper-CBC)

Beide Forschungsrichtungen sollen Mechanismen zur Verfügung stellen, welche es dem Netzwerk ermöglichen schadhafte Elemente bei der Bestätigung von Blöcken im Netzwerk zu bestrafen und somit die Sicherheit des PoS Algorithmus zu gewährleisten. Diese Mechanismen sollen vor allem dafür genutzt werden, um das „Nothing-at-Stake“-Problem zu verhindern und die Verfügbarkeit des Netzwerkes erhöhen.
Der Ablauf des Casper PoS-Algorithmus erfolgt nach dem folgenden Schema: [42]

1. Validatoren setzen ihren Stake auf die möglichen Blöcke
2. Valdatoren werden belohnt, falls der Block bestätigt wird
3. Verlust des kompletten Stakes, falls „Nothing-at-Stake“ oder schadhaftes Verhalten für den Validator festgestellt wird

Dieser Ablauf führt dazu, dass die User, welche die Blöcke bestätigen, wieder ein Risiko eingehen, falls sie sich schadhaft gegenüber dem Netzwerk verhalten.
Weiterhin wird derjenige User bestraft, welcher einen hohen Stake auf einen Block gesetzt hat, anschließend ausgewählt wurde den Block zu bestätigen, aber zu diesem Zeitpunkt offline ist. Diese Strafe soll dazu führen, dass es keine Verzögerungen bei der Bestätigung von Blöcken gibt und die Leistung des Netzwerkes immer auf dem gleichen Niveau bleibt.

5.1.1 Endgültigkeit

Endgültigkeit bedeutet, dass ab diesem Block alle vorhergehenden Transaktionen und der aktuelle Zustand final sind und auch nichts den Zustand wieder auf einen vorherigen Stand zurücksetzen kann. Ruft man sich die Funktionsweise von PoW wieder ins Gedächtnis, so kommt man zu dem Schluss, dass die Endgültigkeit hier schon sichergestellt wird. Dies ist aber nicht der Fall, wie schon der Value Overflow Incident, [43] bei dem die Transaktionen von einem halben Tag im Bitcoinnetzwerk zurückgesetzt wurden, [44] gezeigt hat.
Wirtschaftliche Endgültigkeit tritt für einen Block ein, sobald zum Beispiel 2/3 der berechtigten User eine Wette mit maximalem Einsatz darauf abschließen, dass dieser auch tatsächlich bestätigt werden wird. Die Validatoren, die diese Wette abgeschlossen haben, besitzen nun ein Interesse daran, dass genau dieser Block nicht mehr zurückgesetzt wird oder, dass es keinen neuen Zustand der Blockchain gibt, bei dem dieser Block nicht Teil der Kette ist, da sie sonst ihren kompletten Wetteinsatz verlieren würden.
Hinter wirtschaftlicher Endgültigkeit verbirgt sich nicht die Garantie, dass ein Block nicht mehr zurückgesetzt werden kann, sondern die Garantie, dass diejenigen Stakeholder, welche auf diesen Block gewettet haben, ihr eigenes Kapital verlieren, falls der Block zurückgesetzt wird. [44]
Weiterhin gleicht ein erfolgreicher Angriff bei Casper eher einem Hardfork, als einem Zurücksetzen der Blockchain auf einen alten Zustand. Dies hat zur Folge, dass sich die Nutzer entscheiden können, welcher Version der Kette sie folgen und sind nicht an Entscheidungen von Anderen gebunden.
Grundsätzlich gibt es keine absolute Sicherheit, wenn es um digitale Endgültigkeit geht. Die zusätzlichen Mechanismen, die Casper mit sich bringt, führen aber Risiken für die Validatoren ein, sodass diese etwas zu verlieren haben, falls sie sich entgegen der Interessen des Netzwerkes verhalten.

5.1.2 Casper the Friendly Finality Gadget

Casper-FFG sollte eine Zwischenlösung sein, welche die Umstellung von Ethereum von einer PoW Blockchain auf eine PoS Blockchain vereinfachen und vorbereiten

soll. Hierbei sollte ein PoS Protokoll als zusätzliche Schicht über der PoW Blockchain existieren. Es sollten also die Blöcke immer noch über den PoW Algorithmus Ethash geschürft werden, aber alle 50 Blöcke hätte es einen PoS-Checkpoint gegeben, bei dem die berechtigten Stakeholder über die Endgültigkeit der geschürften Blöcke abstimmen sollten. Inzwischen handelt es sich bei Casper-FFG auch um eine vollständige PoS Lösung. [9, S.322] Dieser neue Lösungsansatz beruht auf der gleichen Grundlage, wie die andere aktuelle Forschungsrichtung von Casper - dem GHOST-Protokoll. [45] Hierbei steht GHOST für Greedy Heaviest Observed Subtree und sollte ursprünglich eingesetzt werden, um den Fall zu behandeln, falls zwei Blöcke bei PoW gleichzeitig gefunden wurden. [46] Bei Casper-FFG wird eine Variante von GHOST eingesetzt, welche die maximale Anzahl an Stimmen aller Blöcke in einer Subchain als Entscheidungsregel verwendet, um zu entscheiden, welche der beiden Subchains nun die „richtige“ Kette ist.

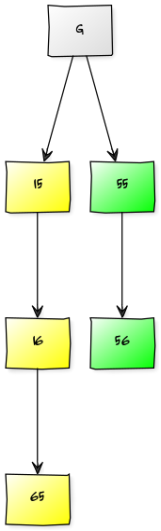


Abbildung 5.1 – Fork Regel Casper-FFG[11].

