DEGREE PROJECT IN INFORMATION AND COMMUNICATION TECHNOLOGY,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2017

Trust and verifiable computation for smart contracts in permissionless blockchains

DOMINIK HARZ

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Abstract

Blockchains address trust through cryptography and consensus. Bitcoin is the first digital currency without trusted agents. Ethereum extends this technology by enabling agents on a blockchain, via smart contracts. However, a systemic trust model for smart contracts in blockchains is missing. This thesis describes the ecosystem of smart contracts as an open multi-agent system. A trust model introduces social control through deposits and review agents. Trust-related attributes are quantified in 2,561 smart contracts from GitHub. Smart contracts employ a mean of three variables and functions and one in ten has a security-related issue. Moreover, blockchains restrict computation tasks. Resolving these restrictions while maintaining trust requires verifiable computation. An algorithm for verifiable computation is developed and implemented in Solidity. It uses an arbiter enforcing the algorithm, computation services providing and verifying solutions, and a judge assessing solutions. Experiments are performed with 1000 iterations for one to six verifiers with a cheater prior probability of 30%, 50%, and 70%. The algorithm shows linear complexity for integer multiplication. The verification depends on cheater prior probability and amount of verifiers. In the experiments, six verifiers are sufficient to detect all cheaters for the three prior probabilities.

Keywords: blockchain; smart contract; trust; multi-agent system; verifiable computation

Referat

Blockchains adresserar tillit genom kryptografi och konsensus. Bitcoin är den första digitala valutan utan betrodda agenter. Ethereum utökar denna teknik genom att möjliggöra agenter i blockchain, via smart contracts. En systemisk förtroende modell för smart contracts i blockchains saknas emellertid. Denna avhandling beskriver ekosystemet för smarta kontrakt som ett öppet multi-agent system. En förtroende modell introducerar social kontroll genom inlåning och granskningsagenter. Tillitrelaterade attribut kvantifieras i 2,561 smart contracts från GitHub. De använder ett medelvärde av tre variabler och funktioner med en av tio som har en säkerhetsrelaterad fråga. Dessutom blockchains begränsa beräkningsuppgifter. Att lösa dessa begränsningar samtidigt som du behåller förtroendet kräver kontrollerbar beräkning. En algoritm för verifierbar beräkning utvecklas och implementeras i Solidity. Den använder en arbiter som tillämpar algoritmen, computation services som tillhandahåller och verifierar lösningar och en judge som bedömer lösningar. Experiment utförs med 1000 iterationer för en till sex verifierare med en snyggare sannolikhet för 30 %, 50 % och 70 %. Algoritmen visar linjär komplexitet för heltalsmultiplicering. Verifieringen beror påfuskans tidigare sannolikhet och antal verifierare. I experimenten är sex verifierare tillräckliga för att detektera alla cheaters för de tre tidigare sannolikheterna.

Keywords: blockchain; smart contract; trust; multi-agent system; verifiable computation

Acknowledgments

I am sincerely grateful for having Magnus Boman as my supervisor, who is as excited as me about blockchains and the future of smart contracts. His input and experiences have been an integral contribution to this research.

I like to thank my examiner Mihhail Matskin for his comments and input on my thesis as well as the delightful discussion during the defence. Also, I like to thank my opponents Tharidu Fernando and Daniyal Shahrokhian for their valuable criticism of my thesis. Without the comments during my first seminar at SICS ICT and during my defence, the thesis would have been much more difficult to read and less concise.

Last, I would like to thank my family and friends who have motivated me to follow my dreams. Your understanding and support have made this thesis possible.

Contents

Abbreviations **ix**

1 Introduction **1**
1.1 Background . . . 1
1.2 Problem . . . 3
1.3 Purpose and goal . . . 3
1.4 Delimitations . . . 4
1.5 Methods . . . 4
1.6 Benefits, ethics, and sustainability . . . 5
1.7 Outline . . . 6

2 Extended background **7**
2.1 Blockchain and smart contracts . . . 8
2.1.1 Cryptography . . . 8
2.1.2 Consensus . . . 10
2.1.3 Ledger principle . . . 11
2.1.4 Application logic . . . 12
2.2 Trust . . . 12
2.2.1 Smart contracts as agent systems . . . 13
2.2.2 Models of trust in multi-agent systems . . . 14
2.2.2.1 Prior research . . . 15
2.2.2.2 Trust model comparison . . . 15
2.2.3 Trust model selection . . . 16
2.3 Verifying computations . . . 17

3 Trust models for smart contracts **19**
3.1 Smart contract ecosystem . . . 19
3.2 Trust model for smart contracts . . . 21
3.2.1 Deposits . . . 21
3.2.2 Gossiping . . . 22
3.2.3 Review agents . . . 22
3.3 Smart contract trust analysis . . . 23
3.3.1 Method . . . 24
3.3.2 Results . . . 25

4 Verifiable computation for smart contracts **28**
4.1 Verifying computation . . . 28
4.1.1 Method . . . 28
4.1.2 Actors . . . 29
4.1.3 Assumptions . . . 30
4.1.4 Algorithm . . . 31
4.1.4.1 Verification algorithm . . . 31
4.1.4.2 Dispute resolution . . . 36
4.1.5 Interactions . . . 37
4.2 Implementation and experiments . . . 39
4.2.1 Method . . . 41
4.2.2 Execution time . . . 42
4.2.3 Gas cost . . . 42
4.2.4 Verification . . . 43

5 Discussion **48**
5.1 Trust model evaluation . . . 48
5.1.1 Smart contract ecosystem . . . 48
5.1.2 Trust model for smart contracts . . . 50
5.1.3 Trust analysis for smart contracts . . . 51
5.2 Verifying computation evaluation . . . 54
5.2.1 Actors and assumptions . . . 54
5.2.2 Algorithm . . . 56
5.2.3 Implementation . . . 58
5.2.4 Experiments . . . 59

6 Conclusion **62**
6.1 Summary of findings . . . 62
6.2 Main contributions . . . 63
6.3 Future work . . . 64
6.4 Outlook . . . 65

List of Figures

2.1 Simplified diagram of a block chain with a genesis block (adjusted from [32, p. 33]). 9

3.1 Overview of objects in the smart contract ecosystem and their relations. 20
3.2 Number of repositories and Solidity versions in GitHub dataset. . . . 26
3.3 Total number of declarations in each Solidity contract in GitHub dataset. Median displayed as an orange line. 26
3.4 Number of security-related issues in Solidity contracts in GitHub dataset. 27

4.1 Overview of actors in the verification algorithm. Black actors are required in the verification algorithm, while grey actors are only part of the dispute resolution. 30
4.2 Agent UML package diagram of verification algorithm and dispute resolution with user, arbiter, solver, one verifier, and judge. 32
4.3 Implementation overview of actors in the verification algorithm. Black actors are required in the verification algorithm, while grey actors are only part of the dispute resolution. 39
4.4 Total execution time with different number of verifiers and 30% of computation services providing incorrect solutions with $N = 1000$. . 43
4.5 Total execution time of algorithm with different number of verifiers and percentage of computation services providing incorrect solutions. Each combination of specific number of verifier(s) and percentage of computation services with incorrect solutions with $N = 100$. Median displayed as an orange line. 45
4.6 Total amount of gas used by algorithm with different number of verifiers and percentage of computation services providing incorrect solutions. Each combination of specific number of verifier(s) and percentage of computation services with incorrect solutions with $N = 1000$. Median displayed as an orange line. 46
4.7 Results of computations with different number of verifiers and percentage of computation services providing incorrect solutions. Each combination of specific number of verifier(s) and percentage of computation services with incorrect solutions with $N = 1000$. 47

Abbreviations

ABCI .... Application BlockChain Interface
ABI .... Application Binary Interface
API .... Application Programming Interface
ASIC .... Application Specific Integrated Circuit
DAO .... Decentralised Autonomous Organization
Dapp .... Distributed Application
EVM .... Ethereum Virtual Machine
MAS .... Multi-Agent System
P2P .... Peer-to-Peer
PoW .... Proof of Work
SDG .... Sustainable Development Goal
SEC .... U.S. Securities and Exchange Commission
tx/s .... transactions per second

Chapter 1
Introduction

Trust is a global challenge. According to the Edelman trust barometer [1], the year 2017 marks a crisis in trust. The survey quantifies trust in institutions including government, business, media, and NGOs since 17 years and covers around 33,000 respondents in 28 countries. In 2017, the study found the highest gap of trust between the informed public and the mass population since its beginning. Over the years trust declined in the four above mentioned institutions, with media having the highest loss overall and government receiving the lowest trust overall.

How can computer science research contribute to establishing or restoring trust? Blockchains and their use cases are discussed as "a novel solution to the age-old human problem of trust" [2, p. 1]. In the 2016 article "Trustless Trust", Werbach argues that blockchains can enable a third system of trust apart from formal institutions and private peers. The term "trustless trust" describes a system based on cryptographic measures and distributed consensus algorithms, where the system itself enables trust without any trusted actor within it [2, p. 5]. Moreover, in October 2015, The Economist's lead article describes blockchain as a "machine for creating trust" [3]. The article explains how the cryptocurrency Bitcoin introduced its underlying database (i.e. the Bitcoin blockchain) to allow a transparent and immutable way to store transactions. Bitcoin transactions are stored publicly in a distributed ledger. Different types of blockchains can be used to create a system of trust without intermediaries and thereby changing long-established procedures. Examples range from land registries, ownership of luxury goods, notary services, or autonomous organisations [3].

1.1 Background

The concept of blockchain started in 2008 when Nakamoto introduced a peer-to-peer (P2P) electronic cash system called Bitcoin [4]. Its objectives are to allow payments online without financial intermediaries or single entities controlling the

cash flow. Based on distributed computing and cryptographic principles, the Bitcoin blockchain allows a decentralised, resilient, and transparent way of spending money [4][5]. By publishing all transactions on a publicly, trust is based on transparency and enforcement of rules through the protocol. The idea of the underlying blockchain inspired others to create systems with different approaches [6]. Ethereum adopted the blockchain approach and advanced the concept with introducing Solidity, a Turing-complete programming language to create programs on top of the Ethereum blockchain called smart contracts [7]. They include a set of rules executed within the blockchain. These systems are by design trustless [2] as they ensure every transaction or interaction is fully transparent to everyone and all users are anonymous.

From a high-level perspective, three different types of blockchains are distinguished. Private permissioned blockchains belong to one organisation, which manages read and write access for its users. The organisation trusts itself with the blockchain and the users in it. Consortium-based permissioned blockchains belong to multiple organisations, with configurable read and write access based on the organisation's needs. Trust lies between the organisations and one organisation needs to trust a certain set of organisations for the system to work. Public permissionless blockchains are decentralised, implying they do not belong to particular organisations or individuals. By default, users can join the blockchain freely and receive access based on the blockchain protocol. All three maintain an immutable distributed ledger of transactions [8].

Current blockchain systems are mainly used for financial or cryptocurrency use cases (e.g. Bitcoin) [9]. Other examples include the startup *Provenance*, which uses the blockchain to store and trace the different components and manufacturing steps of a product to create transparency towards potential customers [10]. *Colony* tries to create a new way how people come together to work by flexibly selecting people based on their skills and projects [11].

Trust is a wide area of research including, but not limited to, psychology, social sciences, computer science, and economics [12]. To describe computational trust two main types of models have evolved [13]. Cognitive models are based on underlying beliefs [14], while game-theoretical models depend on utility functions [15]. These models classify the basis of information into:

- Direct experiences [16]
- Witness information [17]
- Sociological information [18]
- Prejudice and bias [13]

In blockchains, cryptography and consensus algorithms enable trust [19]. However, trust with external actors is not part of the model for example described in [8] and [2].

1.2 Problem

Two problems are covered in this research. First, a trust model for smart contract in permissionless blockchains is missing. Related studies and reports on trust in computer science [20][21], agent systems [22], and the semantic web [22][23] exist. However, the trust models used in previous research need to be adjusted for the inherent transparency and trust implications of blockchain systems. The overview of trust needs to account for the smart contract ecosystem, representation of smart contracts as agent systems, and underlying technologies.

Second, permissionless blockchains limit the complexity of computation tasks. When utilising smart contracts, external services can be required to circumvent the computational limitations. These do not offer transparency to the user. Also, their provided solution or correct execution cannot be verified [24]. Hence, the initial benefit of having full transparency over every transaction and enforced trust through a consensus mechanism cannot be guaranteed [25]. Implemented algorithms to verify computations in smart contract systems do as of June 2017 not exist and are a focus of research (e.g. in Ethereum [26]).

1.3 Purpose and goal

The purpose of this thesis is to provide insight into trust in permissionless blockchains. This enables others to understand entities within the system and trust relationships between different agents in- and outside the blockchain. Moreover, limitations set by smart contracts should be overcome by extending trust to interactions with external systems. This intends to solve the scalability restrictions set by permissionless blockchains. Thereby, it broadens the potential use cases for smart contracts.

The goal is to devise a trust model and to contribute an algorithm for verifiable computation. An overview of the smart contract ecosystem is created, a trust model for smart contracts in permissionless blockchains is investigated, and an algorithm to verify computations is developed. The overview's objective is an accurate description of the current, as of June 2017, smart contract ecosystem. Moreover, the aim of the trust model is to describe trust-related attributes of smart contracts and to introduce measures to steer smart contract behaviours. The goal of the algorithm is to allow verifiable computations for smart contracts to overcome computation limitations of blockchains. The report intends to answer the following research questions:

1: Which models of trust can be applied to smart contract ecosystems to reflect public permissionless blockchains?

2: How can computations be verified in permissionless blockchains utilising models of trust?

1.4 Delimitations

Blockchains are a "hyped" topic. Bitcoin started in 2009 and as of April 2017 has a market capitalisation of around $18 billion, while Ethereum started in July 2015 and as of April 2017 has a market capitalisation of around $4 billion [1]. Ethereum has around 272,000 contracts deployed in its main network [2]. This has resulted in the emergence of new concepts, technologies, and approaches in a short time [27][28][29]. Hence, this project is not able to cover a full view of all existing blockchain solutions, smart contract approaches, or technological aspects. Rather, the report focusses strongly on the market capitalization wise largest[3] permissionless blockchain allowing smart contracts (i.e. Ethereum).

There are different theories regarding the legal application and implications of smart contracts ranging from "code is law" [30] to questioning the legality of Bitcoin as a means of payment [31]. Legal, regulatory, and governance-related issues are covered to a limited extended in the conclusion.

Distributed systems research revolves around solving consensus including Byzantine Generals problem and Fischer-Lynch-Paterson impossibility result [32]. This section covers consensus algorithms related to public permissionless blockchain (i.e. Bitcoin and Ethereum) and shortly introduces other approaches in permissioned blockchains. General protocols like Paxos are not elaborated.

Blockchains are P2P systems employing decentralisation as a core part of their functionality. However, from the perspective of an application executing on top of the blockchain the P2P systems behaves like one homogeneous system. Therefore, trust models relating to P2P systems are not further discussed.

1.5 Methods

A range of methods is applied to answer the research questions stated in section 1.3. This section gives an overview of the employed methods while methodological details are presented in the according chapter investigating the research questions.

The first research question is analysed in three steps. First, a definition of ecosystem in relation to smart contracts in public blockchains is required. This definition and

[1] https://coinmarketcap.com (visited on 04/04/2017)
[2] https://etherscan.io/accounts/c (visited on 04/04/2017)
[3] According to https://coinmarketcap.com (visited on 02/20/2017)

the following description of the ecosystem follows a deductive method of basic system theory based on [33]. The protocol definitions of the two largest permissionesless blockchains i.e. Bitcoin [4] and Ethereum [19] are studied to define objects in the ecosystem. Second, the applicability of agent-based trust models for smart contracts is evaluated by deducting their strong and weak notions based on agent theory in [34]. Third, a trust model suitable for smart contracts in permissionless blockchains is qualitatively developed based on a review of existing multi-agent system trust models in [13]. This includes an empirical study of trust-related attributes in a dataset of 2651 smart contracts from a public code repository. The method of the quantitative assessment is described in detail in section 3.3.1.

The second research question is investigated by developing an algorithm for verifiable computations. The development follows a deductive analysis of existing research of verifiable computations in blockchains [35] [36]. A detailed method description including objective of the algorithm is given in section 4.1.1. Moreover, the algorithm is quantitatively assessed to determine the objects set forth in section 4.1.1. The method of assessment is elaborated on section 4.2.1.

1.6 Benefits, ethics, and sustainability

The project intends to help others understand trust implications in blockchain and specifically smart contracts. As described in section 1, blockchains could help establish a new form of trusted organisation and benefit society in the long run. However, incidents with organisations like "The DAO" [4] showed the early stage of blockchain and smart contracts [38]. Hence, a better understanding of the technology is required. Cryptocurrencies and blockchain technologies are perceived as enablers for illegal activity (e.g. currency for buying illegal goods on the "dark net"), but not as facilitators of a decentralised economy [3]. Developing an algorithm for verifying computations allows creating solutions to increase the acceptance of blockchain. However, this allows further use of blockchain technologies for unethical transactions [39].

Based on the United Nations' 17 Sustainable Development Goals (SDGs) [40] the project's goals are contrasted and the applicable SDGs are listed. Blockchain-based services can give access to persons to a currency account without the necessity of a permanent postal address or official ID [4]. Hence, new insurance services for times of financial distress can be established with minimal administrative effort [41]. Understanding trust in these relationships is a crucial factor to enable such services. Moreover, provenance of products can be stored on a distributed ledger to create transparency for customers, intermediaries, sellers, and suppliers. This applies to various assets including, but not limited to, agricultural products, land, energy,

[4] The DAO suffered security flaws, which led to a severe attack on its system and a loss of around 130 million USD [37].

software, and, cryptocurrencies [42]. Also, blockchains and smart contracts enable new types of business innovations (i.e. autonomous organisations) [28]. Blockchain users are identifiable by their hexadecimal public address. Thus, in blockchains, the gender, age, or ethnicity of a person is irrelevant. Through its immutability, blockchains can promote transparent operation of government processes and lower corruption [43]. Trusted smart contracts support the following SDGs: 1 - No poverty; 2 - Zero Hunger; 4 - Quality Education; 5 - Gender Equality; 7 - Affordable and Clean Energy; 8 - Decent Work and Economic Growth; 9 - Industry, Innovation and Infrastructure; 10 - Reduced Inequalities; 11 - Sustainable Cities and Communities; 12 - Responsible Consumption and Production; and 16 - Peace, Justice and Strong Institutions.

Blockchain-based technologies such as Bitcoin and Ethereum are not environment-friendly because of the current consensus algorithm. Ethereum depends on a proof of work approach, where nodes solve mathematical problems to reach the required consensus [4][7]. This takes considerable amount of processing power which is not energy efficient, due to the high electricity consumption. For example, Bitcoin's proof of work consumes energy compared to Ireland's electricity consumption [44]. To lower this, Ethereum is planning to migrate to a proof of stake approach where the money owned by users will decide the stake and the capability to produce new blocks needed to reach consensus. Hence, it will not depend on the energy-inefficient calculations needed in proof of work [45].

1.7 Outline

This report covers the details of the underlying technologies of smart contracts in chapter 2, including an overview of trust models. Chapter 3 selects a model to understand and evaluate trust in smart contract settings. Moreover, a quantitative analysis of existing smart contracts is presented. In chapter 4, ways of providing transparency and trust by means of algorithms are elaborated. Thereafter, an algorithm to prove the correctness of results from computations between agents is introduced. Chapter 5 discusses the model of trust and its application. Also, the algorithm is examined and contrasted with other approaches. The last chapter concludes the report and presents potential future work.

Chapter 2
Extended background

Blockchains introduce a new type of architecture to implement applications. A blockchain-based technical architecture provides a trustless and tamper-proof distributed ledger in a shared environment. It uses multiple computer nodes to replicate this "database of transactions" that enables a *single global truth* about a transaction. Blockchain applications comparable to Bitcoin [4] can be seen as the first generation of blockchains. They are implementing a single or limited number of use cases. Applications like Namecoin[1] and Coloredcoin[2] are building upon existing blockchains (i.e. Bitcoin). They add extra functionality to store data inside the blocks. However, they are limited to the single purpose of the design without providing flexibility.

The second generation addresses those functionalities by providing users with Turing-complete programming languages. Instead of being limited to storing data, applications can execute code to form a generic programmable blockchain. These applications are typically described as smart contracts or chaincodes [46]. A smart contract can function as a decentralised application (Dapp) executed on top of the blockchain platform. This platform is a distributed P2P network and thus, independent from specific servers as long as there are enough servers. The term smart contract, coined by Nick Szabo, expresses the formalisation of electronic commerce in code to execute the terms of a contract [47]. With the introduction of blockchain, the definition of smart contracts was adjusted. Brown describes smart contracts as applications, that react on events, have a specific state, are executed on a distributed ledger, and are able to interact with assets stored on the ledger [48]. Smart contracts are comparable to "autonomous agents" [49].

