TU WIEN Informatics

A Blockchain-based Computation Offloading Approach with Result Verification

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Ing. Benjamin Körbel, BSc
Registration Number 1228896

to the Faculty of Informatics

at the TU Wien

Advisor: Dr.-Ing. Stefan Schulte
Assistance: Marten Sigwart, MSc
Ing. Dipl.-Ing. Philipp Frauenthaler, BSc

Vienna, 21st December, 2020

Benjamin Körbel	Stefan Schulte

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

A Blockchain-based Computation Offloading Approach with Result Verification

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Ing. Benjamin Körbel, BSc
Matrikelnummer 1228896

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Dr.-Ing. Stefan Schulte
Mitwirkung: Marten Sigwart, MSc
Ing. Dipl.-Ing. Philipp Frauenthaler, BSc

Wien, 21. Dezember 2020

Benjamin Körbel	Stefan Schulte

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Ing. Benjamin Körbel, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 21. Dezember 2020

Benjamin Körbel

Acknowledgements

I would like to thank my advisors Marten Sigwart and Philipp Frauenthaler for their consistent support and their quick and comprehensive feedback. Our periodic discussions about blockchain technology, different concepts and design decisions during the literature search and development phase helped me a lot to bring this thesis to a higher level.

I also would like to express my gratitude to my supervisor Stefan Schulte for his guidance and precise comments.

Kurzfassung

Computation Offloading bezeichnet das Auslagern von Berechnungen an externe Plattformen. Diese Plattformen verfügen über leistungsstarke Ressourcen, die gegen eine Gebühr genutzt werden können. Computation Offloading wird häufig bei Geräten mit einer beschränkten Hardwareausstattung eingesetzt, bspw. um die Zeit für die Abarbeitung von komplexen Berechnungen zu verkürzen oder den Akkustand zu schonen.

Üblicherweise erfolgt die Auslagerung an eine Cloud-Infrastruktur, also Rechenzentren mit nahezu endlosen Ressourcen. Der Cloud-Anbieter trägt die alleinige Verantwortung über die Zuweisung von Ressourcen als auch die Zahlungsabwicklung. Dies erfordert eine gut ausgebaute Infrastruktur, welche nur wenige große Anbieter wie bspw. Amazon, Microsoft und Google bereitstellen können. Für kleinere Anbieter ist der Markteintritt daher schwierig.

Eine Alternative zu traditionellen Cloud-Anbietern stellt die Blockchain dar. Mittels Peer-to-Peer-Netzwerk ermöglicht die Blockchain eine dezentrale Verwaltung von Daten und Programmen, da sämtliche Änderungen von allen Netzwerkteilnehmern durchgeführt und verifiziert werden. Für die Teilnahme am Netzwerk gibt es keine Beschränkungen oder Hürden. Aufgrund dessen bietet sich die Blockchain-Technologie als Grundlage für eine dezentrale, frei zugängliche Computation-Offloading-Plattform an. Da eine Blockchain jedoch nicht für aufwendige Berechnungen geeignet ist bzw. deren Ausführung sehr teuer werden kann, ist es erforderlich, die Berechnungen außerhalb der Blockchain durchzuführen. Dies führt allerdings dazu, dass man auf die korrekte Ausführung der Berechnungen seitens der Netzwerkteilnehmer vertrauen muss.

In dieser Arbeit entwickeln wir auf Basis der Blockchain-Technologie einen Ansatz, der es beliebigen Anbietern ermöglicht, Ressourcen gegen eine Gebühr zu vermieten. Zudem vergleichen wir in einer Evaluierung mehrere Möglichkeiten, Berechnungen außerhalb der Blockchain von Anbietern durchführen zu lassen, ohne dabei einem bestimmten Anbieter für die korrekte Ausführung vertrauen zu müssen. Anhand eines exemplarischen Anwendungsfalls zeigen wir, dass unser Ansatz eine Alternative zu traditionellen Cloud-Anbietern bietet.

Abstract

Computation offloading refers to the outsourcing of tasks to providers with vast computational resources in exchange for a fee. It can be an enabling technique for devices with limited computational resources, e.g., mobile or IoT-devices. These resource-constrained devices can benefit from computation offloading, e.g., by a reduced application runtime and battery savings.

Conventionally, computation offloading takes place in the cloud, i.e., in data centers with nearly endless computational resources. Usually, all layers of the cloud such as the assignment of resources and payment processing are controlled by a single provider. Because of high cost for provisioning and maintaining cloud infrastructure, the cloud provider market is dominated by only a few big organizations such as Amazon, Microsoft, and Google. For small to medium-sized companies it is difficult to enter the market.

Blockchain technology provides an alternative to traditional cloud providers. Organized as peer-to-peer network, the blockchain manages data and programs in a decentralized manner since all participants execute and verify all state changes. Furthermore, blockchain technology allows participants to freely join and leave the network. As such, blockchain technology is a promising solution to realize a decentralized and open computation offloading platform. However, carrying out resource-intensive computations on blockchains is rather expensive or in some cases just not possible. To tackle this problem, computations can be executed off the blockchain. However, if the task is computed off-chain, the correctness of the computation is not verified by the peer-to-peer network. In other words, the resource provider has to be trusted.

In this thesis, we develop a blockchain-based computation offloading approach that allows arbitrary providers to rent out their computational resources for a fee. We compare multiple solutions for executing tasks off-chain without having to trust the resource provider carrying out the computation. As such, the proposed solution provides an open alternative to traditional cloud providers.

Contents

CHAPTER 1

Introduction

Computation offloading has gained a lot of research attention in recent years [54]. The basic idea of offloading is that hardware-constrained devices outsource or offload resource-intensive (mobile) tasks (e.g., natural language processing) to providers offering vast computational resources in exchange for a fee [43, 54]. For instance, Amazon's Echo Dot is a voice-controlled smart speaker that leverages computation offloading to process and interpret voice commands, e.g., a weather forecast request [39]. Besides such smart devices, computation offloading is also used in the field of mobile applications and the Internet of Things (IoT). Mobile applications execute various tasks, e.g., GPS navigation or face recognition, which can be resource-intensive [1, 35]. By offloading computational tasks to powerful resource providers, application running time can be saved. If the device is battery-operated, battery lifetime can be saved as well [54].

Traditionally, computation offloading leverages the cloud—data centers with nearly endless computational resources managed by a single organization such as Amazon Web Services, Microsoft Azure, etc. [43]. Usually, all layers of the cloud such as the assignment of resources and payment processing are controlled by a single provider. As setting up this stack is highly cost-intensive, the cloud market is dominated by only a few companies. Small to medium-sized organizations that are looking to rent out their own idle resources experience high entry barriers [32, 33].

However, the advent of blockchain technology has given rise to an alternative to traditional cloud providers as means to realize computation offloading [23, 25, 42]. Computations in a blockchain network are not carried out by a single actor. Instead, all participants execute and verify all state changes [45]. As such, the idea is that the blockchain takes care of task assignment and payment processing in a completely decentralized manner while anyone can freely join the network to rent out any idle computational resources [45, 63].

While blockchain technology removes the aforementioned entry barriers, two challenges arise. First, task issuers have no guarantee that task processors are honest and send

back correct results. Ideally, before task issuers pay processors for their work, they have an assurance that the returned results can be trusted. Second, due to the fact that all data on the blockchain is public, all network participants are able to see a returned computation result including the task issuer. Therefore, it becomes possible for task issuers to retrieve the result that was submitted by a task processor to the blockchain without paying.

Previous works [15, 23, 25] provide various approaches, i.e., redundant computing, reprocessing fractions of a task locally, or reputation-based systems in order to ensure correct results. However, the aforementioned techniques only reduce the risk of receiving wrong results [63]. The issue of publicly accessible results is also not discussed explicitly in [15, 23, 25].

Besides the aforementioned feature of result verification and correct payment processing, task issuers also want to receive the best possible solution to a given problem. Considering an NP-complete problem, e.g., the Traveling Sales Person (TSP), finding the optimal solution is not feasible for large problem instances but different heuristics can be used to find acceptable solutions in less time. In such use cases, it would be beneficial if a variety of possible solutions (e.g., different heuristics implemented by different task processors) were submitted to the blockchain with only the best one chosen and paid for by the task issuer.

The goal of this work is to implement and evaluate a blockchain-based offloading approach that can prove the proper execution of a particular computation while ensuring that task processors are always paid for their work. By using a blockchain as underlying infrastructure we dissolve the dependency on a single organization. Further, to enhance competition between task processors and to offer task issuers the best possible solution, an auction-based mechanism is integrated. Therefore, the thesis addresses the following research questions:

RQ.1 Which result verification schemes are possible?
The first research question aims to provide an overview of result verification schemes, that are suitable for a blockchain-based environment. Various concepts are compared with respect to applicability, expected cost and security.

RQ.2 How can blockchain-based computation offloading be integrated with the selected result verification scheme and an auction-based mechanism?
The second research question focuses on the offloading procedure in the context of the selected result verification scheme and the auction-based mechanism. Various requirements serve as basis for a transparent, open, universally applicable and secure offloading mechanism. Different approaches regarding result delivery and the open nature of blockchains are discussed. Consequently, a prototype is implemented.

RQ.3 What are the benefits of the proposed solution and when does it make sense to prefer a verification scheme that proves whether a defined computation was executed properly over alternative variants?
The goal of the third research question is to evaluate the proposed solution with regard to the overhead resulting from determining whether a computation was executed honestly. Further, we evaluate under what circumstances the selected result verification scheme is cheaper than alternative variants.

The methodological approach can be grouped roughly into the following four parts:

Literature review As a first step, literature in the field of computation offloading is studied. Different computation offloading approaches, especially those which are suitable for a blockchain-based environment and support result verification schemes, are compared with regard to practicability, flexibility, cost and security.

Definition of requirements According to the selected result verification scheme and the auction-based mechanism of the platform, basic requirements with regard to interactions between parties and their temporal sequences are defined.

Implementation of the platform Within this phase, the selected result verification scheme is implemented and combined with an auction-based mechanism by meeting the predefined requirements. To get early insights with respect to feasibility or possible pitfalls, the defined requirements are underpinned by explorative prototyping and testing. For the purpose of a fully functional prototype, an exemplary use case is implemented.

Evaluation In this part of the thesis, the following points are evaluated

- Overhead included by result verification: The additional time and cost for verifying the result of the selected use case is determined.
- Identification of cost drivers: Potential parameters which influence the cost development in the context of the use case are outlined.
- Cost efficiency: The costs resulting of computing a task with the implemented approach are compared with costs that occur by on-chain result verification, respectively by the means of a smart contract only.
- Practicability of the proposed computation offloading approach with result verification: Beneficial aspects as well as drawbacks of the implemented solution are described and compared to related approaches.

The thesis is structured as follows: In Chapter 2 background knowledge in the field of blockchain technology and Zero-knowledge proofs (ZKPs) is provided. After a summary with regard to the building blocks of a blockchain, smart contracts and ZKPs, recent works and projects in the area of blockchain-based computation offloading solutions are presented in Chapter 3. Chapter 4 compares various off-chain computation and

verification schemes. Additionally, a development tool to realize the so-called verifiable off-chain computation is described. Chapter 5 covers the requirements and architecture, while Chapter 6 describes the prototypical implementation of the proposed solution. By means of an exemplary use case, our blockchain-based offloading solution is evaluated in Chapter 7. Chapter 8 concludes with an overview of the contributions and possible future work of the thesis.

CHAPTER 2

Background

In this chapter, fundamental concepts and preliminary knowledge in the field of blockchain-based systems and ZKPs are explained. Section 2.1 describes the basic techniques and underlying cryptographic primitives used within blockchain technology. Section 2.2 gives a summary of blockchain-based cryptocurrencies, namely Bitcoin and Ethereum. The fundamentals of ZKPs and their variants are stated in Section 2.3.

With regard to the term cryptocurrency, a number of different definitions can be found. The EU parliament compares definitions of policy makers like the European Central Bank and concludes that cryptocurrencies are mostly specified as subset of digital or virtual currencies. They further summarize these definitions as follows: "a digital representation of value that (i) is intended to constitute a peer-to-peer (P2P) alternative to government-issued legal tender, (ii) is used as a general-purpose medium of exchange (independent of any central bank), (iii) is secured by a mechanism known as cryptography and (iv) can be converted into legal tender and vice versa" [36].

A blockchain can be seen as the technology which secures a cryptocurrency and ensures its validity. The EU parliament describes a blockchain as backbone for cryptocurrencies [36]. Besides cryptocurrencies as applications running on top of a blockchain, blockchain technology can be applied in various other fields as well, e.g., healthcare [65].

2.1 Fundamental Cryptography and Data Structures

Like fiat currencies, cryptocurrencies must provide security features to prevent cheating, e.g., spending the same coin twice. Security measures for cryptocurrencies have the additional requirement to work without a central authority. Furthermore, every security policy must be enforced technically. To cover these challenges and realize appropriate security measures, cryptography is employed. Among others, cryptographic hash functions are commonly used within cryptocurrencies [45].

2.1.1 Cryptographic Hash Functions

In general, a hash function transforms an input of type string with arbitrary length into a fixed-sized output. The described transformation can be computed efficiently, so that each string of length n can be hashed within a running time of $\mathcal{O}(n)$ [45]. In the context of blockchains, cryptographic hash functions must fulfill the three properties collision resistance, hiding, and puzzle-friendliness to be considered secure. It is worth mentioning, that puzzle-friendliness is not a general requirement but necessary for hash functions in the field of cryptocurrencies.

Collision resistance is defined as follows [45]: *"A hash function H is said to be collision resistant if it is infeasible to find two values, x and y, such that $x \neq y$, yet $H(x) = H(y)$."*
In other words, it should not be possible, to find two different input strings which produce the same output. In theory, collisions are possible due to the fact that the number of inputs (strings of arbitrary length) is larger than the amount of outputs (strings of fixed length). Therefore, it is valid that collisions exist for a specific hash function. However, such collisions should not be found within a tolerable period of time. Consider a cryptographic hash function with an output length of 256 bit, e.g., SHA256. A collision can be found by hashing $2^{256} + 1$ different inputs. In practice, a collision can be found earlier when more random hashes than the square root of the number of possible outputs are generated. This phenomena is known as the birthday paradox [45]. However, calculating $\sqrt{2^{256}} + 1$ hashes with a hash rate of $100 * 10^6$ hashes per second (comparable with the AMD Radeon VII with a hash power of 90 MH/s, which is at the moment one of the most powerful graphic cards[1]) still takes 10^{23} years.

The second property **hiding** is defined as follows [45]: *"A hash function H is said to be hiding if a secret value r is chosen from a probability distribution that has high min-entropy, then, given $H(concat(r, x))$ it is infeasible to find x."*
Simply put, hiding means that hash functions are one-way functions, so that given the output of a hash function $y = H(x)$ it is not possible to figure out the initial input x. In certain cases, for instance, if the amount of inputs is limited, e.g, the numbers of a dice, the conclusion from outputs to inputs can be derived with less effort. To cover this issue, a randomly chosen value r is concatenated with x. As a consequence, guessing the input is more difficult and requires higher effort. High min-entropy means, that each possible value r of a distribution has the same probability to be chosen.

The third security property **puzzle-friendliness** is defined as follows [45]: *"A hash function H is said to be puzzle friendly if for every possible n-bit output value y, if k is chosen from a distribution with high min-entropy, then it is infeasible to find x such that $H(concat(k, x)) = y$ in time significantly less than 2^n."*
In other words, if a part of the input k has been chosen from a spread-out set with high min-entropy, then it is difficult and time-intensive to find another value that hits exactly that target y. Therefore, puzzle-friendliness ensures, that there is no shortcut for finding an input, which is hashed to y, respectively to one member within the solution space.

[1] https://www.techradar.com/news/best-mining-gpu

Usually, y is not a single value, but rather a set of values. If the solution set y contains exactly one value, the puzzle is maximum hard to solve. Otherwise, the greater the amount of permitted values, the less difficult is the puzzle.

2.1.2 Hash Pointers and Blockchain

Regular pointers and hash pointers can be used to form a data structure. In general, these pointers contain a reference (i.e. a memory address) to retrieve further information, e.g., the address of the next data block. The difference between regular- and hash pointers is that hash pointers additionally store a cryptographic hash of the referenced information. Therefore, further information can not only be located, but also checked for integrity, so that possible subsequent changes can be detected [45].

Hash pointers can be used to build all kinds of data structures such as linked lists or binary trees. A linked list using hash pointers is shown in Figure 2.1, whereby each previous-block pointer is implemented as a hash pointer [45].

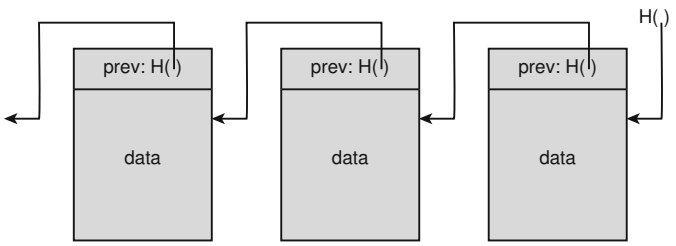


Figure 2.1: Example of a simplified "block chain"

One characteristic of a blockchain is, that it serves as a tamper-evident log. The main features of a tamper-evident log is that new blocks can only be appended as any removal or updating of previous blocks invalidates existing hash pointers. Therefore, changes to any previous block can be detected. To give an example, consider an attacker which manipulates the block k. As a result, the hash of block k also changes as long as the collision resistance property holds. Therefore, the inconsistency can be noticed, unless the subsequent block $k+1$ as well has been modified by the attacker. However, in the case that all blocks from k to the most recent block $k+n$ have been altered, the manipulation is recognized when comparing the hash of the last block $k+n$ with the separately stored hash pointer of the head element. In the described scenario, the separate copy of the head hash pointer plays a key role. As long as a potential attacker has no access to this (local) copy, the obfuscation of subsequent block tampering does not work [45].

2.2 Blockchain-based Crytocurrencies

In recent years, blockchain technology has experienced high interest in research and industry. Wherever an agreement or trade between several parties takes place, blockchain technology is considered as an alternative to existing solutions. Blockchains promise

that no central authority is needed to manage interactions between parties. Due to the absence of a central entity and intermediaries, cost savings can be assumed. A further advantage is that no position of trust to third parties is necessary. Tai et al. describe blockchains as shared append-only transaction ledger operating in a distributed P2P network [59].

Bitcoin, proposed in 2008 by Satoshi Nakamoto [44], is the first implementation of a digital blockchain-based currency which uses a decentralized approach and therefore does not rely on a central authority. The intention of Bitcoin was to build an electronic payment system based on crytographic proofs instead of trust [44]. Therefore, Bitcoin provides value transfers in form of transactions without relying on a trusted intermediary, e.g., a trustee. Cryptocurrencies like Bitcoin, use several techniques to achieve consistency among all network participants. More precisely, Bitcoin makes use of asymmetric cryptography and a consensus mechanism called proof-of-work (PoW) to secure transactions. The consensus protocol is needed to decide one global order of transactions among all nodes. Such methods are required to prevent cheating, e.g., double spending of coins. Furthermore, the PoW is a compute-intensive search puzzle, which gives the node that solves the puzzle first, the privilege to add a new block. Multiple transactions can be included in a block. After a new block was found, it is further broadcasted to the network. The PoW protocol of Bitcoin uses cryptographic hash functions as described in Section 2.1. The goal is to find an input for a hash function which evaluates to a value within the target space. Depending on the difficulty of the PoW, block creation can take more or less time. The average time needed to create a block in the Bitcoin network is 10 minutes. Each node has to verify each transaction, respectively each block. By means of hash pointers, PoW and verification of blocks, the immutability of a blockchain is ensured.

The idea of processing transactions and transferring coins within a trust-less network was extended by the Ethereum platform in 2014 [14]. Ethereum belongs to the second generation of blockchains [34] which provide a Turing-complete programming language to enable on-chain execution of any kind of programs. Such programs encode arbitrary state transitions, allowing users to create their own rules for ownership and transaction formats [14], e.g., one could define rules which specify the conditions for a coin transfer in the context of a purchase of a good. These programs are referred to as smart contracts, which were discussed first by Nick Szabo in 1994 [58]. Like value transfers within Bitcoin, smart contracts in Ethereum are executed without relying on a trusted authority. Each state transition resulting from a smart contract is executed and validated by each network participant. A smart contract, or more precisely its bytecode, is executed within the Ethereum Virtual Machine (EVM) for isolation purposes. Thus, a separation between the host system and the smart contract is obtained. Each Ethereum network participant runs an EVM instance. Due to the redundant computation of smart contracts, blocking operations, e.g., an infinite loop would affect the system's availability and throughput. In the worst case, blocking operations can pave the way for denial-of-service attacks. To cover this issue, an upper bound with regard to computational effort, the so called gas, was introduced by Ethereum. Gas is a pricing value or a fee for executing transactions,

consisting of gas units and a gas limit. Each operation consumes a determined amount of gas units. Therefore, depending on the operations used within a smart contract, more or less gas is consumed, whereby the gas contingent can be specified before execution [64]. Besides the possibility of defining the maximum gas value per function call, the gas limit is fixed on block level (overall transactions per block). Currently, the block gas limit is 10 million [22]. Consequently, any program running on the Ethereum blockchain finally terminates after the provided amount of gas is exhausted. The currency in the Ethereum network is called Ether. Gas can be acquired in sub-units of Ether, referred to as Gwei and Wei, whereby 1 Ether is equal to 10^9 Gwei respectively 10^{18} Wei.

2.3 Zero-Knowledge Proofs

ZKPs were introduced in the 1980s by Goldwasser, Micali and Rackoff [29]. The power and broad applicability of this proof-system was shown by Goldreich, Micali and Wigderson [27, 28]. For instance, Goldreich et al. have proven that there exists a ZKP for all problems of the complexity class NP, e.g., for the graph coloring problem [28]. The idea behind ZKPs is to convince someone that a statement is true without revealing any other information except that the statement is true. When using ZKPs, two parties, a prover and a verifier, are involved whereby the prover tries to convince the verifier of the truthfulness of the statement. When applying ZKPs in real-world use cases, the two parties are computer programs or to be more formal Turing Machines.