In Abbildung 5.1 [45] ist die Situation eines Forks abgebildet. Trifft man nun die Entscheidung auf Grundlage der Standartvariante von GHOST (Summe der Einzelgewichte im Ast), so entscheidet man sich für den grün abgebildeten Baum, da der rechte Ast eine höhere Gewichtung (111) als der linke (96) hat, obwohl die linke Variante mehr Blöcke beinhaltet. Trifft man die Entscheidung nach den Regeln die bei Casper-FFG vorgeschrieben sind, so sucht man nach der maximalen Gewichtung im jeweiligen Ast und trifft nun die Entscheidung für den linken Ast ($65 > 56$).
Grundsätzlich liegt bei Casper-FFG der Fokus darauf, eine funktionierende Versi-

[11] Immediate message-driven GHOST as FFG fork choice rule

on einer PoS Blockchain zu implementieren und weitere Features, wie Sharding einzubinden.

5.1.3 Casper the Friendly GHOST

Casper-CBC ist die andere Forschungsrichtung zu PoS bei Ethereum und behandelt allgemeinere Fragestellungen als Casper-FFG und ist weiter von einem tatsächlichen Einsatz entfernt. [47] Dies hat aber auch den Vorteil, dass das Protokoll auch dafür verwendet werden kann einen Konsens zu finden, egal welche Datenstruktur zugrunde liegt und ist dementsprechend nicht nur für den Usecase Blockchain nützlich.
Unter anderen beinhaltet Casper-CBC die Erlaubnis, dass Bestätigungsknoten die Endgültigkeit bestimmen dürfen, ohne dass es zu Vorfällen kommen kann, bei denen zwei Bestätigungsknoten sich für konkurrierende Forks entscheiden. [47]
Im Weiteren wird die Funktionsweise von Casper-CBC am Usecase Blockchain besprochen.
Angenommen man hat eine vorher festgelegte Anzahl von N Validatoren, welche alle die gleiche Stimmgewichtung haben, so kann jeder Validator k im folgenden Intervall Blöcke bestätigen: $k, k+N, k+2N, ...$ [48] Nun führt man noch die Endgültigkeit von Blöcken im Netzwerk ein, um Angriffe auf das System abzufangen. Bei Casper-CBC erreicht man die Endgültigkeit eines Blocks, indem man sich von der Entscheidungsregel, dass die längste Kette die Richtige sein muss abwendet, und das GHOST-Protokoll anwendet.
Die Entscheidungsregel für das GHOST-Protokoll, welche hierbei zur Anwendung kommt, betrachtet die letzte Nachricht von jedem Validator im Netwerk. In Abbildung 5.2 [48] gibt es 5 Validatoren (blau) im Netzwerk, welche die Blöcke bestätigen.

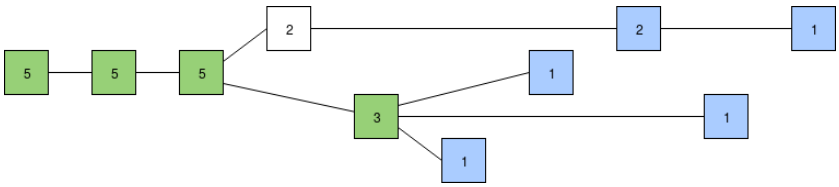


Abbildung 5.2 – Fork Regel Casper-CBC[12].

Wendet man in Abbildung 5.2 die Regel an, dass die längste Kette gewinnt, so entscheidet man sich für den oberen Ast und lässt die letzten Entscheidungen von drei Validatoren außer Acht. Wendet man jedoch die Entscheidungsregel an, dass die letzte Nachricht jedes Validators in die Entscheidung einfließen soll, dann entscheidet man sich für den grün markierten Ast, da der letzte grün markierte Block im unteren Zweig von drei Nachrichten bestätigt wird und der weiß hinterlegte Block im oberen

[12]Latest message-driven GHOST as CBC fork choice rule

Zweig nur von zwei Nachrichten.
Hierbei kristallisiert sich auch der Vorteil der Entscheidungsregel, alle letzten Nachrichten zu betrachten, heraus. Die Entscheidungsregel ermöglicht es, dass alle Kinder eines Elternknotens als Bestätigung für den Elternkonten gezählt werden können und nicht als eine einzige Bestätigung für den gesamten Ast, obwohl die Kinder wiederum miteinander konkurrieren, welches Kind die Kette am Ende fortführen wird. [48] Weiterhin ist die Entscheidung einen Knoten zu bestätigen unnachgiebig, dies bedeutet, dass in einem Angriffsfall den Validatoren nur erlaubt ist auf die „falsche“ Kette zu wechseln, falls sich die Mehrheit unerlaubt dafür entscheidet auf diese zu wechseln. Dieses unerlaubte Wechseln kann jedoch automatisiert bestraft werden. Der Angriffsfall ist in Abbildung 5.3 [48] einzusehen.

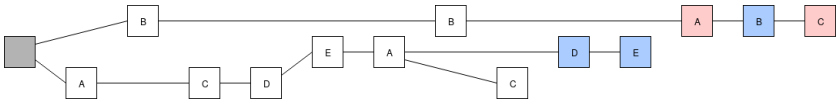


Abbildung 5.3 – Angriff auf LMD GHOST[13].

In Abbildung 5.3 haben sich A und C dazu entschlossen unerlaubt auf die Angriffskette von B (oberer Ast) zu wechseln. Nun können sich D und E, ohne Bestrafung, dazu entscheiden auch auf die Angriffskette von B zu wechseln, da der letzte Block von C (rot) eine Gewichtung von 5 hat, und dies die gleiche Gewichtung ist, die der letzte bestätigte Block im unteren Ast (Block A in weiß) hat. Zur Erkennung von denjenigen Situationen, bei denen sich das Netzwerk auf einen Block festlegt, werden bei Casper-CBC Heuristiken eingesetzt - auch Sicherheitsorakel genannt. Hierbei ist das einfachste Orakel das Clique Orakel. Existiert ein Subset p der Validatoren mit $p = 3/4$, so lässt sich mit Hilfe eines zwei Runden Clique Orakels aussagen, dass eine byzantinische Fehlertoleranz von 25% erreicht werden kann. [48] Diese Fehlertoleranz kann erhöht werden, indem man das Clique Orakel mehr Runden durchführen lässt oder, indem man andere Sicherheitsorakel findet.