This chapter covers the basic building blocks of blockchains and smart contracts. Moreover, it introduces trust-related research in computer science with a strong focus on agent systems. Last, methods to establish trust in code including formal verification and game-theoretic approaches are outlined.

[1]https://namecoin.org (visited on 24/05/2017)
[2]http://coloredcoins.org (visited on 24/05/2017)

2.1 Blockchain and smart contracts

Blockchain implementations consists of (1) cryptographic measures to ensure integrity, authenticity, privacy, and identity, (2) a decentralised consensus protocol for agreeing on transactions, (3) the distributed history of all transactions i.e. the distributed ledger, and (4) the application logic i.e. for smart contracts [50]. Among the second generation of blockchains, Ethereum [7], Hyperledger Fabric [51], Tendermint [52], and Ripple [53] are introduced in this section.

Ethereum is a decentralised platform which offers smart contracts through its own blockchain and cryptocurrency called Ether. The Ethereum Virtual Machine (EVM) handles the states and computations of the protocol. Theoretically, the EVM can execute code of arbitrary algorithmic complexity [7]. Using Ethereum, developers can implement smart contracts, which are lines of code in an account that execute automatically when transactions are sent to that account. Because of its trustless nature, the outcome is final and agreed on by all participants [7].

Hyperledger Fabric is a permissioned blockchain architecture intended for customer enterprise networks. It offers a modular architecture, which allows for implementing different smart contracts, cryptographic algorithms, consensus protocols and data storage solutions [46].

Tendermint is a software including a blockchain consensus engine (Tendermint Core) and a generic application interface (Application BlockChain Interface (ABCI)) with the focus on a permissioned architecture. Tendermint Core implements blockchain functionalities including immutable and ordered storage of transactions, while ABCI allows for development of programs in a variety of programming languages [52].

Ripple is a cryptocurrency and a payment system [54]. Ripple has a consensus protocol based on collectively trusted subnetworks within one large network (i.e. permissioned). Thereby, Ripple seeks to achieve a low latency within the network and tolerate Byzantine failures [53].

2.1.1 Cryptography

Blockchains are built on two core cryptographic measures: hash functions and digital signatures [4]. Hash functions are mathematical functions with three properties [32, pp. 23-24]:

- A hash function's input is a string of any size.
- A hash function's output has a fixed size.
- A hash function is efficient to compute (worst-case computational complexity $\mathcal{O}(n)$).

Blockchains require cryptographic hash functions, which have the following three additional properties:

- A hash function H is resistant against collisions implying that it is not feasible to find x and y such that $H(x) = H(y)$ [32, p. 24].
- A hash function H is hiding if it is infeasible to find x given $H(r||x)$ whereby the secret value r is chosen from a probability distribution with high min-entropy [32, p. 27].
- A hash function H is puzzle-friendly if for every possible n-bit output value y it is infeasible to find x in less than 2^n time given $H(k||x) = y$ where k is chosen from a probability distribution with high min-entropy [32, p. 29].

To create a chain of blocks (i.e. the blockchain itself) hash pointers are used as a data structure. Thereby, a linked list is connected by hash pointers identifying the previous block and the hash of the previous data. This allows checking if the previous block has been manipulated since the hash of the block would have changed. Such a structure creates a tamper-evident log since an adversary would need to change every block in the chain to hide tampering with data in one block [32, pp. 33-34]. Proofing membership and non-membership of blocks in a chain are achieved through Merkle trees [4]. Figure 2.1 shows a simplified blockchain with the genesis block (i.e. the first block in the chain).

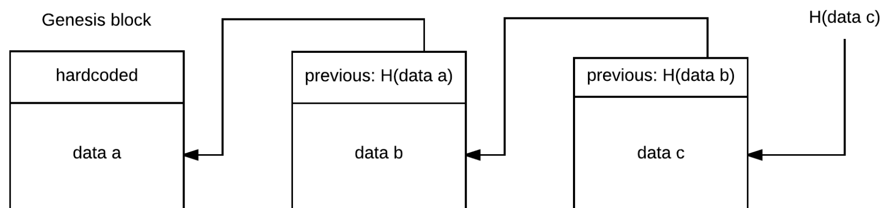


Figure 2.1: Simplified diagram of a block chain with a genesis block (adjusted from [32, p. 33]).

The data stored in each of the blocks are transactions. In Bitcoin, transactions represent the transfer of virtual coins from one participant of the network to another participant [4]. In Ethereum and other blockchains introduced earlier, generic data can be transferred [19][46][52][53]. To verify transactions, digital signatures and a consensus algorithm are used. Digital signatures fulfil two properties. First, signatures are unique to one person implying they cannot be forged but evaluated according to their validity by everyone. Second, signatures are bound to a specific document such that the signature cannot be taken away from one document and sign arbitrary documents without the knowledge of the signer. To build a digital signature scheme, three algorithms are required [32, p. 37].

- An algorithm to create a public pk and a secret (i.e. private) key sk pair based on a key size. With sk, messages are signed and with pk the signature on the message can be verified by anyone having pk.
- A sign algorithm creates a signature sig based on the secret key sk and a message m.
- A verify algorithm evaluates based on the public key pk, the message m, and the signature sig, whether sig is valid for that particular message.

In permissionless blockchains, the public key of the digital signature is often used as the identity of the user. A user of these blockchains creates the identities or so-called addresses and is allowed to create multiple addresses [4][19]. In permissioned blockchains the process of creating identities is controlled by a membership service to authorise new identities [51].

2.1.2 Consensus

Digital signatures are used to verify that a transaction is signed by whoever claims to be the signer. However, this introduces the problem that any one person might send the same coin twice, as it is possible to create valid signatures for both transactions [4]. In centralised systems, a trusted central authority is employed to prevent such double-spending. In a decentralised system participants of the network need to agree on the validity of transactions to prevent double-spending utilising a distributed consensus protocol [4]. A distributed consensus protocol takes into account that in a network with n nodes, an arbitrary number of nodes k might be faulty or malicious. The consensus protocol must ensure that (1) all honest nodes agree on one value (i.e. the chain of transactions) and (2) that this value has to be created by an honest node [32, p. 53].

The Bitcoin consensus algorithm *Proof of Work (PoW)* is based on the idea that the chain with most computational effort is the valid chain. It employs six steps [4].

- First, new transactions are broadcasted to all nodes in the Bitcoin network.
- Second, new transactions are stored into a new block, which is at that point not yet in the agreed blockchain.
- Third, each node (i.e. miner) solves a computational intensive hash puzzle to find a valid hash for the block based on the previous blocks and the current transactions.
- Fourth, once a node has found a valid block hash, it broadcasts this block to all other nodes.

- Fifth, other nodes accept the new block, if all transactions are valid according to their signatures and the coins included are not already spent.
- Sixth, the new block is accepted as part of the blockchain by all nodes, including the new block's hash for creating the next block.

To motivate the miners finding new blocks, they are provided with an incentive of 25 Bitcoins once they find a new block [4].

Ethereum uses a PoW consensus algorithm called Ethash [19]. The algorithm is similar to Bitcoin's, but uses a different PoW function. In Bitcoin's PoW application specific integrated circuits (ASICs) can be used to optimise the computation, which leads to a potential mining centralisation[3]. Therefore, Ethash is optimised for commodity hardware by utilising memory hardness [19]. Performance of Ethash is determined by read and write operations of data in memory, an area where graphic cards are highly optimised. Ethash assumes that specialised ASICs are not able to outperform consumer graphic cards since they are already optimised and graphic card manufacturers are continuously working to improve memory bandwidth (i.e. not only for mining, but also deep learning, computer graphics etc.).

Bitcoin and Ethereum are public permissionless blockchains resulting in the assumption that nodes in the network cannot be trusted and thus, a strong consensus algorithm is required. This affects the network performance in terms of transactions per second (tx/s) and the time a new block is created. In partly decentralised blockchains like Ripple there are restrictions on the nodes, that are able to join the network [53] [55]. In permissioned blockchains such as Hyperledger Fabric and Tendermint even stronger restrictions can be set. Under the assumption that restricted access and authorisation requires a certain level of trust, different consensus algorithms can be deployed. Depending on the use case of the blockchain and the trust between nodes and actors in the system, weaker consensus algorithms can be deployed leading to a decreased latency of the network and a higher throughput in terms of tx/s [46] [56].

2.1.3 Ledger principle

The consensus protocols introduced in the previous section are used to decide upon the state of the distributed ledger. This ledger keeps an immutable record of all accepted transactions [32]. This ledger is in permissionless blockchains accessible to anyone participating in the network and through blockchain explorers even to entities outside of the network. This means everyone is able to see for example which public key owns the most Bitcoins or Ether. Also, each transaction can be

[3]This is happening as of April 2017 in the Bitcoin network, with miners and developers arguing about the use of SegWit or Bitcoin unlimited.

inspected making it possible for participating parties to monitor the progress of their transaction.

To provide an incentive to the miner and prevent unnecessary changes to the ledger, Bitcoin and Ethereum introduce fees on executing transactions [32]. These fees are determined in Bitcoin based on the bytes of the transaction [4] and on the operation cost in Ethereum [19]. In Ethereum the blockchain not only stores transactions but also the code of smart contracts and their state. This means that the state of a smart contract needs to be updated in the same fashion as executing a transaction including fees, consensus, and mining time.

2.1.4 Application logic

In Bitcoin the "application" running on its blockchain is the cryptocurrency itself. The application defines the rules for executing transactions, storing of Bitcoins in accounts, and interactions with others. As described earlier, the second generation of blockchains introduces smart contracts to allow creating arbitrary programs on top of a blockchain. However, this introduces new challenges.

Smart contracts on the Ethereum blockchain are executed by each node participating in the P2P network. As every node has to execute the smart contract, operations are restricted to protect the Ethereum network [19]. The Halting Problem describes a decidability issue in computer science. It is undecidable under certain conditions whether a program terminates or executes infinitely [57]. To circumvent such issues[4], Ethereum introduces a concept to make users "pay" for execution of a smart contract functions. The EVM supports operations as defined in [19]. Each of these operations (i.e. op-codes) have a certain cost referred to as *gas*.

The price a user has to pay to execute a smart contract or conduct a transaction is determined by (1) the gas consumed by the operation i.e. the sum of gas of all op-codes involved in the transaction and (2) the *gas price*, which is expressed as an equivalent to the cryptocurrency within Ethereum *ether*. Before executing a state-changing function or a transaction, the user has to send a certain amount of gas to the function or the transaction. Only if the provided amount of gas is sufficient for the function or transaction to execute, it will successfully terminate. Otherwise, the transaction or function will terminate prematurely. The result of a premature termination depends on the handling of the smart contract function.

2.2 Trust

Section 1.1 describes trust as a wide research area separating cognitive and numerical models [13]. Trust definitions vary [20] and thus, a common understanding of trust

[4]For example, one could use this property to execute a denial of service attack on the network.

and its definition is required. This section provides an overview of smart contracts as autonomous agents and researches trust models applicable to multi-agent systems with respect to smart contracts.

2.2.1 Smart contracts as agent systems

Agents have certain properties separable in weak and strong notions [34, pp. 26-29]. Weak notions include *autonomy*, *pro-activeness*, *reactivity*, and *social ability*.

Autonomy refers to the smart contract ability to operate without a direct intervention of others and include control over their actions and state. In Ethereum the state of smart contracts is maintained on the blockchains, while the actions are coded into the contract itself. These actions can depend on the state of the contract giving it autonomy.

Pro-activeness describes agents' goal-directed behaviour by taking initiative. This is somewhat limited in Ethereum, as smart contracts, as of June 2017, are not capable of initiative behaviour and act on incoming transactions or calls to their functions. However, if one perceives an agent as a collection of multiple different parts, smart contracts might well be extended by external programs triggering such initiatives. Thereby, the limitations set by Ethereum can be circumvented and an agent with pro-active notions can be created.

Reactivity is based on perception of an agent's environment and a timely response to those changes. By design, smart contracts only have access to the state of the blockchain they are operating in, albeit the state of the whole blockchain including every actor, contract, and transaction. Reactivity for state changes in Ethereum is reached via event, transaction, or function implementation. To react to environment changes outside of the blockchain (e.g. executing a function based on changes in stock market prices) requires importing this information to the blockchain via e.g. Oracles [58].

Social ability enables the potential interaction with other agents or humans through a communication language. In Ethereum, users and contracts are identifiable by their public key [19] and interaction is possible through transactions or function calls on smart contracts. Thereby, smart contracts in Ethereum expose an Application Binary Interface (ABI) that defines the rules for executing functions. An ABI is similar to an Application Programming Interface (API). However, it defines interactions on a lower-level, as it includes the data-type for each input and output parameter.

Strong notions include properties such as *beliefs and intentions*, *veracity*, *benevolence*, *rationality*, and *mobility*[34, pp. 31-45]. Since pro-activeness is somewhat limited

in Ethereum smart contracts, these strong properties are presented in a limited aspect.

The two properties *veracity*, which refers to not knowingly communicating false information, and *rationality*, describing the alignment of the agent's actions to its goals, are specific in a blockchain context, especially when employing cryptocurrency. Depending on the incentives an author of a smart contract might develop an agent, which is rational but not truthful, to maximise profits.

2.2.2 Models of trust in multi-agent systems

Multi-agent systems (MAS) describe systems containing multiple agents. In MAS autonomous agents are required to collaborate to achieve their goals [34]. In *open* MAS the intentions of individual agents are unknown [13]. To deal with the uncertainty of agent intentions, three approaches have emerged.

First, security approaches utilise cryptographic measures to guarantee basic properties such as authenticity, integrity, identities, and privacy [13]. Within blockchains, this is mainly achieved through the cryptographic measures introduced in section 2.1.1. These measurements do not provide trust in the content of the messages.

Second, institutional approaches enforce behaviour through a centralised authority. This entity controls agents' actions and can penalise undesired behaviour. It is subjective to the intentions of the central authority [13]. In blockchains, authority is decentralised, which leaves limited institutional control. As described in section 2.1.2 transactions and state of smart contract are accepted based on a consensus protocol. However, governance functions enforcing behaviour not defined in the core protocol do not exist.

Third, social approaches utilise reputation and trust mechanisms to e.g. select partners, punish undesired behaviour, or evaluate different strategies. Agents require a computational model of reputation of trust [13]. In blockchains, transactions are publicly visible in a distributed ledger. However, there is no system of trust implemented in the core protocol of the blockchains, which would rate behaviour according to certain standards.

These three approaches are complementary and can be used to create a system of trust [13]. Trust research and current implementations of Ethereum, Bitcoin, Hyperledger, and other blockchains are primarily focussed on the first two approaches. This allows creating autonomous agents (i.e. smart contracts) on a platform, which enforces these defined trust measurements.

2.2.2.1 Prior research

Trust research in MAS varies in terms of focus on approaches, consideration of information, and other assumptions. In [22] the authors introduce a comparative analysis of existing trust models. They compare models according to their architecture, initial trust, dimension, risk, and reputation. Sabater and Sierra classify models into their paradigm, information sources, visibility, granularity, cheating assumptions, and model type [59]. Pinyol and Sabater-Mir introduce a review of computational trust and reputation models for open MAS [13]. Their classification of models is based on trust, cognitive, procedural, and generality.

Ramchurn et al. focus on the different directions in trust research of MAS [60]. The authors differentiate two levels of trust. First, individual trust levels are researched consisting of socio-cognitive, reputation, and evolutionary and learning models. Second, system level trust is based on trustworthy interaction, reputation, and distributed security mechanisms. Mui et al. introduce a computational model of trust and reputation with a focus on e-commerce systems based on a review of existing literature [18].

Aberer and Despotovic elaborate a trust and reputation model for P2P systems. They argue that previous methods of trust and reputation models require either a central point of truth or global knowledge. This approach and model would not scale to P2P systems, which have unknown agents and numerous participants [17]. Norms as a basis for decision making of agents are introduced by Boman in [61]. The agent's ability to decide on a course of action is based on probability, utility, credibility, and reliability. Thereby, norms are used as global constraints to exclude certain courses of action.

2.2.2.2 Trust model comparison

Blockchains offer a basic set of enforced trust in the system itself by cryptographic measures, consensus, and ledger principle. This requires a detailed comparison of trust models covering the interaction of agents. The method in [59] offers a detailed way of comparison, while [13] compares 25 trust models using this method. The dimensions of the comparison are defined in [59].

Paradigm type classifies the trust model into a numerical or cognitive model. On blockchains agents interact with each other. In Ethereum and other permissionless blockchains measurable values can be transferred (i.e. cryptocurrency). This favours numerical models employing game theory.

Information sources are direct experiences including direct interactions and direct observations. Witness information utilises information provided by other agents. Sociological information refers to relationships between other agents. Last, prejudice or stereotypes is an information resource usable in the absence of other information.

In a permissionless blockchain, the ledger allows to transparently view information from direct interactions, witness information (i.e. interactions from others with one particular agent), and sociological information at the level of transactions. From within the blockchain, the prejudice is that no actor in the system can be trusted.

Visibility is either global or specific to certain agents. In a permissionless blockchain, such as Bitcoin and Ethereum, visibility is global[5].

Granularity of a model evaluates its context dependency for single or multiple cases. For example, a model can consider single interactions or multiple interactions of specific agents to reach reputation scores.

Cheating behaviour is categorised into three different levels, from not considering cheaters, to hiding or biasing information, and to outright lying. As the incentives to cheat in permissionless blockchains can be high[6], the trust model for a permissionless blockchain needs to consider cheating. Last, [13] sorts the compared models into trust and reputation categories.

2.2.3 Trust model selection

A differentiation in existing trust models is the assumption of trust within the network of agents acting on the blockchain. In permissionless blockchains, no trust exists between smart contracts or users since anyone can openly participate and create contracts. There exists further no direct link between a public key and a person in the blockchain. In permissioned blockchains, a certain level of trust between the interacting parties occurs, as they need to be authorised by some entity to interact on the blockchain. Thus, their public key is linked to an identity in a broad sense (e.g. company profile, social media profile, national ID card). This subsection details how trust in a permissionless blockchain can be quantified.

From the 25 models covered in [13], five consider global visibility and nine consider cheaters. The overlap of those models leaves one considering global visibility and cheaters. The model proposed by Rasmusson and Jansson focusses on reputation of actors in electronic markets [63]. The core idea is to use incentives to encourage truthful behaviour of agents in the system by social control. Social control implies that actors in the network are responsible for enforcing secure interactions instead of using an external or global authority. The authors describe two different fraudulent behaviours in detail. *"Con-mania"* refers to a behaviour where an adversary utilises the high rate of transactions in electronic markets and thus can benefit from multiple small gains. *Monopolies* are fast established if the majority of actors use the same source of information.

[5] However, Zcash is a permissionless blockchain employing zero knowledge proofs to reduce the visibility only to the actors involved in a transaction [62].
[6] As seen in The DAO [38].

Both cases are simulated in their research and three different methods of social control are evaluated to tackle these issues. First, promotions describe new agents in the system that offer to provide a deposit for the usage of their service. As they are lacking reputation in the system, they provide the service in combination with the deposit to build up reputation and offer the user of the service a guarantee to receive their money back.

Second, gossip and rumours are discussed to communicate experience with agents. However, this introduces issues on what should be communicated and who would actually gain from this gossiping. Third, reputation and reviewing can be a basis to trust other agents based on independent review agents, which would collect experience with agents offering certain services and provide this reputation information to others. This can also be applied to the agents' own experiences without involving third reviewing agents.

2.3 Verifying computations

A key issue in blockchain development is scalability of computation. The limited computation capability of smart contracts running on top of a blockchain like Ethereum results from the design and the trade-off of having a decentralised P2P network with a consensus protocol. As described in section 2.1.4 Ethereum uses gas to tackle the Halting Problem. While the gas cost and the associated gas price prevent functions from executing indefinitely, it also puts a high price on functions with a time-complexity higher than $\mathcal{O}(n)$. Thus, functions written in smart contract on top of the EVM are limited or are associated with a steep price to execute.

There are two approaches in the blockchain space trying to tackle this issue: state channels and verifiable computation off-chain. State channels primarily aim at limiting the amounts of transactions that are stored on the blockchain to lower the latency and fees involved [64]. However, this is not helpful to execute arbitrary computations. Verifiable computation aims to execute the computation outside the blockchain and get verifiable results back without the need to re-execute the computation.

Executing and verifying arbitrary computations without re-executing them is a research field with two different focus points: formal methods based on mathematical proofs [65] and game-theoretical methods [66] [35]. In [67] Walfish and Blumberg give a comprehensive overview of the current state of verifiable computations. The mathematical proof systems as Zaatar, Pinocchio, Ginger, and TinyRAM, are, as of June 2017, near practical to use. They require a comparable high number of running computational instances and the proof systems are highly complex [35]. As these systems are "near practicality" [65], they are not further considered as part of this thesis.

The second approach of verifiable computations uses game-theoretical models on top of the blockchain. Zyskind et al. introduce privacy-preserving computations supported by blockchains in [66]. Teutsch and Reitwießner propose a scalable verification approach for blockchains [35]. The idea in both concepts is to enforce the rules of the protocol by automated contracts on the blockchain, provide incentives to honest parties, and penalise dishonest parties. The parties in both protocols can be categorised into computational services, verifiers, and judges.