Advanced math, cryptography and knowledge in theoretical computer science are the building blocks of ZKPs. Quisquater et al. stated an illustrative example of ZKPs, which explains the intention and procedure behind zero-knowledge protocols in an easily traceable way [51]. The setting of this example is a cave, the so-called Ali Baba cave, which is shown in Figure 2.2. The cave has one entry and one circular route within it. At half-distance a gate partitions the circular route. The gate can only be opened by voicing a secret word. Consider two persons, Bob and Alice, whereby Bob is the prover and Alice the verifier. Bob knows the secret word, but does not want to share it with Alice. A zero-knowledge protocol can be used to prove Bob's knowledge of that secret. To do so, Alice waits at the entry of the cave, while Bob walks into the cave so that Alice cannot see Bob. Now, Alice tosses a coin and on head, tells Bob to come from the left side of the tunnel. Otherwise, on tail, Alice expects Bob to come from the right side. If Bob actually knows the secret word, he is able to unlock the gate and can fulfill the expectations of Alice, that is showing up at a determined side of the tunnel. Consequently, Alice can assume that Bob knows the secret word. In the case that Bob does not know the secret word, he has a 50% chance to guess the correct side in advance. By repeating this procedure n times, the probability to guess correctly is $1/2^n$. Therefore, an increasing number of rounds leads to a more trustworthy end result. If one round fails, i.e., Bob shows up on the wrong side, Alice can conclude that Bob is dishonest and does not know the secret word. This property, provided by such proof systems, is known as soundness which is described in the next paragraph.

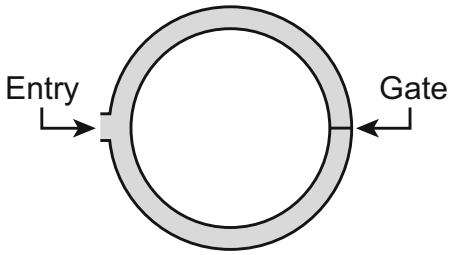


Figure 2.2: Illustration of Ali Baba's Cave

The more general class of ZKPs are interactive proofs which provide the properties completeness and soundness [29]. Informally, they can be described as follows:

1. Completeness: If the prover is honest (the statement is true), then the verifier can be convinced with high probability.
2. Soundness: The probability to convince the verifier is negligibly small, if the prover is dishonest (the statement is false).

Additionally to the mentioned properties, ZKPs fulfill the so-called zero-knowledge property, which means that the verifier learns nothing except that the statement is true.

As indicated, interactive ZKPs entail several rounds or various message exchanges between a prover and a verifier. The verifier or recipient of the proof must ask questions and receive answers from the prover. According to the mentioned soundness property, the probability that a verifier accepts a wrong statement is reduced with an increasing number of rounds performed between both parties [26, 29].

According to Blum et al. in [9, 10] interactive ZKPs have three main ingredients:

(i) Interaction: The prover and the verifier are allowed to talk back and forth.

(ii) Hidden Randomization: The verifier is allowed to flip coins whose result the prover cannot see.

(iii) Computational Difficulty: The prover embeds in his proofs the computational difficulty of some other problem.

In addition, Blum et al. asked themselves if ZKPs can be built "with fewer ingredients". Based on their work, a further variant, the so-called non-interactive ZKPs were introduced. Non-interactive ZKPs show that the aforementioned assumptions (i) and (ii) are not necessary in ZKP systems [9, 10].

In the setting of non-interactive ZKPs, three entities, a prover, a verifier and a uniformly selected reference string, are involved. The reference string is chosen by a trusted third

party and shared between the prover and the verifier. Therefore, the randomly selected reference string replaces the coin toss, respectively point (ii) "hidden randomization". A further difference between interactive and non-interactive ZKPs is that only a single message, sent from the prover to the verifier, is necessary. Without further communication, the verifier can decide if the received proof is convincing or not. Thus, point (i) "interaction" can be discarded [9, 26].

The development around non-interactive ZKPs leads to more efficient proofs which have the property of succinctness. These new proofs, introduced in [24], are referred to as zero-knowledge non-interactive succinct argument of knowledge (zk-SNARKs). Further improvements of zk-SNARKs have been made in the work of Parno [47] and Ben-Sasson [7].

The single parts of the acronym SNARK can be described as follows [52]:

- Succinct: The size of the message sent to the verifier, i.e., the proof attesting the validity of the statement, is tiny in comparison to the length of the actual statement, i.e., the length of the computation. The proof can be verified with less computational effort.
- Non-interactive: Instead of performing multiple rounds between a prover and a verifier, one single message is sent to the verifier. However, a trusted setup has to be executed in advance.
- ARguments (or computational soundness): Verifiers are protected against provers with limited computational power. Provers with enough computational power can create proofs/arguments about wrong statements and can fool the verifier. However, this would mean that any public-key encryption can also be broken with sufficient computing power.
- Of knowledge: For proof construction a so-called witness is required. In other words, without a witness it is not possible for the prover to create a proof. A witness encapsulates a secret, e.g., the preimage x of a hash function, which makes the statement true, e.g., "I know the preimage of hash $h(x)$".

Due to the zero-knowledge property, the verifier is not able to gain additional information, especially not about the witness, besides the fact of validity.

The main feature of zk-SNARKs is succinctness. Succinctness ensures cheap verification of any statement or computation. The complexity of the verification does not depend on the complexity of the computation. Therefore, the verification of a simple problem, e.g., the addition of two integers, and a complex problem, e.g., the graph coloring problem, takes the same effort. The combination of succinctness and non-interactivity paves the way for deploying zk-SNARKs on a blockchain. Zk-SNARKs enable efficient verification and a privacy-preserving execution of arbitrary computations. Due to the aforementioned

properties of zk-SNARKs the cost for verifying a problem on-chain, e.g., on the Ethereum blockchain, is expected to be manageable [52].

One application which uses zk-SNARKs is the cryptocurrency Zcash[2]. Like Bitcoin, Zcash enables value transfers between potential untrusted parties without the need for a central entity. Unlike Bitcoin, which provides only pseudonymity, Zcash leverages zk-SNARKs to provide strong privacy guarantees and anonymity [6]. Therefore, not only the transferred amount but also the sending and receiving party can be kept private. Despite being private, the validity of transactions can still be verified.

In the next chapter, we examine different blockchain-based computation offloading solutions from research and industry. We will see that ZKPs are a promising technology for achieving result verification within these schemes.

[2]https://z.cash/

CHAPTER 3

Related Work

This chapter surveys related work in the field of blockchain-based computation offloading and result verification. As mentioned in Chapter 1, a solution is required which can prove whether a computation was executed correctly in the context of a blockchain-based computation offloading system. Therefore, a result verification scheme is needed which provides universal applicability, low overhead and security. Since we aim to build a computation offloading system, it is beneficial when the result verification scheme is reliable and can be applied to a broad range of problems. Furthermore, it is essential that the result verification scheme is compatible with blockchain technology. Due to a blockchain being not suitable to carry out arbitrary complex computations, we assume that result verification schemes with an expected low overhead are advantageous.

Three commercial blockchain-based offloading solutions are presented in Section 3.1. Section 3.2 outlines recent research projects in the field of blockchain technology and computation offloading. Both, the commercial and the research projects are finally discussed in Section 3.3.

3.1 Commercial Blockchain-based Solutions

In this section, three commercial decentralized cloud/fog solutions are described. Decentralization is achieved with the help of blockchain technology. Fog solutions are based on edge/fog computation, which can be a supplement to cloud computing. The main difference between these architectures is that edge/fog computing uses smaller computation nodes in close proximity to customers, whereby latency can be reduced.

All three solutions aim to share idle resources. Therefore, resource providers and resource consumers are involved. Consumers have to pay resource providers for their service, e.g., for providing the computing infrastructure for a period of time or for processing a specific task. To save fees caused by the use of a blockchain and smart contracts, the

activities within the sharing-workflow are divided into an on-chain and an off-chain part. On-chain activities like payments are conducted in the so-called transaction network, while off-chain activities, e.g., a task execution is done in the so-called side-chain network. The transaction network relies in all three projects on the Ethereum blockchain. The side-chain network can be referred as internal blockchain, e.g., a clone of the Ethereum blockchain, which can be accessed by participants of the solution. Golem, iExec and SONM have their own native currency and were financed by crowdfunding. All three projects raised several million dollars through initial coin offerings from investors.

Golem

Golem[1] is an open-source project launched in 2014 whose vision is to position itself as the first truly decentralized supercomputer, creating a global market for computing power [25]. To achieve this, Golem enables the rental of computational resources from users and by users. Additionally, it is possible to create and deploy software to the Golem network. The system relies on the Ethereum platform and acts as P2P network. The native currency of Golem is called Golem Network Token (GNT). Golem claims to support Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS) and Software-as-a-Service (SaaS). The Golem project is structured into four milestones, Brass-, Clay-, Iron- and Stone Golem, providing new or extended features at each successive version. Brass Golem was launched in 2018, it's current version is 0.22.1[2]. While writing the thesis, Clay Golem is in development with a beta version[3] available. Besides the mentioned milestones introduced in the Golem whitepaper, an additional version, Golem Unlimited[4], is released which focuses on trusted heterogeneous computing resources, e.g., data centers.

The main actors in the Golem ecosystem are software developers, resource providers and requestors. Apart from a few initial use cases, e.g., 3D rendering, Golem gives software developers the opportunity to realize their own applications and deploy them to the network by publishing them to Golem's Application Registry. Resource requestors have demand for computational resources and are able to create and offload tasks. Providers, e.g., individuals with a gaming PC, act as suppliers for computing power and receive payments from requestors for processed tasks.

The general architecture of Golem is depicted in Figure 3.1. As indicated, particular activities are conducted in the side-chain or in the transaction network which relies on Ethereum. With the exception of payments and the reputation system, all activities, e.g., price negotiations, task- orders, executions and verifications are performed in the side-chain network, for cost savings. Tasks can be created by using predefined templates of the task definition framework. A template contains the computational logic, executable source code, information about splitting/merging tasks into subtasks and verification approaches for the specific problem. If no suitable template is available, a new one can be added starting with the Stone Golem version. Furthermore, the developer kit can be

[1] https://golem.network/
[2] https://github.com/golemfactory/golem/releases/tag/0.22.1
[3] https://github.com/golemfactory/golem/releases/tag/0.23.1
[4] https://github.com/golemfactory/golem-unlimited

used to create and register custom applications. New tasks and the reputation of the requestor are published by the transaction framework. Task related data is shared by the InterPlanetary File System (IPFS)[5]. Tasks are executed on the providers' hardware within a sandboxed environment. The result verification depends on the task type but is mostly based on redundant computation. Therefore, the same task is executed by a number of resource providers. Apart from that, recomputing fractions of the entire task locally and (output) log analysis are alternatives. If the result verification is successful, a new due payment is registered in the transaction framework. The reputation is updated based on the behavior, e.g., if a requestor's payment is late or in case a provider does not respond (after making an agreement). Golem supports several payment schemes such as batch- and (probabilistic) nano-payments. Software developers can freely choose the payment model as long as Golem's transaction framework requirements are met [57, 63].

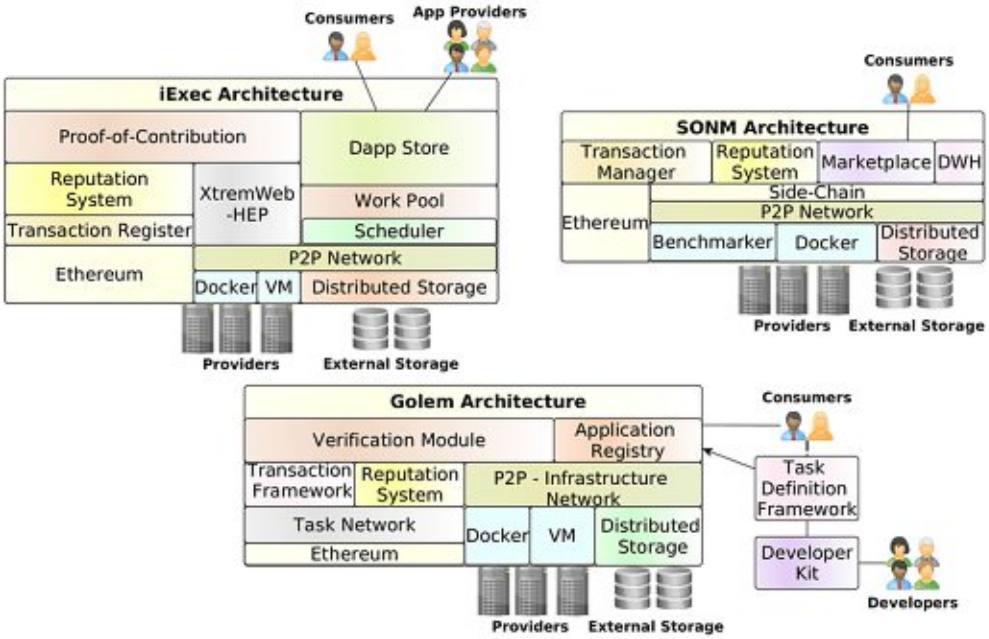


Figure 3.1: General architecture of Golem, iExec and SONM [63]

iExec

The iExec[6] was launched as project in the beginning of 2016. Their vision is to support blockchain-based distributed applications and enable cost effective high-performance computing by building a decentralized cloud infrastructure [23]. Moreover, iExec envisions to establish itself as a marketplace for companies offering computational and storage resources, data providers and SaaS applications. In this context, the cloud models IaaS and PaaS are also supported. iExec's native currency is called Run on Lots of Computers (RLC). The roadmap of iExec presents five versions divided into the three

[5]https://ipfs.io/
[6]https://iex.ec/

stages, the community-, the enterprise- and the research edition. In the beginning of 2020 version 4.1 (belongs to the enterprise edition) was released allowing blockchain-based high performance computing. The release of version five (the research edition) is planned for December 2020.

The main actors can be separated into users, computation-, app- and data providers. Users are individuals or smart contracts which order and pay for task executions. Computation providers are workers, e.g., companies with free CPU-cycles, processing the tasks of users and receiving rewards in RLC. Workers can join a worker pool where a scheduler organizes and distributes tasks. App providers create and deploy decentralized apps on the iExec platform. The fee for deployment and app registration has to be paid in Ether. Depending on the app provider's decision, apps can be used for free or for a fee. Data providers have valuable datasets which can be shared in a secure way [23].

The iExec platform is based on research projects in the field of Desktop Grid computing, developed at the INRA[7] and CNRS[8] resarch institutes. The architecture of iExec is shown in Figure 3.1. The platform uses the so-called XtremWeb-HEP desktop Grid framework for resource provisioning, data management, security and the coordination of off-chain computations. The first step in iExec is to deploy a task smart contract and the according application. Thereupon, a user can launch a computation by depositing RLC coins. Each computation is registered on the Ethereum blockchain. After completion the user can retrieve the result. To verify results, iExec has developed the proof-of-contribution[9] module. Here, the reputation and stakes of workers plus redundant computation come together. Users can specify a level of confidence with regard to the result. Based on that, the task is executed on more or less workers. The more workers are involved, the higher the cost. The confidence level is a composition of three factors: the reputation of participating workers, the score of workers with the same task result and the level of redundancy. After verification, the payment is split between workers with verified results and their reputation is updated [23, 63].

Next to this variant, iExec also supports Software Guard Extension (SGX) and therefore enclave-based off-chain computations (see Chapter 4). Consequently, the features of this verification scheme allows to prove correct computation and to protect data against unauthorized access. However, it is worth mentioning that researches have found potential security issues regarding Intel SGX enclaves [30, 55].

SONM

SONM[10] is an open-source project launched in 2017. Their aim is to establish a global decentralized marketplace of computing power by providing a decentralized fog computing platform. The whitepaper [42] states anything from web hosting to scientific calculations as potential use cases. Currently, solutions for web hosting (content delivery networks), machine learning, rendering and the hosting for decentralized apps are available. Seen

[7] https://www.inrae.fr/en
[8] http://www.cnrs.fr/en/cnrs
[9] https://docs.iex.ec/key-concepts/proof-of-contribution
[10] https://sonm.com/

from a higher level, the platform concentrates on IaaS and PaaS. SONM has its own internal currency called SNM which is based on Ethereum. News and updates are announced via the SONM Blog[11], where the roadmap can be found as well.

The SONM workflow involves at least three entities - the marketplace, a computation power supplier and a buyer. The marketplace is a smart contract where orders can be placed. It acts as the meeting point for suppliers and buyers for making deals. Orders are separated into BID and ASK orders, whereby a BID order can be created by a buyer to request resources. ASK orders originate from suppliers to offer resources. After a buyer and a supplier agree on a deal, tasks can be processed on the supplier's hardware. The use of resources is paid per use (time) in SNM.

The general architecture of SONM is displayed in Figure 3.1. Payments from resource buyers to suppliers are managed by the transaction network. All other components, e.g., the marketplace and the reputation system are executed on the side-chain. To transfer funds between the transaction- and side-chain network, a gatekeeper is used, to keep fees low. The data warehouse (DWH) monitors all operations within the side-chain and holds a copy of it to provide quick and comfortable access for users, e.g., in case of a search query. Rented resources are protected from malicious activities with the help of Docker. Kubernetes can be used to scale and manage a number of Docker containers. Besides the predefined solutions of SONM, individual applications encapsulated in Docker containers can be run on rented hardware. Before suppliers can participate and share their hardware, benchmarks are executed to classify its performance. This paves the way for fair prices on the marketplace, e.g., reduced costs for slower hardware. Like Golem and iExec, resource suppliers are connected by a P2P network. The verification of results is only addressed in SONM's whitepaper. Besides the approach of querying connection and resource usage metrics, no concrete strategy for result verification is outlined.

	Golem	**iExec**	**SONM**
Billing Model	Pay-per-Task	Pay-per-Task	Pay-per-Use
Cloud Model	SaaS	Iaas, PaaS, SaaS	IaaS, PaaS
Communication	Whisper	P2P	Whisper
Data Transfer	IPFS	URI	BtSync
Reputation System	Own Solution	Own Solution	Own Solution
Sandboxing	Docker, VM	Docker, VM	Docker
Transaction Network	Ethereum	Ethereum	Ethereum
Verification	Logs, Correctness, Redundant/High Stakes	Redundant/High Stakes	Planned

Table 3.1: Comparison commercial blockchain-based cloud solutions [63]

[11] https://sonm.com/blog/

3.2 Blockchain-based Offloading Solutions in Research

In research, several works [15, 46, 50, 66] have studied the field of blockchain-based offloading. These works mostly involve mobile- and IoT-devices plus edge/fog computing infrastructure. Tang et al. [60] additionally focus on security and privacy concerns, while Liu et al. [41] proposes a blockchain-based framework for video streaming systems.

FlopCoin is a blockchain-based offloading framework with a distributed incentive- and reputation scheme [15]. The framework's internal currency - Flopcoin - is used to reward devices that execute offloadable tasks, e.g., certain parts of a mobile application. Among other metrics, the reputation of participants is used as input for the offloading decision. Computation offloading can take place from device-to-device (D2D) or between a mobile device and cloudlets or cloud servers. In case of D2D offloading, the task issuer has to run an auction to get resource offers. The user interface of FlopCoin allows task processors to specify the amount of resources which can be allocated, e.g., 50% CPU power. The underlying blockchain processes FlopCoin transactions and is executed on the cloudlets.

EdgeChain [46] uses a blockchain and smart contracts to link edge computation resources and IoT-devices. Therefore, IoT-devices with limited computational capabilities are able to offload tasks to edge computing nodes. Due to the system being based on a permissioned blockchain the transaction throughput is increased while the need for a resource-intensive consensus protocols is eliminated. The downside is that security can be affected in permissioned blockchains, e.g., if a group of participants have an internal and secret agreement to influence transactions according to their own interests. The blockchain is used to monitor the offloading procedure and payments from task issuers to processors. To detect malicious or compromised IoT-devices, EdgeChain monitors and controls them based on their past behavior. Each IoT-device has its own profile on-chain, including parameters like past storage or CPU requests. In case of high deviations, e.g., continuous and increased resource requests, the IoT-device can be blocked.

In the work of Qiu et al. a deep reinforcement learning (DRL)-based computation offloading approach for blockchain-empowered mobile edge computing is introduced [50]. DRL is used to maximize the long-term offloading performance by enabling adaptions to highly dynamic environments. With regard to computation offloading, both data-processing and mining tasks, e.g., solving a mathematical puzzle like PoW, are supported. The approach of Qiu et al. involves IoT-devices, edge computing infrastructure and a blockchain network. IoT-devices can outsource tasks to edge servers based on a task queue concept. In addition, IoT-devices are able to participate in the mining process by instructing proxy miners that further allocate resources at edge servers to perform PoW like puzzles.

Xiong et al. also introduced a concept of edge computing for a mobile blockchain [66]. The aim of their work is to make edge computing resources accessible for IoT- and mobile devices to support their applications. The main barrier is that devices with limited resources cannot perform the resource-intensive mining process. Therefore, IoT-devices can use edge computing nodes to run the consensus process and the mobile blockchain. To

provide efficient resource management, an economic model the so-called Stackelberg game which analyzes the interaction between resource providers and miners, is introduced.

Tang et al. use blockchain techniques in a fog environment to verify each fog server's authenticity. In addition, a blockchain-based offloading approach for smart vehicles is proposed [60]. Mobile devices, such as smart vehicles are able to query closely located fog servers and their workloads. A blockchain is used to coordinate, log and monitor the offloading process. Each offloading decision but also the current workload and location of a fog server is recorded on-chain. Fog servers charge smart vehicles for processing tasks over the blockchain.

Video transcoding is a resource-intensive and time-consuming task and therefore a challenge to offload within a decentralized and blockchain-based environment. As alternative to traditional video streaming technologies Liu et al. [41] propose a blockchain-based framework for video streaming systems with mobile edge computing. To optimize the performance of a blockchain-based video system, an adaptive block size scheme is presented. The adaptive block size, the offloading scheduling and resource allocation is formulated as optimization problem with the aim of maximizing the profit for transcoders. Two offloading modes are supported, D2D and device to mobile edge computing resources. Furthermore, an incentive mechanism is introduced to enhance the collaboration between content creators, video transcoders and video consumers.