5.1.4 Fazit

Vergleicht man Casper-FFG mit Casper-CBC, so kann man sagen, dass Casper-CBC theoretisch die bessere Lösung ist, da bei der Entscheidungsregel bei Casper-FFG die Validatoren, welche einen höheren Stake haben, die Entscheidung zu ihren Gunsten beeinflussen können, sobald sie einen Block bestätigt haben. Andererseits hat Casper-FFG den Vorteil, dass es keine Sicherheitsorakel benötigt, um die Endgültigkeit eines Blockes zu garantieren und ist somit auch einfacher umzusetzen. Dementsprechend ist auch die Entscheidung zuerst Casper-FFG umzusetzen und sich danach auf eine Umsetzung von Casper-CBC zu konzentrieren, nachvollziehbar.

[13] Attack on latest message-driven GHOST

5.2 Skalierbarkeit

Im Moment speichert jeder Knoten im Netzwerk den gesamten Zustand (Kontostände, Quellcode der Smart Contracts, ...) der Blockchain und jeder Knoten bearbeitet alle Transaktionen der User. Dies garantiert ein hohes Maß an Sicherheit, beschränkt aber auch die Skalierbarkeit des Netzwerkes. Der limitierende Faktor für die Anzahl der Transaktionen im Netzwerk, welche bestätigt werden können, ist die Leistung des einzelnen Knotens. Eine Folge der Limitierung ist, dass die Anzahl der Transaktionen pro Sekunde bei Bitcoin ungefähr auf $3-7$ und bei Ethereum ungefähr auf $7-15$ im Moment beschränkt ist. [49]
Es stellt sich nun die berechtigte Frage, wie man die Limitierung aufheben und die Leistung im Netzwerk verbessern könnte. Eine mögliche Lösung ist es die Bearbeitung aller Transaktionen aufzuteilen und nur einen kleinen Teil an Transaktionen durch alle Knoten im Netzwerk bestätigen zu lassen. Hierbei muss aber sichergestellt werden, dass die Sicherheit des Netzwerkes nicht zu stark abnimmt. Diese Lösung ist bei Ethereum unter dem Begriff Sharding bekannt und wird im Folgenden weiter behandelt.

5.2.1 Wichtige Aspekte

Bei den Überlegungen zu Sharding kristallisieren sich folgende Aspekte heraus, welche beachtet werden müssen: [49]

1. Dezentralisierung
2. Skalierbarkeit
3. Sicherheit

Eine Verbesserung im Netzwerk sollte keinen möglichen Knoten davon ausschließen aktiv im Netzwerk zu agieren, während der Durchsatz im Netzwerk grundsätzlich größer sein soll, als der des einzelnen Knotens. Diese Verbesserung soll aber auf eine Art und Weise durchgeführt werden, die die Sicherheit des gesamten Netzwerkes nicht zu stark vermindert.

5.2.2 Naive Lösungsansätze

Ein möglicher Lösungsansatz zur Steigerung des Durchsatz im Netzwerk ist es die verschiedenen Anwendungen, die auf der Blockchain existieren, in Kategorien zu unterteilen, wie zum Beispiel in Glücksspiel, und auf ihre eigenen Subchains aufzuteilen. Dies hat jedoch zur Folge, dass die Sicherheit des gesamten Netzwerkes proportional mit der Anzahl an Minern, die von der Hauptchain auf die Subchain

wechseln, sinkt. [49]
Ein anderer Lösungsansatz ist es die Blockgröße zu erhöhen. Hierbei muss man aber die Hardwarelimitierungen der Knoten wieder in Betracht ziehen, da es kontraproduktiv ist, die Anzahl der möglichen Knoten im Netzwerk dadurch zu limitieren, dass das Schürfen, aufgrund eines hohen Speicherverbrauchs, nur von Supercomputern durchgeführt werden kann. [49]
Weiterhin könnten unterschiedliche Tokenchains erstellt werden, welche auf dem Miningpool von der Hauptchain aufbauen und durch die Hauptchain ihre Sicherheit erreichen. Angenommen, dass alle Schürfer an diesem System teilnehmen, ergibt sich ein höherer Durchsatz, da das System insgesamt mehr Knoten besitzt. Dies hat aber auch zur Folge, dass sich der Speicherverbrauch für jeden Knoten erhöht und dies führt wiederum die Nachteile mit sich, die eine Steigerung der Blockgröße mit sich bringt. [49]

5.2.3 Sharding bei Ethereum

Beim Ansatz zu Sharding, den Ethereum verfolgt, wird der Gesamtzustand des Netzwerks in einzelne Teile (Shards) unterteilt, welche ein voneinander unabhängiges Stück der Transaktionsgeschichte verwalten. [49] Folgende Knoten sind bei dem Shardingansatz bei Ethereum möglich: [49]

Super-Full Knoten:
beinhaltet den vollen Datensatz der PoS-Chain und jeden referenzierten Block der Shards

Top-Level Knoten:
verarbeitet die Blöcke der PoS-Chain aber beinhaltet nicht die Daten der Blöcke der Shards

Einzel-Shard Knoten:
verhält sich wie ein Top-Level Konten, bestätigt aber auch die Dateiköpfe der Datenabgleiche eines Shards, welcher dem Knoten als wichtig erscheint