Chapter 3
3 Trust models for smart contracts

With the introduction of smart contracts, a new ecosystem evolved [2]. This ecosystem consists of entities including, but not limited to, developers, entrepreneurs, investors, and large organisations distributed around the globe [28]. In this chapter, the smart contract ecosystem is defined. Based on the trust models in section 2.2 a model suitable for smart contracts is introduced. The entities in the smart contract ecosystem are described within the trust model including their trust related attributes. Last, a quantitative analysis of smart contracts hosted on GitHub is conducted.

3.1 Smart contract ecosystem

Although the term "ecosystem" is used in cryptocurrency, blockchain, and smart contract related literature (e.g. [2], [8], and [32]), no common definition of this ecosystem exists. The Oxford dictionary defines an ecosystem as a "complex network or interconnected system" [1]. This definition requires further specification of "networks" and "systems". Describing these terms is subject to multi-disciplinary research in e.g. network and system theory, which includes mathematical models and a variety of approaches [33]. However, the intention here is to give the reader an elementary overview of the smart contract ecosystem with a relation to trust, independent of specific research areas and use cases. Thus, the smart contract ecosystem is depicted by four basic components of a system: objects, attributes, internal, and external relationships [33]. The following describes the objects in the ecosystem and their relations as displayed in figure 3.1. Attributes related to trust for smart contracts are covered in section 3.2.

The core of the system is defined as the *blockchain platforms* enabling smart contracts (i.e. EVM, Hyperledger Fabric) and all direct relations to other objects. The blockchain platform is subject to its *blockchain protocol* (i.e. Ethereum, Hyperledger), which specifies implementations on a technical level. This includes for example the

[1] https://en.oxforddictionaries.com/definition/ecosystem (visited on 08/03/2017)

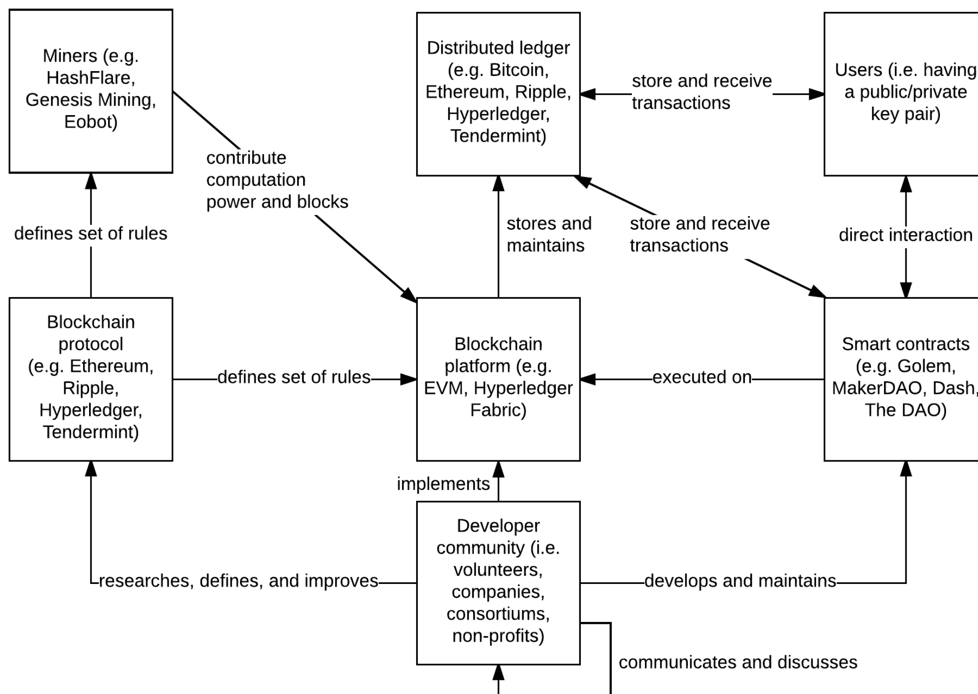


Figure 3.1: Overview of objects in the smart contract ecosystem and their relations.

employed consensus algorithm, the cryptographic hash functions, or the incentives given to miners (if any). Also, the blockchain platform stores and maintains the *distributed ledger*. Within the distributed ledger, transactions are stored as described in section 2.1.3. These transactions are sent and received by *users* and *smart contracts*. Users and smart contracts are characterised by having a public key (i.e. address) in the blockchain. Their interaction is achieved indirectly through transactions stored in the distributed ledger or directly through function calls of smart contracts. Direct interactions between those two have no impact on the state of the distributed ledger. Thus, any state changing interactions like sending currency, changing state variables in a smart contract, or creating new contracts, have to be conducted via transactions [4][7]. Smart contracts can interact through function calls via their ABI. Single smart contracts or multiple smart contracts together can act as decentralised autonomous organisations by encoding the rules of interaction for the organisation's inner and outer relationships (for example The DAO, MakerDAO, Golem, Augur). Next, a *developer community* implements the blockchain platform, develops and maintains smart contracts, as well as researches, defines, and improves the blockchain protocol. This community consists of volunteers, companies, consortium, not-for-profit organisations and other organisations. *Full nodes* store the distributed ledger and validate new blocks in the chain pro bono. *Miners* provide computational power

and new blocks by solving computational puzzles to extend the distributed ledger and maintain the blockchain platform. The aforementioned objects depict the core system of smart contracts.

External relationships outside this core system exist to multiple other objects. *Regulators* like the U.S. Securities and Exchange Commission (SEC) enforce laws and regulations mostly concerning cryptocurrency aspects of blockchains. *Investors* promote the development and expansion of blockchain-based business models and technologies by providing funding and support. *Central banks* such as the European Central Bank and *legislators* such as the EU parliament or US congress create regulations or laws governing blockchain platforms. *Wallet services* help users secure their cryptocurrencies and aim to provide an entrance into the cryptocurrency world. *Exchanges* like Kraken or Coinbase allow exchanging fiat and cryptocurrencies for private and professional customers. *Explorers* like Etherscan provide insight into transactions, accounts, and contracts (i.e. the state of the distributed ledger) through web portals.

3.2 Trust model for smart contracts

From the 25 trust models found, the model by Rasmusson and Jansson covers information sources, visibility, granularity, and cheating behaviour comparable to their manifestation in permissionless blockchains and smart contracts. The model is focussed on electronic markets and offers an abstract implementation of these social controls. It needs to be extended and detailed by utilising further models to fit permissionless blockchains.

3.2.1 Deposits

First, deposits can be used to establish a level of trust between new and existing agents. This approach is taken by Kumersan and Bentov for agents to participate in an incentive-based protocol to provide correct (i.e. verifiable) computations [68]. This approach is adapted by [66] for a privacy-preserving computation protocol and by [35] for verifiable computations on Ethereum. Each agent that wants to participate in these protocols is not trusted by default. This distrust results from the openness of the blockchain platform (i.e. no central authority gives access) and the direct accessibility of cryptocurrency.

Assuming a rational agent, there is a motivation to e.g. breaking privacy or allowing incorrect solutions if this optimises its own utility function. To provide a certain level of trust new agents have to deposit a certain cryptocurrency value for participation. In the aforementioned protocols, this deposit is returned when an agent decides to stop participating. However, dishonest or corrupt agents can be penalised by either destroying their deposit or distributing it to honest agents.

3.2.2 Gossiping

Second, gossiping can be used to communicate experiences with other agents in a P2P fashion and thereby establish trust or reputation scores. In the core protocol of Bitcoin or Ethereum gossiping is the basis for propagating new transactions and subsequently validating blocks [69]. A similar approach can be taken for smart contracts, whereby agents could exchange knowledge or experiences of other agents. In [70] a decentralised reputation management system is introduced. Reputation of an agent is based on its interaction with other agents, whereby agents mutually need to sign a transaction if they are satisfied with the interaction. Over time, an agent collects these signed transactions to build up its reputation. However, this model is prone to colluding agents boosting their reputations. Can and Bhargava introduce SORT, a distributed trust model to evaluate trust of agents based on past interactions and recommendations [71]. Similar, Zhao and Li propose VectorTrust in [72] to calculate trust scores based on distributed algorithms.

Smart contracts could utilise these approaches to quantify trust between agents and propagate information about cheaters through gossiping. However, they assume certain maximum amounts of malicious agents in the system and their detection rate of malicious behaviour is correlated with the assumptions. Moreover, VectorTrust's malicious behaviour detection rate depends on the complexity of the network.

3.2.3 Review agents

Third, trust can be implemented by relying on independent review agents. In [73], FIRE is introduced calculating a reputation score based on interactions, role-based relationships, witness information and references, which the agent itself provides. Jakubowski, Venkatesan, and Yacobi develop a quantitative trust model that results in groups of agents with maximised trust and minimised trust towards agents outside the group based on local and transitive trust [74]. Cerutti, Toniolo, Oren, et al. present a model for computation of trust based on the opinion of trustworthiness, confidence, and a combination thereof [75].

Both, gossiping and review agents, are subject to detection rate issues. As a purely rational agent might be considered malicious in these protocols, their implementation needs to consider the context of the trust or reputation rating. As of 22 April 2017, a total of 295,394 smart contracts are deployed in the Ethereum main network [2]. Hence, quantifying trust between agents (i.e. smart contracts) is complex and subject to attacks on the model due to imprecision of models and the complexity of the network. However, as the blockchain offers transparency it is possible to inspect transaction data to draw conclusions on the behaviour of users of a blockchain as

[2] https://etherscan.io/accounts/c (visited on 22/04/2017)

presented in [5]. Additionally, information sources from outside the blockchain can be used to provide insight into agents behaviour. This external information can be used to link users in the blockchain to other data available on the Web or other sources.

The trust attributes presented are partly applicable to permissionless blockchains. Deposits or financial incentives offer a method of promoting and penalising agent behaviour while no prior trust is required assuming the guarantees provided by the blockchain platform (i.e. validity of transactions, immutable state). Therefore, protocols on the blockchain should consider a game-theoretic setting where the dominant strategy of each agent equals the desired agent behaviour in the protocol's context.

3.3 Smart contract trust analysis

Applying gossiping in P2P networks and implementing review agents is complex. However, a quantitative approach to measure complexity and security related issues of smart contracts can be established by reviewing their source code.

To analyse smart contracts, a suitable dataset of such contracts needs to be established. There are two options to achieve this: On the one hand, the compiled code of a smart contract can be inspected from the Ethereum blockchain. On the other hand, the source code of smart contracts hosted at a public code repository such as GitHub can be reviewed. In Ethereum, addresses (i.e. the public key) are used to identify users and contracts. However, these addresses are not different in any kind of way, thus the only way to separate "regular addresses" from "contract addresses" is to check if the code field of the account has any content greater than zero [76]. The second approach delivers directly the source code enabling analysis of the functions inside the contract. However, those contracts might actually not be in use on the Ethereum blockchain.

The dataset used hereafter is compiled from public GitHub repositories[3]. GitHub offers an API to search for specific code files[4]. As described in section 1.4 the focus of this research centres on Ethereum.

The analysis' objective is to determine attributes of smart contracts in the dataset. The dataset contains the source code of smart contracts, hence behaviour of the contracts cannot be analysed as that would require the inspection of the distributed ledger and transactions issued by smart contracts. Rather, characteristics and complexity of smart contracts are assessed to give an indication of the development

[3]GitHub is in terms of users and number of projects the largest source code hosting service. See https://en.wikipedia.org/wiki/Comparison_of_source_code_hosting_facilities (visited on 05/06/2017) and https://github.com (visited on 08/03/2017)

[4]https://help.github.com/articles/searching-code/ (visited on 08/03/2017)

status. Complexity of code is measured based on the number of declarations in each smart contract. Moreover, coding best practices in relation to security of contracts is assessed through automated code analysis through the Oyente tool [77].

3.3.1 Method

Ethereum smart contracts can be written in three different programming languages namely *Solidity*, *Serpent*, and *LLL* [78]. However, it is recommended to use Solidity as it is actively developed and robust in comparison to the other two [78]. In Solidity a smart contract has a certain structure. It has to contain the word "contract" to define a contract and has the file extension "sol" [76].

To compile a dataset of Solidity contracts the following steps are taken: First, Solidity contracts are queried by searching for keyword "contract" and extension "sol" every day for a period of one month from February 17, to March 17, 2017. The results are sorted by indexed date (recent first). The Solidity contracts are stored on a local file system. Their metadata including, but not limited to, repository name, owner, and timestamp of Solidity file are saved in a PostgreSQL database. GitHub restricts the number of results to 25 files, but gives the total number of matching files. Moreover, only repositories with fewer than 500,000 files are searchable and only the default branch is considered[5]. The only way to filter files in GitHub code search is by file size. If the total number of results is greater than 25, the search range is changed by applying an upper and lower limit of the file size return. The maximum file size returned by the search API is 384 KB. In the first step, the search ranges from 0 to 384 KB. In the next step, two search queries are issued with 0 to 192 KB and 192 KB to 384 KB. The query algorithm recursively splits the subsequent size ranges in two, until either the total number of results is returned or it has been running for eight hours. The time limit prevents the GitHub account from getting blocked by too many requests. GitHub restricts the rate of queries to its search API to 30 searches per minute [6]. In case a Solidity contract is updated during the period of the query, the timestamp in the SQL database is updated and the Solidity file is replaced. The code for querying Solidity contracts is available on GitHub [7].

Second, ConsenSys has developed a Solidity parser which creates a JSON of Solidity source files [79]. This JSON can be used to analyse the source file as it gives structure to the declarations, modifiers, and functions as well as further metadata (i.e. the Solidity version). The resulting JSON depends solely on the source file and has a dynamic length and depth. To allow for a structured analysis, this JSON is stored in the document-based database MongoDB. Metadata collected in the SQL database are also transferred to MongoDB. The source code to load the Solidity files and

[5]https://developer.github.com/v3/ (visited on 18/04/2017)
[6]https://developer.github.com/v3/search/ (visited on 18/04/2017)
[7]https://github.com/nud3l/github-search-crawler

metadata into MongoDB is available on GitHub [8]. Based on this dataset, describing attributes of smart contracts like Solidity version and amount of Solidity files in a repository are extracted. Moreover, the complexity of Solidity code hosted on GitHub is analysed.

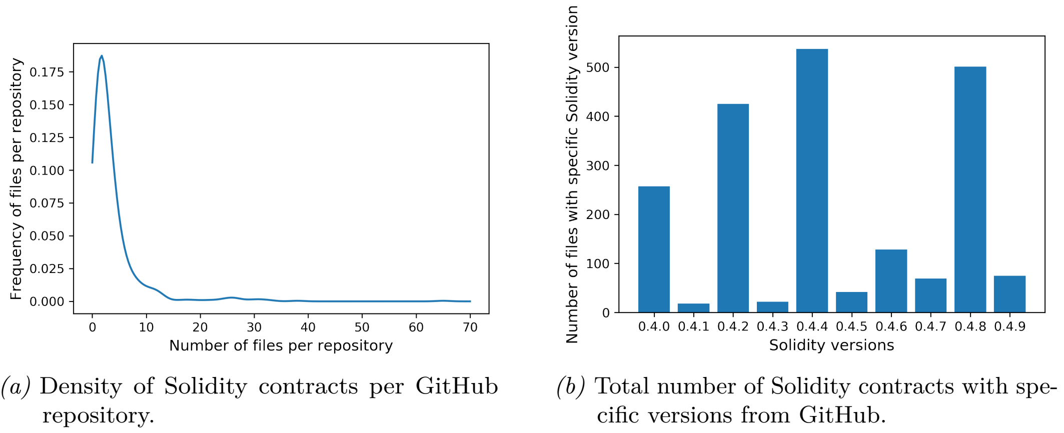
Third, Luu et al. introduced Oyente, a tool to identify for different security issues in EVM op-codes [77]. The tool compiles Solidity source files using *solc* into EVM op-codes to check for security related issues in the code. (1) Callstack: When calling other contracts, exceptions need to be handled properly. Ethereum has a callstack depth limit of 1024, which can be used to attack contracts. (2) Time dependency: Outcome of the contract is dependent on a time constraint. For example, random numbers can be generated based on the mining time of a block, which can be influenced by the miners. (3) Re-entrance: A call can be repeated multiple times, although not desirable. For example, a function of a contract can be called and while this function waits to be finished might be called again. (4) Concurrency: Order of transactions can influence smart contract outcomes. For example, in a Puzzle game, the owner might set the reward lower shortly after a solution has been submitted.

3.3.2 Results

Applying the method results in a total of 2,561 Solidity contracts crawled from February 17 to March 17, 2017. This covers a total of 693 repositories. Figure 3.2a shows the frequency of files of each repository whereby the number of Solidity contracts per repository shows a long tail distribution. The μ number of Solidity contracts per repository is around 3.696 with a σ of approximately 5.266. In figure 3.2b the version of Solidity used in the contract is displayed. In the dataset, the Solidity versions range from 0.4.0 and to 0.4.9. The latest Solidity release in the observed timespan is 0.4.10 released on March 15, 2017. The number of files with specific versions varies.

Next, the declarations in each contract are quantified to measure the complexity of smart contracts. Solidity offers following declarations in contracts: Imports are used to import other contracts and their functions. The address of the imported contract needs to be defined (if known in advance) or provided later on. Libraries are similar to contracts, but whenever library functions are called the code is executed in the calling contract and not the library contract. Variables can either be state variables, permanently stored in the blockchain (i.e. storage type), or function variables, only stored in memory in a function. Events are used as a logging functionality for either debugging or outside of smart contracts (e.g. in the Node.js part of Ethereum Dapps). Mappings provide a key value store only declarable as a storage type, while structs and enums can either be declared in storage or memory. Functions are either

[8] https://github.com/nud3l/smart-contract-analysis/tree/master/data-loader

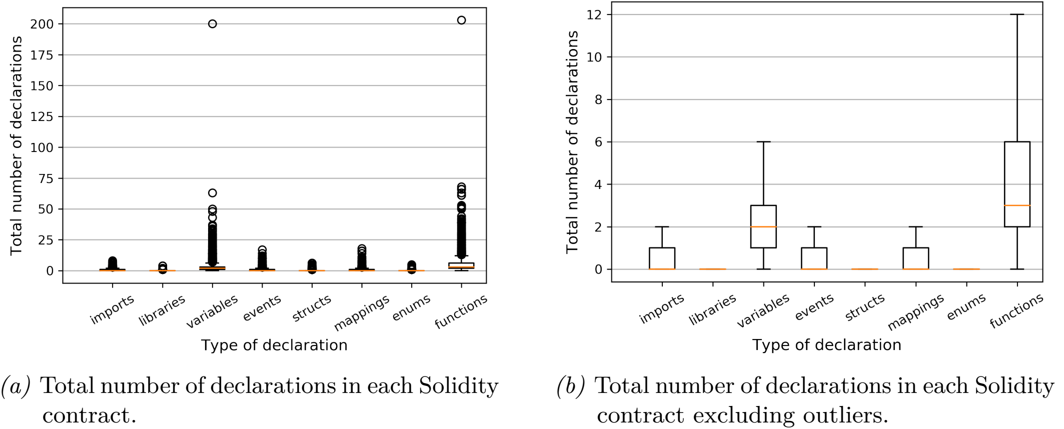


(a) Density of Solidity contracts per GitHub repository.

(b) Total number of Solidity contracts with specific versions from GitHub.

Figure 3.2: Number of repositories and Solidity versions in GitHub dataset.

external, and thereby callable by others, or internal, i.e. only accessible from within the contract. Moreover, they can contain the "payable" modifier, which accepts the transfer of Ether to the function. Figure 3.3 displays the number of declarations in each smart contract with and without outliers. The μ of imports is 0.646 and has a σ of 1.080. Libraries are seldom used with μ 0.017 and σ 0.172. There are μ 3.333 variables per contract with σ 6.392. Events have a μ of 0.633 and σ 1.496. Structs and enums are comparably rare with μ of 0.240 and 0.046 as well as σ 0.679 and 0.265, respectively. Mappings have a μ of 0.689 and σ 1.500. Last, functions are common with μ 5.677 and σ 8.503.



(a) Total number of declarations in each Solidity contract.

(b) Total number of declarations in each Solidity contract excluding outliers.

Figure 3.3: Total number of declarations in each Solidity contract in GitHub dataset. Median displayed as an orange line.