3.3 Discussion

The three commercial projects described in Section 3.1 operate in a very similar field based on internet connected infrastructure and blockchain technology. The technologies used by Golem, iExec and SONM are compared in Table 3.1. The vision and go-to-market strategy vary between the projects. Golem's first goal was to enable the offloading of 3D rendering tasks. iExec concentrates on the support of decentralized applications and SONM aims to establish a fog/edge computing environment. Based on the roadmaps on their websites, iExec seems to be most progressed. With regard to planned or already realized result verification schemes, we find that the state of development is moderately advanced. The result verification is mainly based on redundant computation, reputation and economic penalties (stake). In some cases, enclave-based computations (described in Chapter 4) are supported as well. However, it is worth mentioning that security issues have been detected in enclave-based systems (see Chapter 4). Redundant computation, recomputing fractions of tasks locally or reputation-based methods only reduce the risk of receiving wrong results [63]. Redundant computation can be vulnerable, e.g., if someone has the majority of resources. Additionally, the overall effort for offloading increases by redundant computation by executing the same task multiple times on different resources. Reputation systems can be useful, but can also hinder the market entry for new resource providers, e.g., if existing ones have a high reputation.

The recent works in research, presented in Section 3.2, do not (directly) address result verification. In the work of Pan et al. [46] a mechanism to detect malicious actions based

on past behavior is introduced. Economic penalties or banning the originator are quoted for prevention.

Overall, we conclude that methods for result verification are used, but all above described approaches only reduce the risk of receiving wrong results. We aim to focus on mechanisms that can prove if a computation was carried out correctly and therefore make the risk of receiving a wrong result negligible. In the next chapter, various result verification schemes are examined.

CHAPTER 4

Off-chain Computation and Verification schemes

When computational tasks are outsourced, e.g., by means of cloud computing providers, a task issuer has to trust the implementing entity that all steps are performed correctly. This leap of faith should be eliminated by providing mechanisms that can verify results. In other words, a result verification scheme is needed, so that task issuers have the opportunity to force the verification, which proves that a submitted result was computed properly. A result verification scheme must be selected which fulfills the requirements described in the following paragraphs.

The reasons for off-chaining in the context of computation offloading and blockchain technology are two fold. First, blockchain technology is not built for executing arbitrary complex computations. In case of the Ethereum platform, the computation's complexity is bounded by the gas limit per block. Even if the computation does not exceed this limit, the more complex the computation, the more gas is consumed and therefore the more expensive is the corresponding transaction. Second, by moving computations off the chain, it can be assumed that the redundancy of executions (each blockchain participant verifies each transaction) is reduced which improves throughput and scalability. Furthermore, off-chaining paves the way for establishing more privacy, insofar that data regarding the computation can be hidden from network participants [18, 20].

Overhead and cost
The verification of results lead to an additional value for both, task issuers and resource providers. Therefore, we assume that the statement about a proper computation justifies extra effort, since the task issuer but also the resource provider have an attestation, e.g., a cryptographic proof. The overhead resulting from result verification should be kept as low as possible with regard to cost and resource consumption.

Scalability
The result verification must be independent of the task's complexity in order to not impair the applicability. In other words, the effort for result verification should not scale with the complexity of the outsourced computational problem. Ideally, the effort for result verification is constant for every task. Otherwise, it is possible to run into limits such as the gas limit per block, when the complexity of a problem increases.

Universal applicability
The design of the platform should allow the implementation of different use cases. Therefore, a universal usability and flexibility of the result verification scheme is required, so that new categories of tasks for offloading, respectively the verification of them, can be added with low effort.

Security
The result verification scheme has to be secure, in terms of fraud and manipulation. True statements should be accepted with high probability, while the acceptance of false statements should be negligible small. In other words, the proof system or result verification scheme has to satisfy the two properties completeness and soundness as described in Section 2.3.

In research, different off-chain computation schemes have been studied. In the work of Eberhardt and Heiss [18] off-chain computation schemes including result verification are divided into 4 categories, which are described in the following sections in detail.

4.1 Verifiable Off-chain Computation

Verifiable off-chain computation entails the provisioning of cryptographic proofs that witness correct processing. After a computation is performed, a cryptographic proof is generated and published together with the result on a blockchain by the processor. Subsequently, the validity of the computation can be verified on-chain by a verifier. In case of a successful verification, the verifier can persist the result. According to [18] certain requirements must be fulfilled by verifiable off-chain computation schemes:

- Non-interactivity: As mentioned in Section 2.3 the verification step should be done in a single message, i.e., via a single transaction on a blockchain. This keeps verification cost to a minimum. Conversely, when multiple messages are necessary as in interactive schemes, higher transaction costs can be assumed [18].
- Cheap verification: In contrast to on-chain execution, on-chain verification should be cheap. Otherwise, there would be no advantage regarding cost and scalability. Cost drivers for on-chain verification are the proof size and the verification complexity.
- Weak security assumptions: The security assumptions should be as weak as possible. Stronger security assumptions lead to the necessity of additional trust which is contrary to the blockchain paradigm.

- Zero-knowledge: The zero-knowledge property ensures that private inputs on a computation are not visible on-chain. By means of the cryptographic proof, no conclusions regarding the computation or its result are possible. Thus, privacy concerns can be fulfilled.

Verifiable off-chain computation can be realized by zk-SNARKs, Zero-Knowledge Scalable Transparent ARguments of Knowledge (zk-STARKs) and Bulletproofs. As described in Section 2.3 zk-SNARKs are non-interactive and satisfy cheap verification by its succinctness. The verification effort is not proportional to the complexity of the computation. Before generating a proof and performing the verification step, a one-time setup must be carried out by a trusted party. If the setup is done by a malicious party, fake proofs can be created. To tackle this issue, multiparty computation protocols have been introduced in [7, 11, 12] that prevent fake proofs as long as one participant is honest. Unlike zk-SNARKs, zk-STARKs and Bulletproofs do not require a trusted one-time setup. In zk-SNARKs and Bulletproofs, computations are abstracted with arithmetic circuits, while zk-STARKs leverage higher degree polynomials. As a consequence, no tools are available to specify programs for zk-STARKs [18]. On the other hand, high level languages like the ZoKrates domain-specific-language (DSL) (see Section 4.6) which compile to arithmetic circuits have been introduced in [21, 40].

4.2 Enclave-based Off-chain Computation

Enclave-based systems rely on Trusted Execution Environments (TEE) which enable code execution while preserving confidentiality and integrity [18]. In other words, computations are performed off the blockchain within a trusted and isolated enclave. An enclave is a private region of memory that allows user-level code execution, but is not accessible to higher level processes and the operating system. More specifically, the application code and the corresponding data in memory are isolated from the host system. In a blockchain environment, private inputs are optionally stated by the off-chain node, while public inputs stem from the blockchain. The authenticity of an enclave can be ensured by remote attestation and a trusted third party, or local attestation [2]. The integrity of the outputs can be verified also by the attestation of the enclave. In case of a successful verification, the output can be stored on the blockchain. The Keystone-enclave project[1] and Intel SGX[37] are implementations of TEEs. Regarding Intel SGX, it is worth mentioning that a particular instruction set is required to use their enclave-based implementation. If potential task processors want to participate in such an environment, they have to check that their CPU supports Intel SGX.

Enigma [49] and Ekiden [16] are implementations that combine blockchain technology with TEEs, respectively the enclave-based approach. Enigma allows to execute computations on a blockchain, but also off the chain within an enclave. Ekiden supports only enclave-

[1] https://keystone-enclave.org/

based computations and uses the blockchain as state storage layer. Enclave-based systems are controversial, as researchers have discovered security flaws in Intel SGX [30, 55].

4.3 Secure Multiparty Computation-based Off-chaining

Secure Multiparty Computation (SMPC) protocols enable the construction of privacy-preserving off-chain computation schemes [18]. The basic idea behind SMPC is to distribute a computational task to several processing nodes whereby each node can only access a certain and meaningless share of data. That means, none of the participating nodes is able to access the whole data set, whereby privacy can be provided. After each processing node has computed its part, the outputs are merged to the final result on-chain. Proper computation can be checked by auditors on-chain.

Enigma [67] is a decentralized computation platform which leverages SMPC and a blockchain for result verification. Their goal was to pave the way for decentralized applications where privacy by design is important. However, current SMPC protocols add too much overhead to be practical [21]. Meanwhile Enigma relies on a TEE as described in Section 4.2

4.4 Incentive-driven Off-chain Computing

As the name suggests, incentive-based off-chain computation rewards nodes which are doing verification work to check if a computation is correct or not. Within the incentive-driven scheme, task solvers and competing verifiers are involved. Task solvers publish their computation results on a blockchain and verifiers check these solutions and try to detect errors. If a verifier comes to the conclusion that a submitted result is incorrect, the network participants act as collective judge and decide if the initial computation node (task solver) or the verifier is right. Depending on this decision, the solver or the verifier receives a reward.

One implementation of incentive-based off-chain computation is TrueBit [61]. A challenge when using the described scheme is to keep nodes motivated for performing verifications continuously. If only correct solutions are published, there is no chance to get rewarded as verifier. TrueBit, for example, occasionally enforces processor nodes to submit incorrect results. Consequently, a verifier is rewarded for each discovery of a wrong result. Another potential issue is the computational effort required to provide the judge with a basis for their decision. To keep that effort as low as possible an interactive verification game [38] is used.

4.5 Discussion

After comparing the concepts of the aforementioned off-chaining schemes, we decided to use the verifiable off-chain computation scheme, more precisely zk-SNARKs. Zk-SNARKs

can be applied universally and have acceptable costs compared to Bulletproofs and zk-STARKs.

In our decision, we have also taken into account the limitations regarding result verification of recent works presented in Chapter 3. By the use of zk-SNARKs, we offer an alternative that can prove correct computations and that satisfies the properties completeness and soundness (as described in 2.3).

SMPC is accompanied by high overhead, so that it is not practical [18]. The enclave-based scheme allows universal computations but has potential security issues [30, 55] and is therefore not an option. The incentive-based computation scheme works as long as there is one honest verifier. However, the throughput of completed computation tasks and the general service can be hindered by malicious verifiers by marking each computation result as faulty [18].

The differences regarding the implementations of the described off-chain computation schemes are depicted in Table 4.1. If not stated otherwise, n represents the number of multiplication gates in circuit. With regard to the verifiable scheme, it can be seen that, the off-chain part is costlier than the on-chain part for all three ZKP-systems. However, this is negligible, as the off-chain computation does not influence the transaction cost. In contrast, the verification complexity and the proof size does affect the transaction cost. The fact that the proof size and the on-chain verification of zk-STARKs and Bulletproofs grow with circuit complexity, limits their applicability. Zk-SNARKs are independent of the circuit complexity and get along with compact proves [18]. In contrast to zk-STARKs and Bulletproofs, zk-SNARKS require a one-time setup which has to be executed by a trusted party. To avoid having to trust a single party, multiparty protocols to distribute the computation of the one-time setup were introduced. Since the one-time setup is performed off-chain, it has no impact on the transaction cost.

	Off-chain computation	On-chain verification
zk-SNARKs [48]	$O(n)$	Verify: $O(1)$, Proof size: 3 group elements [31], i.e., 127 bytes for BN128 curve
zk-STARKs [5]	$O(n)$	Verify: $O(log(n))$, Proof size: a few hundred kilobytes, $log(n)$
Bulletproofs [13]	$O(n)$	Verify: $O(n)$, Proof size: a few kilobytes, $log(n)$
SMPC	$O(n)$, n number of gates in circuit [4]	On-chain Auditor: $O(n)$ n number of gates in circuit, Off-chain Auditor: None
Enclave Systems [37]	Native Execution and attestation overhead	Validate enclave's attestation: $O(n)$, signature verification
Incentive Systems	Virtual Machine Overhead (Execution History)	Binary Search and one computation step: $O(log(n))$, n number of computation steps [62]

Table 4.1: Comparison of off-chain computation schemes [18]

Now that the result verification scheme has been determined, a development tool is needed that supports the employment of zk-SNARKs on a blockchain. In Section 4.6 such a tool is presented.

4.6 ZoKrates

ZoKrates was introduced in 2018 by Eberhardt and Tai [21]. The open-source development tool is designed for the Ethereum blockchain and supports the entire process from specification, integration and deployment of non-interactive ZKPs. The toolbox consists of a DSL, a compiler and generators for proofs as well as smart contracts for verification. The generated proofs attest proper computation or vice versa a faulty one. ZoKrates uses zk-SNARKs and enables easy handling by hiding the complexity of ZKPs.

ZoKrates[2] intends to be used in the setting of off-chain computation schemes. In such a scheme, a computational task is executed off-chain. Afterwards, the result and the corresponding proof are written back on a blockchain. The proof can be verified on-chain that attests correct (or incorrect) computation. Therefore, the computational effort on a blockchain is reduced, while privacy can be preserved due to ZKPs.

The overall process is shown in Figure 4.1 and consists of the following steps:

[2] https://github.com/Zokrates/ZoKrates

1. Specification of a provable program.
2. Execution of the one-time trusted setup.
3. Generation of the verification smart contract
4. Computation of a witness.
5. Generation of a proof.
6. Verification of the proof.

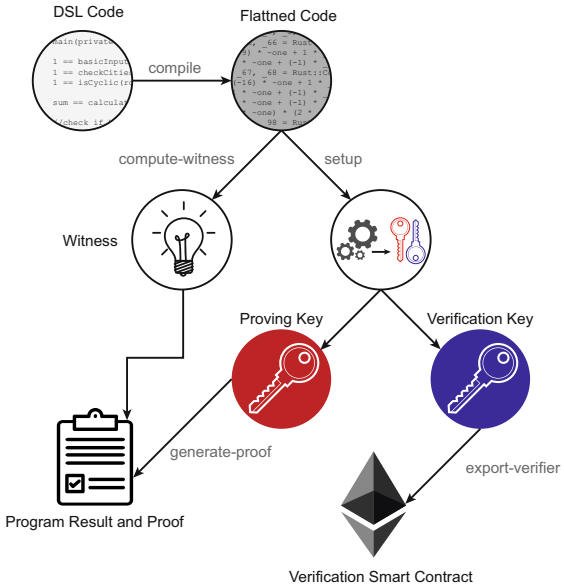


Figure 4.1: ZoKrates process at a glance [21]

First, when starting with the specification, a program has to be written to verify the validity of a witness which proves the according computation. Zk-SNARKs expect programs to be supplied in mathematical equations such as a Quadratic Arithmetic Program (QAP) or Rank-1-Constraint-System (R1CS) [21]. Since it is cumbersome and difficult to specify complex computations as QAP or R1CS, ZoKrates introduced a high-level DSL which translates to R1CS. In such a DSL program, some logic, e.g., conditions, can be stated, which allows to attest correct computation. In other words, the program takes a number of inputs (depending on the use case) and verifies if all specified conditions are met. On R1CS level this means that it is checked whether all mathematical equations hold.

Consider the offloading task of calculating the square root of a number x. In this case, the ZoKrates program must check if the computed square root in fact corresponds to the predefined number x. In other words, the equation $y = \sqrt{x}$ must hold, whereby y is

the solution of the task, computed by a worker. To realize this example, the mentioned equation must be encoded in the ZoKrates DSL. If an input parameter should not be revealed, e.g., the result y, it can be specified as private.

Listing 4.1 shows the DSL code to check the square root of x. The signature of the main function can be seen in the first line. The main function has two inputs and one (mandatory) output. Due to y being a private input, it can be kept in secret during the verification process. In line two, the actual check is performed. Since the direct calculation of the square root is not supported, the calculation is done by multiplying y. The result of $y * y$ has to be equal to x, otherwise an exception is thrown and the program aborts. If the check was successful, the value one is returned in line three.

```
1 def main(private field y, field x) -> (field):
2 x == y * y
3 return 1
```

Listing 4.1: Exemplary DSL code for verifiying the square root

After the program has been specified, the compiler converts it into so-called flattened code which consists of constraints, i.e., variable definitions and assertions. The structure of the flattened code is compatible with zk-SNARKs by a further intermediate conversation step which is executed by ZoKrates. As soon as the flattened code exists, the trusted setup can be executed, which generates the common reference string. In fact, two public keys, a verification and proving key, result. Then the Solidity[3] smart contract for verification can be generated and deployed on the Ethereum blockchain. The first line of Listing 4.2 shows the command for compiling the DSL code. After the option $-i$ the path and filename of the DSL code is appended. In line two and line three the commands for performing the setup and creating the verification smart contract are shown.

```
1 > zokrates compile -i sqrt.zok
2 > zokrates setup
3 > zokrates export-verifier
```

Listing 4.2: Statement to compile the DSL code, execute the setup and to create the verification smart contract

Based on the flattened code, a potential prover can perform the compute-witness step. Afterwards, a proof by using the witness and proving key can be created. In this context, computing a witness means that, the DSL code is executed with a certain parameter assignment that eventually evaluates to true if all resulting constraints of the DSL program are fulfilled. In other words, the goal is to find a parameter assignment so that all conditions of the program are satisfied. As mentioned and to proceed further in the process, a witness is required to create a proof. Here, ZoKrates uses the library libsnark[4] which implements zk-SNARK schemes. Both steps, compute-witness and proof

[3] https://solidity.readthedocs.io/en/v0.7.0/
[4] https://github.com/scipr-lab/libsnark

generation are conducted off-chain. Listing 4.3 depicts the commands for computing a witness and generating a proof. As can be seen in line one, the inputs (according to the signature of the main function) are passed by the option $-a$. In this example, the task was to compute the square root of 16. Therefore, the value four represents the result y and 16 corresponds to x.

```
1 > zokrates compute-witness -a 4 16
2 > zokrates generate-proof
```

Listing 4.3: Statement to compute a witness and to create a proof

The resulting proof has the structure of a json file as depicted in Listing 4.4. The proof always consists of four arrays a, b, c and inputs. The first three arrays represent three elliptic curve points that make the zk-SNARKs proof and are therefore necessary for the verification. The inputs array depicts the public inputs and the expected return value of the DSL code, in our example 16 and one. As can be seen, all arrays are in hexdecimal format. Usually, each value of an array is 32 bytes long. For space reasons the length was reduced to one byte in Listing 4.4.

```
1 {
2 "proof": {
3 "a": ["0x1a", "0x2a"],
4 "b": [["0x1b", "0x2b"], ["0x3b", "0x4b"]],
5 "c": ["0x1c", "0x02v"]
6 },
7 "inputs": ["0x10", "0x01"]
8 }
```

Listing 4.4: Exemplary structure of a proof

Finally, a potential verifier can perform the verification step. To do so the proof, the verification key and the smart contract for verification are needed. After the proof has been submitted to the Ethereum blockchain, i.e., the particular verification function of the smart contract was called, it can be determined whether the worker has computed the solution properly. By publishing the proof on the blockchain, conclusions regarding the computation and concrete result are not possible. Besides the deployment of the smart contract, merely the verification step is carried out on-chain.

Listing 4.5 outlines the verification function of our square root example. To verify a computation on-chain the function verifyTx has to be executed. VerifyTx requires the three aforementioned elliptic curve points in form of the arrays a, b, c and the inputs array. In case of a successful verification the boolean true is returned and a Solidity event is triggered.

```
1 function verifyTx(
2 uint[2] memory a,
3 uint[2][2] memory b,
4 uint[2] memory c,
5 uint[2] memory input
6 ) public returns (bool r) {
```

```
7 ...
8 }
```

Listing 4.5: Verification function in the Solidity smart contract

The entire process can be split into recurring- and one-time steps. One-time steps are executed only once per program, e.g., the specification of the program, the trusted setup and export of the verification smart contract. The generation of a witness and proof as well as the verification of the proof can be performed arbitrary times for a program.

Until now, we have decided to use zk-SNARKs for result verification. To implement such a result verification scheme, the above described toolbox ZoKrates can be used. With the help of zk-SNARKs it is possible to prove if a particular computation was correct or not. In the context of zk-SNARKs, a proof, e.g., generated with ZoKrates, attests computational validity, but does not state anything about the solution or its quality. Especially in the case of outsourcing optimization problems, it is beneficial to compare solutions by their quality. Therefore, we aim to integrate an auction-based mechanism to the proposed solution approach, as described in Chapter 5.

CHAPTER 5

Blockchain-based Offloading with Result Verification

This chapter describes the requirements and the fundamental architecture of the blockchain-based offloading approach including result verification. Based on the requirements defined in Section 5.1, design decisions are derived and discussed in Section 5.2. The design consists of two main components, the auction smart contract and the result verification. The auction smart contract acts as basis for the auction-based mechanism of our computation offloading solution. The second part, the result verification is responsible to check if the computation of a particular problem was conducted in a proper way.

5.1 Task Offloading Requirements

When offloading tasks two parties are involved. Task issuers potentially have limited computational capabilities and therefore are interested in outsourcing particular tasks. Conversely, task processors may have idle computational resources and offer their CPU-cycles to process these tasks. To enable task offloading, the solution approach has to act as platform for both the task issuers and processors. Potential task issuers have the possibility to create and publish their tasks for offloading. Task processors should be able to pick these tasks, compute them and return the results. For each processed task, respectively for renting resources out, the task processor receives a payment from the task issuer.

The task offloading procedure has to be implemented in such a way that the requirements defined in the following paragraphs are met.

Transparency
The offloading procedure should be as transparent as possible, so that each step of the procedure is comprehensible to every participant of the proposed solution. Due to

complete transparency, it is not necessary to trust other entities, because every step can be checked anyway. Another reason for transparency is that malicious activities can be detected easily. What we understand by a malicious activity is for example when someone wants to generate an advantage at the expense of others. In the context of our offloading procedure, malicious behavior can be undertaken at both ends, e.g., a task issuer never collects the result so that a task processor receives no reward or a task processor submits fake results. Regardless of the originator, malicious activities have to be prevented.

Incentive structure
Task processors are rewarded for computing tasks of task issuers. An appropriate pricing scheme is necessary that motivates potential processors for participation. At the same time, the fees should be attractive for task issuers as well. Overall, it can be helpful to keep an eye on the prices of other offloading solutions in order to be able to compete. For this reason, operational costs should be considered in design decisions and kept low from the beginning.

To prevent the aforementioned malicious behavior originating from the task issuer, a stake can be useful. In case of irregularities which can be attributed to the issuer, the stake is withhold and used to reimburse the task processor. In case of an honest task issuer, at least a part of the stake can be used to reward the task processor.

Entry barriers and openness
The participation as task issuer and task processor should be as simple as possible. Ideally, the participants can freely join and leave the platform without any barriers. A user registration and deposit of payment details as it is common with traditional cloud providers is not needed. An existing relationship and a position of trust between a task issuer and a task processor is not required.