Leichter Knoten:
bestätigt nur di Blockköpfe von Blöcken der Hauptkette und verarbeitet die Dateiköpfe der Datenabgleiche eines Shards nur, falls der Knoten Zugriff auf den Zustand eines spezifischen Shardes benötigt

Weiterhin gibt es beim Sharding Knoten, die Vergleicher (collators) genannt werden, die Transaktionen auf Shard k akzeptieren und Datenabgleiche (collation) durchführen. [50] Jeder bestätigte Block im Netzwerk, welches Sharding einsetzt, besitzt einen Vergleichskopf (collation head) und der Block ist gültig falls er folgende Eigenschaften erfüllt: [50]

- Die Wurzel von allen Shards stimmt mit der aktuellen Zustandswurzel des dem Block zugehörigen Shards überein
- Alle Transaktionen in allen Shards sind gültig
- Der angegebene Zustand der Ausführung stimmt mit dem Ergebnis der Ausführung der Transaktion über dem angegebenen Vorzustand
- Mindestens 2/3 der für den Shard registrierten Vergleicher unterzeichnet den Datenabgleich

Es steigt der Durchsatz des gesamten Netzwerkes mit der Anzahl der Shards, während jeder Knoten noch aktiv am Netzwerk teilnehmen kann. Es werden also die Aspekte Dezentralisierung und Skalierbarkeit durch diesen Lösungsansatz erfüllt. Weiterhin wird auch der Aspekt der Sicherheit erfüllt, da Blöcke nur gültig sind, falls die Ausführung der Transaktion zum angegebenen Vorzustand mit dem angegebenen Endzustand übereinstimmt und falls genügend Kollatoren den Block auch bestätigen.

5.3 Fazit

Erste theoretische Analysen haben ergeben, dass die Umstellung des PoW-Algorithmus auf einen PoS-Algorithmus in Verbindung mit Sharding einen enormen Leistungszuwachs des Netzwerkes zur Folge hätte. Diese Analyse beruht auf der momentanen technischen Spezifikation, wie Ethereum 2.0 aufgebaut werden soll. Hierbei wird die Blockchain auf 1.024 Shards aufgeteilt, welche jeweils eine Blockzeit von 8 Sekunden haben. Im Moment beträgt die Blockzeit bei PoW 15 Sekunden, was zu 7-15 Transaktionen pro Sekunde führt. Vergleicht man nun die Transaktionen pro Sekunde von PoW mit dem erwarteten theoretischen Wert von ca. 13.000 Transaktionen pro Sekunde, der sich bei der theoretischen Analyse ergibt, so ergibt sich eine Leistungssteigerung um den Faktor 800, falls man davon ausgeht, dass PoW unter optimalen Bedingungen operiert. [51]Dieser Leistungszuwachs bestätigt die Aussage von Vitalik Buterin, dass Ethereum und Blockchain allgemein als der Weltcomputer von Morgen agieren soll und auch kann.
Diese Werte sind vielversprechend, wenn man berücksichtigt, dass das Erstellen von Blöcken bei Ethereum weiterhin so dezentral wie möglich sein soll. Im Gegensatz dazu, erreichte Bitshares auf dem eigenen Testnetz einen Durchsatz von 3.300 Transaktionen pro Sekunde und kann theoretisch, sobald die Hardware der Delegates verbessert wurde, einen Durchsatz von 100.000 Transaktionen pro Sekunde erreichen. [52] Jedoch ist die Erstellung von Blöcken bei Bitshares durch den Einsatz von DPoS auf wenige Delegates zentralisiert und der aktuelle Durchsatz beläuft sich auf ungefähr 12 Transaktionen pro Sekunde.

Kapitel 6

Implementierung

Zur Veranschaulichung einiger Design Muster für Smart Contracts wurde eine dezentralisierte Firma implementiert. Diese Firma besteht aus wenigen angestellten Mitarbeitern, die darüber entscheiden, welche Aufträge sie ausschreiben wollen. In der Implementierung wird davon ausgegangen, dass jeder Mitarbeiter frei über die Ausschreibung von Aufträgen entscheiden kann.
Zur Ausschreibung erstellt der Mitarbeiter ein Bounty, indem er createBounty(uint256, uint256) mit den gewünschten Parametern aufruft und hinterlegt anschließend den vorher festgelegten Betrag durch den Aufruf von depositReward(address), welcher die Bezahlung für den Freelancer darstellt.
Nun können sich die Freelancer entscheiden, den Auftrag durch den Aufruf von claimBounty(address) anzunehmen und zu bearbeiten. Die Lösung kann der Freelancer durch den Aufruf von provideSolution(string, address) hinterlegen.
Sobald die Lösung durch den Freelancer hinterlegt wurde, wird zur Bestätigung des Auftrags eine Abstimmung durch createBountyProposal(address) erstellt, bei dem die Mitarbeiter abstimmen, ob sie die Lösung akzeptieren oder nicht. Die Anzahl der Stimmen, welche für die Wahl benötigt werden, passt sich automatisch mit der Zahl der Mitarbeiter an. Im Falle, dass die Lösung abgelehnt wurde, wird das Bounty zurückgesetzt und ist wieder für andere Freelancer verfügbar. Wird die Lösung jedoch von den Mitarbeitern akzeptiert, so wird die Belohnung für den Freelancer freigegeben und dieser kann sich diese durch den Aufruf von withdraw(address) abholen.
Die Mitarbeiter können die erstellten Aufträge solange durch cancelBounty(address) abbrechen, solange keine Lösung durch den aktuellen Freelancer hinterlegt wurde.

In folgendem Sequenzdiagramm in Abbildung 6.1 ist die Auftragserstellung und die Bearbeitung des Freelancers abgebildet. Hierbei wird die Lösung des Freelancers von der Mehrheit der Mitarbeiter akzeptiert und der Freelancer wird bezahlt.