The analysis of security issues depends on compiling the Solidity source code and then performing a symbolic execution with the Z3 theorem prover [80]. There can be

two issues when running this analysis: First, as contracts on GitHub might not be used in practice and reflect a current status of development, they might not be able to compile with the Solidity compiler *solc*. These contracts would not be analysed for security issues, as the resulting EVM bytecode would not be available to Z3. Second, the Z3 theorem prover executes different states of the contract which is comparably time-consuming. To circumvent execution of potential infinity loops and limit the time of execution, Z3 is provided with a timeout that aborts its analysis even if the result is incomplete. The analysis is conducted on a computer with four Intel i7 cores at 2.20 GHz and 16 GB of RAM for 97 hours. All contracts in the dataset are complied and checked for the callstack issue with 250 of 2561 being affected. The other three methods required a longer time or ran into infinity loops, resulting in 2057 of 2561 contracts being analysed. Out of these 2057 contracts, 38 are potentially affected by the time dependency issue, 246 by re-entrance, and 64 by concurrency issues as displayed in figure 3.4. The aforementioned results, with regards to data exploration, complexity, security issues, and shortcomings of the analysis, are discussed in chapter 5.

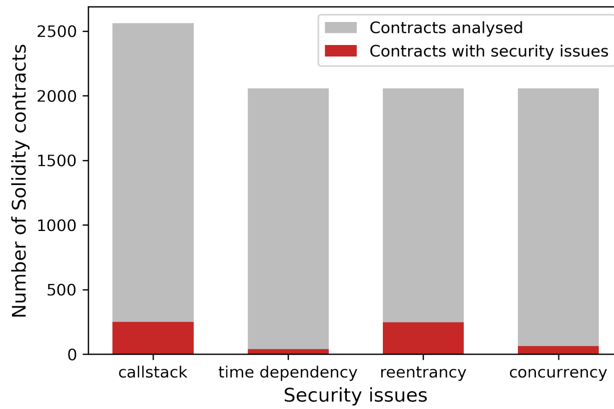


Figure 3.4: Number of security-related issues in Solidity contracts in GitHub dataset.

4 Chapter 4

Verifiable computation for smart contracts

Permissionless blockchains assume no trustworthy agents or entities exist in the system. By applying social control, behaviour of agents can be enforced. Due to the restrictions set by the EVM (i.e. gas cost of operations), implementing functions in Ethereum with a time and space complexity greater than $\mathcal{O}(n)$ is not feasible. Hence, complex computations like matrix multiplications can only be achieved for small-sized matrices. Otherwise, the function implementing the multiplication will use more gas than the upper gas limit set by the EVM and thus, throw an error before finishing. To circumvent these limitations, computations can be executed outside of Ethereum and results stored on the blockchain. This chapter covers the development of an algorithm for verifying computations requested within the blockchain and executed by external services under the assumption of a trustless system.

4.1 Verifying computation

Formal methods based on mathematical proofs and game-theoretic approaches can be used to achieve verifiable computations. Since formal methods are not yet practical, a game-theoretic approach is taken to develop an algorithm for verifiable computations. This section describes the method used to develop the algorithm, agents in the algorithm, and their interactions.

4.1.1 Method

The development of the algorithm follows a deductive method of considering existing research in two fields. First, verifiable computation concepts using blockchains presented in [36] and [35] are analysed. Second, cloud and distributed systems research as presented in [81] and [82] are studied. Specifically, the assumptions made by the aforementioned research are critically assessed to develop a new algorithm. The main

aspect of [36] revolves around preserving privacy of user data, whereby aspects of the blockchain are used to enforce the algorithm. [35] focusses on verifiable computation for Ethereum using computation services inside the blockchain. Their suggestion is a verification algorithm with a dispute resolution and an incentive layer. Their proposal has two practical issues: First, the verification game includes a "jackpot" to reward solvers and verifiers for their work. This introduces an incentive to steal the jackpot by solvers and verifiers colluding to receive the jackpot without providing a correct solution. Second, they propose to implement the computation tasks in C, C++, or Rust code using the Lanai interpreter implemented as a smart contract on Ethereum. Using this approach, execution of the judge requires a comparable large amount of gas. Therefore, with many services providing a wrong solution to a task, a costly computation is triggered. Moreover, it limits the flexibility of computation services by forcing them to use one of the three programming languages. The objective of the here presented algorithm is to achieve:

1. Execution of arbitrary computations requested from a smart contract in Ethereum and executed outside the blockchain.
2. The verification of the computation result should be achievable in a reasonable time, that is $\mathcal{O}(n)$.
3. Ensure that the result of the computation is correct without having to trust the providing service.

This development is further based on a creative aspect to experiment with different agents, incentive models, and interactions. Hereby, different parameters and involved agents are considered in a "pen and paper" exercise. Afterwards, the parameters are verified by a qualitative assessment and a quantitative analysis. The qualitative assessment discusses the first design objective of executing arbitrary computations on the basis of previous research in section 5.2. The implementation is described in section 4.2 and the experiment method for the quantitative analysis in section 4.2.1. The quantitative experiments constitute an evaluation basis for the last two algorithm objectives.

4.1.2 Actors

The actors involved in the verifying computation algorithm are presented in figure 4.1. First, *users* request solving a specific computation problem based on input data, and an operation to be performed on the data. This could be e.g. multiplying two matrices or inverting a matrix. Users can be smart contracts or any other entity holding a public Ethereum address. They provide an incentive for solving and verifying the problem. Second, *computation services* provide computation power outside of the Ethereum blockchain in exchange for receiving a compensation. They eventually receive the operation and data from users and return a result from the computation

that is either correct or incorrect. Thereby, one of the computation services acts as a *solver* and at least one other computation service acts as a *verifier*. They have a smart contract counterpart handling the data exchange in and out of Ethereum through oracles [83]. Oracles in Ethereum are programs having a smart contract, and a “regular” application part. The smart contract receives requests from users, stores them on the blockchain, and issues an event about a new request. Typically, the application counterpart is an implementation using the *web3* JavaScript API for Ethereum listening for these events. It then collects the query data and executes it. Upon completion, the result of the query is stored in Ethereum. An example of such a service is [84]. Third, *judges* are smart contracts that decide whether basic mathematical operations including multiplication, subtraction and addition of two integers are correct or not. They are neutral parties and are not receiving any incentives. Fourth, an *arbiter* enforces the verifiable computation algorithm when users request a new computation. The arbiter is a smart contract that moderates the algorithm.

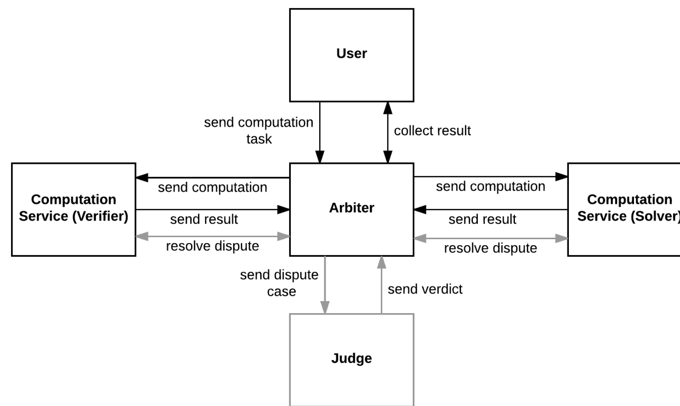


Figure 4.1: Overview of actors in the verification algorithm. Black actors are required in the verification algorithm, while grey actors are only part of the dispute resolution.

4.1.3 Assumptions

Certain assumptions are made during the development of the algorithm. Users are assumed as rational agents with the objective to receive a correct computation. They are required to send a fee in Ether to execute the computation to cover the costs of using the oracle service and to reward solvers and verifiers. This fee depends on the complexity of the computation to be performed, the complexity of the input data, and the number of verifiers. However, it cannot exceed 5 Ether as the creation of random numbers in Ethereum depends on properties set by the miners. This is discussed in detail in section 5.2.1.

Computation services are assumed as rational agents trying to optimise their incentive. However, they might purposely communicate false information to maximise their incentive. Possible agent behaviours are introduced in 4.1.5. Further, enough computation services are available (i.e. a minimum of 2) to execute the computation with at least one verifier. The probability of detecting a false computation depends on the number of verifiers in the algorithm.

The arbiter and judge are trusted agents, respectively enforcing the algorithm and reaching a verdict. This is a strong assumption in a trustless system and needs to be justified. To limit their incentive for undesired behaviour (i.e. cheating) in the algorithm, these two agents are not rewarded for taking part in the computations. Thus, their work is *pro bono* and only the operational cost in gas are covered. Another approach is to formally verify the implementation of the algorithm and publish the proof alongside the smart contract, so users can verify the behaviour of the agent [85][86].

Computation services offer a range of computation types, that are implemented in the same way. For example, a matrix multiplication can be implemented in a naive and a parallel execution way. Both would represent different computation types. Also, in floating point operations, computation services need to have the same precision, or otherwise results or intermediary results differ.

4.1.4 Algorithm

Before the algorithm starts, an arbiter is present to moderate the algorithm. Computation services can subscribe to one or multiple arbiters for solving tasks. They need to provide a deposit defined by the arbiter to be able to participate in the algorithm. Furthermore, they can unsubscribe from an arbiter and receive their deposit back in case they did not break the algorithm (elaborated on detail later). Judges also register themselves with one or multiple arbiters, but do not need to provide a deposit. To track the status of the algorithm each computation request has an assigned status. The user of the algorithm can track the status of computations and trigger actions accordingly. The states of the algorithm are introduced in table 4.1. Actions taken by the actors involved are displayed in an Agent UML package diagram in figure 4.2 [87]. Details are explained in the following paragraphs.

4.1.4.1 Verification algorithm

The algorithm is initiated when a user requests a computation by sending the input data, the operation to be performed, and the desired number of verifiers to the arbiter. The input data can be in a binary format or a link to the data. The arbiter can be agnostic towards the input data, while the computation service implementing the operation needs to be able to read the data. For example, the input data could

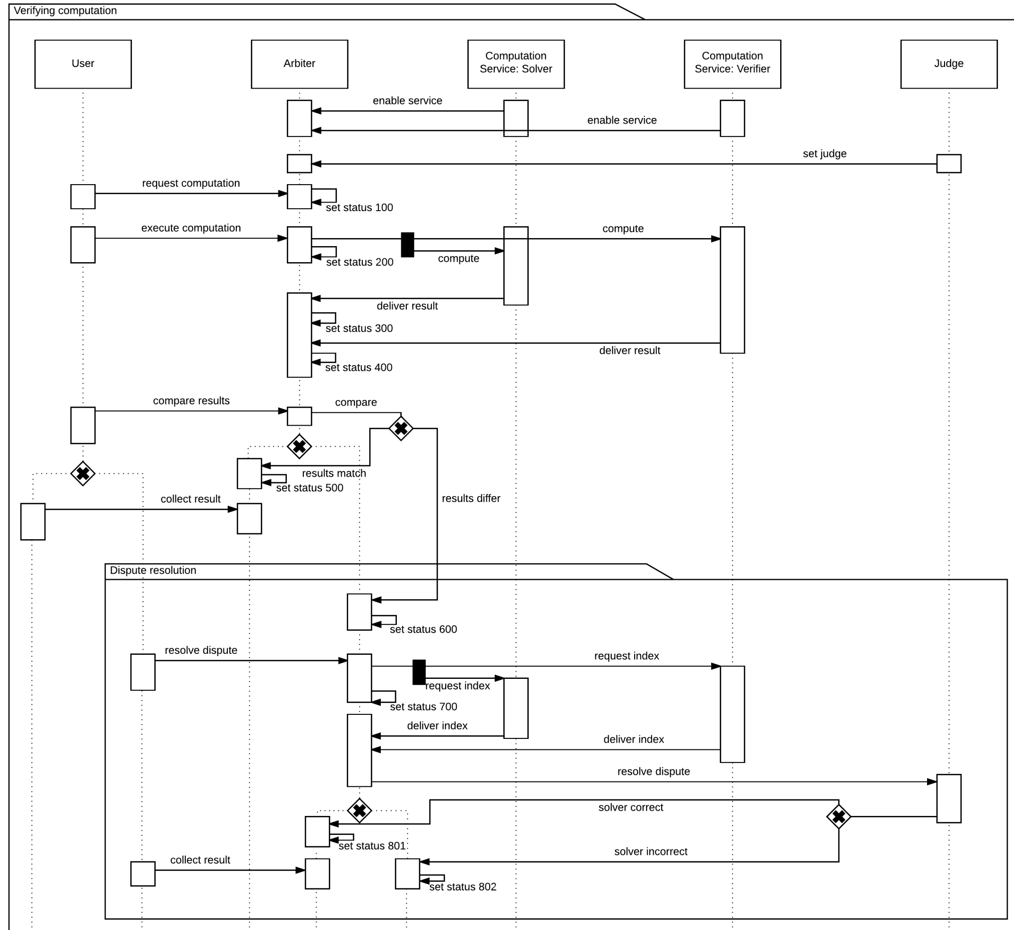


Figure 4.2: Agent UML package diagram of verification algorithm and dispute resolution with user, arbiter, solver, one verifier, and judge.

Table 4.1: Overview of verifying computation algorithm states.

Status	Description
100	The computation request is created, but no solutions are yet provided.
200	Arbiter sent the computation requests to solver and verifier(s).
300	A part of the results are in and the arbiter is waiting for the remaining results.
400	All results from solver and verifier(s) have arrived.
500	Results are compared and solver and involved verifier(s) agree on one solution.
600	Results are compared and solver and at least one verifier disagree on the solution.
700	Dispute resolution started to determine, if the solver provided an incorrect solution.
800	Dispute is resolved with either the solver being correct (status 801) or incorrect (status 802).

be two matrices, the operation “multiplication”, and the user requires five verifiers. The arbiter then checks if there are enough computation services available that can perform the operation. In the example, at least six need to be available as it requires one solver and five verifiers. In case enough computation services are available, the request is registered in the blockchain.

Computation request and execution functions are listed in algorithm 1. One computation service is randomly determined as a solver, and the other(s) are randomly assigned as verifiers (status is set to 100). The user instructs the arbiter to forward the input data and operation to the computation services smart contracts (status is set to 200), triggering the off-chain computation by sending a request through an oracle. This requires sending a fee for the computation as well as providing the fee for using the oracle (see details in section 4.2).

Receiving and comparing results is described in algorithm 2. Verifiers and the solver report their result back to the arbiter. The arbiter in turn receives the results provided by solver and verifier(s) to store them for comparison (status is set to 300). If all results are reported back, then the status is set to 400. In case a time-out condition is reached, the arbiter stops the computation[1].

If the status is set to 400, the user can trigger the arbiter to compare the available results. If the solver and all participating verifiers agree on one solution, the status is set to 500 and the user is able to receive the result. However, if at least one verifier disagrees with the solver the status is set to 600.

[1]Handling computation services unsubscribing during computation is discussed in section 5.2.2.

Algorithm 1: Arbiter requesting and executing computation functions.

```
1  function requestComputation (input1, input2, operation, numVerifiers);
   Input  : Two strings input1 and input, the operation to be performed, and the
            number of verifiers numVerifiers.
2  if numVerifiers = 0 then
3  |  throw;
4  else
5  |  computationId = rand(0, 2^64);
6  |  requests[computationId].input1 = input1;
7  |  requests[computationId].input2 = input2;
8  |  requests[computationId].operation = operation;
9  |  remainingServices = computationServices;
10 |  requests[computationId].solver = random(remainingServices);
11 |  remainingServices = removeService(solver, remainingServices);
12 |  for i = 0 to numVerifiers do
13 |  |  requests[computationId].verifiers[i] = random(remainingServices);
14 |  |  remainingServices = removeService(verifiers[i], remainingServices);
15 |  end
16 |  updateStatus(100, computationId);
17 end
18 function executeComputation ();
19 AbstractComputationService mySolver =
   AbstractComputationService(requests[computationId].solver);
20 mySolver.compute( requests[computationId].input1, requests[computationId].input2,
   requests[computationId].operation, computationId );
21 for i = 0 to requests[computationId].verifier.length do
22 |  AbstractComputationService myVerifier =
   |  AbstractComputationService(requests[computationId].verifier[i]);
   |  myVerifier.compute( requests[computationId].input1,
   |  requests[computationId].input2, requests[computationId].operation,
   |  computationId );
23 end
24 updateStatus(200, computationId);
```

Algorithm 2: Arbiter receiving and comparing results functions.

```
1  function receiveResults (result, computationId);
   Input : The result delivered by the computation service to a specific
           computationId.
2  if msg.sender = requests[computationId].solver then
3  |  requests[computationId].resultSolver = result;
4  else
5  |  i = requests[computationId].verifier.index;
6  |  requests[computationId].resultVerifier[i] = result;
7  end
8  if allresultsarein then
9  |  updateStatus(400, computationId);
10 end
11 function compareResults ();
12 for i = 0 to requests[computationId].verifier.length do
13 |  if requests[computationId].resultSolver !=
   |  requests[computationId].resultVerifier[i] then
14 |  |  recordChallenger[i];
15 |  end
16 end
17 if recordChallanger.length = 0 then
18 |  updateStatus(500, computationId);
19 else
20 |  updateStatus(600, computationId);
21 end
```

4.1.4.2 Dispute resolution

If the status is set to 600, the user can initiate a dispute resolution algorithm. The dispute resolution is inspired by a technique introduced in [82], [81] and [35]. The idea is to split up the operation into simple parts with intermediary results until the computation is simple enough for the judge to solve it. Overall and intermediary results are stored in a Merkle tree for the solver, and each verifier challenging the solver. The comparison is achieved through a binary search on the trees. The root of the tree encodes the overall result, while the leaves in the lowest layer encode the input data. Leaves in between represent intermediary results. For simplification, the following example considers the comparison of a matrix multiplication without hashing overall and intermediary results as well as input data. Also, the example does not cover the construction of a Merkle tree, but rather gives an indication how comparison on computations can be achieved by splitting up the comparison to intermediary results. In practice, Merkle trees and binary search are used to efficiently limit the search time to $\mathcal{O}(\log n)$ [88].

When a user requests the dispute resolution, the arbiter sends a request to the computation service(s) acting as verifier(s). They are required to point out an index of the output data, and a type of operation for which they receive a different result (the status is updated to 700). The verifier(s) receive the result of the solver to determine the index for which their solution defers. Consider the simple example of a matrix multiplication with two input matrices A and B. The result provided by the solver is matrix S. At least one verifier provided the differing matrix V.

$$A = \begin{bmatrix} 2 & 7 \\ 6 & 3 \end{bmatrix} B = \begin{bmatrix} 1 & 5 & 9 \\ 6 & 3 & 4 \end{bmatrix}$$

$$S = \begin{bmatrix} \mathbf{43} & 31 & 46 \\ 24 & 39 & 66 \end{bmatrix} V = \begin{bmatrix} \mathbf{44} & 31 & 46 \\ 24 & 39 & 66 \end{bmatrix}$$

The verifier provides the index of his result matrix deferring from the solvers result (i.e. $V_{1,1}$) to the arbiter. The arbiter checks whether the result at index $1, 1$ of the solver and verifier are different. In case the verifier would not have provided a different result at the index (i.e. the value at index $1, 1$ is 44 for both results) the algorithm ends. The solution provided by the solver would be marked as correct. If the results deviate, the solver needs to provide the intermediary steps leading to his result. The solver repeats until the input of this intermediary result is an index of the input matrices. Hence, the solver would provide two intermediary results $I(S)$ and their according index. If the solver fails to provide these results, the status is set to 802.

$$S_{1,1} = I(S)_1 + I(S)_2 = 1 + 42 = 43$$

$$I(S)_1 = A_{1,1} * B_{1,1} = \mathbf{1 * 2 = 1}$$
$$I(S)_2 = A_{2,1} * B_{1,2} = 7 * 6 = 42$$

The arbiter forwards the two intermediary results to the verifier including the index of the input matrices. Following this, the verifier provides an intermediary result $I(V)$ and the index of the input matrices from the verifier such that $I(V)_n = A_{i,j} * B_{j,i}$ and $I(S)_n \neq I(V)_n$. In our example, the verifier needs to send $I(V)_1$ with $A_{1,1}$ and $B_{1,1}$. Then the arbiter checks if $I(S)_n \neq I(V)_n$ and triggers a verdict by the judge, if it holds true. Otherwise, the solver is declared to be correct. A verdict is reached by the judge receiving the value of the input matrices at the two indices, the intermediary result of the solver, and the operation to perform from the arbiter. The judge returns a boolean based on those inputs. In the example, the judge receives $A_{1,1} = 1$, $B_{1,1} = 2$, $I(S)_1 = 1$, and the operation “multiplication” which results in $false$. The solver provided a false solution, resulting in a status of 802.

If the solver had been correct and the judge returning $true$, the status would have been set to 801. Also, if during the intermediary steps, the verifier would have not been able to provide intermediary results deviating from the solver’s, the status of the algorithm would have been set to 801. In a last step, the user is able to collect the result, if the status of the algorithm is set to either 500 or 801. Otherwise, the solver’s result is incorrect. The verifier proving the solver wrong at one index might have errors in his computation. These were not checked in the algorithm. Therefore, the result cannot be retrieved if the status is set to 802.

4.1.5 Interactions

The computation service is not trusted. In fact, the algorithm is intended to resolve the issue of having services providing incorrect results or trying to modify the algorithm to optimize their incentive. Computation services can act either as solver S or verifier V, which is determined by a random selection through the arbiter. This section focusses on the behaviours of the computation services.