Computation offloading should be possible directly between a task issuer and a task processor. In other words, to initiate and perform the computation offloading procedure, trusting a centralized third party should not be necessary. In addition, by doing the offloading without trusted third parties, potential commission cost can be saved.

Auction-based mechanism
Competition between task processors should be promoted so that task issuers have the opportunity to receive the best possible result. Especially, when optimization problems are offloaded, an auction-based mechanism can be useful, because more than one solution is valid. To cover such a functionality, it is necessary that several solutions can be submitted per task. This further enables the possibility that task processors can compete with each other. A prerequisite for competition is that solutions can be ranked. Therefore a criteria to derive the quality of the solution is required. Another necessity is that an offloading cycle has a limited duration to prevent a permanent lock of the task issuer's stake in case no solution is provided. In the end of the offloading cycle, it would be beneficial if the task issuer has the possibility to select between the submitted solutions.

5.2 Design

This section covers the fundamental architecture of the blockchain-based offloading approach. Design- and architectural decisions as well as possible implications are discussed extensively. It is essential that the requirements defined in Section 5.1 are taken into account and fulfilled by these decisions. Based on that, the technology stack for the auction-based offloading mechanism is determined. Since we have discussed various result verification schemes in Chapter 4 and in order decided to use zk-SNARKs, a concrete process to employ result verification with zk-SNARKs in the context of our auction-based mechanism is outlined.

Figure 5.1 shows the components of our solution approach. As can be seen, the system consists of a task issuer client, a task processor client, an auction smart contract and a further smart contract for result verification. The smart contracts are executed on a blockchain. Each component is precisely described in the following sections.

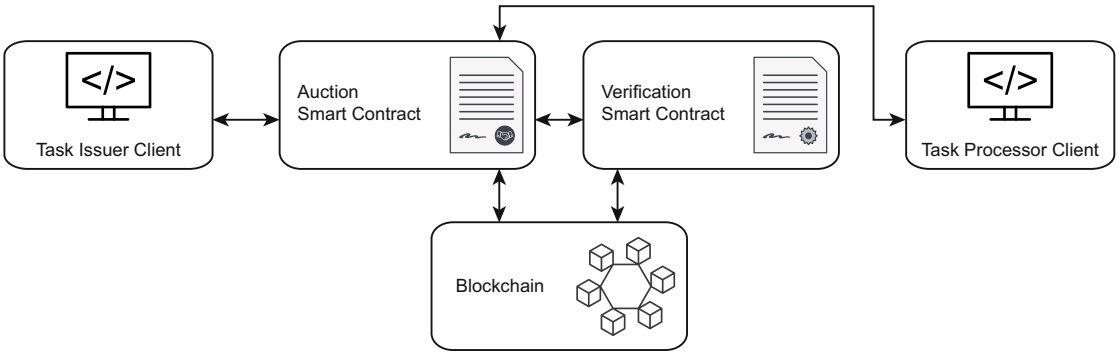


Figure 5.1: Overview: Blockchain-based computation offloading with result verification

5.2.1 Blockchain as Infrastructure

As depicted in Figure 5.1, blockchain technology is involved in the system. It is planned to use a blockchain for payments and as underlying infrastructure for the offloading and result verification process. The main reasons for this decision are transparency, low entry barriers, the public nature of blockchains, the fact that no centralized entity is necessary and network participants do not need to trust each other. To realize the blockchain-based offloading approach, a suitable blockchain has to be selected. Here, it is essential that the selected blockchain supports the execution of smart contracts. In addition, a broad acceptance and market penetration of the respective blockchain is advantageous.

Among others, the Ethereum blockchain but also Hyperledger Fabric support smart contracts. Hyperledger Fabric is an open-source gerneral-purpose permissioned blockchain which is primarily used in business-to-business scenarios. A permissioned blockchain provides a way to secure the interactions among a group of entities that have a common goal but which do not fully trust each other [3]. Compared to public blockchains, the use

of permissioned blockchains requires a registration process and participants must know (and trust) each other. Ethereum is one of the public blockchains where everyone can join. For this reason but also due to the wide distribution and community we decided to use Ethereum as underlying blockchain. In addition, ZoKrates the tool which is used for result verification (see Section 4.6) targets the Ethereum blockchain.

5.2.2 Auction Smart Contract

To cover the auction-based mechanism, as required in Section 5.1, appropriate logic additionally to the offloading procedure is needed. Both functionalities are combined in one smart contract.

By means of smart contracts and an underlying blockchain, trust in centralized third parties is dispensable. Therefore, direct interactions between task issuers and processors can be achieved. In other words, the auction smart contract acts as meeting point for task issuers and processors to close the offloading deal. To perform one offloading cycle, no preexisting relationship and no position of trust between a task issuer and potential task processors is required. These properties are ensured by the blockchain network. As described in Section 2.2 smart contracts allow to encode actions that can be called by network participants after the contract was deployed on a blockchain. These actions can be abstracted, respectively summarized within functions. As we decided to use the Ethereum platform, the auction smart contract can be written in various languages, e.g., Solidity[1] or Vyper[2]. Again, by reason of the wide distribution, Solidity is used for the implementation.

States and functions

The auction smart contract passes through several stages between the start and the end of an auction. Consequently, different states are used to structure an auction's lifecycle. Figure 5.2 depicts the auction smart contract as state diagram. It consists of the states *ready*, *running* and *ended*. For the transition between the states, particular functions must be called. In the following paragraphs, the states and functions shown in Figure 5.2 are described in detail.

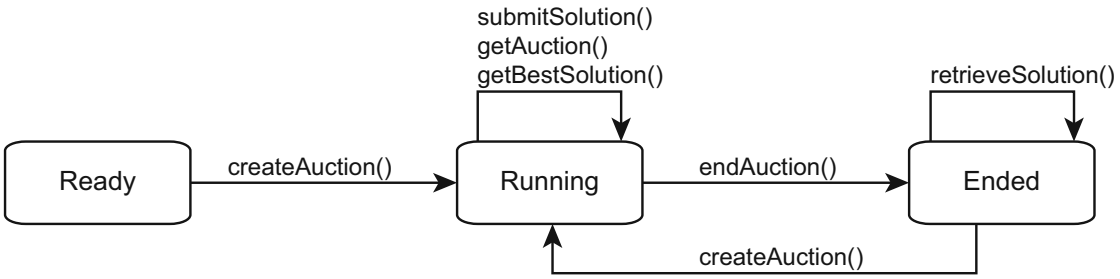


Figure 5.2: Simplified state diagram of the auction smart contract

[1] https://solidity.readthedocs.io/en/v0.7.0/
[2] https://vyper.readthedocs.io/en/latest/index.html

Before any interaction is possible or a function is callable, the auction smart contract must be deployed on the blockchain. After the deployment, it is in state *ready*. At this point, a task issuer is able to create a new auction, more specifically a new task.

As mentioned in Section 5.1 a stake is required for creating an auction. To fulfill this requirement, the function *createAuction()* can be executed only if the call includes a coin transfer. The stake must be deposited in the cryptocurrency of the underlying blockchain. To give an example, the cryptocurrency of the Ethereum blockchain is called Ether. Therefore, a task issuer has to deposit the stake in Ether. Further inputs of *createAuction()* concern information about the task and if the result has to be verified.

By making the result verification optional, the task issuer has more flexibility and can choose if verification should be enforced. Apart from that, both versions can be compared easily plus possible impacts of conducting result verification can be derived when our approach is evaluated (see Chapter 7). When the verification is set to true, the task issuer also accepts potentially higher cost. The mentioned inputs plus the account address of the task issuer (caller) and the earliest auction end time are temporarily stored in the smart contract and thus on a blockchain, e.g., the Ethereum blockchain. The previously mentioned auction end time represents the time limit, respectively the minimal duration of an auction. As clarified in Section 5.1, the temporal restriction is required to prevent a permanent lock of the entire system by one single auction. After *createAuction()* was executed successfully, the auction is in state *running*.

When entering the state *running*, the task order is completed and task processors are able to retrieve information about the task, e.g., by calling *getAuction()* or *getBestSolution()*. If a potential task processor has decided to contribute to a particular auction, processed results can be submitted by calling *submitSolution()*.

The function *getAuction()* returns all necessary information to process the task. From the perspective of a task processor, we assume that several factors are relevant to participate in an auction. Besides a financial reward, one decision-making criteria can be related to already submitted solutions. By getting an insight in the quality of these solutions, a potential task processor might be able to estimate if a better solution can be found with its own algorithms and resources. Therefore, the auction contract provides the possibility to retrieve information about the currently best submitted solution. As mentioned, the ranking is done by a quality criteria. This value can be fetched by calling the function *getBestSolution()*.

The submission of results can be done by executing the function *submitSolution()*. Calling this function requires that the caller has already computed the particular task. Depending on whether the result verification is mandatory, different inputs have to be provided by the task processor. More precisely, according to verification is mandatory or not, two different functions for submission are available (this separation is omitted in Figure 5.2 for simplicity). When calling *submitSolution()*, the solution and its quality criteria have to be delivered. At this point, network participants including the task issuer are able to see the submitted solution due to the public nature of the blockchain. In theory, a task

issuer is able to freely read the submitted solution without properly ending the auction. However, payment for task processors is ensured as a stake was locked at task creation.

In the described scenario, the payment is conducted before result delivery. Equally, it would be possible to pay after result delivery. Both approaches have pros and cons and are discussed in section Solution Delivery within this section.

After successfully calling the function *endAuction()* the auction smart contract is in state *ended*. In addition, the reward is transferred to one or more task processors which have submitted a result. Detailed information regarding the payment model can be found under Payment within this Section. A prerequisite for executing this function without getting an error is that the minimal duration has been exceeded. After this point of time, every network participant has the possibility to end the current auction by calling *endAuction()*. When the auction was ended, the task issuer is able to collect a solution by calling the function *retrieveSolution()*. Certainly as mentioned above, the solution can also be read by the transaction history, due to the public nature of a blockchain. Regardless of whether *retrieveSolution()* was executed or not a new auction can be created. Calling *createAuction()* in state *ended*, triggers a state transition to state *running*, which closes the auction's cycle.

In the current architecture, the auction smart contract is able to process one auction at a time. Parallel auctions are only possible by deploying further auction smart contracts. However, the auction smart contract is designed in a way so that the implementation of parallel auctions is possible with manageable effort, e.g., by adding an unique identifier to each new auction.

As can be seen in Figure 5.2 three states and five transitions are used. Based on this state diagram, optimizations are possible, e.g., by adding states for a clearer separation of the auction's lifecycle. However, more states lead to potentially higher cost caused by an increased amount of write operations on the blockchain. Conversely, less states, e.g., only *ready* and *running* or *running* and *ended*, would also be feasible, but possibly lead to a lack of clarity.

With regard to the state changes in Figure 5.2, a state transition to state *ready* (instead of *ended*) would be plausible and probably more intuitive when executing the function *retrieveSolution()*. Thus, the read operation would turn into a write operation causing higher cost. To keep cost as low as possible, this was not implemented. Likewise, a dedicated time frame for calling *retrieveSolution()* after an auction has ended would benefit the task issuer, because the solution can be retrieved regularly and in an comfortable way. This can be done by adding a further time limit, which is set in *endAuction()*. Consequently, the task issuer would have a fixed time window to collect the result. Currently, a new auction can be created as the state *ended* is reached which overrides the previous one including the results. However, in case of a task issuer does not collect the result before a new auction is created, it can be read out by means of the transaction history. Due to cost savings and the transaction history as possible fallback, the above mentioned additional time limit for *retrieveSolution()* is omitted.

Figure 5.3 shows one possible sequence regarding the interactions between the involved entities. First, the task issuer I_A creates an auction. Thereupon the task processor P_A fetches relevant information regarding the task and decides to contribute. After a result was processed off the chain, P_A writes the solution back on the blockchain. The second task processor P_B also fetches the task of I_A plus the current best solution. Task processor P_B may assume that they cannot outperform the current best result and does not participate. After the minimal duration of the auction is passed, the auction can be ended, in this case it is done by the task issuer itself. Subsequently, the task issuer retrieves the result with the best quality criteria.

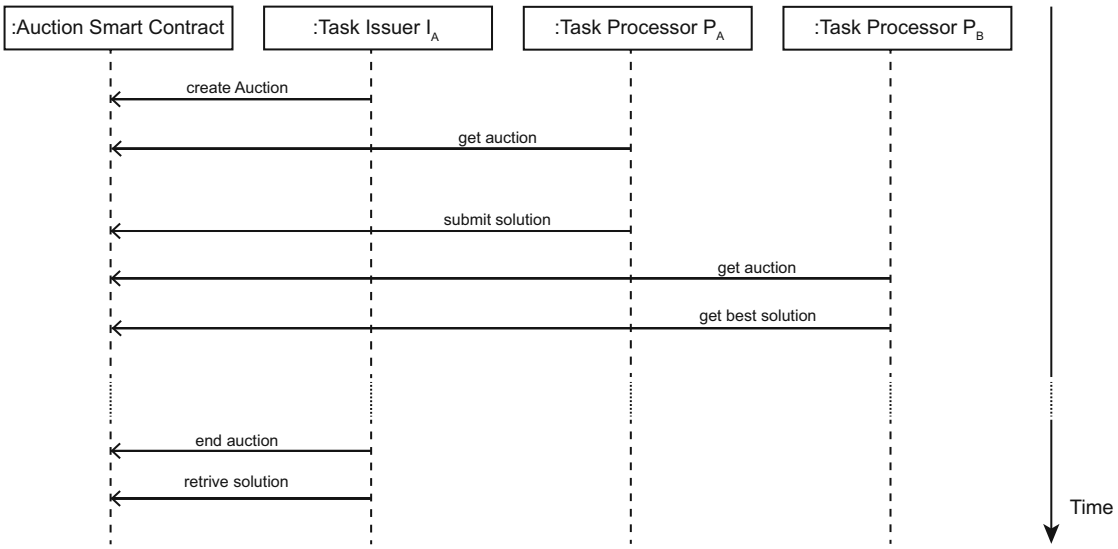


Figure 5.3: Exemplary process between the auction smart contract, one task issuer and two task processors

Access-levels

It is essential, that the provided functions of the auction smart contract are called in the correct order. To achieve this, the aforementioned states *ready*, *running* and *ended* are introduced to restrict the sequence. For particular functions, a restriction based on the caller's role is required. This especially concerns the function *retrieveSolution()* insofar as only the task issuer is permitted to call it. Due to the public nature of blockchain technology, transactions can be read by every participant. Therefore, participating entities are able to read also submitted solutions by means of the transaction history, independently of the the access-level of a smart contract function. However, to ensure the regular procedure of an auction, we decided to implement role-based access, e.g., for the function *retrieveSolution()*. Both, the sequence and the role-based access can be covered by using so-called function modifiers provided by the smart contract language Solidity.

Solution selection

With several solutions submitted per task, the task issuer can potentially choose any one of these solutions. Consequently, it is conceivable that task processors define the price for

their solution so that the task issuer can weigh between the solution's quality and price. This increases flexibility but also requires a more complex logic. Due to higher expected execution cost, we consider a simpler approach based on the quality of the solution only. To keep the amount of interactions and cost low, the auction smart contract selects always the solution with the best quality criteria, after an auction was ended. As a result, the automatically selected solution can be retrieved by using *retrieveSolution()*.

Solution delivery

Before different versions regarding the solution delivery are discussed, it must be decided how the solution is stored within the auction smart contract. One option is to store solutions as plaintext within the auction smart contract. This way, the plain solution is an input parameter of *submitSolution()*, subsequently it is also possible that other network participants can read it in the transaction log of the blockchain. Depending on the task, privacy might suffer and the solution can be stolen by the task issuer without payment. The latter problem can be mitigated by the placement of a stake before the auction starts. When the auction smart contract knows the plain solution before the auction is ended, the solution delivery can be realized as read-only function. This approach is simple and does not need enhanced logic which can be seen as advantage. The downside is a higher storage space demand on the blockchain.

Due to the public accessibility of a blockchain's transactions, solutions that are stored as plaintext can be read by anyone. As mentioned previously, we ensure the payment of a task processor by the stake deposited at task creation and consequently assume that there is no incentive for a task issuer to read a possible solution in advance before the auction was ended. However, the fact that solutions are publicly readable can only be circumvented by not storing the solution as plaintext. One option is to only store a hash of the solution on-chain. By doing this, the solution itself remains private. However, if task processors do not submit plain solutions during an auction, another mechanism is needed so that the task issuer is able to retrieve it after an auction was ended.

As such, the solution delivery can be conducted in two steps. While an auction is running, only hashed solutions plus its quality in plaintext are submitted. After the auction was ended, the solution with the best quality criteria is selected and the corresponding task processor is notified. Subsequently, the task processor submits the solution as plaintext. After the smart contract has ensured that the plain solution match with the hash, the task processor receives payment and the task issuer is able to retrieve the plain solution. Participating task processors have to deposit a stake, to ensure the subsequent upload of the plain solution after the auction was ended. The stake is refunded to those participants whose solution is not selected. The task processor with the winning solution gets a payment consisting of its own stake and the reward for processing from the task issuer. Since task processors have to deposit a stake within the described two-step submission variant, it is questionable, if both the hash and the quality criteria is in fact necessary at the first submission step. We assume that further optimizations are possible by removing the hash of the first submission step. In other words, the stake and the quality criteria are sufficient. In case of the task processor with the winning solution does

not upload a plaintext solution (in the second step) which corresponds to the quality criteria (submitted in the first step), his stake can be withhold. Therefore, the input parameter for the hash in the first submission step can be discarded, which results in cost savings.

The advantage of the two-step variant is, that the plain solution is only available after the end of an auction and overall less data must be stored on the blockchain because only the winning solution is written on the blockchain. Consequently, cost caused by the reduced on-chain storage demand can be saved. Disadvantageous is that more logic and interaction is necessary. Additionally, privacy cannot be achieved as long as the solution is submitted as plaintext (in the second step) on the blockchain instead of ciphertext. By means of, e.g., public key encryption like OpenPGP[3] it can be ensured that only the respective task issuer can read the solution (after decryption). However, in this case the necessary exchange of the public key have to be kept in mind. As can be seen, a trade-off between transparency and privacy has to be accepted.

Another variation of the described two step delivery process can be done by replacing the blockchain storage for task solutions with other storage services, e.g., IPFS. IPFS is an decentralized system for data sharing, where participating entities do not need to trust each other [8]. When IPFS is used, the link to the solution can be submitted (instead of the plain solution) which potentially saves cost. By comparing the hash of the data shared via IPFS with the hash on the blockchain, the integrity of the solution can be ensured.

However, to keep the solution delivery as simple as possible, we decided to use the plaintext solution as input parameter for the function *submitSolution()*. Therefore, both the solution submission of a task processor and the solution delivery to a task issuer can be carried out in each case by a single function call.

Payment
As mentioned in Section 5.1, a task processor is incentivised for participation by a financial reward. The question arises who determines the price or reward and how it is composed. As computational tasks are offloaded, it is possible that a task issuer pays per task or also per time unit. Further, it can be questioned if its useful to pay only one or more participating task processors.

As assessment basis for the price, the fixed charges on the task processor's side can be considered. The fixed charges consist of two main parts, the cost for consumed energy and the transaction fee for submitting the solution. The energy cost can be calculated by means of the current energy price, the amount of time units used for processing the task and the energy consumption per time unit. The transaction fee caused by calling *submitSoluiton()* depends on the amount of data written on the blockchain. As long as the processing time cannot be forecasted, the payment per time unit seems to be fair for both ends. In case of a pay per task model, a discrepancy regarding the price

[3] https://www.openpgp.org/

and the processing time is possible, which can be an advantage for the task issuer and disadvantage for the task processor, or vice versa.

Further it is conceivable that the task issuer proposes a price and task processors can decide if they want to participate for this price or not. However, when task issuers or processors are able to define or even negotiate a price, we assume an increased need of interaction, e.g., caused by a varying pricing per time coming from the resource availability of a task processor. In general, we do not expect an added value or impact on the evaluation of the proposed solution approach from a pricing model. Therefore no particular pricing, e.g., which aims at maximizing profit of task processors is implemented. For reasons of simplicity, we decided to implement a minimal price per task which is set by the owner of the auction smart contract. If the solution approach is positioned on the market in the future, an appropriate pricing model must of course be integrated. In this respect, a pricing model based on the processing effort can be provided. Furthermore, by rewarding, e.g., the three best solutions, the competition between task processors can be promoted. Recent works [23, 25] in the field of blockchain-based computation offloading have introduced various pricing models. Golem, for example, has implemented a probabilistic pricing, where only a few processors are paid per transaction [63].

Besides the kind of payment, the time of payment has to be considered as well. Payment before, at and after result delivery are possible. Due to simplicity, we have planned to implement the former.

Payment before result delivery means, that a task processor receives the payment before a task issuer is able to call the appropriate function to retrieve the result. In our scenario, the payment is triggered within the function *endAuction()*. The result delivery is possible by calling *retrieveSolution()* whereby as a precondition the auction must be ended. As mentioned before, this has the upside of a simpler implementation and less demand of interaction. From the perspective of the involved entities, payment before delivery is beneficial for task processors as their effort is compensated before the respective task issuer has confirmed the solution delivery. On the other hand, task issuers run the risk of paying for a potentially useless solution. However, due to the result verification employed in our solution, task issuers have a statement about the correctness of computed solutions.

In the context of our running example, payment at result delivery means that the coin transfer is conducted when the function *retrieveSolution()* is called. Here, a fallback is needed in case of the task processor never retrieves the solution by calling *retrieveSolution()*.

Payment after result delivery can be implemented as alternative to the above mentioned variants. Payment after result delivery can be realized by adding a confirmation function, which triggers the coin transfer and is only callable by the task issuer. A task issuer has the possibility to retrieve and take a look at the result, before the payment is triggered. Further, if the task issuer accepts the solution, the confirmation function can be called to complete the auction. This implies additional interaction and exception handling, due to the auction completion depends on the task issuer's reaction. In addition, we assume

that there is less demand to inspect the solution before payment, since result verification takes place. Compared to payment before delivery, a task processor runs the risk of not receiving a reward with payment after delivery.