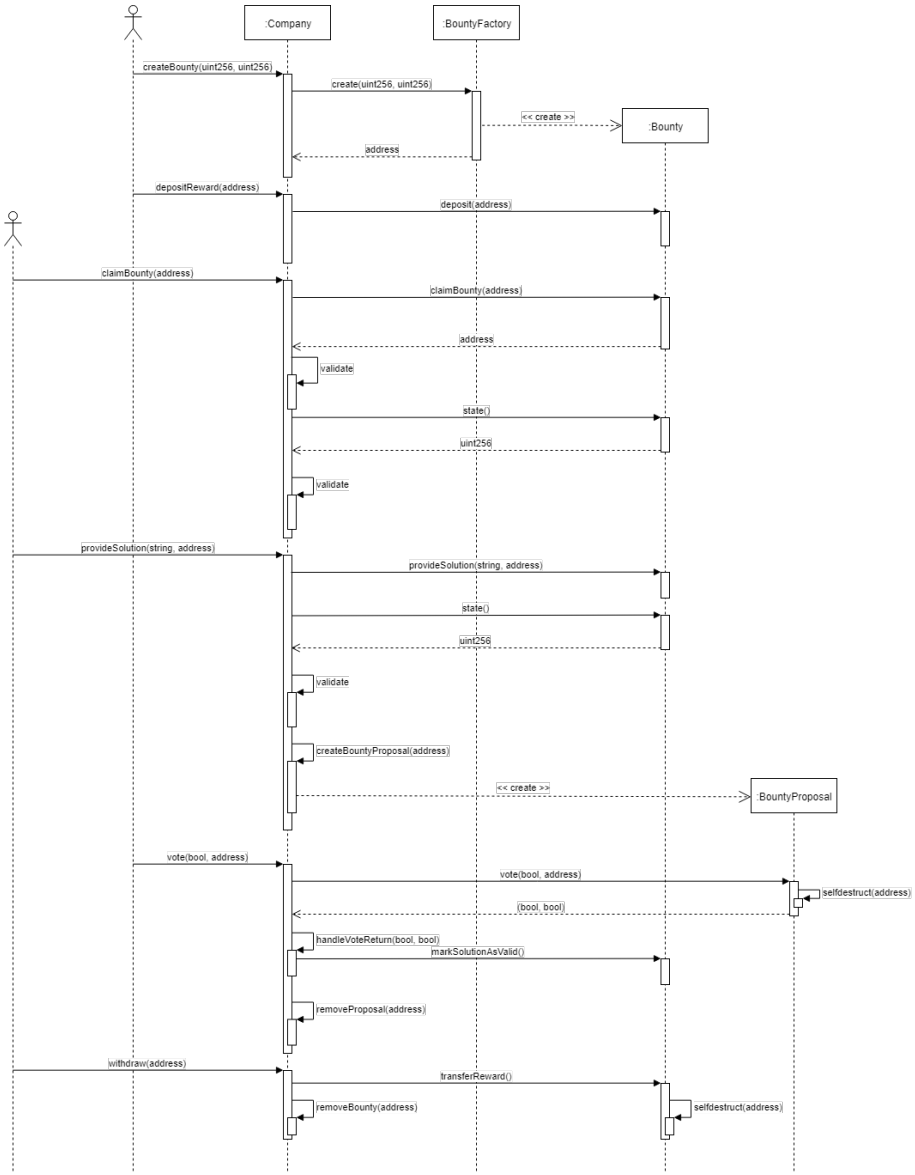


Abbildung 6.1 – Dezentrale Firma

Die Implementierung kann unter folgendem Link eingesehen werden:

https://github.com/dhohner/bachelor_thesis

Kapitel 7

Fazit

Der Fokus der Arbeit lag darin, Lösungsstrategien zu entwickeln, welche bei der Entwicklung von dezentralisierten autonomen Organisationen eingesetzt werden können, um bekannte Sicherheitslücken zu vermeiden und eventuell sogar unbekannte Sicherheitsprobleme abzufangen. Hierfür wurde ein allgemeines Verständnis für die bekannten Sicherheitslücken geschaffen und die verschiedenen Design Muster, welche dazu beitragen diese Sicherheitslücken zu vermeiden, vorgestellt.
Hierbei muss geklärt werden, wie sich Weiterentwicklungen der virtuellen Maschinen, welche die Smart Contracts ausführen, auf die Notwendigkeit der Design Muster auswirkt. Diese Frage stellt sich vor allem beim Abhebemuster, da dieses Muster für den normalen Benutzer eine Einschränkung darstellt, da er mehr Schritte als notwendig durchführen muss, um das Vermögen abzuheben.
Weiterhin sollten Weiterentwicklungen bewertet werden, welche zu einer Leistungsverbesserung bei dem Einsatz einer Blockchain führen könnten.
Bei der Analyse des Istzustandes der bekanntesten Blockchains (Bitcoin, Ethereum) wurde ein Vergleich der drei bekanntesten Konsensalgorithmen durchgeführt, bei dem festgestellt wurde, dass sowohl PoS als auch DPoS gegenüber PoW einen höheren Transaktionsdurchsatz bei geringerem Energieeinsatz erreichen. Während der weiteren Betrachtung von Ethereum wurden festgestellt, ein Umstieg auf einen PoS-Algorithmus (Casper) geplant ist, welcher die besprochenen Probleme von PoS durch die Slashing-Conditions lösen soll. Bei der Analyse der Leistungswerte des ersten Testnetzes, das Sharding umsetzt und auf Casper aufbaut, konnte ein signifikant höherer Transaktionsdurchsatz in der Sekunde festgestellt werden.
Jedoch ist noch unklar, wie sicher der eingesetzte PoS-Algorithmus beim alltäglichen Betrieb ist, und inwiefern der Leistungszuwachs der beim Testnetz beobachtet wurde auf das Hauptnetz übertragbar ist.