Under the assumption that arbiter, judge, and user behave rational and follow the algorithm, computation services have a combination of four different behaviours with respect to their role as S or V. The behaviours are summarised in table 4.2 with either all verifiers accepting the solution (i.e. V_A) or challenging the solution (i.e. V_C). S profits the most if it provides a correct solution, which is challenged by V, while V profits the most when S provides a false solution and V is able to challenge it. The problematic case is that the incentives for accepting a false or correct solution are the same. To prevent this from happening we will consider the behaviour of V and S in detail.

Case 1: S provides a correct solution and no V challenges the solution. In this case agents behave exactly as intended by the algorithm. As no V challenges the

Table 4.2: Possible behaviours of computation services as solver S and verifier V, whereby all verifiers behave the same.

		S: correct solution	S: false solution
V	challenge	S receives S fee share S receives V_C fee share V_C receives nothing	S receives nothing V_C receives V_C fee share V_C receives S fee share
	accept	S receives S fee share V_A receives V_A fee share	S receives S fee share V_A receivesV_A fee share

solution, the judge is not triggered and the fee is equally split among S and the involved V.

Case 2: S provides a correct solution and at least one V challenges the solution. This is an undesired behaviour, since the solution provided is actually correct. This triggers the dispute resolution with a verdict by the judge determining S as correct. In this case S profits from the extra work due to the additional dispute steps by receiving the fee share of V_C. V_A receive their part of the fee, since they acted correctly and their amount of work remained the same.

Case 3: S provides a false solution and no V challenges the solution. In this case S and all V would receive their share of the fee. This is a highly undesired behaviour in the algorithm as it would flag a false result as correct. To prevent this from happening two measures are used. First, computation services do not know their role in advance as they are randomly assigned by the arbiter. If several services collude to provide false solutions to, e.g. damage the user or save computation cost (i.e. they could prepare an arbitrary solution in advance), all of them would need to work together to provide the "same wrong" result. However, if just one V_C exists, it profits by gaining the fee shares of itself, S, and all V_A. Thus, second, the user is able to determine the number of V for each computation. Thereby, the probability of having at least one V_C depends on the prior probability p of V providing correct or false solutions and the number n of V in the computation. As computation services are drawn at random, the probability of having at least one V_C can be calculated by $P(V_C) = 1 - p^n$. With an increase of n or a decrease of p, the probability of having at least one V_C can be increased. Thus, the user is able to balance the probability of achieving a correct result and the cost of the algorithm.

Case 4: S provides a false solution and at least one V_C challenges the solution. Hereby, S and V_A are not receiving their share of the fee, which goes to all V_C. This is based on the verdict by the judge. However, this is also an undesired case since the user does not receive a solution to his computation (i.e. algorithm results in status 802).

Considering the four scenarios, rational S is trying to receive its share of the incentive and get a chance to receive fees of any V challenging a correct solution. Hence,

the strategy for S considering V is to provide a correct solution to the problem. Moreover, V profits the most form challenging a false solution. A rational V provides the correct solution to a computation to receive its fee share or to have the chance of becoming a challenger to a false solution. Arguably, S and V could try to deliver a false solution to save up on computation cost or trick the user. In this case, the probability of discovering the false solution relies on the number of Vs and the prior probability of cheating Vs. If a V delivers a false solution, it must be the same solution as S' to not trigger the dispute resolution. As the assignment is random, this is a valid strategy for a low number of Vs. Moreover, by destroying the services' deposits and excluding them from the algorithm after detected cheating, the prior probability of having such a service can be reduced.

4.2 Implementation and experiments

The algorithm is implemented using smart contracts, oracles, and external computation services. Implementation details are presented in figure 4.3, which extends the previous actor overview figure from section 4.1.2.

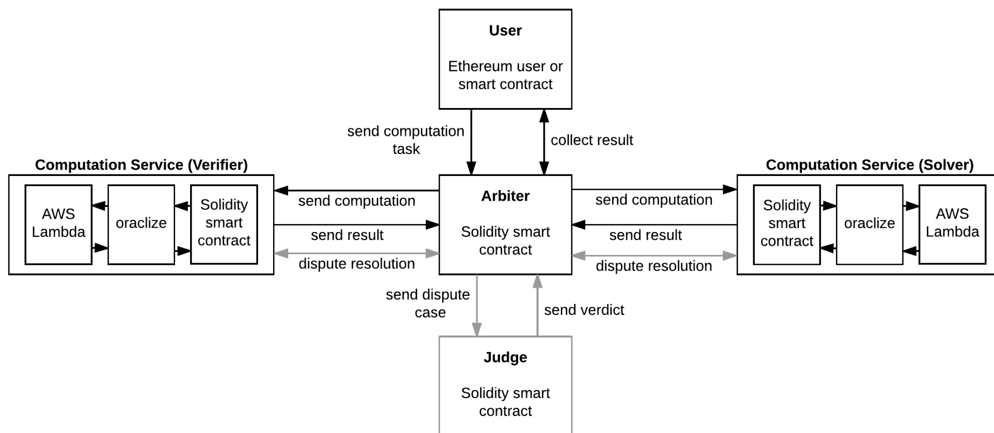


Figure 4.3: Implementation overview of actors in the verification algorithm. Black actors are required in the verification algorithm, while grey actors are only part of the dispute resolution.

Smart contracts are written in Solidity 0.4.8 and are available on GitHub [2]. Users are assumed to have an address in the Ethereum network and can request computations. The arbiter and judge are implemented as smart contracts. The computation service is implemented using a smart contract as an interface for receiving computation requests and providing solutions to the arbiter. To execute the computation, AWS Lambda[3] functions are created and accessed through an API. The communication

[2]https://github.com/nud3l/verifying-computation-solidity (visited on 16/05/2017)
[3]https://aws.amazon.com/lambda/ (visited on 16/05/2017)

between the service's smart contract and AWS Lambda is achieved by using oraclize [84].

```
pragma solidity ^0.4.8;

contract AbstractArbiter {
  function enableService();
  function disableService();
  function requestComputation(string _input1, string _input2, uint
    _operation, uint _numVerifiers);
  function executeComputation() payable;
  function receiveResults(string _result, uint256 _computationId);
  function compareResults();
  function requestIndex();
  function receiveIndex(uint _index1, uint _index2, uint _operation,
    uint256 _computationId, bool _end);
  function setJudge(address _judge);
  function getStatus() constant returns (uint status);
  function getCurrentSolver(address _requester) constant returns (
    address solver);
}
```

Code excerpt 4.1: Abstract arbiter smart contract to define its ABI excluding internal functions.

Smart contracts communicate either through sending transactions or function calls. In both cases, the initiating smart contract needs to know the ABI definition of the smart contract it wants to interact with. To separate the definition of the ABI from the implementation of the functions, the arbiter and computation service are defined through an abstract contract as presented in code excerpt 4.1 and 4.2.

```
pragma solidity ^0.4.8;

contract AbstractComputationService {
  function __callback(bytes32 _oraclizeID, string _result);
  function compute(string _val1, string _val2, uint _operation, uint256
     _computationId) payable;
  function provideIndex(string _resultSolver, uint _computationId);
  function registerOperation(uint _operation, string _query);
  function enableArbiter(address _arbiterAddress);
  function disableArbiter(address _arbiterAddress);
  function getResult(bytes32 _oraclizeID) constant returns (string);
}
```

Code excerpt 4.2: Abstract computation service smart contract to define its ABI excluding internal functions.

The abstract contract is comparable to an abstract class in object oriented programming. The abstract contract defines the required functions including their parameters and return values without implementing the function logic. This is sufficient to define the ABI of the actual contract implementation. The abstract contracts can be used to call or send a transaction to an implemented type of that contract at a specific

Ethereum address. Therefore, the implementation of each computation service can be different, while the ABI remains constant.

Experiments are executed within *TestRPC*. TestRPC is a framework to simulate the EVM on a local computer to allow development of smart contracts and interaction using cryptocurrency without actual value [89]. For the development of the smart contracts *Truffle* is utilised. It offers some degree of automation for compiling, deploying, and testing smart contracts [78]. The experiments are conducted by scripts written in JavaScript simulating a full execution of the algorithm from the perspective of a user. Each iteration of the experiment starts with deploying a new set of ten computation services, one arbiter, and one judge. Judge and computation services are registering themselves with the arbiter before the user sends a computation request. The testing method is elaborated on the next section. Results are discussed in section 5.2.4.

4.2.1 Method

The quantitative analysis is conducted by executing experiments based on an implementation of the algorithm in Ethereum with one exemplary type of computation. The computation is a multiplication of two integers to simplify the verification steps in the algorithm. To test the algorithm's objective two and three (refer to section 4.1.1), the algorithm is simulated multiple times. The execution time of the algorithm, the gas consumption of the algorithm, and the verification result are reported. The results depend on external and internal parameters of the algorithm. Externally, the prior probability of computation services providing false solutions is considered. Internally, the number of verifiers the user requests for each computation are examined.

The percentage of verifiers providing false results is the prior probability of receiving false results in the verification. In essence, the algorithm is a variation of a consensus algorithm, where a judge decides between arguing parties. However, if the dispute resolution is not triggered, the judge is not involved. Hence, different prior probabilities need to be tested. [90] introduce a *38.2% attack* on Nakamoto consensus algorithms, where a minority of rational miners can incentive other miners to accept a blockchain of the minority's choice. In the introduced algorithm also computation services could collude to save on computation power. [4] describes a *51% attack* on consensus algorithms, where a majority of the miners' computation power can dictate the blockchain. If one entity would be able to control 51%, that entity is able to control the blockchain. Again, with the introduced algorithm a majority of computation services can collude to dictate the solution. Hence, the experiment is conducted with 30%, 50%, and 70% prior probabilities to test the result of the algorithm. Also, the computation services are assumed to collude i.e. they provide the same false solution.

Experiments are executed 1000 times for each different configuration of parameter to determine execution time, gas consumption, and outcome of the computation. Assuming a potentially large number of computation services ($> 10,000$), this gives a confidence level of 95% and a maximum confidence interval of 3.1 for the three different prior probabilities. Before each iteration of the experiment the environment is initialised with a new set of smart contracts. TestRPC is operated on a computer with two Intel i5 cores at 2.5 GHz and 8 GB of main memory.

During test runs of the experiments, it was discovered that utilising an oracle is comparably time-consuming. The mean time for issuing a request through the oraclize service until receiving the result back is around 150 seconds. Since the experiment aims to show the probability of accepting false solutions, numerous executions are required. Hence, the implementation of the computation service is changed in the experiments. Instead of using an oracle and AWS Lambda, the smart contract of the computation service directly provided a solution by executing the integer multiplication itself.

4.2.2 Execution time

The first experiment is used to measure the time to complete the algorithm. Specifically, it measures the time of executing the functions, calling contracts, and sending transactions. Thus, the overall execution time depends on the algorithm implementation and also on TestRPC. During experiments the performance of TestRPC varied. With an increasing number of calls TestRPC utilises more main memory. Figure 4.4 shows how after around 370 rounds of executing the experiments, the execution time with the six different numbers of verifiers increases. This is caused by TestRPC requiring more RAM than available on the test computer. The swap fills up at around round 680 and leads TestRPC to crash. Restarting TestRPC and continuing the experiments reduces the overall response time again.

Therefore, the sample size for the execution time is reduced to $N = 100$. Figure 4.5 presents the results with 30%, 50%, and 70% of computation services providing false solutions and different numbers of verifiers. Time testing is conducted to determine time complexity of the algorithm and communication overhead. Adding verifiers to the algorithm increases the execution time linearly. Overall, σ increases when including more verifiers. At 50%, the execution time grows stronger than with 30% or 70%.

4.2.3 Gas cost

Reporting the amount of gas used equals the time and space complexity of the algorithm, as gas consumption is determined by the type and number of operations in the EVM. It further excludes the time used for sending transactions or calls.

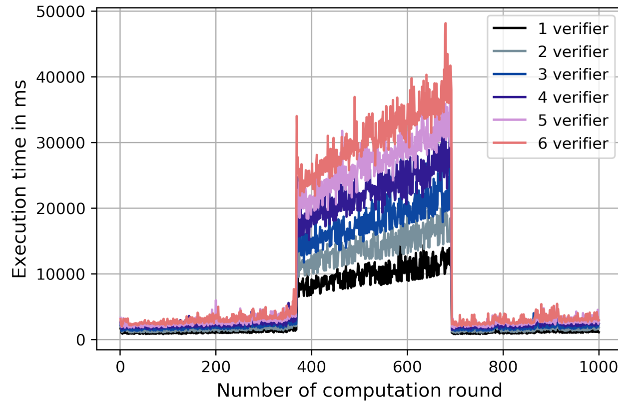


Figure 4.4: Total execution time with different number of verifiers and 30% of computation services providing incorrect solutions with $N = 1000$.

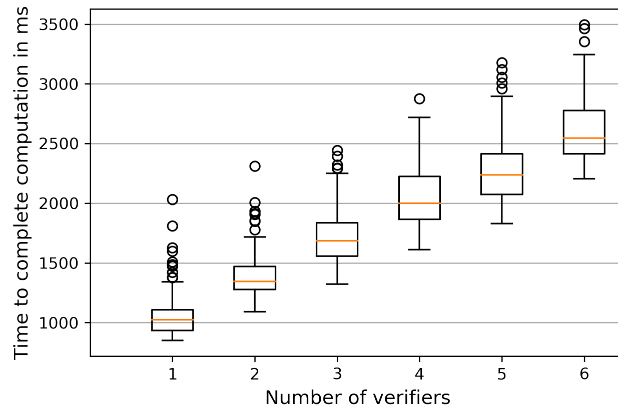
Independent of the prior probability of false solutions, the μ gas consumption increases linear as presented in figure 4.6. Further, σ decreases with an increasing number of verifiers. At a low number of verifiers, the dispute resolution is less likely triggered, leading to a higher σ in gas consumption. With an increasing number of verifiers, the probability of triggering the dispute resolution increases. As the dispute resolution is almost always triggered, σ is reduced.

4.2.4 Verification

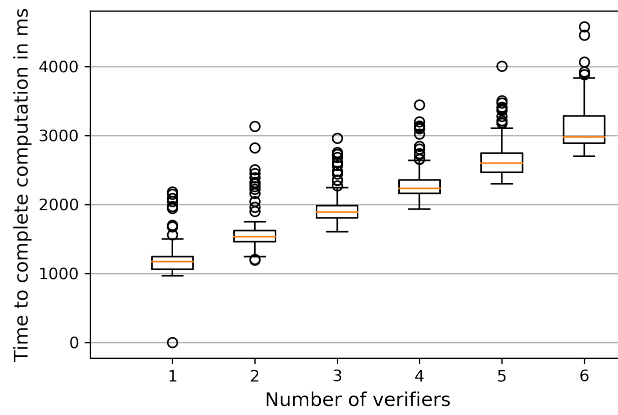
Last, the verification of the algorithm is observed. The algorithm is tested for five different cases: First, the algorithm can accept a correct solution (i.e. status 500 or 801). Second, each verifier agrees with the solver although the solution is not correct (i.e. status 500). In this case the dispute resolution is not triggered and the user receives a false solution marked as correct. Third, at least one verifier disagrees with the solver providing a false solution and the judge rules that the solver's solution is false (i.e. status 802). Fourth, the dispute resolving is triggered, but a false solution is accepted (i.e. status is 801, but should be 802). Fifth, the dispute resolution is triggered, but a correct solution is denied (i.e. status is 802, but should be 801).

Figure 4.7 presents the results of the experiment. With 30% prior probability of false solutions, the number of correct solutions provided is μ 776.5 and σ 20.5. The number of false accepted solutions is μ 4.5 and σ 11. No solutions provided (i.e. status 802) has μ 219 and σ 24.8. At 50%, correct solutions has μ 440.8 and σ 17. False solutions are accepted with μ 77.7 and σ 112. No solutions occur with μ 481.5.7 and σ 121.7. Having 70% results in μ 334.2 and σ 14.4 correct solutions accepted. False solutions are accepted with μ 142.5 and σ 158.3. Last, no solution is reached

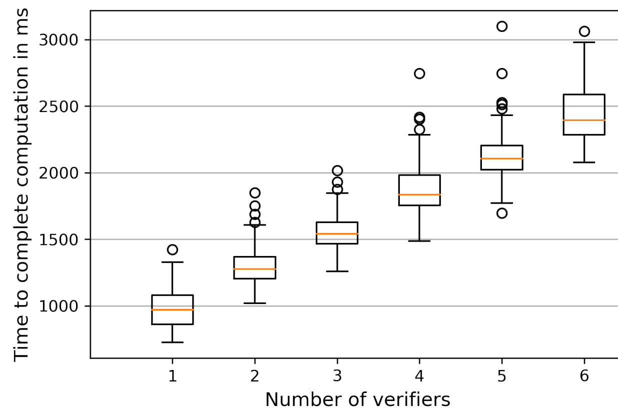
with μ 523.3 and σ 154.7. The cases four and five of the experiment did not occur. Thus, in no case the verdict of the judge was false.



(a) 30% of computation services providing incorrect solutions.

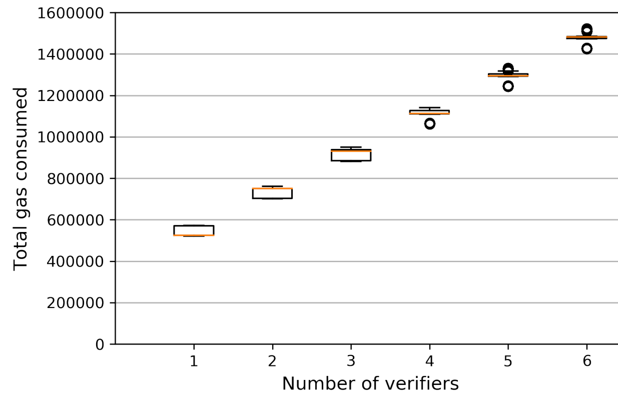


(b) 50% of computation services providing incorrect solutions.

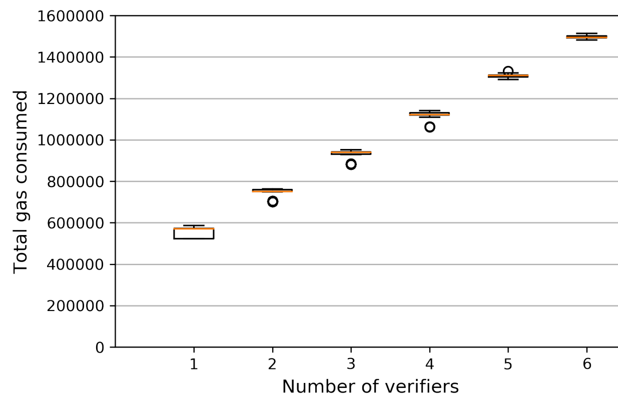


(c) 70% of computation services providing incorrect solutions.

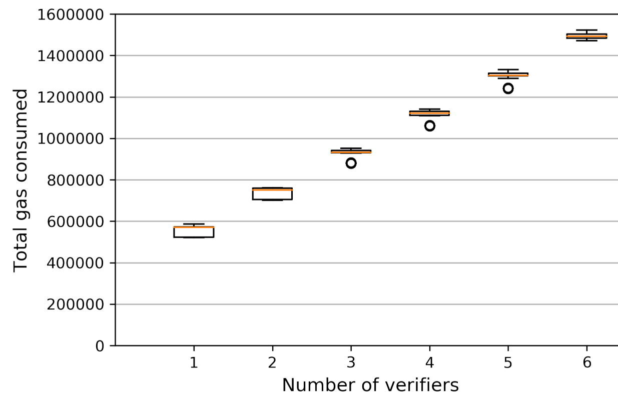
Figure 4.5: Total execution time of algorithm with different number of verifiers and percentage of computation services providing incorrect solutions. Each combination of specific number of verifier(s) and percentage of computation services with incorrect solutions with $N = 100$. *Median displayed as an orange line.*



(a) 30% of computation services providing incorrect solutions.

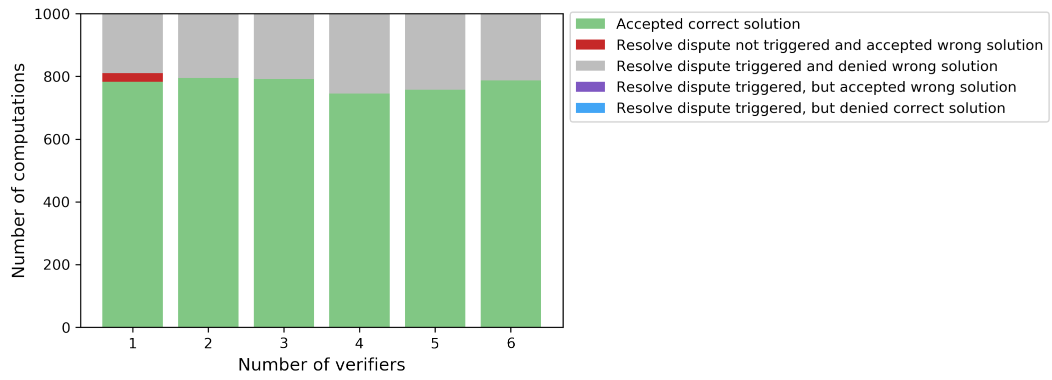


(b) 50% of computation services providing incorrect solutions.