5.2.3 Result Verification

The second fundamental part of the proposed solution, is that it can be determined whether a submitted result was computed properly. Thus, potential task issuers obtain a statement about the validity of solutions to their corresponding task.

Until now, we have decided to use the Ethereum blockchain as underlying technology. As a consequence, we need a result verification scheme which can be built on top of the Ethereum blockchain. Various result verification schemes are described in Chapter 4. Based on the comparison in Chapter 4, zk-SNARKs seem to be the most suitable approach for our blockchain-based offloading solution. Due to its succinctness, very short proofs (in the ballpark of bytes) can be provided which is beneficial when blockchain technology is involved. We further decided to use ZoKrates (described in Section 4.6) to employ result verification because it can be used in combination with the Ethereum blockchain.

ZoKrates is a toolbox to run zk-SNARKs on Ethereum. While the computation and a few preparation steps regarding the result verification are carried out off the blockchain for cost-savings, the verification step is done on-chain.

Figure 5.4 gives a high level overview of result verification within the proposed solution approach. Solutions of verifiable tasks are written to the blockchain only if the result verification is successful. Otherwise, the submission is discarded. Solutions of tasks for which verification is not mandatory are stored on-chain without any checks. The activity *verify Solution* is part of the result verification while all other activities in Figure 5.4 belong to the auction smart contract. The verification is performed within a separate smart contract generated by ZoKrates. Therefore, the auction smart contract has to trigger the verification function *verifyTx()* in the verification smart contract.

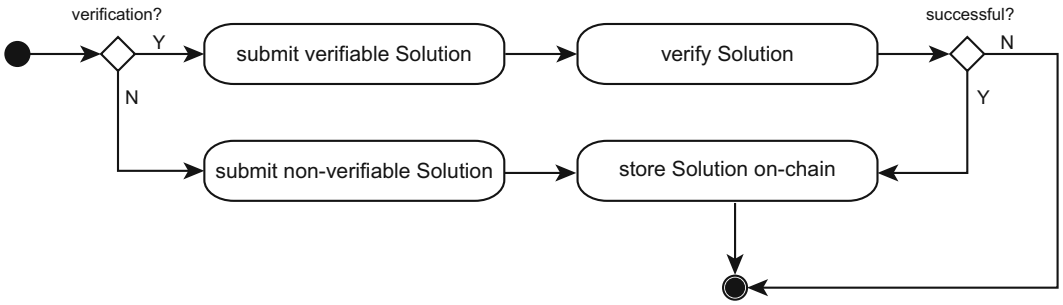


Figure 5.4: High level process of the result verification

The procedure of the result verification is illustrated in Figure 5.5, whereby the yellow activities are supported by the ZoKrates toolbox. First, a task processor decides to contribute to a particular auction where verification is mandatory and retrieves all

relevant data. Based on that, the task processor computes the task locally (and therefore off the chain) with its own algorithm. After a solution was found, a witness has to be computed. This means that the previously mentioned program specified in DSL code is run with the task processor's result as input. If it succeeds, it returns a witness that proves proper computation. Otherwise, it can be assumed that an error occurred during the computation phase or a wrong result was entered. Based on the witness, a proof can be generated, which is needed for the on-chain verification in the next step. Both actions, compute witness and generate proof are performed locally on the task processor's hardware. Afterwards, the solution and the proof can be submitted. When a solution is submitted, the auction smart contract calls the verification function *verifyTx()* of the verification smart contract. If the verification function returns true, we can assume that the computation was executed honestly. As a result, the solution is stored on-chain. Otherwise, if false was returned, the submission of the task processor is discarded entirely.

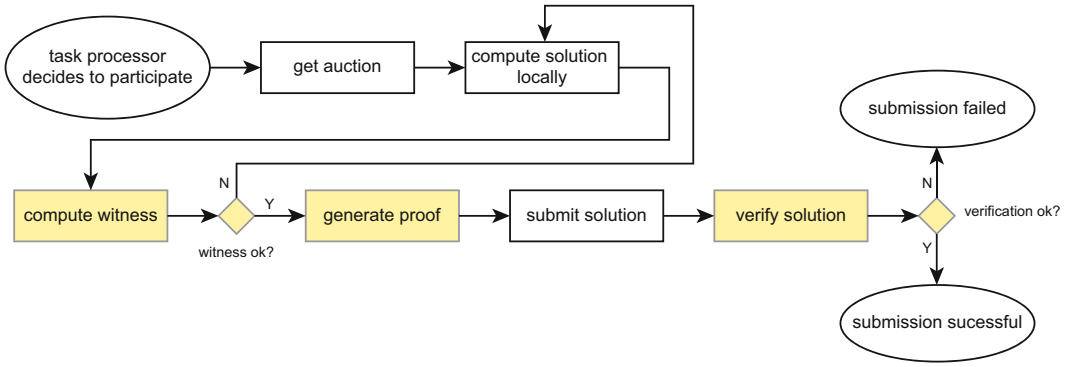


Figure 5.5: Result verification procedure

Implementing specific use cases

As described in Section 4.6 the result verification with ZoKrates is tied to specific computation problems. This means that a certain part of the result verification, more specifically the DSL code, is not generic and must be changed whenever a different computation problem has to be served. The DSL code contains appropriate checks to prove that a computation is done correctly. The possibility of creating or adopting these programs enables flexibility and adds universal applicability to the result verification. In other words, new use cases can be added and thus potential demands of task issuers for new computation problems can be met.

To verify results of a specific computation problem, several one-time preparation steps have to be conducted. As indicated, a program in form of DSL code has to be written. Then, keys and a verification smart contract have to be generated. Afterwards, the on-chain verification for the specific computation problem is ready for deployment. The mandatory steps to add a new use case are depicted in Figure 5.6. Since it is not planned that users can add additional computation problems by their own, all these actions have to be done by the developer. When the result verification scheme is integrated in the auction smart contract, the inputs of the function *submitSolution()* as well as the

storage format for solutions of the new problem should be kept in mind. Under certain circumstances, it can be useful to draw up a detached auction smart contract for each use case. A separation per use case would lead to more compact code artifacts and better maintenance.

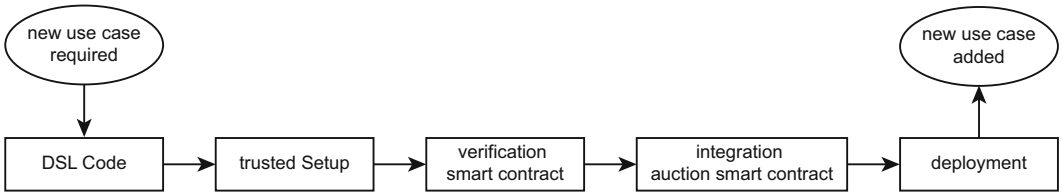


Figure 5.6: Process for adding result verification for additional use cases

To demonstrate how our blockchain-based offloading approach with result verification can be adopted to specific use cases, we describe the process of implementing an exemplary use case in the next chapter.

CHAPTER 6

Implementation

To demonstrate and evaluate the effectiveness of the proposed solution, we implement our blockchain-based offloading approach using the TSP as exemplary use case. The TSP is an optimization problem which involves finding the shortest path between predefined cities. Each city has to be visited once and the start city must be identical to the end city. Figure 6.1 shows a possible solution for an exemplary TSP instance.

Figure 6.1: Exemplary TSP solution path for the cities Budapest, Vienna, Prague, Berlin, Amsterdam, Brussels

The TSP is an NP-complete problem and potentially hard to compute. Therefore, offloading TSP instances can be a promising solution approach, e.g., for resource-constrained devices. Especially for larger problem instances, it is unknown if an optimal solution can be found within polynomial time. Once a solution has been found, its validity can be verified in less time, e.g., by traversing the path of a TSP solution. In research the TSP is used as benchmark to test optimization approaches solving combinatorial

problems [56]. Therefore, but also because of more than one solution or path per TSP instance is possible, the TSP is truly suitable for our auction-based offloading approach.

The terms task issuer and task processor are generally described in Section 5.1. With regard to the TSP use case, a task issuer is able to create a TSP instance, by publishing the cities of interest. Subsequently, task processors can retrieve key data of this instance and further decide if they want to participate. In case of participation, a task processor computes a solution for the TSP instance and provides it for the task issuer. The effort of calculating an TSP instance is compensated by a reward at the end of the offloading cycle.

Note that the TSP is merely used as exemplary use case. It is fairly easy to apply the solution to other optimization problems as well (see Section 5.2). However, particular parts of the implementation are directly related to properties of the TSP such as the logic of the result verification. To verify TSP solutions, the following requirements have to be considered:

- Each city appears exactly once in the stated path (Hamiltonian cycle)
- The given path length must correspond to the sum of the connections between the cities on the basis of the path and the adjacency (distance) matrix

The result verification has to ensure that the stated path is a Hamiltonian cycle and its given length is equal to the calculated path length based on the distance matrix. A distance (adjacency) matrix represents a map consisting of a number of cities and its distances.

This chapter focuses on the implementation of the TSP as an exemplary use case and is structured as follows: First, the implementation details regarding the auction smart contract are outlined in Section 6.1. Afterwards, we describe the implementation of the result verification with zk-SNARKs by means of the toolbox ZoKrates in Section 6.2. Here, we go along with the defined process illustrated in Figure 5.6 to add a new use case. In Section 6.3 an optimized variant of result verification with zk-SNARKs is presented. Finally, Section 6.4 comprises two client interfaces, which simplify interacting with the implemented prototype.

6.1 Auction Smart Contract

According to Section 5.2.2, the auction smart contract covers the auction-based mechanism and the integration of the result verification. Due to the result verification being tied to the number of cities (see Section 6.2), several verification smart contracts have to be linked with the auction smart contract. Apart from the employment of result verification, the auction smart contract allows to create auctions (tasks), submit bids (solutions) including their verification, rank bids, end an auction and to retrieve the best solution.

For the development and evaluation phase, the Truffle Suite[1] is used. The Truffle Suite enables the operation of a local Ethereum blockchain and supports the creation, deployment and testing of smart contracts. All smart contracts are implemented by means of the language Solidity (v5.8.0).

Data structures and storage demand

The auction smart contract stores task details and its solutions on the blockchain. The corresponding data structures are depicted in Listing 6.1. Each `Task` can contain several instances of the struct `Solution` (see line 13). The names of the cities specified by the task issuer are stored as `bytes32[]`. The variable `path` within a solution is of type `uint[]`, because ZoKrates supports only numbers (see Section 6.2 for further details). The transformation from city names to numbers (or city Ids) is carried out by a separate function within the auction smart contract. The necessary mapping between city names and its Ids is stored on-chain as `bytes32[]`, whereby the array index is equal to the Id of a city. Since TSP instances are computed off-chain, it is not required to hold the distance matrices on-chain. The variable `mapnumber` within a solution indicates which map was used to compute the solution.

```
1  struct Solution {
2  address payable processor;
3  uint[] path;
4  uint mapnumber;
5  uint sum;
6  }
7
8  struct Task {
9  address issuer;
10 bool verification;
11 bytes32[] cities;
12 uint stake;
13 Solution[] solutions;
14 State state;
15 uint auctionEnd;
16 }
```

Listing 6.1: Data structures for storing tasks and solutions

Skeleton of the auction smart contract

As specified in Section 5.2.2 the lifecycle of an auction is divided into the three stages *ready, running, ended*. Therefore, we apply the state machine pattern[2]. This pattern allows to pass trough different states and can be used to offer various functionality depending of the current state. The ingredients for the state machine pattern are states, state transitions and interaction controls for the functions.

Listing 6.2 shows the basic code skeleton of the auction smart contract. The states are defined as `enum` in line five. Based on the definitions in Section 5.2.2, the state transitions are represented by corresponding functions. A state transition function is only callable

[1] https://www.trufflesuite.com/
[2] https://fravoll.github.io/solidity-patterns

if the contract is in the correct state. This interaction control is realized with so-called modifiers which can be applied to functions. For example, the modifier *inState()* starting in line eight ensures that the actual state is equal to the expected one. Otherwise, an exception is thrown. This modifier is applied to the function *submitSolution()* in line 22. As can be seen, the modifier *inState()* guarantees that function *submitSolution()* can only be called in state *running*. In contrast to *inState()*, the modifier *orState()* allows function calls within two states `op1` and `op2`. Thus, it is possible to create an auction in the states *ready* or *ended*.

Besides the state machine pattern, also the guard check and the access restriction pattern are utilized[3]. The guard check pattern makes sure that, e.g., the input parameters of a function are as expected. The guard check pattern can also be a part of modifiers as can be seen in line nine. By means of the access restriction pattern, the ability of calling functions can be restricted. For example, the function *retrieveSolution()* can be called only by the task issuer (ensured by the modifier *equalsAdress()*).

```
1  contract Auction {
2
3  struct Solution {...}
4  struct Task {...}
5  enum State {Ready, Running, Ended}
6  Task task;
7
8  modifier inState (State actual, State expected){
9  require(actual == expected, 'useful-err-msg');
10 _;
11 }
12
13 modifier orState (State actual, State op1, State op2){
14 require(actual == op1 || actual == op2, 'useful-err-msg');
15 _;
16 }
17
18 function createAuction(...)
19 external payable orState(task.state, State.Ready, State.Ended) {}
20
21 function submitSolutionVerifiable(...)
22 external inState(task.state, State.Running) {}
23
24 function endAuction()
25 external inState(task.state, State.Running) timeExceeded() {}
26
27 function retrieveSolution()
28 external view inState(task.state, State.Ended) equalsAddress(task.issuer) {}
29 }
```

Listing 6.2: Simplified code skeleton of the auction smart contract

Create auction

A new auction (or task) can be created by calling the function *createAuction()*. The city names and whether result verification is mandatory or not have to be stated as input. Moreover, the function call must include a coin transfer which is used as stake. Otherwise,

[3] https://fravoll.github.io/solidity-patterns

an exception is thrown. Consequently the struct `Task` is filled with the consigned data. The variable `auctionEnd` represents the minimal duration of an auction and is set to the current blocknumber + 5. Therefore the auction can be ended five blocks later at the earliest.

Submit verifiable solution

In case the task issuer requires a verified solution, a task processor can submit its result by calling *submitSolutionVerifiable()*. Two functionalities are provided by *submitSolutionVerifiable()*. First, it ensures that solutions for tasks can be submitted. Second, it acts as integration point for the result verification and therefore triggers the verification with zk-SNARKs.

The function signature of *submitSolutionVerifiable()* is illustrated in Listing 6.3. The inputs from line 2 - 5 are part of the proof that was created off-chain based on the task processor's computation. The input `_hp` in line nine represents the hashed path. The function *submitSolutionVerifiable()* first compares `_hp` with the hash of `path`. Second, the on-chain verification step is executed, i.e., calling the function *verifyTx()* in a verification smart contract (which was generated with ZoKrates). Before that the input variables for *verifyTx()* must be prepared. If the on-chain verification step was successful, the result is stored within the struct `Solution`.

```
1  function submitSolutionVerifiable(
2  uint[2] calldata _a,
3  uint[2] calldata _b1,
4  uint[2] calldata _b2,
5  uint[2] calldata _c,
6  uint[] calldata _path,
7  uint _mapnumber,
8  uint _sum,
9  uint[2] calldata _hp
10 ) external inState(task.state, State.Running) {...}
```

Listing 6.3: Function signature of *submitSolutionVerifiable()*

The function signature of *verfyTx()* is depicted in Listing 6.4. Here, the cryptographic proof is verified (as described in Section 4.6). In case of a successful verification, we can assume that the task processor was honest and computed the solution properly. In the positive case, a Solidity event is emitted and the boolean value true is returned. Otherwise, false is returned.

```
1  function verifyTx(
2  uint[2] memory a,
3  uint[2][2] memory b,
4  uint[2] memory c,
5  uint[14] memory input
6  ) public returns (bool r) {...}
```

Listing 6.4: Function signature of *verifyTx()* to verify a path of length 10

End auction

The function *endAuction()* terminates the currently running auction if the minimal

duration is already exceeded. Furthermore, the solution with the best quality criteria is determined automatically and its task processor gets paid. In the current version the stake of the task issuer is transferred.

Retrieve solution
With regard to the auction's lifecycle, the function *retriveSolution()* can be accessed only by the task issuer after the auction has been ended. If these requirements are fulfilled, the solution with the best quality criteria is determined and returned. Here it has to be kept in mind, that all data on a blockchain is public. Therefore, solutions can be read as soon as they are submitted. Since the task issuer has to deposit a stake at task creation, the payment for the task processor is ensured.

Supportive functions
Further functions have been implemented to support the auction-based procedure. From the view of a task processor it is necessary to know details with regard to the current task, e.g., the cities of interest. Moreover, a task processor may want to know whether a solution has already been submitted and what quality it has. To cover such cases, two functions *getAuction()* and *getBestSolution()* are provided. The first function returns the cities names entered by the task issuer. To learn about the currently best solution, the sum of the shortest path is given back by *getBestSolution()*.

6.2 Result Verification

As described in Section 5.2.3, we aim to use zk-SNARKs for result verification. To realize this plan, the toolbox ZoKrates is utilized which accompanies the entire verification process. The basis and beginning for the result verification is a program written in the DSL which proves program inputs against the defined checks (see Section 4.6). For example, if task processors intend to prove that they know the preimage of a hash, the DSL program has to check if the preimage hashes to the expected result, i.e., $e = h(p)$ whereby e is the expected result, $h()$ the hash function and p the input representing the preimage. Basically, arbitrary problems can be covered by such programs. With regard to our running example, the result verification has to verify whether TSP solutions were computed properly. In other words, task processors should be able to prove that they know a valid solution for an TSP instance. To accomplish that, the requirements to verify a TSP solution outlined in the introduction of this chapter have to be considered. In short, the stated path has to be Hamiltonian and the path length must match.

As indicated in the paragraph Data structures and storage demand in Section 6.1, maps or distance matrices are identified by a map number. The idea behind identifying maps is that we aim to support several maps, e.g., one map with European cities and another with North-American cities to enhance flexibility for task issuers. To tackle this, a couple of components must be aware of the predefined maps. At least, the auction smart contract and the DSL program must know which maps and cities are valid. To verify the validity of a path and its length, the result verification component needs the distance matrix of each map.

Before we go into further details about the implementation of these requirements, we find it useful to outline supported features and limitations of ZoKrates as there are implications for the implementation.

Restrictions of ZoKrates
Currently, the DSL of ZoKrates (v0.5.1) supports only a small set of types and control flow tools, in contrast to common programming languages like Java. There are two primitive types, *field* and *bool* available. The type *field* is a positive integer between 0 and $p - 1$ (p represents a large prime number between $2^{253} < p < 2^{254}$). The second primitive type *bool* can be used in conditions only. Further, two complex types, arrays and structs can be used. An array consists of several *field* or *bool* types and can be one- or two-dimensional. The size of an array must be defined at compile-time. A struct allows to combine various primitive types within one container variable [19].

With regard to the control flow, function calls, if-expressions and for-loops are possible. By defining functions the program can be better structured. The condition within an if-expression supports the operators <, <=, >, >= and ==. However, equality operators should be preferred due to the evaluation (the compute witness step) can be done with less effort. When using for-loops, it must be kept in mind that the bounds must be known at compile-time which implies that only constants (and no variables) can be applied [19].

Furthermore, ZoKrates provides a standard library, which comprises functions for hashing and public-key cryptography.

Implications on the specification of the DSL program
The mentioned restrictions have effects on the design of the proposed solution. Because only numbers are supported by ZoKrates, a mapping from city names to numbers is necessary. Since bounds of loops have to be defined at compile-time and the lack of dynamic fields (like an array list in Java which can grow during program execution), considerations have to be made regarding different amounts of cities per task. Out of the box, it is not possible to write a DSL program which supports a varying number of cities as input. Instead, the fields for the path consisting of cities must be fully defined. This means, that if five fields for the cities are provisioned by the DSL program, all five fields have to be filled. Since NULL Values are not supported, an alternative approach must be provided.

Based on the described limitations, the following design decisions are made:

- Depiction of a map: As mentioned before, each map covers particular cities and its distances whereby each city is connected with each other (equivalent with a completed graph in graph theory). These data structures can be stored as array within the DSL program. The identification of particular cities can be done with the help of the array indices since only numbers are possible. The maps provided in this solution approach are fitted for the symmetric TSP, i.e., the distances between two cities (A to B and B to A) are equal. The asymmetric TSP can be easily added in the future, e.g., by adding new maps. When new maps are added

	0	1	2	3
0	0	100	200	300
1	100	0	400	500
2	200	400	0	600
3	300	500	600	0

Table 6.1: Exemplary distance matrix

Name of City	Id
A	0
B	1
C	2
D	3

Table 6.2: Exemplary mapping

or existing ones are changed, the whole solution must be redeployed. Table 6.1 illustrates an exemplary distance matrix which consists of four cities. While the auction smart contract must only know which indices and city names are valid, the verification module and the task processor client interface need the distance matrices for operation. When task processors participate on a TSP instance, they use the distance matrices and the indices of the cities in their algorithms. Therefore, task processors do not necessarily need to know the names of the cities.

- Mapping: A conversion from cities to numbers and vice versa is mandatory since the DSL of ZoKrates does not support chars or strings. Based on the definition of maps, where the array index identifies the city, the mapping can be built on these indices. That means, the set of valid numbers is fixed by the distance matrix. As a consequence, a city can only be uniquely identified by the map. The assignment of cities to indices is determined by the distance matrix, which can be arbitrarily structured by the developer team. Table 6.2 illustrates a mapping according to the distance matrix in Table 6.1. At least, the mappings must be accessible for the auction smart contract. Since task processors submit a sequence of numbers representing the path to the auction smart contract, it is beneficial if the task issuer client is already aware of the mapping. As such, an on-chain conversion from numbers to cities can be omitted, saving additional cost.

- Lack of dynamic fields: To circumvent this limitation, the maximum amount of cities is partitioned by providing several programs in the ZoKrates DSL. The separation is done in steps of 10, so that there is a verification program for 10 cities, 20 cities etc. Depending on the quantity of cities, task processors have to execute the compute witness step with a different program. For example, if a path for four cities has to be computed, a task processor runs the verification program for 10 cities. In our example six fields from the fifth to the tenth city are occupied by placeholders. In our prototype, the first invalid index of the distance matrix (the array) is used as placeholder. Accordingly, when the distance matrix of Table 6.1 is used, the first invalid index is five. One alternative is to use a fixed number as placeholder, e.g., 254, which is valid for all maps. However, this limits the maximum number of cities on a map which we consider as disadvantage.