Abkürzungsverzeichnis

P2P-Netzwerk	Peer-to-Peer-Netzwerk
Peer	Nutzer
Node	Knoten
PoW	Proof of Work
ASIC	application specific integrated circuit
PoS	Proof of Stake
DPoS	Delegated Proof of Stake
MKR	Maker
Casper-FFG	Casper the Friendly Finality Gadget
Casper-CBC	Casper the Friendly GHOST: Correct-by-Construction

Abbildungsverzeichnis

Literaturverzeichnis

[1] "Bitcoin-Block #0," https://www.blockchain.com/de/btc/block/000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f/, Accessed: 2018-12-28.

[2] "Historical Snapshot - December 23, 2018," https://coinmarketcap.com/historical/20181223/, Accessed: 2018-12-28.

[3] A. M. Antonopoulos, *Mastering Bitcoin - Programming The Open Blockchain*, 2nd ed. O'Reilly, 2017.

[4] "Bitcoin und Co.: So verbreitet sind Kryptowährungen in Deutschland," https://www.handelsblatt.com/finanzen/banken-versicherungen/info-des-bundesfinanzministeriums-sechs-banken-in-deutschland-handeln-mit-kryptowaehrungen/, Accessed: 2018-12-28.

[5] W. Dai, "b-money," 1998, Accessed: 2018-12-29.

[6] Y. Börner, "Was ist Fiatgeld? Einfach erklärt," https://praxistipps.focus.de/was-ist-fiatgeld-einfach-erklaert_101338/, Accessed: 2018-12-29.

[7] A. F. Michler, "Warengeld - Definition | Gabler Wirtschaftslexikon," https://wirtschaftslexikon.gabler.de/definition/warengeld-49163/, Accessed: 2018-12-29.

[8] "Was sind Fiat-Währungen? - Coininvestoren," https://coinvestoren.com/fiat-waehrungen, Accessed: 2018-12-29.

[9] A. M. Antonopoulos and G. Wood, *Mastering Bitcoin - Programming The Open Blockchain*, 1st ed. O'Reilly, 2018.

[10] "The Basics | Zcash," https://z.cash/the-basics/, Accessed: 2019-02-23.

[11] A. Chumbley, K. Moore, and J. Khim, "Merkle Tree | Brilliant Math & Science Wiki," https://brilliant.org/wiki/merkle-tree/, Accessed: 2019-03-09.

[12] Daniel, “What’s A Merkle Tree? Komodo’s Guide To Understanding Merkle Trees,” https://komodoplatform.com/whats-merkle-tree/, Accessed: 2019-03-09.

[13] “What is a Peer to Peer Network (P2P)? | Lisk Academy,” https://lisk.io/academy/blockchain-basics/how-does-blockchain-work/what-is-a-peer-to-peer-network, Accessed: 2019-03-10.

[14] K. Vaidya, “The Byzantine Generals’ Problem - All Things Ledger - Medium,” https://medium.com/all-things-ledger/the-byzantine-generals-problem-168553f31480, Accessed: 2019-02-13.

[15] Z. Witherspoon, “A Hitchhiker’s Guide to Consensus Algorithms - Hacker Noon,” https://hackernoon.com/a-hitchhikers-guide-to-consensus-algorithms-d81aae3eb0e3, Accessed: 2019-02-17.

[16] K. Shirriff, “Bitcoin mining the hard way: the algorithms, protocols, and bytes,” http://www.righto.com/2014/02/bitcoin-mining-hard-way-algorithms.html, Accessed: 2019-02-17.

[17] BTCEcho, “Was ist Proof-of-Work und wie funktioniert der Konsens-Mechanismus?” https://www.btc-echo.de/tutorial/was-ist-proof-of-work-wie-funktioniert-konsens-mechanismus/, Accessed: 2019-02-17.

[18] “Ethash - ethereum/wiki Wiki,” https://github.com/ethereum/wiki/wiki/Ethash, Accessed: 2019-02-17.

[19] “Top Miners over the last 24h - etherchain.org,” https://www.etherchain.org/charts/topMiners, Accessed: 2019-02-25.

[20] “What is Proof of Stake? (PoS) | Lisk Academy,” https://lisk.io/academy/blockchain-basics/how-does-blockchain-work/proof-of-stake, Accessed: 2019-03-10.

[21] BitFury-Group, “Proof of Stake versus Proof of Work,” https://bitfury.com/content/downloads/pos-vs-pow-1.0.2.pdf, Accessed: 2019-03-11.

[22] “Delegated Proof-of-Stake Consensus | BitShares Blockchain,” https://bitshares.org/technology/delegated-proof-of-stake-consensus, Accessed: 2019-03-11.

[23] T. K. Sharma, “Could Blockchain Replace DNS? | Blockchain Council,” https://www.blockchain-council.org/blockchain/blockchain-replace-dns/, Accessed: 2019-03-12.

[24] M. Biederbeck, “Der DAO-Hack: Ein Blockchain-Krimi aus Sachsen | WIRED Germany,” https://www.wired.de/collection/business/wie-aus-dem-hack-des-blockchain-fonds-dao-ein-wirtschaftskrimi-wurde, Accessed: 2019-01-05.

[25] C. Kyriasoglou, “The DAO bricht Crowdfunding-Rekorde und sammelt fast 160 Millionen ein,” https://www.gruenderszene.de/allgemein/ethereum-dao, Accessed: 2019-02-11.