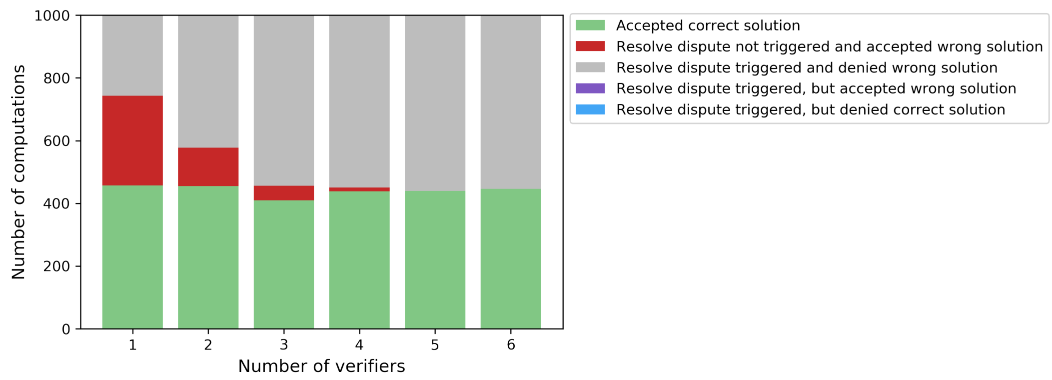


(c) 70% of computation services providing incorrect solutions.

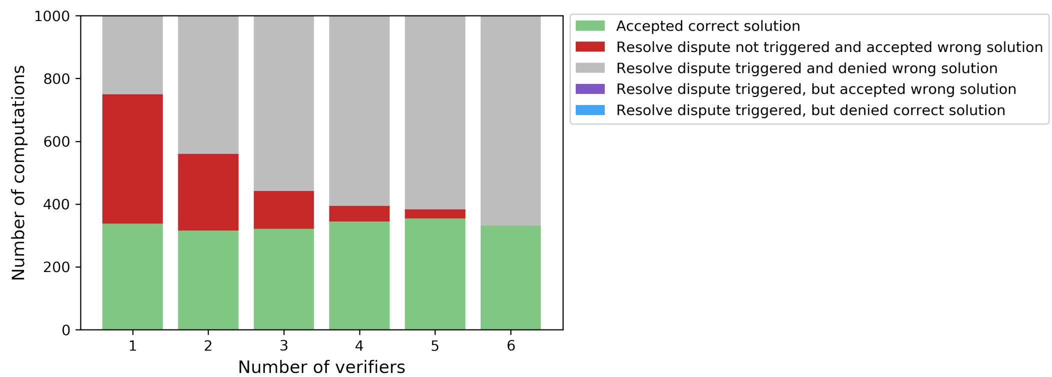
Figure 4.6: Total amount of gas used by algorithm with different number of verifiers and percentage of computation services providing incorrect solutions. Each combination of specific number of verifier(s) and percentage of computation services with incorrect solutions with $N = 1000$. *Median displayed as an orange line.*



(a) 30% of computation services providing incorrect solutions.



(b) 50% of computation services providing incorrect solutions.



(c) 70% of computation services providing incorrect solutions.

Figure 4.7: Results of computations with different number of verifiers and percentage of computation services providing incorrect solutions. Each combination of specific number of verifier(s) and percentage of computation services with incorrect solutions with $N = 1000$.

5 Chapter 5
Discussion

This chapter covers the discussion of the two research questions and their investigation in chapters 3 and 4. In the first section, the results from chapter 3 including the selected trust model and the findings from the quantitative analysis of Solidity smart contracts are discussed. The second section elaborates on the verifying computation algorithm proposed in chapter 4.

5.1 Trust model evaluation

The trust model is developed upon describing the ecosystem of smart contracts and developing a trust model under the assumption of smart contracts as agent systems. This section discusses the ecosystem description as well as the trust model itself.

5.1.1 Smart contract ecosystem

Section 3.1 introduces objects, attributes, and relations. It gives a basic definition of a system and elaborates which the different objects are and how they interact. The intention is to give the reader a simple overview. However, existing system theory and agent modelling techniques offer more detailed models. The model can be enhanced by including environmental factors influencing the objects in it [33]. The presented model of the system is in fact one specific way to view it. A more detailed system view might consider the distributed ledger, and all objects directly on it, as a core part of the system. With such an approach, objects could also be described with a state. The state of the blockchain depends on transactions and operations of users and smart contracts. Due to the consensus protocol, each state in the blockchain is deterministic. A formal description of the system consists of users and smart contracts as objects in a specific state. Transformations (i.e. transactions or calls to contract functions) are used to change the state of the system [19].

Also, the presented model is biased towards public permissionless blockchains. In permissioned blockchains, an authority has to grant access to objects on the blockchain [8]. Moreover, miners are not present in this type of blockchains as the consensus algorithm differs. The authority granting permissions is an agent that (1) is trusted to give the permissions, and (2), in turn, provides a certain level of trust to others by giving them the permission to interact on the blockchain. In established organisations or consortia the authorities typically already exist to grant access rights and permissions.

Permissioned blockchains can be perceived as part of the technology stack of an existing organisation or consortium, while permissionless blockchains are the enabler of new types of organisations. Consequently, permissioned blockchains are part of an existing organisational framework and technology stack, and, thus, are not necessarily at the centre of the organisational system. Additionally, established organisations might employ multiple different blockchains. For example, applying consensus algorithms depending on the use case is a way of balancing trust and tx/s. A weaker consensus algorithm allows a higher transaction rate as fewer parties need to agree. However, these few should be trusted to detect and disregard invalid transactions [52] [51].

The ecosystem can be modelled differently for other permissionless blockchains. In Bitcoin, the blockchain platform is more or less the distributed ledger and there is little separation between the two as it only implements one use case (i.e. cryptocurrency). Bitcoin offers a limited set of instructions to write applications on top of it, but it does not have a Turing-complete programming language. Moreover, the model in section 3.1 lacks possible extensions of blockchains. For example, *pegged sidechains* introduced in [91] enable fast transactions between entities. Upon closing this sidechain, the result of the sum of transactions is stored in the actual blockchain (e.g. Bitcoin).

Last, the presented model is a snapshot of the current ecosystem. In [92] Rao discusses the evolving of a new era of communication, whereby “machines” are able to communicate directly with each other on a common platform. [93] applies this idea to Ethereum and compares the ABI to a universal way of smart contracts communicating with each other. Hence, one can think of Ethereum as an enabler for decentralised autonomous organisations being able to communicate with each other by a standardised, enforced, and universal (at least within the blockchain) protocol. Combining this with an increase of smart contract pro-activeness and autonomy, it can lead to an extensive ecosystem, where autonomous organisations shape the system without human interaction [94]. This idea is further elaborated on the conclusion.

5.1.2 Trust model for smart contracts

Section 3.2 introduces a trust model for smart contracts in permissionless blockchains. Within the trust model, smart contracts are perceived as agents in an open MAS. The trust model considers agents to be rational, including cheating behaviours to increase their utility. As described by [13], a trust model covers three different aspects including security, institutional, and social approaches to steer agents' behaviour. In permissionless blockchains the security and institutional aspects are primarily covered by the blockchain platform implementing its protocol. As a result, the trust model for smart contracts in permissionless blockchains is based on [63] covering social controls including deposits, gossiping, and independent review agents.

Using deposits is used in other proposal in e.g. privacy preserving or verifiable computation protocols as presented in [95], [35], and [36]. Deposits are comparably simple to implement in permissionless blockchains, that already have a cryptocurrency in place. However, the deposit value can be volatile based on the cryptocurrency itself. This poses two risks: Either the escrow or independent entity maintaining the deposit may be motivated to steal the deposits based on its value, or the deposit value might be so little that its trust-building attribute vanishes. To prevent this, the deposit value could be bound to a fiat currency or a stable asset. Otherwise, the deposit can also be dynamically adjusted and deposits only kept for e.g. a short or one iteration of interactions (i.e. for the algorithm presented in chapter 4).

Gossiping could be used as a basis to communicate experiences with other agents. In permissionless blockchains, the agents can use a common protocol to exchange this information and use an approach as presented in [96] to rate reputations. Zhou, Hwang, and Cai propose a protocol for scoring and ranking in unstructured P2P networks. Yet, gossiping can be misused by agents to boost their own reputations. As creating new agents in permissionless blockchains is fairly simple, an adversary could boost his own reputation by deploying new agents vouching for his reputation. To circumvent this, either correlation analysis of transactions or outside information is required as elaborated on [5]. Thus, based on the origin of transactions i.e. which user created which smart contracts and the interaction between agents, a reputation model can be built.

Independent review agents are used in chapter 4 as well as by [35] in the form of judges. This is a simplified version of reviews, where the agents do not keep track of the entirety of the network or trust of individual agents. Rather, they reach a verdict on a specific issue or problem. Thus, their implementation is simple and potential scenarios to manipulate agents' reputations are prevented. However, the judge or review agent needs to be trusted by other agents.

Within blockchains, agents need to assess information and make decisions under uncertainty. At the same time, the blockchain platform enforces social norms [61] in

the system. They prevent double-spending or tampering with the distributed ledger. However, this still leaves room for interactions with a high degree of uncertainty. To cope with it and to predict agents' behaviour, game-theoretic methods can be used. Thereby, agents want to optimise their utility while being constraint by the rules set by the blockchain protocol. To control interactions within these constraints, algorithms need to ensure that they consider all different strategies an agent or other entity (like miners) have. This also includes the underlying assumptions made as well as the possible interactions of an agent. However, assumptions can change or not hold in reality. Hence, it is cumbersome to predict an agent's behaviour, in particular in complex scenarios. Therefore, a thorough trust model for permissionless blockchains needs to employ methods of social control such as deposits and review agents, while focussing on assumptions and agent's possible strategies.

A trust model for permissioned blockchains needs to account for the assumption differences. As these enable restrictive access and employ authorities granting identities and access rights, there exists some prior trust between agents. Entities giving access could for example be used as independent review agents keeping track of the reputation. Agents' access right can then be adjusted based on their behaviour and reputation. Also, agents manipulating their reputation is more difficult, since new agents cannot freely join or leave the system. However, deposits are harder to implement as permissoned blockchains typically do not employ a cryptocurrency[1]. Moreover, the prior trust in agents in the system depends on the permissiveness of the blockchain.

5.1.3 Trust analysis for smart contracts

In section 3.3, trust-related implications of smart contracts are quantified. Specifically, 2561 Ethereum smart contracts written in Solidity and hosted on GitHub are analysed. The analysis covers the distribution of smart contracts in different repositories, the Solidity version, code complexity, and security-related issues. Using GitHub as a source to compile smart contracts offers the benefit of having the source code available for analysis. However, those contracts might never be deployed in Ethereum. Alternatively, the EVM bytecode could be used as in [77] and [6]. In both papers, the authors collected smart contracts directly from the blockchain and conducted their analysis.

The contracts covered in section 3.3 are not checked for duplicates or similarity. Anderson et al. discovered that from 19,528 contracts they analysed 309 are exact copies and 2937 are very similar measured based on the Levenshtein distance of the EVM bytecode [6]. Potentially, on GitHub this similarity or copying is higher since there is no cost involved in forking or copying contract code. Hence, the

[1] Ripple is an exception as it is a permissioned blockchain with its own cryptocurrency.

unique number of contracts is likely lower than the total number of contracts in the analysis.

During the work on this thesis, from beginning of November 2016 to May 2017, Solidity was updated from version 0.4.4 to 0.4.11 [2]. The fast development is reflected in the number of different Solidity versions found on GitHub as shown in figure 3.2b. Two observations can be made: First, the oldest version in the dataset is Solidity 0.4.0, and second, certain versions including 0.4.0, 0.4.2, 0.4.4, 0.4.6, and 0.4.8 are more common than others. One of the reasons why no version prior to 0.4.0 is available in the dataset could be caused by the fact the Solidity version did not have to be included prior to 0.4.0. However, all 2561 contracts include the version number. Another reason, could be that version 0.4.0 introduced breaking changes making older Solidity code files incompatible with the EVM. Hence, developers are required to use 0.4.0 or later. Most likely this caused developers to update their contracts as the compiler released on 8 September 2016 would not accept any Solidity version prior to 0.4.0.

The frequency variation of different Solidity version is potentially caused by developers using frameworks to create smart contracts. The two most used ones, Truffle [3] and Embark [4], support different Solidity versions. They need to update their code base in order to support new Solidity versions. The release between Solidity version 0.4.1 and 0.4.2 are eight days, between 0.4.3 and 0.4.4 six days, between 0.4.5 and 0.4.6 one day, between 0.4.7 and 0.4.8 29 days. For versions 0.4.2 through 0.4.6 the time to apply the new updates might have taken longer than the new release of Solidity, so the Truffle and Embark developers might have decided to directly switch to the latest version and not support in between versions. However, between 0.4.7 and 0.4.8 is a considerable amount to time. Apart from Holiday season between the two release dates, the author does not have an explanation.

The number of declarations show outliers with one contract having around 200 variables and functions. The majority of the contracts have a mean of three variables and two to four functions as presented in figure 3.3. This represents a low complexity of the source code. Likely the majority of the contracts are used for trying out Solidity and Ethereum. This is supported by the low average number of imports and libraries. While smart contracts can have low code complexity, they can be combined together using imports of other contracts and libraries. However, this feature is rarely used. Moreover, seldom use of structs and enums for state variables indicate simple models of data in the set of smart contracts. A recommended way to store more complex data-models is to define a struct and access it either through arrays or mappings. Events used for controlling and debugging smart contracts are also fairly rare. This supports the hypothesis that the majority of contracts are for experimenting with Solidity. Additionally, a more detailed analysis of the functions

[2] https://github.com/ethereum/solidity/blob/develop/Changelog.md (visited on 22/05/2017)
[3] http://truffleframework.com (visited on 22/05/2017)
[4] https://github.com/iurimatias/embark-framework (visited on 22/05/2017)

could be done. This can include the number and type of parameters and returns as well as *modifiers* used. Modifiers are a way to apply certain checks before executing a function. For example, a modifier could be defined to check if the sender of the call or transaction is the owner of the contract. Instead of defining this piece of code for every function individually, a modifier could be defined and applied to functions. Also, the fixed “payable” modifier needs to be set for functions accepting payments in ether. It can be analysed which contracts accept payment and which functions are responsible for receiving payments.

The Oyente tool presented in [77] is used to analyse security-related issues in the 2561 smart contracts as presented in 3.4. Callstack and re-entrancy are the two most common issues found in the set of smart contracts. The majority of smart contracts are not showing any of the four tested security-related issues. However, there is a chance that security issues arise from the smart contracts, that are not tested through the Oyente tool. Hence, an automatic checking of the smart contracts alone is not sufficient to ensure the smart contract implements the intended functionality. Moreover, the contracts showing security-related issues are not checked for false positives. In [77], the authors recognise that Oyente can produce false positives and the source code needs to be further analysed. Their reported false positive rate is at 6.4%, which needs to be verified as part of future work for the smart contracts from GitHub as well.

A critical aspect of trusting smart contracts concerns their implementation. If their implementation defers from their intention, it can have severe consequences. The DAO is an infamous example of smart contracts not being implemented as intended, leading to a loss of around USD 150 million [38]. In the design of The DAO, a re-entrancy was possible where users were able to extract funds recursively before their balance in the state of the contract was updated. Ethereum supports a cryptocurrency and thus, the incentive to exploit such implementation flaws is considerable. Moreover, the immutable nature of smart contracts makes it impossible to quickly update or fix such issues. The problem leads to two questions: What are the consequences of having issues in smart contracts, and how can those issues be prevented? The first question is discussed in chapter 6 of this thesis as it covers the understanding of code in blockchains. The second question is primarily concerned with ensuring that implementation follows the intention of the author. First, [97] presents common issues and design best practices for the creation of smart contracts. Developers need to be aware of the immutability of smart contracts and plan ahead for safety features, such as contract suicide functions. Second, formal verification methods as stated in [86] and [85] are aiming to mathematically proof the implementation of smart contracts according to a specification. Similar to critical infrastructure or military applications, the functions of a smart contract can be abstractly defined and their implementation verified by using mathematical proofs. This can be simplified by using a theorem proofer such as Lem or Isabelle/HOL. However, a complete proof of a smart contract requires a considerable amount of work and is as of May 2017 not yet a common practice [86].

5.2 Verifying computation evaluation

This section discusses the algorithm introduced in chapter 4. It is structured to reflect chapter 4 such that actors, assumptions, algorithm, implementation, and last, experiments are critically analysed. The assessment focusses on how the algorithm achieves the objectives stated in section 4.1.1.

5.2.1 Actors and assumptions

The algorithm is based on its actors and their interaction. The idea of arbiter, judge, user, and computation services is strongly influenced by [35] and [36]. The main differences between the presented algorithm and [35] are in the idea of using a jackpot to reward verifiers as well as the implementation either entirely on Ethereum or using external computation services. Moreover, the algorithm defers from [36] as its goal is to deliver verifiable computations for entities (i.e. users or smart contracts) on the blockchain, while Zyskind primarily delivers privacy preserving computations, where blockchain provides useful technical characteristics enabling his algorithm. A justification for the design differences is given in section 4.1.1.

Moreover, there is an economic threshold to using verifying computation services. An oracle introduces a fixed fee for using the service. For example, oraclize requires to send the ether equivalent of USD 0.01 with every query [84]. Users may test the gas consumption and the assumed gas price using for example the Remix IDE [98]. Thereby, they determine which computations are economically feasible to outsource to a verifiable computation solution. Computations that are cheaper to execute in a smart contract itself, should be executed there.

Miners also need to be considered as actors of the algorithm due to the restrictions in creating random numbers. Ethereum is a deterministic system as every node in the network needs to be able to verify each transaction and function. Therefore, the results of transactions or function calls depend distinctly on the inputs. However, this introduces an issue in the creation of random numbers in Ethereum. As Ethereum is deterministic "true" randomness cannot be achieved. Functions creating random numbers, need to create the same pseudo-random number during each verification step of a node. To achieve pseudo-random numbers, values provided by the blockchain itself like the block hash, timestamp, or nonce can be used. Miners are able to influence these values, and hence, can influence the creation of pseudo-random numbers [99]. A miner receives five ether for every new block it finds. If a miner is able to influence the algorithm, he might adjust the value to influence the randomness. Thereby, a miner would be able to influence the selection of solvers and verifiers according to his preference. A miner might run the risk of another miner finding a suitable block, while it decides to change a value (like timestamp or withholding a block). In that case, it looses its mining work as the other miner proposes the new block and other nodes verify it. Hence, the incentive the miner can gain needs to

be more than five Ether, otherwise the miner is better off proposing the new block. Therefore, the overall incentive for solver and verifiers combined should not exceed five Ether. The implementation of pseudo-random numbers is further discussed in section 5.2.3.

Miners also need to be considered because of their validation work. [100] introduces the *Verifier's dilemma*. Rational miners have an incentive to accept not validated blockchains, if the computational effort for validating blocks is not trivial. This opens up consensus algorithms to attacks where 38.2% of the miners can influence the majority of the system. Hence, the algorithm outsources the complex computations outside the blockchain and simplifies the judgement on results. As such, miners do not gain an incentive for not validating the computations part of the algorithm.

The algorithm is further based on the assumption of cheaters in the system. [101] lists and categorises a total of seven cheating techniques that can be employed by agents. First, the user might get cheated on after submitting the fee for the computation by not receiving any result (or status update). To ensure the user is properly refunded, the fee could be stored in an escrow by arbiter and judge. Only upon finishing the computation the fees are distributed.

Second, computation services might provide correct solutions on simple computations, but provide false results on complex ones. In such cases, they save up on computation cost, while relatively maintaining their reputation. However, as no reputation system is employed, computation services are penalised independent of the fee provided. Moreover, the deposit that needs to be provided, needs to exceed the maximum possible fee to prevent this as a profitable strategy.

Third, computation services and users can enter freely into the blockchain. Thus, cheaters might reappear in new accounts. For computation services, this is prevented by deposits. Yet, users can also provide bogus challenges, wasting computation resources. To prevent this from happening, computation services could receive the choice to refuse certain computations as described in [35].

Fourth, in an initial state no trust between agents exists. Hence, users cannot know which agent to trust. This can be resolved by the formal verification of arbiter and judge, as well as the deposits and incentive structure of the algorithm.

Fifth, a computation service planning to exit the system might cheat, because there is no reputation to lose. This again, is prevented by the deposit. However, combining this with the volatility of cryptocurrencies might impose the problem, that the gain through not providing a correct solution outweighs the potential punishment of burning the deposit.