6.2.1 Specification of the DSL program

This section presents the main parts of the implementation for verifying TSP solutions. As previously mentioned, several DSL programs are necessary, because ZoKrates does not support dynamic fields. Due to the fact that all these programs are very similar and differ only in minor details, e.g., the amount of loop iterations, we present the code artifacts for the verification of up to 10 cities. Each DSL program supports two maps or distance matrices to achieve flexibility. The presented version contains one map with 30 and one with 70 cities.

Input parameters for the ZoKrates program
To verify that a task processor has computed the solution for a TSP instance properly, the following information is crucial:

- The path consisting of a sequence of numbers representing cities, e.g., 0, 1, 2, 3
- The map number which was used for solving the TSP instance
- The length of the path, e.g., 1400
- The cities, more specifically their numbers, which were stated by the task issuer at auction creation, e.g., 0, 3, 2, 1
- The hashed path, e.g., the value of *sha256(0123)*

The path and the map are needed to calculate the overall distance and to compare it with the stated length of the task issuer. Additionally, by the help of these two parameters, the validity of the city numbers can be ensured. The path and the cities are used to determine if all cities are covered exactly once. The hashed path is necessary to prevent malicious behavior originating from the task processor when submitting a solution. Without the hash, it would be possible to decouple the proof from the path. In other words, if the task processor submits a valid proof (based on a valid witness) but an invalid (e.g., random) path, it possibly cannot be detected on-chain. The result verification would succeed even though an incorrect path would be stored on blockchain. To recognize and prevent such scenarios, we decided to compare the hash and the path within the ZoKrates program and in the auction smart contract.

As described in Section 4.6, input parameters can be declared as private or public in the ZoKrates DSL. Private inputs are not part of the on-chain verification whereas public parameters are required to perform the verification step on the blockchain. Therefore, private inputs can be kept in secret by the task processor. We decided to make the parameters path and map as private so that its basically possible to hide the solution. Since we defined in Section 5.2 that the plain solution has to be submitted, the advantage of keeping these two inputs private is neutralized. However, if another solution delivery mechanism is implemented which does not disclose the path, private parameters are beneficial. The main function of the DSL program to verify TSP solutions, is depicted

in Listing 6.5. As can be seen in the function's signature, a path up to 10 cities can be verified.

```
1  def main(
2    private field[10] path, private field mapnumber,
3    field sum, field[10] cities, field[2] hashOfPath
4    ) -> (field):
5
6  1 == basicInputCheck(path, cities, mapnumber)
7  1 == checkCities(path, cities, mapnumber)
8
9  sum == calculateSum(path, mapnumber)
10
11 field[2] hashedPath = hash(concat(path))
12 hashOfPath[0] == hashedPath[0]
13 hashOfPath[1] == hashedPath[1]
14
15 return 1
```

Listing 6.5: Main function of the DSL program

Input parameters for the verification function *verifyTx()*
The verification function *verifyTx()* is executed on-chain, while the DSL program runs off-chain. The inputs for *verifyTx()* differ slightly from the inputs of the DSL program. Instead of the private inputs, the proof of the task processor has to be stated. The proof generated by ZoKrates contains three elliptic curve points, which are mandatory for the verification step. Additionally, all public inputs and the expected return value of the DSL program have to be provided by the verifier to perform the verification. In our case, the verifier is the auction smart contract that holds the needed information and triggers the verification. With regard to the return value of the DSL programs, we decided to return always the integer one.

Bounds check
A bounds check can be useful to sort out solutions with invalid indices or map numbers like a format check. Listing 6.6 shows a snippet of the function which performs the bounds check. Since the map numbers 30 and 70 are supported (see line three), indices between 0 - 29, 0 - 69 and the placeholders 30 and 70 for unused path or city slots are valid. The for-loop (see line 5 - 10) ensures that the arrays `path` and `cities` only contain valid indices and placeholders. In addition, the placeholders of both fields are summed up and compared in line 12. The function *inputCheck()* returns the value one if a valid input was entered. Otherwise in case of an invalid input, e.g., the path or cities array contains the number 71, an exception is thrown and the processing is stopped.

```
1  def inputCheck(field[10] path, field[10] cities, field mapnumber) -> (field):
2  ...
3  1 == if mapnumber == 30 || mapnumber == 70 then 1 else 0 fi
4  ...
5  for field i in 0..10 do
6    0 == checkBounds(path[i], mapnumber)
7    0 == checkBounds(cities[i], mapnumber)
8    iCitySlots = iCitySlots + if cities[i] == mapnumber then 1 else 0 fi
9    iPathSlots = iPathSlots + if path[i] == mapnumber then 1 else 0 fi
```

```
10 endfor
11 ...
12 iCitySlots == iPathSlots
13 ...
14 return 1
```

Listing 6.6: Function to conduct the bounds check

Hamiltonian cycle

Next, it has to be examined if the stated path contains each city exactly once. The corresponding function specified in the DSL is illustrated in Listing 6.7. The examination is conducted within the for-loop from line 3 - 6. The function *containsOneTime()* (line four and five) checks if an element of the path occurs exactly once in the cities array and vice versa. Due to placeholders being possible (which are equal to the map number), the check has to be done in two ways. If the placeholders could be replaced by dynamic fields, line four and the comparison of the length of both arrays would last out to verify the Hamiltonian property. If all conditions shown in Listing 6.7 hold, the value one is returned and the execution continues with the comparison of the path length.

```
1 def checkCities(field[10] path, field[10] cities, field mapnumber) -> (field):
2
3 for field i in 0..10 do
4   1 == if cities[i] == mapnumber then 1 else containsOnetime(cities[i], path) fi
5   1 == if path[i] == mapnumber then 1 else containsOnetime(path[i], cities) fi
6 endfor
7
8 return 1
```

Listing 6.7: Function to check the Hamiltonian cycle property

Comparison of the path length

This part checks if the stated path length (field `sum` in the signature of the main function) is equal to the sum resulting from the distances of the stated path based on the distance matrix (map). While the stated path length and the calculated path length are compared in the main function of the DSL program (Listing 6.5, line 9), the summation is shown in Listing 6.8.

In line four, the appropriate distance matrix is fetched. We decided to use a one-dimensional array for distance matrices because its processing is cheaper in contrast to two-dimensional arrays. As mentioned before, two maps (distance matrices) of potential different size are supported currently. However, it can be extended with low effort, so that, e.g., three or four maps are possible. Since the summation is done in one function and the size of arrays has to be known at compile-time, the size of the array `map` in line four follows the largest map. In our case, the largest map contains 70 cities and requires therefore 4900 fields. Because of the aforementioned placeholders for unused cities in the path, a padding is necessary to overcome addressing issues (e.g., see line 11). The overall size results from calculating $map_{size} * map_{size} + map_{size} = array_{size}$, which is in our example $70 * 70 + 70 = 4970$. Since the field with index 4970 has to be addressable as well, the array in line four is 4971 fields long. Thus, the fields from 0 - 4899 contain

valid distances, while fields between 4900 and 4970 serve as padding and are set to the value zero.

With regard to the summation, from line 7 - 8, the distance between the last and the first city is calculated. The for loop from line 10 - 13 sums the distances between the first and the penultimate cities. Line 11 covers the transition from a one-dimensional to a two-dimensional array by computing the according position in the distance matrix. Line 12 accesses the (distance) value of the previous calculated position and adds it to the `field sum`. Afterwards, the computed total sum is returned in line 15 to the caller, i.e., the main function.

```
1  def calculateSum(field[10] path, field mapnumber) -> (field):
2
3  field sum = 0
4  field[4971] map = getMap(mapnumber)
5
6  //end - start
7  field pos = (path[getIndexOfEndpoint(path, mapnumber)] * mapnumber) + path[0]
8  sum = sum + map[pos]
9
10 for field i in 0..9 do
11   pos = (path[i] * mapnumber) + path[i + 1]
12   sum = sum + if path[i + 1] == mapnumber then 0 else map[pos] fi
13 endfor
14
15 return sum
```

Listing 6.8: Function to calculate the path length

Hashing

As described at the beginning of Section 6.2.1, it is necessary to embed the hash of the path in the verification procedure. Otherwise, cheating originated by the task processor is possible by submitting a valid proof (based on a correct computation) and an invalid plain text path to the auction smart contract. Therefore, the hash of the path is needed as input. Consequently, we have to compute the hash of the stated path and compare it with the input `field[2] hashOfPath`, to prevent the aforementioned malicious action. Fortunately, the standard library of ZoKrates provides an implementation of SHA256[4].

Nevertheless, a few challenges have to be solved to meet the above requirements. First, the SHA256 function of ZoKrates expects a binary input (of type `field[256]`). Therefore, the input parameter path, which is a (decimal) array of type `field`, has to be concatenated and transformed to a binary format so that it can be further passed to the SHA256 function. Second, it has to be kept in mind that a path can start with index zero. This must be accounted for, otherwise a leading zero can be truncated. Finally, the value range of type `field` is between $2^{253} < p < 2^{254}$, where p is a large prime number. Depending on the amount of elements in the path, bit flips can occur if the value range of type `field` is exceeded.

[4] https://zokrates.github.io/toolbox/stdlib.html

To overcome these challenges, we decided to slice the stated path into groups of 10 elements and insert the number one before each group. For example, the path $[0, 2, 1, 3, 70, 70, 70, 70, 70, 70]$ is transformed to 10213707070707070. This separation has two advantages. First, the partition size of 10 matches employed partitions of cities, therefore no additional logic, e.g., via padding is mandatory. Second, the number resulting from concatenating 10 elements is smaller than 2^{128}. Assuming that each element is smaller than 100, 128 binary digits are sufficient. Furthermore, up to 20 path elements, respectively, two slices can be passed to the SHA256 function. For computing a hash for 30 or 40 elements, the function SHA512 can be used which expects two inputs of type `field[256]`. Equally SHA1024 is leveraged for 50 or 60 path elements. The function *unpack128()* allows to transform a variable of type `field` to its binary representation.

Listing 6.9 shows the function for computing a hash with a path length of 10. As this function expects only one `field d1` as input, the concatenation function *concat()* (see 6.5, line 11) has to be applied prior to that. With the help of *pack128()* in line 10 and 11, the binary representation is converted to a decimal format. In line 13 the computed hash (divided into two parts `res0` and `res1`) is returned. The comparison of the computed and the stated hash of the path is conducted in the main function of the DSL program (see 6.5 line 12 and 13).

```
1  def hash(field d1) -> (field[2]):
2  field[256] block = [0; 256]
3  field[128] dummy = [0; 128]
4
5  field[128] b1 = unpack128(d1)
6
7  block = [...dummy, ...b1]
8
9  block = sha256Padded(block)
10 res0 = pack128(block[..128])
11 res1 = pack128(block[128..])
12
13 return [res0, res1]
```

Listing 6.9: Function to compute a sha256 hash

6.3 Result Verification at Constant Cost

As described in [21] and based on our experiments with ZoKrates, the number of inputs of the verification function *verifyTx()* is a cost factor. With regard to our use case and the specification of the input parameters in Section 6.2.1 the number of cites scales with the instance size and therefore is a cost factor. In the first version, the input for cities in the DSL program is public and therefore also part of the verification function *verifyTx()*. The more cities are stated, the higher the gas cost for executing *verifyTx()*. Therefore, we have decided to conduct optimizations targeting the number of input parameters. In the following, the optimized variant is referred as result verification at constant cost.

To get rid of the above mentioned scale factor, we have set the input for the cities to

private and added one public input which expects the hashed cities. By this change, the fee for executing *verifyTx()* can be pushed to constant gas cost, regardless of the size of the TSP instance.

To provide an auction-based mechanism with result verification at constant cost, primarily adoptions of the DSL programs are required. Since the signature of the main function within DSL programs change, the verification function *verifyTx()* is also affected. Minor changes have to be carried out within the auction smart contract.

By implementing the previously mentioned optimizations, a task processor has to provide the path, the cities, the path length and the hashes of the path and the cities, when executing the compute witness step. The verification function expects the proof, the path length, both hashes and the return value of the ZoKrates program, which should be the value one. The adopted signature of the main function within the DSL program is illustrated in Listing 6.10.

```
1 def main(
2 private field[10] path, private field mapnumber, private field[10] cities,
3 field sum, field[2] hashOfCities, field[2] hashOfPath
4 ) -> (field):
5 ...
6 return 1
```

Listing 6.10: Function signature for verification at constant cost

In contrast to the original smart contract the main difference can be seen in the function signature *verifyTx()* in Listing 6.11, as the size of the `input` array (line five) is six. The input array consists only of the path length, hash values for the path and the cities plus the expected return value of the DSL program. Therefore the structure follows this pattern: [pathLength, hashOfCities1, hashOfCities2, hashOfPath1, hashOfPath2, 1]. This version can deal with solutions independently of its size, e.g., a solution of length 10, but also 60.

```
1 function verifyTx(
2 uint[2] memory a,
3 uint[2][2] memory b,
4 uint[2] memory c,
5 uint[6] memory input
6 ) public returns (bool r) {...}
```

Listing 6.11: Function signature of *verifyTx()* with constant input parameters

With regard to the adoptions within the auction smart contract, only the signature of the function *submitVerifiableSolution()* is affected, because additionally the hash of the cities is necessary. The auction-based mechanism itself including the data structures and solution delivery remain the same, e.g., solutions are sill written as plaintext on the blockchain.

Within the auction smart contract, another scalability problem comes up when a solution is submitted (regardless of verification is mandatory or not). Here, the number of elements

of the input variable path can be a cost factor.

The composition of a path has been considered as the successor city is stated only, e.g., (0, 1, 2, 3), instead of predecessor and successor, e.g., ((0,1), (1,2), (2,3), (3,0)). This path layout halves the demand of inputs and storage space on the blockchain. Further optimizations of the path length are not possible, since we have decided in Section 5.2.2 that the plain solution has to be submitted for a simple solution delivery.

6.4 Client Interfaces

For reasons of usability and to make it easy for task issuers and processors to interact with the proposed solution, two client interfaces are implemented. All possible actions of task issuers and processors, e.g., collecting and submitting a result are supported by the interfaces. The client interfaces are implemented as command line interfaces written in Java which allow the input of commands according to the available interactions of the solution approach.

The task issuer client supports the following actions:

- Creating an auction
- Ending an auction (after the auction's time limit is exceeded)
- Retrieving the solution

The task processor client supports the following actions:

- Ending an auction (after the auction's time limit is exceeded and in case of the task issuer does not respond)
- Retrieving necessary data regarding an auction
- Getting the quality criteria of the currently best solution
- Preparation and delivery of a solution

6.4.1 Nearest Neighbor Heuristic

With regard to solution strategies for TSP instances, exact algorithms, e.g., a branch and bound algorithm [17], and heuristics were developed [53]. Providing such algorithms and heuristics is not a core part of this thesis, task processors ideally implement their own approach to solve TSP instances. The idea is to reach a greater variety of different algorithms and thus enhance the probability to find the shortest path. However, we implemented the nearest neighbor heuristic as Java application for evaluation purposes.

As the name nearest neighbor indicates, the nearest city to the current one is visited next. According to Rosenkrantz et al., a path with the nearest neighbor heuristic is constructed as follows [53]:

1. Start with an arbitrary node.
2. Find the node not yet on the path which is closest to the node last added and add to the path the edge connecting these two nodes.
3. When all nodes have been added to the path, add an edge connecting the starting node and the last node added.

Each node can be referred as city. At the beginning, all cities entered by the task issuer are marked as not visited. Then, an arbitrary city is picked as starting point and marked as visited. Afterwards, the city with the shortest distance to the current one is selected. This is continued until all cities are visited.

Our implementation works as described above, with the exception that the start city is always the first city stated by the task issuer. The Java application expects distance matrices plus the cities as text-based files and outputs a path as possible solution.

Since we have outlined implementation details of the auction-based offloading approach with result verification in the context of the TSP as exemplary use case, we analyze and evaluate our approach in the next chapter.

CHAPTER 7

Evaluation

This chapter presents the evaluation of the blockchain-based computation offloading approach with result verification. The evaluation can be divided into two main parts, i.e., the requirements and the quantitative analysis. The latter is done by means of the TSP, which is implemented as exemplary use case (see Chapter 6). Overall, the evaluation aims to examine the following aspects of the proposed solution:

- Requirements analysis: Requirements for the task offloading procedure and the result verification scheme have been defined. The fulfillment of these requirements is analyzed in detail.
- Overhead of result verification: Possible overheads in terms of time and cost are determined.
- Identification of cost drivers: Potential indicators of the cost development are outlined.
- Cost efficiency: The cost caused by the submission of a solution and its verification with zk-SNARKs are compared to on-chain verification (without cryptographic proofs). Based on that, the preferable verification variant with regard to cost and instance size is derived.

The evaluation chapter is structured as follows: In Section 7.1 the requirements analysis is presented. Before the quantitative analysis is outlined, the evaluation setup is described in Section 7.2. Next, in Section 7.3 the overhead of result verification with zk-SNARKs in the setting of the auction-based environment is investigated. In Section 7.4 different variants of result verification are analyzed and compared in detail, so that potential cost drivers can be identified. Finally, the results with regard to the (cost) overhead and cost-efficiency are discussed in Section 7.5.

7.1 Requirements Analysis

Prior to the design and development of the blockchain-based offloading approach, we defined requirements for the task offloading procedure and the result verification scheme (see Section 5.1 and the introduction of Chapter 4). Our claim was to meet these requirements by our design decisions and also by our implementation. The following two sections briefly describe each requirement and evaluate how they are fulfilled.

7.1.1 Task Offloading

This section evaluates the requirements defined for the task offloading procedure (see Section 5.1). The following paragraphs examine why and how transparency, openness, an auction-based mechanism and an incentive structure are achieved.

Transparency
Each step of the task offloading procedure should be transparent to all participants. As a consequence, no participant has to be trusted. Furthermore, it is required to easily detect malicious activities.

We decided to use a blockchain (i.e., the Ethereum blockchain) as underlying infrastructure. Caused by the public nature of blockchains, data and programs on the chain are visible for all participants. Furthermore, each step in the context of our offloading procedure that changes the state on the Ethereum blockchain is logged and publicly visible. For example, each order of a task issuer can be traced by all participants. Thus, transparency can be achieved. Unlike traditional cloud providers, our solution approach does not require trust in a third party since the state or the validity of a transaction is determined through blockchain technology. In other words, each state change is executed and verified by all participants.

A malicious activity can be understood as someone trying to gain an advantage at the expense of others, e.g., when a task processor submits a fake result. For such cases, we provide a result verification scheme. The result verification is performed on-chain, meaning that all participants verify this operation. Therefore, malicious activities can be detected easily. Due to the transparency and public nature of a blockchain, submitted results can be read by every participant. Therefore, it is possible that a task issuer reads the result and does not pay the task processor. Such a scenario is mitigated since a stake has to be deposited at auction creation which ensures the payment of a task processor.

Entry barriers and openness
Task issuers and processors should be able to freely join and leave the platform. No trust or relationship between task issuers and processors is necessary.

As mentioned in the paragraph above, we leverage blockchain technology to realize computation offloading. Due to its openness, task processors with spare resources can freely join and leave the computation offloading system. Equally, task issuers can join without any barriers and can start to offload tasks. Caused by the fact that a blockchain

is organized in a decentralized manner and participants agree on state changes, no relationship or position of trust is needed between task issuers and processors.

Since we use the Ethereum blockchain, both task issuers and processors need Ether, e.g., to deposit the stake or to pay the transaction fee when a smart contract function is called. In case that a participant does not contribute to the mining process and therefore earn a reward, an exchange of fiat money to crypto coins, e.g., Euro to Ether, is necessary. To conduct such an exchange, a registration at a cryptocurrency exchange platform, e.g, Bitpanda[1] or Coinbase[2] is required, which can be seen as entry barrier. Unlike conventional cloud providers, our solution approach does not require an upfront registration before participation.

Auction-based mechanism
To enable competition between task processors and to provide the best possible result to task issuers, an auction-based mechanism allowing the submission of multiple results per task is needed.

This requirement is covered by a separate smart contract, which specifies the auction-based mechanism. An auction is divided into the stages *ready, running* and *ended*. While an auction is in stage *running* multiple results can be submitted by task processors. A submission contains the computed solution and its quality criteria. With the help of the quality criteria, solutions can be ranked and compared. Task processors are able to retrieve the quality criteria of the currently best submitted solution. Through these functionalities, task processors can compete with each other.

Incentive structure
When a task is offloaded by a task issuer, the resource provider, i.e., the task processor receives a fee in exchange for a task computation. The price or reward should be attractive for both, task issuers and processors.

As mentioned previously, a deposit of a stake is necessary to successfully create an auction, i.e., the request to offload a task. Thus, the payment for a task processor is ensured. After an auction ends, the task processor who submitted the solution with the best quality criteria (i.e., the shortest path) receives the stake. In the current version, a minimal and fixed price for each arbitrary task is required. Since the minimal price is the lower bound for the stake, task issuers can influence the attractiveness of their offloading request and the responsiveness of task processors. In future work, a dynamic pricing scheme based on the computational complexity of the task may be advantageous.

7.1.2 Result Verification

For the result verification part separate requirements have been defined in the introduction of Chapter 4. This section evaluates why and how scalability, low cost, security and universal applicability are achieved.

[1] https://www.bitpanda.com/
[2] https://www.coinbase.com/

Scalability
The effort to execute the result verification should not scale with the complexity of the offloaded computational problem. Otherwise the applicability of the result verification scheme can be limited.

Blockchains can be used to realize computation offloading, but they are not built for executing resource-intensive tasks. An off-chain computation scheme can be beneficial, because limitations such as the block gas limit can be avoided. In addition, the effort to verify a result on the blockchain ideally does not correspond to the problem's complexity. Both, off-chain computation and the independence of an off-chain computation and its effort for verification can be achieved by zk-SNARKs [21].

Overhead and cost
The cost to verify a submitted result on-chain should be kept as low as possible.