[26] K. Schiller, “Ethereum Classic (ETC) | Übersicht, DAO und Hard Fork,” https://blockchainwelt.de/ethereum-classic-dao-und-hard-fork/#Die_-Hard_Fork_von_Ethereum, Accessed: 2019-02-11.

[27] BTCEcho, “Was sind Soft und Hard Fork?” https://www.btc-echo.de/tutorial/der-fork-guide-was-ist-eine-fork-und-welche-arten-gibt-es-soft-fork-hard-fork-uasf-masf/, Accessed: 2019-02-11.

[28] P. Glazer, “An Overview of MakerDAO - Hacker Noon,” https://hackernoon.com/an-overview-of-makerdao-21e9f34aa1f3, Accessed: 2019-03-17.

[29] K. Schiller, “Was ist eine DApp (dezentralisierte App)? | Blockchainwelt,” https://blockchainwelt.de/dapp-dezentralisierte-app-dapps/, Accessed: 2019-01-06.

[30] B. Garner, “What Is Aragon (ANT) | The Complete Guide - CoinCentral,” https://coincentral.com/aragon-ant-beginners-guide/, Accessed: 2019-01-06.

[31] “Ethereum (ETH) price, charts, market cap, and other metrics | CoinMarketCap,” https://coinmarketcap.com/currencies/ethereum/, Accessed: 2019-01-05.

[32] “Security Considerations - Solidity 0.5.3 documentation,” https://solidity.readthedocs.io/en/develop/security-considerations.html#re-entrancy, Accessed: 2019-01-05.

[33] “Units and Globally Available Variables - Solidity 0.5.3 documentation,” https://solidity.readthedocs.io/en/develop/units-and-global-variables.html?highlight=transfer#address-related, Accessed: 2019-01-07.

[34] “Common Patterns - Solidity 0.5.3 documentation,” https://solidity.readthedocs.io/en/v0.5.3/common-patterns.html, Accessed: 2019-02-09.

[35] P. Humiston, “Smart Contract Attacks [Part 1] - 3 Attacks We Should All Learn From The DAO,” https://hackernoon.com/smart-contract-attacks-part-1-3-attacks-we-should-all-learn-from-the-dao-909ae4483f0a, Accessed: 2019-01-07.

[36] M. Fowler, "CircuitBreaker," https://martinfowler.com/bliki/CircuitBreaker.html, Accessed: 2019-01-07.

[37] M. Mulders, "Smart Contract Safety: Best Practices & Design Patterns - SitePoint," https://www.sitepoint.com/smart-contract-safety-best-practices-design-patterns/, Accessed: 2019-01-26.

[38] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley Pearson Education, 1994.

[39] J. Bishop, "C# 3.0 Design Patterns," https://www.oreilly.com/library/view/c-30-design/9780596527730/, Accessed: 2019-02-04.

[40] W. Sanders, "Learning PHP Design Patterns," https://www.oreilly.com/library/view/learning-php-design/9781449344900/ch10.html, Accessed: 2019-02-04.

[41] "Casper Proof of Stake compendium | ethereum_wiki Wiki," https://github.com/ethereum/wiki/wiki/Casper-Proof-of-Stake-compendium, Accessed: 2019-02-25.

[42] "What is Ethereum Casper Protocol? Crash Course - Blockgeeks," https://blockgeeks.com/guides/ethereum-casper/, Accessed: 2019-02-25.

[43] "Value Overflow Incident - Bitcoin Wiki," https://en.bitcoin.it/wiki/Value_overflow_incident, Accessed: 2019-02-26.

[44] V. Buterin, "On Settlement Finality," https://blog.ethereum.org/2016/05/09/on-settlement-finality/, Accessed: 2019-02-26.

[45] ——, "Immediate message-driven GHOST as FFG fork choice rule - Casper - Ethereum Research," https://ethresear.ch/t/immediate-message-driven-ghost-as-ffg-fork-choice-rule/2561, Accessed: 2019-02-27.

[46] "White Paper - ethereum_wiki Wiki," https://github.com/ethereum/wiki/wiki/White-Paper#modified-ghost-implementation, Accessed: 2019-02-27.

[47] "FAQ - ethereum_cbc-casper Wiki," https://github.com/ethereum/cbc-casper/wiki/FAQ, Accessed: 2019-03-02.

[48] V. Buterin, "A CBC Casper Tutorial," https://vitalik.ca/general/2018/12/05/cbc_casper.html, Accessed: 2019-03-02.

[49] "Sharding FAQs - ethereum_wiki Wiki," https://github.com/ethereum/wiki/wiki/Sharding-FAQs, Accessed: 2019-03-04.

[50] K. Schiller, "Sharding erklärt - Skalierung von Ethereum | Blockchainwelt," https://blockchainwelt.de/ethereum-sharding-skalierung/, Accessed: 2019-03-04.

[51] E. Conner, "Ethereum network throughput under Shasper - Eric Conner - Medium," https://medium.com/@eric.conner/ethereum-network-throughput-under-shasper-390e219ec2b5, Accessed: 2019-03-04.

[52] "Industrial Performance and Scalability | BitShares Blockchain," http://bitshares.org/technology/industrial-performance-scalability/, Accessed: 2019-03-16.