Sixth, multiple agents might collude to maximise their utility. This is the case, when all computation services agree to provide the same arbitrary solution to a

computation request. Thereby, they save the cost for computation and receive their share of the fee. Moreover, if the judge is colluding with them, even a truthful verifier, might be deemed wrong. Thus, previously mentioned methods i.e. formal verification, deposits, and incentive fees are crucial to prevent this behaviour.

Seventh, cheating computation services might multiply themselves to increase their prior probability of being chosen. In the current implementation, a computation service might be deployed numerously with forwarding computations to the same AWS Lambda API. As shown in section 4.2, the result of the outcome depends on the prior probability of cheaters. Hence, an adversary could provide a large percentage of cheating services with a probability that its false results are accepted. The adversary could check if all computation services, in a particular task, belong to it. In that case, it could just provide the same false answer while receiving the incentives.

The algorithm further assumes that arbiter and judge are trusted. This might not hold in practice as to the users the entities are unknown and no deposits are provided. A way to create this form of trust, can be formal verification as explained in section 5.1.3. Both, arbiter and judge can publish a proof through a smart contract function so that a user is able to verify their implementation according to the algorithm specification. We leave this for future work.

5.2.2 Algorithm

In the theoretic design of the algorithm, the likelihood to accept a correct result depends on the prior probability of computation services providing correct services. However, the algorithm does not consider multiple rounds of execution. Potentially, the prior probability can be influenced by excluding computation services that are detected of cheating. Moreover, if the deposits of these computation services are "burned", it might reduce the incentive to cheat for other computation services as well. That is, they need to account for penalising of undesired behaviour and thus, might adjust their utility functions. Moreover, the number of false accepted results can be reduced by the same means.

The algorithm cannot guarantee to detect false solutions. It is based on the assumption that solvers and verifiers behave as desired (i.e. delivering correct solutions), as their strategy is aligned with the incentives provided by the algorithm. This assumption is based on game-theoretic properties. Agent decision making utilises primarily four different concepts: dominant strategies, Nash equilibria, Pareto efficiency, and social welfare [34, pp. 229-235].

Dominant strategy and Nash equilibria can serve as solutions to agent decision problems. A dominant strategy for an agent is one that gives this agent the highest utility independent of the other agents' strategies. A Nash equilibrium is defined by two agents i and j choosing a strategy that is the best response to each others'

strategy. For example, if agent i plays strategy S_1 then agent j is best of playing strategy S_2 and vice versa. However, in practice a Nash equilibrium might not exist, as agents make decisions under uncertainty and cannot know the probability of actions by other agents. Also, multiple Nash equilibria might exist.

Applying those two concepts to the algorithm leaves no dominant strategy considering the interactions in table 4.2 and description in section 4.1.5. S can choose either to provide a correct or false solution and V can challenge or accept. Only when considering both agents, a Nash equilibrium exists. If there is a (high) probability that a V_C exists, the only valid strategy for S is to provide a correct solution. Consequently, V in turn has to provide a correct solution, which accepts correct S and challenges false S.

Pareto efficiency and social welfare are properties of a decision's outcome. Pareto efficiency describes a solution (i.e. choice of strategy by all agents involved) where an agent's utility cannot be increased without decreasing another agent's utility.

In the algorithm, both S and V providing correct solutions gives a Pareto efficient result. If they change their strategy under the assumption that no V_C exists, their utility remains the same. However, a V has an incentive to challenge a false solution, which would increase his utility and reduce the others utility. Social welfare in turn considers the sum of all agent's utilities depending on their strategy. Social welfare can be disregarded in permissionless blockchains, since overall the agent wants to optimise his utility independent of the overall utility. Specifically, the overall utility is potentially unknown to an individual agent, since he is unable to determine with certainty the utility of other agents.

Moreover, the algorithm depends on the verdict of the judge. Simplified judges are an efficient way to reach a decision in case of a dispute between agents [102]. To ensure that the judge is implemented correctly formal verification as elaborated earlier can be used. In case a judge's verdict is incorrect, the algorithm fails to detect correct and false solutions. This leads to penalising and rewarding the wrong agents and, potentially, to removing agents providing correct solutions.

Within the algorithm, computation services might drop out. This case is currently not handled in the algorithm, but can be approached as follows. If computation services have not yet been assigned as solver or verifiers for a current computation, they can replace the dropped out ones. This can be repeated until the gas provided by the user is finished. When the gas limit is reached, the computation is aborted, since the probability of detecting false results is influenced by the number of verifiers. Otherwise, the computation can continue as planned. If computation services drop out during the dispute resolution phase, their deposits are burned to penalise their behaviour. Drop out caused by network or infrastructure issues are highly unlikely due to the P2P and serverless architecture of Ethereum.

Last, the algorithm description leaves the exact fee for a computation open. The fee for the computation is supposed to cover the cost of using the oracle service as well as the computation cost for a computation service. The fee is therefore determined by the work of computation services and their implementation. It should not exceed five ether due to the potential influence of miners on the process. The exact fee could further be realised by an arbiter setting a fixed fee or utilising auction protocols. However, this issue is left for future work.

5.2.3 Implementation

The implementation of the algorithm is based on smart contracts in Solidity, using oraclize as an Orcale service, and AWS Lambda to implement the computation services. The design of the implementation is based on having an abstract arbiter and abstract computation service contract to allow changes and individual implementations, but keeping the ABI constant. Different operations e.g. integer multiplication, matrix multiplication, or matrix inverse can be implemented as different operation types. The operation types are characterised by their query URL (i.e. to the respective AWS Lambda or other service) and their dispute resolution protocol. An author of a computation service needs to implement the handling of pointing out indexes and providing intermediary results. Therefore, each type of computation requires its own custom implementation. This provides a common framework for the computation and resolution. However, it requires creating a custom implementation for each type of computation.

Moreover, the algorithm is only suitable for a set of deterministic computations. A dispute resolution for results reported by an application which relies on randomness and probabilities are cumbersome to implement. For example, a verification of a result delivered by a convolutional neural network on image classification would require the comparison of initialisation weights and numerous intermediary results. However, mostly initial weights are chosen at random and depending on the depths of the network, the required dispute resolution steps might consume too much gas for the dispute resolution to finish.

As previously mentioned, randomness in Ethereum is a challenge. Code excerpt 5.1 presents the function used in the arbiter contract. The input values are used to determine in which range to choose a random number. This reflects the available computation services minus the ones already assigned as solver or verifiers. The random number depends on the block number and hash of the longest block in the chain (i.e. *block.number - 1*). However, this sort of random number creation introduces the influence of miners as discussed previously. If a miner can overall profit from withholding a block, it will likely influence the random number. There are two approaches to circumvent the problem [99] [103]. First, oracles can be used to generate random numbers outside the blockchain and write them back into it.

However, this introduces a high delay and a relatively high cost. Second, a *RANDAO*[5] is an autonomous organisation creating random numbers in Ethereum. It is based on users sending transactions with hash values and determines the random number based on those hashes. Thereby, it excludes the miners from influencing the random number process. However, it requires a certain number of blocks to generate the number and is therefore slower and also involves a higher cost.

```
1 function rand(uint min, uint max) internal constant returns (uint256
     random) {
2   uint256 blockValue = uint256(block.blockhash(block.number-1));
3   random = uint256(uint256(blockValue)%(min+max));
4   return random;
5 }
```

Code excerpt 5.1: Random number generator in Solidity based on block number and hash.

The objective of the algorithm is to minimise any kind of trust requirements in the system. Using oracles requires a certain level of trust, since the oracle could also manipulate the solution provided by the computation service. Oraclize offers a *TLS Notary proof* to verify that the result written to the blockchain confirms to the information delivered by the API or website that was queried. However, this comes with an extra cost and moreover, one needs to trust the correct implementation of the proof. In [35], the authors circumvent this issue. The computation is written as C, C++, or Rust code and complied as well as executed on the blockchain. Overall, this limits the flexibility of the computation services, but reduces the trust requirements. The presented algorithm utilises oracles to reduce the implementation requirements and provide a more flexible framework.

5.2.4 Experiments

The experiments in section 4.2 cover the total execution time, gas consumption, and result of algorithm. It uses a simple computation, namely multiplication of two integers. The dispute resolution for an integer multiplication requires only one step i.e. the judge multiplying the two integers and comparing this to the initial solver's result. Thus, the execution time and the gas consumption reported in figures 4.5 and 4.6 are on the low end. With an increasing number of resolution steps, time and gas consumption are higher. However, both values indicate a linear time and space complexity of the algorithm. Therefore, with an increased complexity of dispute resolution the time and gas consumption of the overall protocol in relation to the number of verifiers grows linear. This would further apply to other computations, as long as it can use Merkle trees and binary search to limit the additional dispute resolution steps to $\mathcal{O}(n)$.

An issue not shown in the experiments is verification steps that require too much gas. The user triggers the dispute resolution protocol by calling a function in the arbiter.

[5]https://github.com/randao/randao (visited on 23/05/2017)

Ethereum has an upper gas limit of 4.7 million that can be sent to any function. If the gas sent exceeds the maximum value an error is thrown and the transaction or call is not executed. Similarly, if the gas required by the function exceeds the delivered gas, the function will terminate prematurely with an out of gas error. Thus, the overall dispute resolution gas consumption of any computation cannot exceed the upper gas limit set by Ethereum. Figure 4.6 shows that the most expensive dispute resolution with six verifiers consumes a total of 1.5 million gas. The dispute resolution function consumes around 0.5 million gas, hence it is considerably far from the upper limit of 4.7 million. Moreover, the prior probability of cheaters seem to not affect the overall gas consumption.

The result of the algorithm as presented in figure 4.7 shows that it detects false solutions and that the judge, if invoked, rules correctly. However, invoking the dispute resolution depends on the prior probability of computation services providing false solutions. The experiments are conducted with 30%, 50%, and 70% prior p. The probability of having at least one verifier challenging the solver is determined by the number of verifiers n in the computation, where the probability is described by $P(V_C) = 1 - p^n$. Thus, the probability of accepting a false result is described by the prior probability of having a false result (i.e. p) times not detecting the false result (i.e. $1 - P(V_C)$). Table 5.1 compares the theoretical expected number of false accepted results to the actual false accepted results from the experiments in section 4.2. The experiments show that the expected and actual value are similar for $p = 0.5$. However, for $p = 0.3$ and $p = 0.7$ the actual values are below the expected ones. Since the experiment is executed with a confidence level of 95% and interval of 3.1, those changes are accounted towards sampling size not being a perfect representative for the actual distribution. Moreover, the random assignment of false and correct computation services in the JavaScript tests could be a cause for having a higher detection rate.

Table 5.1: Comparison of expected and actual probabilities of accepting a false solution in the algorithm.

Prior p	**Verifiers** n	**Expected false [%]**	**Actual false [%]**
0.3	1	9.0	2.7
0.3	2	2.7	0.0
0.3	3	0.81	0.0
0.3	4	0.243	0.0
0.3	5	0.0729	0.0
0.3	6	0.02187	0.0
0.5	1	25.0	28.6
0.5	2	12.5	12.2
0.5	3	6.25	4.6
0.5	4	3.125	1.2
0.5	5	1.5625	0.0
0.5	6	0.78125	0.0
0.7	1	49.0	41.2
0.7	2	34.3	24.4
0.7	3	24.01	12.1
0.7	4	16.807	4.9
0.7	5	11.7649	2.9
0.7	6	8.23543	0.0

6 Chapter 6 Conclusion

This chapter summarizes the key findings as well as the critical evaluation in chapters 3, 4, and 5. Moreover, the major contributions of this project are outlined and potential future work is described. Last, an outlook of future development is presented.

6.1 Summary of findings

This thesis elaborates on trust and verifiable computations for smart contracts with a focus on permissionless blockchains. The purpose is to answer two research questions as stated in section 1.3.

RQ1: Which models of trust can be applied to smart contract ecosystems to reflect public permissionless blockchains?
RQ2: How can computations be verified in permissionless blockchains utilising models of trust?

Investigation of the first research question is covered in chapter 3. The **ecosystem** of smart contracts is described with its objects, trust-related attributes, and relations. The core of the system consists of a blockchain platform, blockchain protocol, distributed ledger, smart contracts, users, nodes, miners, and a developer community. Users and smart contracts can either directly interact through function calls, or transactions that are stored in the distributed ledger. The blockchain platform implements its protocol and thereby enforces the rules of the system.

In the **trust model**, smart contracts are perceived as rational agents in an open MAS. They are capable of reactive and, to a certain extent, autonomous and proactive behaviour. However, their intentions are not necessarily known and agents have to be able to make decisions under uncertainty. Trust and reputation research for MAS focusses on three aspects: security, institutional, and social approaches [13]. The security and institutional aspects are enforced by the blockchain through cryptography and consensus mechanisms. It ensures that valid transactions are

immutably stored. However, further interactions by smart contracts or users are not covered by these measures. As permissionless blockchains like Ethereum employ a cryptocurrency, rational agents are motivated to maximise their utility by honest and dishonest behaviour. Hence, further social control aspects including deposits, gossiping, and independent review agents as introduced by the model in [63] are required. In permissionless blockchains especially deposits are an effective control measure, since no prior trust needs to be established and the access to cryptocurrency allows implementation through already available measures. Next, the quantitative analysis of trust-related attributes in smart contracts show that Solidity contracts hosted on GitHub are: (1) mostly still in a try-out phase with a low code complexity indicated by few declarations in the source code, and (2) security-related issues in around one in every ten smart contracts exists.

The second research question is explored in chapter 4. Trust can be extended to entities outside of permissionless blockchains through an algorithm implementing **verifiable computation**. The algorithm uses deposits and review agents as social control from chapter 3. Moreover, it employs a game-theoretic setting for participating agents. It includes users requesting computational tasks, computational services providing solutions and acting either as solver or verifier, arbiter enforcing the algorithm, and a judge resolving disputes. Due to the incentive structure and the potential penalty cause by cheating, providing correct solutions to the computation task is a Nash equilibrium (as elaborated on section 5.2.2). Under the assumption that arbiter and judge are trusted, the algorithm detects false solutions provided based on a probability distribution. The algorithm is realised as Solidity smart contracts and AWS Lambda functions, implementing verification of multiplying two integers. Experiments show that with six verifiers the algorithm detects all cheaters with prior probabilities of 30%, 50%, and 70% dishonest computation services. Moreover, the experiments show that the algorithm performs overall with a linear time and space complexity depending on the number of verifiers.

6.2 Main contributions

The main contributions of this thesis revolve around smart contract ecosystems, trust models, and verifiable computations. First, the ecosystem of smart contracts in permissionless blockchains is described including their objects, attributes, and relations.

Second, from 65 trust models presented in [13] a trust model for smart contracts in permissionless blockchains is deducted. The model extends [63] and is detailed by further research in the field on deposits, reputation, and review agents.

Third, 2,561 Solidity smart contracts are analysed for trust-related attributes. This provides an insight in the current development status regarding code complexity

and security related issues. It includes a reusable crawler utilising a greedy algorithm to receive code files from GitHub. Also, the dataset of 2561 smart contracts including source code, analysed code, and meta-data is available for further analysis.

Fourth, an algorithm to allow verifiable computation in permissionless blockchains is presented and evaluated. It is implemented as Solidity smart contracts and AWS Lambda functions to use it for integer multiplications. The implementation is designed to allow extensibility of the algorithm for other computations. All code written throughout this thesis is publicly available GitHub[1].

6.3 Future work

Among others, four main focusses of future work are identified. Gossiping in permissionless blockchains could be a viable option to evaluate the trustworthiness of agents and use it as part of the trust model in chapter 3. Since the distributed ledger allows a transparent view of previous transactions, a formal model can be developed based on this information. This could be used for arbitrary protocols or algorithms involving both smart contracts and users. It can also serve as a selection criteria for the arbiter in the algorithm introduced in chapter 4. The random selection can be adjusted as such that computation services with previously successful computations are assigned at a higher likelihood.

Moreover, extending the computation algorithm to support more computations and testing it on the test or live network of Ethereum is necessary to confirm the assumptions and interaction scenarios depicted in chapter 4. Thereby, the test should monitor the number of computation services available and their behaviour. The solution proposed by Teutsch and Reitwießner is supposed to be implemented within 2017 [35]. A quantitative experimental comparison of the here proposed algorithm and their TrueBit solution would allow identifying differences in the incentive structure, behaviours of agents, and complexity of the algorithms. Also, this can include experiments to show the upper computational complexity of computations, that do not incentivise miners.

Eliminating trust towards arbiter and judge requires either a fully decentralised algorithm or formal verification. Formal verification of arbiter and judge requires the specification of the implementation presented in section 4.2 in a theorem prover. [104] presents an example of an Ethereum smart contract verification process. However, as [86] points out, the methodology of verifying smart contracts is still a work in progress. It requires the exact replication of the EVM, otherwise the verification

[1] A list of all repositories is available at: https://github.com/nud3l/TrustedSmartContracts (visited on 05/06/2017)

process would not yield a valid result. Also, the algorithm could be re-designed to be fully decentralised without trusted entities.

Last, finding a method to determine the computation fee is left for future research. Depending on who sets the fee (i.e. arbiter, user, computation service), the strategies of the different agents can be affected. Users will naturally try to minimise the fee, while computation services try to maximise their profits. Hence, auctions could be an option to find a price both parties agree on. However, there is a variation of auction protocols, each with their own advantages and disadvantages [34].

6.4 Outlook

Chapter 1 opens with the question on how computer science can contribute to providing trust. The continuing acceptance of cryptocurrencies, shows that trusts exists in decentralised economies and currencies enforced by a protocol rather than traditional institutions [2]. Price increases of those currencies opens up the door for speculation on its value, rather than using it as a means of payment. However, having a system, which does not require trust, minimises potential malicious activities from the perspective of the protocol. While Bitcoin is used for buying illegal goods and has a notorious reputation, its basic design introduced an opportunity to showcase P2P payment systems preventing double-spending and storing transactions immutably. This decentralised architecture later on adapted by Ethereum and other blockchains, makes the system tamper-proof and resilient against influences from single or colluding parties.

This immutability introduces new challenges. In [30], Lessig argues whether code should be treated as law. Assuming a smart contract is not “just an application”, but the equivalent of a contract in a legal sense, then how are flaws in the contract treated? Ideally, an application receives a fix of the flaw after it is discovered. However, legal contracts cannot be altered in the sense of an application. Rather, a new contract needs to replace the existing one or adjustments need to be made with participating parties agreeing mutually on these changes. Courts decide whether a contract was legally binding in the first place. Yet, Ethereum does not have courts that would be able or have the authority to reach verdicts. The case of The DAO led to the split of the Ethereum project into Etheruem and Ethereum Classic, because of a debate how to treat smart contracts. Ethereum supporters argue that the application was not implemented according to its specification and thus, users that are damaged, should be refunded and the blockchain platform modified to prevent further incidents.

[2]During work on this thesis from 1 November 2016 to 24 May 2017, the Bitcoin and Ethereum market capitalisation has risen from USD 11 billion to USD 39 billion and USD 924 million to USD 18.6 billion, respectively. See https://coinmarketcap.com/currencies/bitcoin/ and https://coinmarketcap.com/currencies/ethereum/ (visited on 24/05/2017)

Ethereum Classic followers contrary pointed out that since it should be treated as a legal contract, the contract is binding and the damage done should not be reverted. Users should be aware of flaws as they would be with other legal contracts. As of May 2017, no regulations regarding the legal applicability of smart contracts in the EU exist. However, the state of Arizona has declared that they will treat smart contracts as legal contracts[3].

Blockchain is characterised by decentralisation and trust enforcement through technology. The here presented model of trust can be applied to a variety of permissionless blockchains. Thus, it can help others understand the nature and control measures to develop protocols or algorithms on top of these platforms. Security of these protocols and algorithms is crucial due to the immutability and high impact of faults[4]. Potentially everyone with the ability to program a smart contract can become a developer working with cryptocurrencies at a large scale [105]. Therefore, secure coding techniques or formal verification methods are required as well as a consideration of agent interactions through game theory. Buterin and others proposed the emergence of a specialised field called “cryptoeconomics” focussing on combining computer science (i.e. cryptography, consensus, agents), and economics (i.e. game theory) to research those aspects [58]. Ethereum continues to develop[5], while the number of new decentralised autonomous organisations grows with it [6]. With an increasing ability of smart contracts (i.e. inclusion of pro-activeness), trust models need to be revised and the decision making capabilities of agents can be greatly increased. In the future this may lead to truly autonomous organisations, capable of complex behaviours and interactions on a homogeneous and decentralised platform.

[3] http://www.trustnodes.com/2017/04/03/arizona-gives-legal-status-blockchain-based-smart-contracts (visited on 24/05/2017)

[4] An exchange for cryptocurrencies implemented a faulty split function in an Ethereum contract resulting in an equivalent of USD 14.7 million worth of Ether being trapped inside the contract. See: http://www.coindesk.com/ethereum-smart-contract-exchange-14-million/ (visited on 04/06/2017)

[5] https://github.com/ethereum/EIPs (visited on 23/05/2017)

[6] Overview of start-ups seeking funding based on issuing their own tokens: https://www.icoalert.com (visited on 24/05/2017)

References

[1] Antoine Harary, David M. Bersoff, Sarah Adkins, et al. *2017 Edelman Trust Barometer*. Tech. rep. 17. Edelman, 2017.