We decided to leverage zk-SNARKs to integrate result verification in our blockchain-based computation offloading approach because the properties of zk-SNARKs are promising with regard to keeping cost a minimum. As mentioned in the above paragraph, the on-chain verification cost does not depend on the problem's complexity. Due to zk-SNARKs being succinct and non-interactive, proofs are short compared to the computation length and can convince a verifier with a single message. With regard to our implementation (see Section 6.3) and quantitative analysis (see Section 7.4.2), the result verification can be done at constant cost.

Security
The result verification scheme has to be secure in terms of fraud and manipulation.

Zk-SNARKs satisfy the properties completeness and soundness as outlined in Section 2.3. Therefore, correct computations are accepted with a very high probability while incorrect computations are accepted with a negligibly small probability. Apart from that, zk-SNARKs require a trusted one-time setup that has to be executed by a trusted party. If the trusted party is dishonest, it is possible to create fake proofs. To overcome this potential issue, multiparty computation protocols have been introduced [7, 11, 12]. In the setting of our implementation, the trusted setup was executed by the development team (without using a multiparty computation protocol). In case of deploying our solution on the Ethereum mainnet, the multiparty computation protocol would provide greater assurance that the trusted setup was executed honestly.

Universal applicability
Since the platform should support new use cases, a universal applicable and flexible result verification scheme is required.

In general, universal applicability is satisfied because zk-SNARKs can be applied to all problems in the complexity class NP in a generic fashion [52]. Consequently, problems of the class P can also be covered, since P is a subset of NP. Therefore, it is possible to add further use cases, e.g., the graph coloring problem, the bin packing problem or the calculation of the greatest common divisor.

Our implementation uses the toolbox ZoKrates to deploy zk-SNARKs on the Ethereum blockchain. The basic elements of the ZoKrates DSL might have an impact on universal applicability, e.g., by the fact that the number of loop iterations has to be known at compile time or that only numeric types are supported (see Section 6.2). As alternative, other circuit generation languages, e.g., jsnark[3] can be used.

As we now have analyzed the requirements in the context of the task offloading and result verification, we continue with the quantitative analysis. Therefore, we present the evaluation setup in the next section.

7.2 Evaluation Setup and Dataset

As mentioned in Chapter 5, the computation offloading procedure consists of an on-chain and an off-chain part. To evaluate on-chain activities, the particular smart contracts have to be deployed on a Ethereum blockchain. The off-chain part is conducted with the help of ZoKrates for computing witnesses and generating proofs. ZoKrates can be installed natively on the operating system, but the tool is also available as Docker container[4]. We decided to use the dockerized variant and assigned 8 GB RAM and 4 CPU cores to the container.

With regard to the deployment, two options are possible. The blockchain-based computation offloading solution can be deployed on a local Ethereum blockchain or on an online Ethereum testnet, e.g., the Rinkeby[5] testnet. As previously mentioned, we mainly examine the cost development (gas usage) of the submission and verification of solutions. Additionally, the required time to perform off-chain activities, i.e., compute-witness and generate-proof, is measured. Therefore, the local deployment is sufficient to cover both dimensions of interest. If latency were to be investigated as well, the deployment to a testnet would be necessary. Since latency is not in scope of the evaluation, we use a local Ethereum blockchain provided by the Truffle Suite.

Our implementation allows to verify TSP instances up to 60 cities. First and foremost, the limiting factor is the map size within our DSL programs (the maps have to be known by our DSL programs to calculate the path length). In case a map contains more than 74 cities, we run into a stack overflow when compiling the program. We discussed the issue with the ZoKrates community, but were thereupon not able to solve it. The issue is possibly fixed in upcoming ZoKrates versions. Since we figured out in the early development phase that an instance size of 60 is enough to undercut the gas consumption of alternative result verification schemes, no further investigations with regard to the mentioned stack overflow issue have been done. However, when the issue is resolved, our solution can easily be extended to handle larger instances, e.g., of size 120 by means of the ZoKrates standard library. In case instances greater than 120 have to be supported, the hash functions of the ZoKrates standard library can be extended.

[3] https://github.com/akosba/jsnark
[4] https://hub.docker.com/r/zokrates/zokrates
[5] https://www.rinkeby.io/

The dataset for evaluation corresponds to the lodged maps, respectively distance matrices of the TSP use case. Here, the above described map and instance size limit has to be considered. That means, a map size greater than 60 but smaller than 74 is needed so that instance sizes up to 60 cities can be verified and the stack overflow issue does not occur. To meet these conditions, we decided to use a map of size 70 or 70 cities respectively. Since our DSL programs support two different maps (see Section 6.2), we have decided to integrate another map of size 30 to find out if there are differences caused by the map size with regard to the overhead of the result verification. Based on that, we have chosen two publicly available maps, one with 70 North-American[6] cities and one with 30 European[7] ones. The map with the North-American cities is illustrated in Figure 7.1. Equally, the map with the European cities is shown in Figure 7.2.

As alternative to predefined datasets, individual distance matrices can be build with the help of (non-free and free) APIs, e.g., Google's[8] distance-matrix API or the matrix API of openrouteservice[9].

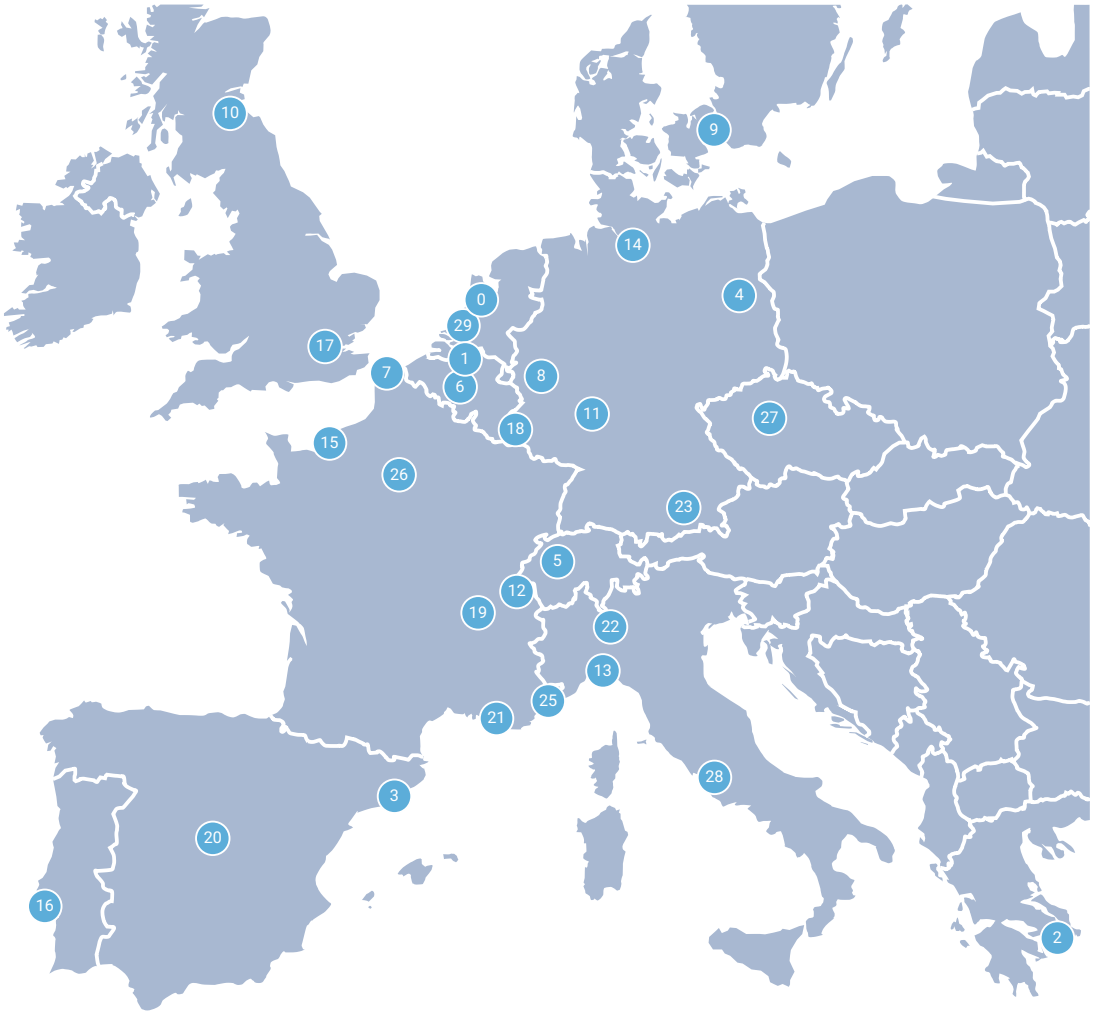
7.3 Overhead of Result Verification

First, we aim to determine the overhead caused by result verification with zk-SNARKs, as described in the introduction of this chapter. Since the verification is part (or a sub-function) of the submission of a solution, we have to examine the submission step within the auction smart contract. More precisely, two variants of submitting a solution are considered to conduct a comparison. In the first variant, a solution can be submitted and verified with zk-SNARKs. In the second variant, a solution can be submitted without performing any verification. Differences of both versions with respect to the overhead caused by result verification are determined. Additionally, possible discrepancies between instances of equal size addressing different maps, i.e., map 30 and map 70, are investigated. Next, the overhead with regard to cost, then the additional time needed to perform the verification with zk-SNARKs is evaluated.

Additional cost
To get a first insight into extra cost, i.e., gas consumption, we compare the functions *submitSolutionVerifiable()* and *submitSolution()* within the auction smart contract. In general, *submitSolutionVerifiable()* verifies the solution with the help of zk-SNARKs in a separate smart contract and stores it afterwards if the verification was successful. The function *submitSolution()* merely stores the submitted solution on-chain without performing any check. Both functions finally store the solution but *submitSolutionVerifiable()* performs also the result verification (abstracted in Figure 5.4). Therefore, we expect a higher effort caused by the verification within *submitSolutionVerifiable()*. To get a

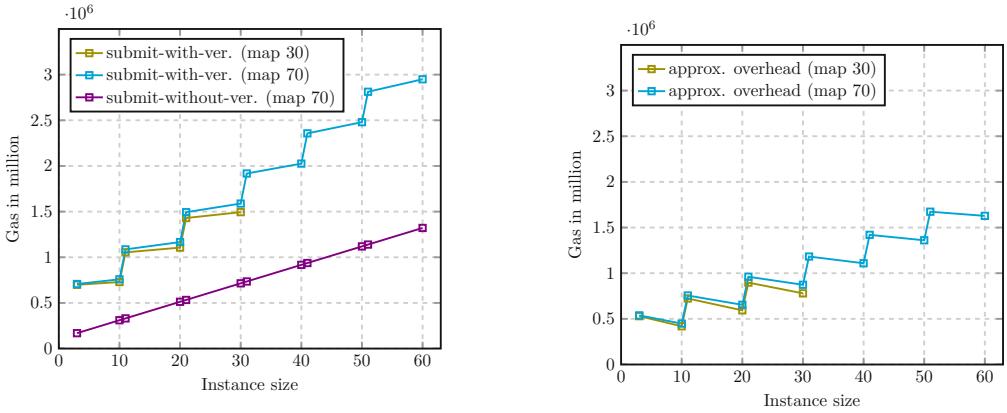
[6] https://people.sc.fsu.edu/ jburkardt/datasets/cities/cities.html
[7] http://www.mapcrow.info/european_travel_distance.html
[8] https://developers.google.com/maps/documentation/distance-matrix
[9] https://openrouteservice.org/#/dev/api-docs/v2/matrix

Figure 7.1: Map with North-American cities



0 Amsterdam, NL	6 Brussels, BE	12 Geneva, CH	18 Luxembourg, LU	24 Naples, IT
1 Antwerp, BE	7 Calais, FR	13 Genoa, IT	19 Lyon, FR	25 Nice, FR
2 Athens, GR	8 Cologne, DE	14 Hamburg, DE	20 Madrid, ES	26 Paris, FR
3 Barcelona, ES	9 Copenhagen, DK	15 Le Havre, FR	21 Marseille, FR	27 Prague, CZ
4 Berlin, DE	10 Edinburgh, GB	16 Lisbon, PT	22 Milan, IT	28 Rome, IT
5 Bern, CH	11 Frankfurt, DE	17 London, GB	23 Munich, DE	29 Rotterdam, NL

Figure 7.2: Map with European cities



(a) Gas consumption of submitting a solution with and without verification

(b) Approximated overhead

Figure 7.3: Comparison of the cost development of the submitting solutions and its overhead

complete picture of the cost overhead, instance sizes between 3 - 30 (map 30) and 3 - 60 (map 70) are used.

Figure 7.3 shows two plots. The left one, Figure 7.3a, depicts the gas consumption of *submitSolutionVerifiable()* and *submitSolution()*. The second plot on the right side, Figure 7.3b, displays the approximated overhead of result verification with zk-SNARKs. Next, both are analyzed in detail.

Figure 7.3a contains three lines. The olive and cyan colored line present the gas consumption for submitting and verifying a solution that is related to map 30 and map 70, respectively. Since the gas usage for submitting a solution without verification does not depend on the underlying map, no separation by map is done. At a glance, it can be seen that submitting a solution including result verification is more expensive (independently of the used map) than conducting no verification. Additionally, it also becomes clear that the gas demand of all three variants increases with the instance size. While submitting a solution of instance size 3 for map 70 needs 168K (no verification) and 729K (with verification) gas, submitting a solution of instance size 60 for map 70 requires 1.3M, respectively 2.9M gas.

It is also noticeable that, the cost level of *submit-without-ver. (map 70)* in Figure 7.3a rises continuously, while the levels of *submit-with-ver. (map 30)* and *submit-with-ver. (map 70)* increase step-wise. This is caused by the partitioned verification, due to the lack of dynamic fields within the DSL of ZoKrates. For solutions of size 3 - 60, separate DSL programs with a varying number of inputs (starting with an instance size of 10, and increased by steps of 10) are provided. For example, if a TSP solution for 11 cities is computed, the resulting path has to be padded, according to the expected number of inputs of the DSL program. Therefore, in our example, the path within the solution needs nine placeholders and has consequently a length of 20. Due to the fact that such a

padding is not necessary when no verification is performed, the gas consumption merely rises continuously.

Finally, Figure 7.3a clarifies, that the submission of solutions based on the smaller map (with 30 cities) is slightly cheaper than solutions based on the larger map (with 70 cities). On average, 4% (standard deviation: 1.6%) less gas is needed when a solution is based on map 30. Within the auction smart contract, a mapping array from city names to its Ids is lodged to conduct a conversation for each supported map. The aforementioned difference by map can be explained by a varying iteration effort, when the conversation from city names to its Ids is executed.

The actual overhead can be approximated by subtracting the gas cost of *submitSolution()* from *submitSolutionVerifiable()*. The result is illustrated in Figure 7.3b. As determined above, the overhead increases by the size of an instance. The zigzag of both lines in Figure 7.3b is again reasoned by the verification logic of our DSL programs, which does also influence the cost ratio between a submission with and without verification. For example, submitting and verifying a solution with three cities of map 70 is 4.1 times costlier than performing no verification. When no placeholders are necessary, i.e., a solution with 10 cities of map 70 the cost ratio is 2.3. Simply put, the cost ratio is better, if less padding is necessary when a solution is submitted and verified.

Additional time

Another difference between both versions is that a witness and a proof must be generated off-chain in advance to submit a verifiable solution (see Section 5.2.3). In contrast to the procedure without verification, a solution must be only computed (off-chain) and is then ready for submission. Figure 7.4 depicts the time needed to execute the compute-witness and generate-proof step for TSP instances of size 3 - 60. The values on the x-axis can be interpreted as follows: 3/m30 means that the instance is of size three and belongs to map 30. As can be seen, the run-time overall increases by the size of instances. The step generate-proof takes on average 3.2 (standard deviation: 0.28) times longer than the compute-witness step. Since the run-time hardly varies between, e.g., 3/m30 to 10/m70, it seems that the size of the map and instance does not influence the overall run-time. This is again caused by partitioning logic within the result verification plus the map organization within the DSL programs. Our implementation allows two maps per DSL program. The arrays for the maps within these DSL programs, respectively the array sizes are dominated by the larger one due to restrictions of ZoKrates.

Summing up, submitting and verifying a solution is more expensive than a simple submission without verification. However, in the latter case it is also not possible to prove whether a solution was computed correctly. With regard to the relation between the gas consumption and the instance size, we observed that especially larger instances are more expensive to submit and to verify, compared to a submitting them without verification. The average cost ratio between a submission with and without verification is 2.6 (standard deviation: 0.63), meaning that conducting a verification is 2.6 times costlier than a simple submission without any check. To gain a better insight into the composition of costs, a further analysis is necessary. One already known cost driver of

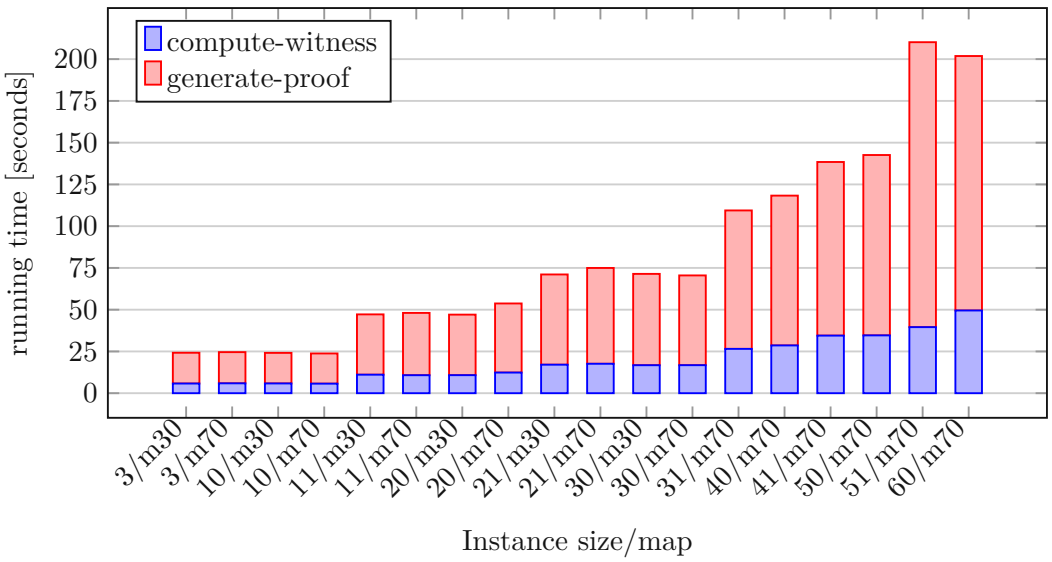


Figure 7.4: Time in seconds to conduct compute-witness and generate-proof with ZoKrates

result verification is the number of public inputs of our DSL programs, because these public inputs have to be supplied as well, when the verification is executed on-chain (see Section 6.3). Therefore, a reduced number of public inputs decreases also the gas consumption. An optimized variant of result verification with zk-SNARKs is evaluated in Section 7.4.

7.4 Variants of Result Verification and Cost Efficiency

In this section, the submission and verification of solutions is further analyzed, so that cost factors can be identified. In addition, an optimized version - result verification at constant cost - is evaluated and compared with on-chain verification. By on-chain verification we mean a separate smart contract that ensures whether all requirements for a valid TSP solution are fulfilled (without cryptographic proofs). Since the difference of cost between the submission of a solution belonging to map 30 or map 70 is negligible, we mainly consider instances of map 70 in this section. In total, three versions of result verification - basic result verification, result verification at constant cost and on-chain verification are evaluated. Finally, adoptions to the auction smart contract with regard to cost development are made and analyzed.

7.4.1 Basic Variant of Result Verification

A basic comparison of our auction-based mechanism with and without result verification has been conducted in Section 7.3. We have found that the total effort for the submission and verification is rather expensive compared to a simple submission without verification. Therefore, we aim to learn more about the composition of costs, within the function

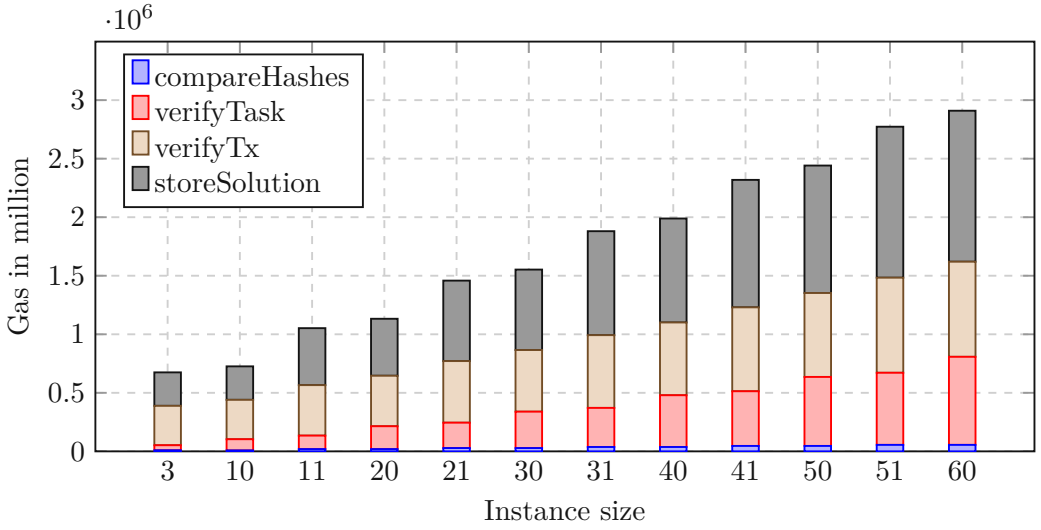


Figure 7.5: Cost overview of submitting a solution plus result verification

submitSolutionVerifiable(), to identify cost factors and to derive possible optimizations. Here, it can be helpful, to keep the input parameters of the submission function in mind. Besides the cryptographic proof, the computed path, its hash, the path length and the map number have to be supplied. The function *submitSolutionVerifiable()* can be divided into sub-functions (see Section 6.1). First, the hash of the path is computed and compared with the input parameter containing the hashed path of the task processor. Next, preparations for the verification and finally the result verification (with the help of a smart contract generated by ZoKrates) are performed. If the verification succeeded, the supplied solution is stored on-chain.