Anhang

Smart Contract mit Reentrance Schwachstelle

```
1  pragma solidity ^0.4.8;
2
3  contract HoneyPot {
4      mapping (address => uint) public balances;
5
6      function HoneyPot() public payable {
7          put();
8      }
9
10     function put() public payable {
11         balances[msg.sender] = msg.value;
12     }
13
14     function get() public {
15         /* solium-disable-next-line security/no-call-value */
16         if (!msg.sender.call.value(balances[msg.sender])()) {
17             /* solium-disable-next-line security/no-throw */
18             throw;
19         }
20         balances[msg.sender] = 0;
21     }
22
23     function() public {
24         /* solium-disable-next-line security/no-throw */
25         throw;
26     }
27 }
```

Smart Contract verwundbar durch Reentrance Angriffe

Smart Contract für Reentrace Angriff

```
pragma solidity ^0.4.8;
/**
    Credits to Gustavo Guimaraes

    https://medium.com/@gus_tavo_guim/reentrancy-attack-on-smart-
    contracts-how-to-identify-the-exploitable-and-an-example-of-
    an-attack-4470a2d8dfe4

    accessed: 2019-01-05

    edited by Daniel Hohner
 */
import "./HoneyPot.sol";

contract HoneyPotCollect {
    // instanciate honeypot contract to enable communication
        with HoneyPot-Contract
    HoneyPot public honeypot;

    function HoneyPotCollect (address _honeypot) public {
        honeypot = HoneyPot(_honeypot);
    }

    function kill () public {
        suicide(msg.sender);
    }

    // put small amount of wei into honeypot contract to
        enable reentrance draining
    function collect() public payable {
        honeypot.put.value(msg.value)();
        honeypot.get();
    }

    // fallback function - gets called by
        msg.sender.call.value(balances[msg.sender])()
    // drains HoneyPot contract
    function () public payable {
        if (honeypot.balance >= msg.value) {
            honeypot.get();
        }
    }
}
```

Reentrance Angriff - Smart Contract

Verhindern von Reentrance

Anwendung von Checks-Effects Interaction

```
pragma solidity ^0.4.8;

contract HoneyPot {
    mapping (address => uint) public balances;

    function HoneyPot() public payable {
        put();
    }

    function put() public payable {
        balances[msg.sender] = msg.value;
    }

    function get() public {
        uint256 amount = balances[msg.sender];
        if (amount > 0) {
          balances[msg.sender] = 0;
          msg.sender.transfer(amount);
        }
    }

    function() public {
        /* solium-disable-next-line security/no-throw */
        throw;
    }
}
```

Anwendung des Checks-Effects-Patterns

Anwendung von Mutex

```
1  pragma solidity ^0.4.8;
2
3  contract HoneyPot {
4      mapping (address => uint) public balances;
5      // Mutex
6      bool public mutex = false;
7
8      function HoneyPot() public payable {
9          put();
10     }
11
12     function put() public payable {
13         balances[msg.sender] = msg.value;
14     }
15
16     function get() public {
17       uint256 amount = balances[msg.sender];
18
19       if (amount > 0) {
20         balances[msg.sender] = 0;
21         // check if mutex is locked - reverts all changes if ↘
               mutex is locked
22         require(!mutex, "transfer is locked");
23         // lock transfer for others
24         mutex = true;
25         // transfer funds
26         msg.sender.transfer(amount);
27       }
28       // unlock mutex
29       mutex = false;
30     }
31
32     function() public {
33         /* solium-disable-next-line security/no-throw */
34         throw;
35     }
36 }
```

Anwendung eines Mutex

Sicherung

```
1  pragma solidity ^0.5.0;
2
3  contract CircuitBreaker {
4      bool public isStopped = false;
5
6      modifier frozen {
7          require(!isStopped, "execution was frozen");
8          _;
9      }
10     modifier enableIfFrozen {
11         require(isStopped, "ony executable if contract is ↘
               frozen");
12         _;
13     }
14
15     mapping(address => uint256) public balances;
16     address private owner;
17
18     constructor() public {
19         owner = msg.sender;
20     }
21
22     function transfer() public payable frozen {
23         balances[msg.sender] = msg.value;
24     }
25
26     function withdraw(uint256 _withdrawAmount) public frozen {
27         uint256 amount = balances[msg.sender];
28         /* transfer _withdrawAmount to msg.sender if possible ↘
               locks contract otherwise */
29         if (_withdrawAmount <= amount) {
30             balances[msg.sender] = amount - _withdrawAmount;
31             msg.sender.transfer(_withdrawAmount);
32         } else {
33             isStopped = true;
34         }
35     }
36
37     function unlockContract() public enableIfFrozen {
38         require(msg.sender == owner, "not the owner");
39         isStopped = false;
40     }
41 }
```

Anwendung des Sicherungsmusters

Speed Bump

```
pragma solidity ^0.5.0;

contract SpeedBump {
    mapping(address => uint256) public balances;
    mapping(address => uint256) public requestedWithdrawalAt;
    uint256 public waitTime = 4 hours;

    function transfer() public payable {
        balances[msg.sender] += msg.value;
    }

    // announce msg.sender wants to withdraw money
    function requestWithdrawal() public {
        requestedWithdrawalAt[msg.sender] = now;
    }

    function withdraw() public {
        // check if msg.sender has waited long enough to withdraw
        require(requestedWithdrawalAt[msg.sender] >= now + ↘
            waitTime, "did not wait long enough");
        // get balance of msg.sender and transfer
        uint256 amount = balances[msg.sender];
        balances[msg.sender] = 0;
        msg.sender.transfer(amount);
    }
}
```

Anwendung des Speed Bumps

Balance Limit

```
1  pragma solidity ^0.5.0;
2
3  contract BalanceLimit {
4      uint256 public limit;
5      mapping(address => uint256) public balances;
6
7      constructor(uint256 _limit) public {
8          limit = _limit;
9      }
10
11     // deny all transfers over limit
12     function() external payable {
13         require(address(this).balance + msg.value <= limit,
               "contract holds too much ETH");
14         balances[msg.sender] += msg.value;
15     }
16
17     function withdraw() public {
18         uint256 amount = balances[msg.sender];
19         balances[msg.sender] = 0;
20         msg.sender.transfer(amount);
21     }
22 }
```

Anwendung des Balance Limits