[2] Kevin Werbach. “Trustless Trust”. 2016.

[3] The Economist. *The promise of the blockchain: The trust machine*. 2015. URL: http://www.economist.com/news/leaders/21677198-technology-behind-bitcoin-could-transform-how-economy-works-trust-machine (visited on 02/17/2017).

[4] Satoshi Nakamoto. *Bitcoin: A peer-to-peer electronic cash system*. 2008.

[5] Fergal Reid and Martin Harrigan. “An Analysis of Anonymity in the Bitcoin System”. In: *Security and Privacy in Social Networks*. New York, NY, USA: Springer New York, 2013, pp. 197–223.

[6] Luke Anderson, Ralph Holz, Alexander Ponomarev, et al. “New kids on the block: an analysis of modern blockchains”. June 2016. URL: http://arxiv.org/abs/1606.06530.

[7] Vitalik Buterin. *A Next-Generation Smart Contract and Decentralized Application Platform*. 2013. URL: https://github.com/ethereum/wiki/wiki/White-Paper (visited on 10/09/2016).

[8] Tim Swanson. “Consensus-as-a-service: a brief report on the emergence of permissioned, distributed ledger systems”. 2015. URL: http://www.ofnumbers.com/wp-content/uploads/2015/04/Permissioned-distributed-ledgers.pdf.

[9] Jamie Burke. *6 Things We Learned from Analysing 227 Corporates in the Blockchain Ecosystem*. 2016. URL: https://www.linkedin.com/pulse/5-things-we-learned-from-analysing-227-corporates-blockchain-burke?trk=hp-feed-article-title-share (visited on 12/12/2016).

[10] Project Provenance Ltd. *Provenance | Building trust in great businesses and products*. 2016. URL: https://www.provenance.org/ (visited on 12/01/2016).

[11] Colony. *The Future of Work*. 2015. URL: https://blog.colony.io/the-future-of-work-cf99211e7ac4%7B%5C#%7D.nsm29c621 (visited on 12/01/2016).

[12] Denise M. Rousseau, Sim B. Sitkin, Ronald S. Burt, et al. “Not So Different After All: A Cross-discipline View of Trust”. In: *Academy of Management Review* 23.3 (July 1998), pp. 393–404.

[13] Isaac Pinyol and Jordi Sabater-Mir. “Computational trust and reputation models for open multi-agent systems: A review”. In: *Artificial Intelligence Review* 40.1 (2013), pp. 1–25.

[14] Babak Esfandiari and Sanjay Chandrasekharan. “On how agents make friends: Mechanisms for trust acquisition”. In: *Proceedings of the Fourth Workshop on Deception, Fraud and Trust in Agent Societies*. Vol. 222. 19 June. Montreal, Canada, 2001, pp. 27–34.

[15] Diego Gambetta. “Can we trust trust?” In: *Trust: Making and breaking cooperative relations* 13 (2000), pp. 213–237.

[16] Alfarez Abdul-Rahman and Stephen Hailes. “Supporting trust in virtual communities”. In: *Proceedings of the 33rd Hawaii International Conference on System Sciences - Volume 6*. Washington, DC, USA: IEEE Computer Society, 2000, p. 6007.

[17] Karl Aberer and Zoran Despotovic. “Managing trust in a peer-2-peer information system”. In: *Proceedings of the tenth international conference on Information and knowledge management - CIKM'01*. New York, New York, USA: ACM Press, 2001, p. 310.

[18] Lik Mui, Mojdeh Mohtashemi, and Ari Halberstadt. “A computational model of trust and reputation”. In: *HICSS. Proceedings of the 35th Annual Hawaii International Conference on System Sciences*. Vol. 5. IEEE, 2002, pp. 2431–2439.

[19] Gavin Wood. “Ethereum: a secure decentralised generalised transaction ledger”. In: *Ethereum Project Yellow Paper* (2014), pp. 1–32.

[20] Donovan Artz and Yolanda Gil. “A survey of trust in computer science and the Semantic Web”. In: *Web Semantics: Science, Services and Agents on the World Wide Web* 5.2 (June 2007), pp. 58–71.

[21] Matt Blaze, Joan Feigenbaum, John Ioannidis, et al. “The Role of Trust Management in Distributed Systems Security”. In: *Secure Internet Programming* (1999), pp. 185–210.

[22] Vimala Balakrishnan and Elham Majd. “A Comparative Analysis of Trust Models for Multi-Agent Systems”. In: *Lecture Notes on Software Engineering* 1.2 (2013), pp. 183–185.

[23] Daniel Olmedilla, Omer F Rana, Brian Matthews, et al. “Security and Trust Issues in Semantic Grids”. In: *Proceedings of the Dagstuhl Seminar, Semantic Grid: The Convergence of Technologies*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2006, pp. 191–200.

[24] Gideon Greenspan. *Why Many Smart Contract Use Cases Are Simply Impossible*. Apr. 2016. URL: http://www.coindesk.com/three-smart-contract-misconceptions/ (visited on 12/01/2016).

[25] Ahmed Kosba, Andrew Miller, Elaine Shi, et al. “Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts”. In: *2016 IEEE Symposium on Security and Privacy (SP)*. Vol. 2015. IEEE, May 2016, pp. 839–858.

[26] Christian Reitwießner. *TrueBit - Off-Chain Computations for Smart Contracts*. 2016. URL: https://chriseth.github.io/notes/talks/truebit/%7B%5C#%7D/ (visited on 02/01/2017).

[27] European Central Bank. *Distributed Ledger Technology*. Tech. rep. 1. European Central Bank, 2016.

[28] Maximilian Friedlmaier, Andranik Tumasjan, and Isabell Welpe. “Disrupting industries with blockchain: The industry, venture capital funding, and regional distribution of blockchain ventures”. 2016.

[29] Huw Van Steenis, Betsy L. Graseck, Fiona Simpson, et al. *Blockchain in Banking: Disruptive Threat or Tool?* Tech. rep. Global Insight 2016. Morgen Stanley Research, 2016.

[30] Lawrence Lessig. “Code Is Law”. In: *The Industry Standard* 18 (1999). URL: http://www.slate.com/articles/technology/future_tense/2015/01/cfaa_reform_how_laws_are_determining_the_ethics_of_code.html.

[31] Joshua J Doguet. “The Nature of the Form: Legal and Regulatory Issues Surrounding the Bitcoin Digital Currency System”. In: *Louisiana Law Review* 73.4 (2013), p. 9.

[32] Arvind Narayanan, Joseph Bonneau, Edward Felten, et al. *Bitcoin and Cryptocurrency Technologies - Draft*. Princeton, NJ, USA: Princeton University Press, 2016.

[33] Ludwig von Bertalanffy. *General System Theory: Foundations, Development, Applications*. Braziller, 1969.

[34] Michael Wooldridge. *An Introduction to MultiAgent Systems*. 2nd. Wiley Publishing, 2009.

[35] Jason Teutsch and Christian Reitwießner. “A scalable verification solution for blockchains”. 2017.

[36] Guy Zyskind. “Efficient Secure Computation Enabled by Blockchain Technology”. Master Thesis. Massachusetts Insitute of Technology, 2016.

[37] Tom Simonite. *The Autonomous Corporation Called the DAO Is Not a Good Way to Spend $130 Million*. 2016. URL: https://www.technologyreview.com/s/601480/the-autonomous-corporation-called-the-dao-is-not-a-good-way-to-spend-130-million/ (visited on 10/09/2016).

[38] Phil Daian. *Analysis of the DAO exploit*. 2016. URL: http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/ (visited on 10/09/2016).

[39] John Bohannon. *Why criminals can't hide behind Bitcoin.* 2016. URL: http://www.sciencemag.org/news/2016/03/why-criminals-cant-hide-behind-bitcoin (visited on 12/02/2016).

[40] United Nations. *Sustainable development goals - United Nations.* 2016. URL: http://www.un.org/sustainabledevelopment/sustainable-development-goals/ (visited on 02/13/2017).

[41] Brett Scott. "How Can Cryptocurrency and Blockchain Technology Play a Role in Building Social and Solidarity Finance?" Geneva, Switzerland, 2016.

[42] Konstantinos Christidis and Michael Devetsikiotis. "Blockchains and Smart Contracts for the Internet of Things". In: *IEEE Access* 4 (2016), pp. 2292–2303. URL: http://ieeexplore.ieee.org/document/7467408/.

[43] Juri Mattila. "The Blockchain Phenomenon: The Disruptive Potential of Distributed Consensus Architectures". 2016.

[44] Karl J. O'Dwyer and David Malone. "Bitcoin Mining and its Energy Footprint". In: *25th IET Irish Signals & Systems Conference 2014 and 2014 China-Ireland International Conference on Information and Communities Technologies (ISSC 2014/CIICT 2014).* 2014, pp. 280–285.

[45] Vitalik Buterin. *A Proof of Stake Design Philosophy.* 2016. URL: https://medium.com/@VitalikButerin/a-proof-of-stake-design-philosophy-506585978d51%7B%5C#%7D.ebrfrh9c4 (visited on 02/20/2017).

[46] Marko Vukolić. *Hyperledger fabric: towards scalable blockchain for business.* Tech. rep. Trust in Digital Life 2016. IBM Research, 2016. URL: https://www.zurich.ibm.com/dccl/papers/cachin_dccl.pdf.

[47] Nick Szabo. *Formalizing and Securing Relationships on Public Networks.* 1997. URL: http://ojphi.org/ojs/index.php/fm/article/view/548/469 (visited on 04/07/2017).

[48] Richard Brown. *A Simple Model for Smart Contracts.* 2015. URL: https://gendal.me/2015/02/10/a-simple-model-for-smart-contracts/ (visited on 03/07/2017).

[49] Ethereum. *Ethereum Homestead documentation: Account Types, Gas, and Transactions.* 2016. URL: http://www.ethdocs.org/en/latest/contracts-and-transactions/account-types-gas-and-transactions.html?highlight=autonomous%20agent (visited on 02/25/2017).

[50] IBM Corporation. *Making Blockchain Real for Business.* 2016. URL: http://www.ibm.com/systems/data/flash/it/technicalday/pdf/Making%20blockchain%20real%20for%20business.pdf.

[51] Christian Cachin. "Architecture of the hyperledger blockchain fabric". 2016.

[52] Tendermint. *Introduction to Tendermint - Tendermint.* 2016. URL: https://tendermint.com/intro (visited on 02/02/2017).

[53] David Schwartz, Noah Youngs, and Arthur Britto. “The Ripple protocol consensus algorithm”. 2014.

[54] Frederik Armknecht, Ghassan O. Karame, Avikarsha Mandal, et al. “Ripple: Overview and Outlook”. In: *Lecture Notes in Computer Science*. Vol. 9229. 2015, pp. 163–180.

[55] Matteo Biella and Vitorrio Zinetti. *Blockchain Technology and Applications from a Financial Perspective*. Tech. rep. 1. UniCredit, 2016.

[56] Diego Ongaro and John Ousterhout. “In Search of an Understandable Consensus Algorithm”. In: *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA, USA: USENIX Association, 2014, pp. 305–319.

[57] A. M. Turing. “On Computable Numbers, with an Application to the Entscheidungsproblem”. In: *Proceedings of the London Mathematical Society* s2-42.1 (1937), pp. 230–265.

[58] Vitalik Buterin. *Chain Interoperability*. Tech. rep. 1. R3CEV, 2016.

[59] Jordi Sabater and Carles Sierra. “Review on computational trust and reputation models”. In: *Artificial Intelligence Review* 24.1 (2005), pp. 33–60.

[60] Sarvapali D. Ramchurn, Dong Huynh, and Nicholas R. Jennings. “Trust in multi-agent systems”. In: *The Knowledge Engineering Review* 19.01 (Mar. 2004), pp. 1–25.

[61] Magnus Boman. “Norms in artificial decision making”. In: *Artificial Intelligence and Law* 7.1 (1999), pp. 17–35.

[62] Daira Hopwood, Bowe Sean, Taylor Hornby, et al. *Zcash Protocol Specification*. Tech. rep. 2016-1.10. Zerocoin Electric Coin Company, 2016.

[63] Lars Rasmusson and Sverker Jansson. “Simulated social control for secure Internet commerce”. In: *Proceedings of the 1996 workshop on New security paradigms - NSPW ’96* (1996), pp. 18–25.

[64] Jeff Coleman. *State Channels - an explanation*. 2015. URL: `http://www.jeffcoleman.ca/state-channels/` (visited on 04/03/2017).

[65] Michael Walfish and Andrew J Blumberg. “Verifying computations without reexecuting them: from theoretical possibility to near-practicality.” In: *Electronic Colloquium on Computational Complexity (ECCC)* 20.165 (2013).

[66] Guy Zyskind, Oz Nathan, and Alex Sandy Pentland. “Decentralizing Privacy: Using Blockchain to Protect Personal Data”. In: *2015 IEEE Security and Privacy Workshops*. IEEE, May 2015, pp. 180–184.

[67] Michael Walfish and Andrew J Blumberg. “Verifying computations without reexecuting them”. In: *Communications of the ACM* 58.2 (Jan. 2015), pp. 74–84.

[68] Ranjit Kumaresan and Iddo Bentov. "How to Use Bitcoin to Incentivize Correct Computations". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security - CCS '14*. New York, New York, USA: ACM Press, 2014, pp. 30–41.

[69] Christian Decker and Roger Wattenhofer. "A Fast and Scalable Payment Network with Bitcoin Duplex Micropayment Channels". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Ed. by Andrzej Pelc and Alexander A. Schwarzmann. Vol. 9212. Cham: Springer International Publishing, 2015, pp. 3–18.

[70] Davide Carboni. "Feedback based Reputation on top of the Bitcoin Blockchain". Feb. 2015.

[71] Ahmet Burak Can and Bharat Bhargava. "SORT: A Self-ORganizing Trust Model for Peer-to-Peer Systems". In: *IEEE Transactions on Dependable and Secure Computing* 10.1 (Jan. 2013), pp. 14–27.

[72] Huanyu Zhao and Xiaolin Li. "VectorTrust: trust vector aggregation scheme for trust management in peer-to-peer networks". In: *The Journal of Supercomputing* 64.3 (June 2013), pp. 805–829.

[73] Trung Dong Huynh, Nicholas R. Jennings, and Nigel R. Shadbolt. "An integrated trust and reputation model for open multi-agent systems". In: *Autonomous Agents and Multi-Agent Systems* 13.2 (2006), pp. 119–154.

[74] Mariusz Jakubowski, Ramarathnam Venkatesan, and Yacov Yacobi. "Quantifying Trust". 2010.

[75] Federico Cerutti, Alice Toniolo, Nir Oren, et al. "Context-dependent Trust Decisions with Subjective Logic". Sept. 2013.

[76] Ethereum. *Solidity 0.4.10 documentation*. 2017. URL: `https://solidity.readthedocs.io/en/develop/` (visited on 03/08/2017).

[77] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, et al. "Making Smart Contracts Smarter". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS'16*. New York, New York, USA: ACM Press, 2016, pp. 254–269.

[78] ConsenSys. *Intro to Programming Smart Contracts on Ethereum*. 2015. URL: `http://consensys.github.io/developers/articles/101-noob-intro/%7B%5C#%7Dclients_languages` (visited on 03/08/2017).

[79] ConsenSys. *ConsenSys Solidity parser*. 2017. URL: `https://github.com/ConsenSys/solidity-parser` (visited on 03/29/2017).

[80] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceed.* Ed. by C R Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340.

[81] Ran Canetti, Ben Riva, and Guy N. Rothblum. "Practical delegation of computation using multiple servers". In: *Proceedings of the 18th ACM conference on Computer and communications security - CCS '11*. New York, New York, USA: ACM Press, 2011, p. 445.

[82] Ran Canetti, Ben Riva, and Guy N. Rothblum. "Refereed delegation of computation". In: *Information and Computation* 226 (May 2013), pp. 16–36.

[83] Fan Zhang, Ethan Cecchetti, Kyle Croman, et al. "Town Crier". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS'16*. New York, New York, USA: ACM Press, 2016, pp. 270–282.

[84] Oraclize. *blockchain oracle service, enabling data-rich smart contracts*. 2017. URL: `http://www.oraclize.it/` (visited on 05/07/2017).

[85] Karthikeyan Bhargavan, Nikhil Swamy, Santiago Zanella-Béguelin, et al. "Formal Verification of Smart Contracts". In: *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security - PLAS'16* (2016), pp. 91–96.

[86] Yoichi Hirai. *Formal Verification of Ethereum Contracts*. 2017. URL: `https://github.com/pirapira/ethereum-formal-verification-overview` (visited on 05/12/2017).

[87] James J. Odell, H. Van Dyke Parunak, and Bernhard Bauer. "Representing agent interaction protocols in UML". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 1957 LNCS. 2001, pp. 121–140.

[88] Sanjay Jain, Prateek Saxena, Frank Stephan, et al. "How to verify computation with a rational network". June 2016.

[89] Ethereum. *Ethereum TestRPC*. 2017. URL: `https://github.com/ethereumjs/testrpc` (visited on 04/01/2017).

[90] Jason Teutsch, Sanjay Jain, and Prateek Saxena. "When cryptocurrencies mine their own business". 2016.

[91] Adam Back, Matt Corallo, Luke Dashjr, et al. "Enabling Blockchain Innovations with Pegged Sidechains". 2014. URL: `http://www.blockstream.com/sidechains.pdf`.

[92] Venkatesh Rao. *The Mother of All Disruptions*. 2013. URL: https://www.ribbonfarm.com/2013/10/11/the-mother-of-all-disruptions/ (visited on 03/28/2017).

[93] Simon de la Rouviere. *Love, The End of the World, and The Benefits of Verifiable Computing*. 2016. URL: https://medium.com/@ConsenSys/love-the-end-of-the-world-and-the-benefits-of-verifiable-computing-1697658e3143%7B%5C#%7D.nn9024jpp (visited on 03/28/2017).

[94] Trent McConaghy. *AI DAOs, and Three Paths to Get There*. 2016. URL: https://blog.bigchaindb.com/ai-daos-and-three-paths-to-get-there-cfa0a4cc37b8 (visited on 05/16/2017).

[95] Iddo Bentov, Ariel Gabizon, and Alex Mizrahi. "Cryptocurrencies Without Proof of Work". In: *Financial Cryptography and Data Security* 9604 (2016).

[96] Runfang Zhou, Kai Hwang, and Min Cai. "GossipTrust for Fast Reputation Aggregation in Peer-to-Peer Networks". In: *IEEE Transactions on Knowledge and Data Engineering* 20.9 (2008), pp. 1282–1295.

[97] ConsenSys. *Ethereum Contract Security Techniques and Tips*. 2017. URL: https://github.com/ConsenSys/smart-contract-best-practices (visited on 05/18/2017).

[98] Ethereum. *Remix - Solidity IDE*. 2017. URL: https://remix.ethereum.org/%7B%5C#%7Dversion=soljson-v0.4.11+commit.68ef5810.js (visited on 05/16/2017).

[99] Vitalik Buterin. *Could Ethereum do this better? [Tor Project is working on a web-wide random number generator]*. 2016. URL: https://www.reddit.com/r/ethereum/comments/4mdkku/could_ethereum_do_this_better_tor_project_is/d3v6djb/ (visited on 05/18/2017).

[100] Loi Luu, Jason Teutsch, Raghav Kulkarni, et al. "Demystifying Incentives in the Consensus Computer". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - CCS '15*. New York, New York, USA: ACM Press, 2015, pp. 706–719. URL: http://dl.acm.org/citation.cfm?doid=2810103.2813659.

[101] Reid Kerr and Robin Cohen. "Smart Cheaters Do Prosper : Defeating Trust and Reputation Systems". In: *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems*. Budapest, Hungary, 2009, pp. 993–1000.

[102] Christian Reitwießner. *From Smart Contracts to Courts with not so Smart Judges - Ethereum Blog*. 2016. URL: https://blog.ethereum.org/2016/02/17/smart-contracts-courts-not-smart-judges/ (visited on 05/18/2017).

[103] Tjaden Hess. *How can I securely generate a random number in my smart contract? - Ethereum Stack Exchange*. 2016. URL: https://ethereum.stackexchange.com/questions/191/how-can-i-securely-generate-a-random-number-in-my-smart-contract/2074%7B%5C#%7D2074 (visited on 05/18/2017).

[104] Yoichi Hirai. "Formal Verification of Deed Contract in Ethereum Name Service". 2016.

[105] Nick Johnson. *Dividend-bearing tokens on Ethereum*. 2017. URL: https://medium.com/@weka/dividend-bearing-tokens-on-ethereum-42d01c710657 (visited on 05/11/2017).

TRITA TRITA-ICT-EX-2017:79

www.kth.se