Figure 7.5 depicts the gas cost considering the aforementioned sub-functions of *submitSolutionVerifiable()*. Based on Figure 7.5, it becomes clear that the verification and the storage operation account for the highest share of the costs, independently of the instance size. At this point it is worth mentioning that the sub-function *verifyTask()* preprocesses the inputs for the function *verifyTx()*, which performs the actual verification with zk-SNARKs. Therefore, the cost for the complete verification step can be determined by forming the sum of *verifyTask()* and *verifyTx()*. Both sub-functions require more gas the larger the instance is. This has two reasons. First, the larger the instance, the more data or inputs are necessary. Second, a higher amount of inputs also increase the processing effort on-chain. The same can be applied with regard to storing the solution. The more data, the costlier the storage process.

7.4.2 Result Verification at Constant Cost

Since we now have a breakdown of costs, targeted optimizations can be carried out. In Section 5.2, we have decided to write the solution of a task as plaintext on the blockchain

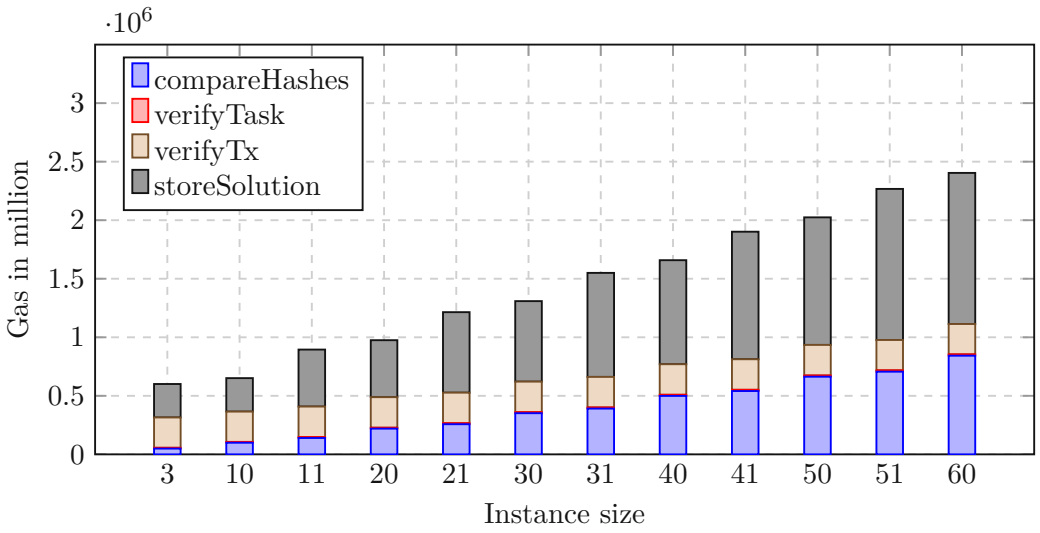


Figure 7.6: Cost overview of submitting a solution plus result verification at constant cost

because of simplicity. To keep up with this, the storage process and its location is not changed. Instead, optimizations are done at the result verification (see Section 6.3). We call the improved variant result verification at constant cost.

Eberhardt and Tai point out in [21] that the gas cost rise linearly with the number of inputs provided for the verification function *verifyTx()* in the verification contract. Therefore, the goal of this optimization is to reduce the number of public input parameters of our DSL programs, because these inputs have to be supplied to *verifyTx()* as well.

The main difference between the basic and optimized version is that the input parameter for cities (which has an equal length as the instance size) is changed to private and a new public input (of length two) expecting the hash value of the cities is added. Consequently, the overall number of input parameters for *verifyTx()* remain the same, regardless of the instance size. Further implementation details can be found in Section 6.3.

Figure 7.6 shows the gas consumption when TSP solutions of size 3 - 60 are submitted and verified in the context of the aforementioned optimization. As expected, the sub-function *verifyTx()* requires the same amount of gas for each instance. The sum of *verifyTx()* and *verifyTask()* gives the total cost of the verification. The sub-function *compareHashes()* needs more gas, compared to the unmodified variant of result verification (see Figure 7.5). The reason for this is that, the hash function must now be computed two times (cities and path) and additional preparatory steps are necessary before the cities can be passed to the hash function. More precisely, the city names must be converted to its city indices (Ids). With regard to the auction smart contract, less changes are necessary to integrate result verification at constant cost.

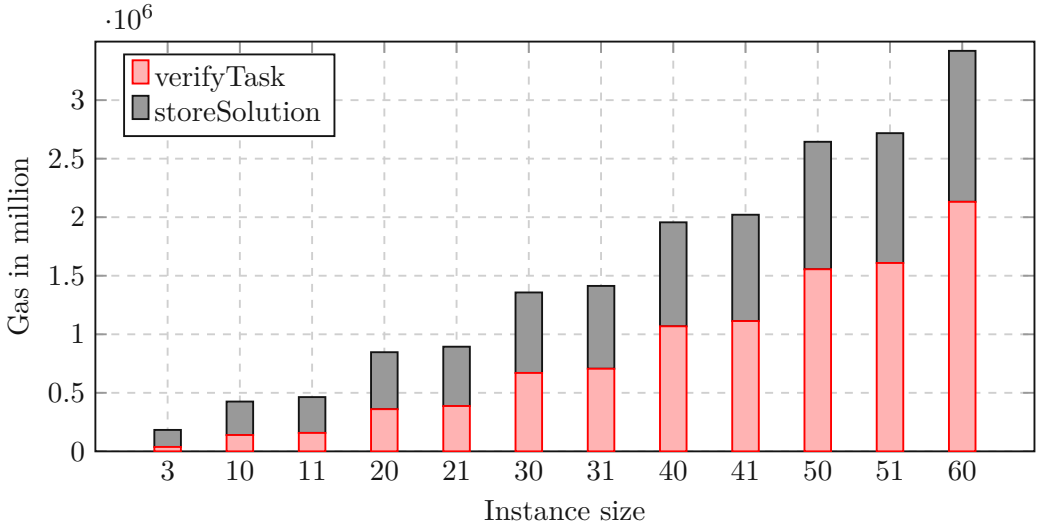


Figure 7.7: Cost overview for submitting a solution plus on-chain verification

7.4.3 On-chain Result Verification

So far, we have examined the result verification variants based on zk-SNARKs. Apart from that, it is also possible to verify results without the help of cryptographic proofs. In the following, this version is called on-chain result verification. Here, we aim to inspect on-chain result verification and further compare it with the zk-SNARKs-based verification functions, whereby the actual comparison is carried out in the next section.

In the context of our scenario, on-chain verification means that a solution for a TSP problem is verified by a separate smart contract deployed on a blockchain. Instead of calling the verification function *verifyTx()* in the auction smart contract when a solution is submitted, a separate function is triggered which encodes the requirements of a TSP solution as outlined in the introduction of Chapter 6. To verify the TSP on-chain the supported distance matrices have to be available on-chain as well.

Figure 7.7 shows the cost distribution of solution submissions, in the setting of on-chain verification. Concretely, Figure 7.7 visualizes the cost for verification and for writing the solution on the blockchain. Again, the same instance sizes from 3 - 60 are used. The way how the solution is stored has not changed. Therefore the costs are nearly the same as for the variants described above (small deviations are caused by the padding). On the contrary, the overall cost (sum of both sub-functions) and the effort for the on-chain verification differs. In addition, it is visible that smaller instances have a moderate gas consumption, while large instances require a large amount of gas. A detailed comparison of the introduced variants of result verification is carried out in the next section.

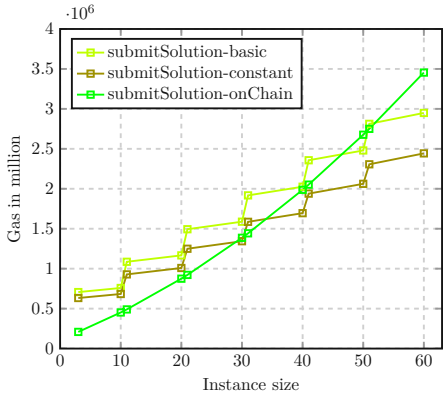
7.4.4 Comparison of Different Result Verification Variants

Until now, we have separately analyzed the gas consumption of three different result verification variants with the main focus on the cost composition of the submission and verification function. The first two verification variants use zk-SNARKs and the third one the on-chain variant (without cryptographic proofs), to verify TSP solutions. Now all three variants are compared directly by its cost caused by calling *submitSolutionVerifiable()*. Similarly, the comparison for the sub-function *verifyTask()* is given.

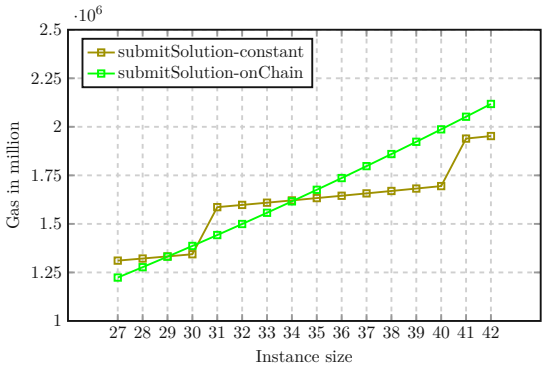
Figure 7.8a compares the total cost of executing *submitSolutionVerifiable()* for each verification variant by TSP instances of map 70 and of size 3 - 60. At a glance, it can be seen that the result verification at constant cost (see line "submit-constant") is continuously cheaper than the non-optimized version of result verification (see line "submit-basic"). Verifying TSP solutions on-chain is cheaper than the other two variants for small instances, but more expensive for large ones. Up to an instance size of 29, the on-chain verification is cheaper than the other two variants. For instances of size 30, the constant version should be preferred, whereby on-chain verification is at a lower price for an instance size of 31. Finally, the cost of on-chain verification exceeds the cost of the constant variant at instance size 40 and the basic variant when a solution with a path length of 60 is submitted. Overall, we can clearly see, which version is cheaper for instances of size up to 29 and from 40 ascending. Around the instance size 30, three intersections with regard to the on-chain and constant variant are visible. To determine the exact break-even point(s) between 29 and 40, a more detailed analysis is needed.

Since the constant version is continuously less expensive than the basic version of result verification with zk-SNARKs, we compare hereafter only the on-chain and the constant result verification in detail. In Figure 7.8b the gas consumption of both previously mentioned versions, respectively their submission functions, is shown. Here, instance sizes from 27 - 42 are used, allowing us to examine the above mentioned intersections in detail. It is noticeable, that instances of size smaller or equal than 29 should be verified on-chain due to lower cost. The verification of instances of size 30 are cheaper when zk-SNARKs are used. Further, on-chain verification is more cost-efficient for instances between the size 31 and 34. Ascending from instance size 35, again the verification with zk-SNARKs is more favourable (see Table 7.1).

In Figure 7.9 the gas consumption of the verification functions is illustrated. At a first glance, it can be seen that the ascent of line "verifyTask-onChain" is steeper compared to the ascent of "verifyTask-basic". Possible reasons are the various number of inputs (per instance size) and the effort for conducting the actual verification. The different number of inputs of "verifyTaks-onChain" and "verifyTask-basic" is negligible, as it differs only by the possible padding (required to tackle the lack of dynamic fields in the ZoKrates DSL). To verify a TSP solution, the length of the supplied path has to be calculated and checked against the supplied path length. This can be encoded by a loop, which sums up the distances between the cities of the path. The difference between both versions is that, this loop is executed off-chain in case of "verifyTask-basic" and on-chain in case



(a) Gas required to submit a solution for each verification variant



(b) Detailed cost comparison between the submission with on-chain and constant result verification

Figure 7.8: Cost development of the solution submission and of the verification function

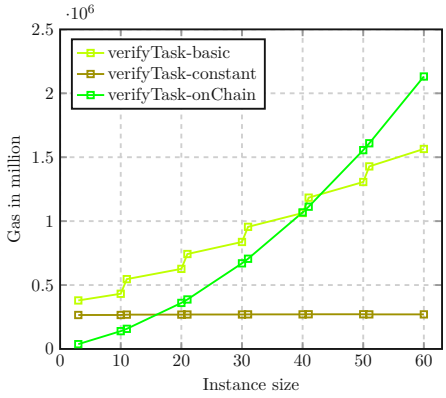


Figure 7.9: Comparison of on-chain and verification with zk-SNARKs

of "verifyTask-onChain". We conclude that the execution of the loop on the Ethereum blockchain causes this high cost increase.

With regard to the break-even point, it can be observed that the on-chain verification is cheaper until (approximately) an instance size of 15. If more than (approximately) 15 cities are of interest, leveraging zk-SNARKs at constant cost is more cost-efficient. The basic version of verification with zk-SNARKs is cheaper than on-chain verification if more than (approximately) 45 cities have to be verified. The main statement reasoned by the evaluation within Figure 7.9 is that, the actual verification at constant cost can be very cheap and independent of the instance size. The prerequisite to keep the cost so low is, that all inputs for the verification step are already in an appropriate form. This can be achieved by preprocessing or if the input structure of, e.g., *submitSolutionVerifiable()* corresponds exactly to that of *verifyTx()*. We have decided to preprocess the inputs, because of assumed lower costs. In the following section, adoptions within the auction

smart contract are presented to reduce the need of preprocessing and to save cost.

We assume that the cost discrepancy between on-chain verification and verification with zk-SNARKS becomes even clearer when instance sizes above 60 are used. However, our current implementation, more precisely our DSL programs, limits the the instance size to 60.

7.4.5 Optimized Auction Smart Contract

When an auction is created, the task issuer can enter the city names of interest, which is rather comfortable than cost-conscious. The fact is that costs can be saved if the indices of the cities (and the map no.) are given, instead of the city names at task creation, because the conversation from city names to its indices can be omitted.

The above described versions execute this conversation within the function *submitSolutionVerifiable()*, to prepare the input parameters for *verifyTx()*. Among others, the city indices of the requested cities are necessary (see Section 6.2.1). Since the city indices are not supplied as parameter of *submitSolutionVerifiable()* they have to be processed with the help of the city names and the map number. By removing this step, the gas consumption can be reduced by further 13% on average (standard deviation: 8.9%). It is worth mentioning, that larger instances benefit more, e.g., at the instance size of 60, 30% gas is saved. This is caused by the fact that larger instances entail a higher conversion effort. This adoption can be applied to all three versions (basic, constant, on-chain) and decreases the preprocessing effort to conduct result verification. Consequently, the cost difference between submitting a solution with and without result verification is also reduced.

Figure 7.10 shows the cost development when the previously described feature is removed

Besides this adoption, more observations should be made in future, maybe with regard to the input structure of the function *submitSolutionVerifiable()* for the purpose of further reducing the total cost.

7.5 Discussion

The result verification within the auction-based environment has been analyzed in detail. The overhead between the submission with and without verification has been determined. Furthermore, optimizations with regard to the result verification with zk-SNARKs have been done. Two different variants of zk-SNARKs based result verification schemes, have been compared to on-chain verification.

The cost gap between verification and no verification in the auction-based environment depends on the utilization of the partitions within the result verification. On average, result verification is costlier by a factor of 2.6 (standard deviation: 0.63).

The adoption of the basic variant of result verification, by reducing the input parameters of *verifyTx()* to a constant amount also reduces the overall cost of submitting a verifiable

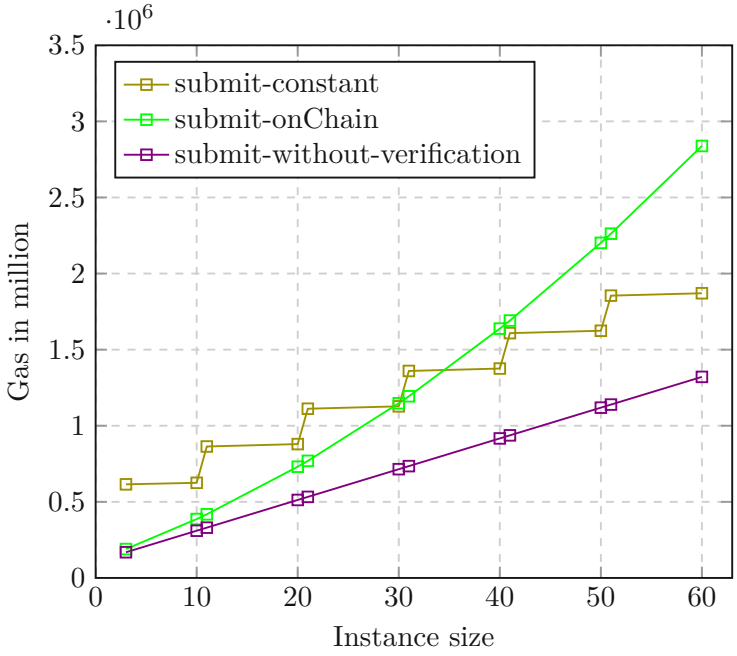


Figure 7.10: Comparison of different submission variants in the setting of lightweight auction smart contract

solution to the auction smart contract. On average, 15% of the costs could be saved (standard deviation: 2.7%). The difference between no verification and verification at constant cost, decreases as well. On average, submitting and verifying a solution is only 2.3 times more expensive (standard deviation: 0.59).

The comparison of the gas consumption of the two zk-SNARKs based variants and the on-chain verification, shows which variant is better suited for which instance size. Since result verification at constant cost is consistently cheaper than the basic variant, we consider in the following only the constant version. Based on the cost development over different instance sizes, we conclude that on-chain verification should be preferred if the offloaded task contains less or equal than 29 cities. If 30 - 40 cities are requested, it depends on the utilization of the partition implemented in the result verification, which variant is more economical. When the instance size is exactly 30, then the constant version of result verification should be used. However, when instances of size 31 - 34 offloaded, then on-chain verification is more beneficial. As soon as more or equal than 35 cities have to be computed and verified, the zk-SNARKs based variant at constant cost is more cost-efficient. Table 7.1 summarizes which verification variant should be preferred for each instance size with regard to low-cost.

Instance Size			
[3; 29]	[30]	[31; 34]	[35; 60]
on-chain	zk-SNARKs	on-chain	zk-SNARKs

Table 7.1: Recommended result verification variant with regard to cost

With regard to the cost distribution of the submission function of all three variants, it turned out that storing the solution of a task on-chain is a massive cost factor. However, the storage procedure and possible locations are not in the scope of this theses. Since costs of the current (on-chain) storage procedure are very high, it may be useful to further investigate alternative storage locations. Here it is important that the correctness and the integrity of the solution is still preserved. Besides the storage part, it can be observed in Figure 7.6, that the effort for computing and comparing hashes of the constant version increases with larger instance sizes. We figured out that not only the increased effort of computing hashes is responsible but also its preprocessing caused by the conversion of city names to its indices (see Section 7.4.2).

CHAPTER 8

Conclusion

Our proposed solution uses blockchain technology to realize computation offloading. By means of a blockchain, the assignment of resources and the processing of payment can be carried out in a decentralized manner. Furthermore, task issuers and processors can freely join and leave the computation platform without any barriers. Offloaded tasks are computed off the chain, because blockchains are not built for executing resource-intensive tasks. By using off-chain computations, a task processor must be trusted, due to no information about the validity of the computation can be provided. To remove the need of trust between task issuers and processors with regard to computed solutions, a result verification scheme is integrated which allows to prove whether a computation of a defined problem was conducted properly. We use zk-SNARKs to verify submitted results.

We aimed to determine the overhead through result verification with zk-SNARKs and its cost-efficiency. More details and results are outlined in the next paragraph - the discussion of the research questions.

Discussion of research questions
Three research questions in the field of blockchain-based computation offloading and result verification schemes were introduced in Chapter 1. Here, concrete answers to these questions are summarized.

RQ.1 Which result verification schemes are possible?
As mentioned in the introduction of this chapter, blockchain technology is not built for executing arbitrary complex computations. Therefore, when a blockchain is involved in a computation offloading setting, such as the Ethereum blockchain, off-chain computation should be preferred to preclude the possibility of exceeding the gas limit per block or cause very high execution cost.

When using off-chain computation, it is questionable, if the result of such a computation is trustworthy and correct. To gain information with regard to the

trustworthiness of a result, we have compared four result verification schemes in Chapter 4. According to our literature research, verifiable-, enclave-based, secure multiparty and incentive-driven off-chain computation are potential schemes for result verification.

With regard to the selection of a result verification scheme, we take practicability, expected cost, security and the limitations of related projects presented in Chapter 3 into account. We decided to use the verifiable off-chain computation scheme for our solution approach, because of its universal applicability. In the field of verifiable off-chaining, three implementations zk-SNARKs, zk-STARKs and Bulletproofs exist. We use zk-SNARKs to employ result verification in our system, due to the complexity of the off-chain computation does not influence the complexity of result verification.

RQ.2 How can blockchain-based computation offloading be integrated with the selected result verification scheme and an auction-based mechanism?

In Chapter 5 the architecture of the proposed solution approach is described. The offloading procedure and the auction-based mechanism are both combined within one smart contract, the auction smart contract, which is deployed on the Ethereum blockchain. The auction smart contract serves as meeting point for those, who want to offload a task (task issuers) and task processors, which offer their computational resources against a fee. The auction smart contract allows to create an auction (task), submit bids (solutions), end an auction and to retrieve the solution. All submitted solutions are ranked by a quality criteria. A task issuer is able to retrieve the solution with the best quality criteria.

The second part, the result verification is achieved by an own smart contract generated by ZoKrates, a tool which allows to employ zk-SNARKs on the Ethereum blockhain. The verification smart contract is integrated in the auction smart contract. In other words, the verification smart contract contains a verification function, which is called when a solution is submitted. Only if the verification succeeds, the submitted solution is accepted.

Solutions for tasks are stored on-chain and as plaintext. This decision has pros and cons. Storing solutions on-chain is easy to achieve, but could be expensive, the larger a solution is. Storing solutions as plaintext and on-chain is equally straight forward, but privacy can suffer due to the public nature of blockchains. Furthermore, doors for malicious activities are opened if data can be accessed publicly. What we understand by a malicious activity is for example when a task issuer reads a submitted result, but does pay the task processor for the computation. We set economic motives, i.e., a stake to prevent such a scenario and discussed alternative storage and solution delivery concepts in Chapter 5.

RQ.3 What are the benefits of the proposed solution and when does it make sense to prefer a verification scheme that proves whether a defined computation was executed properly over alternative variants?

The benefits of the proposed solution are two fold. First, by means of zk-SNARKs a task processor can convince a task issuer whether the computation of an offloaded task was executed properly. Zk-SNARKs fulfill the two requirements completeness and soundness (see Section 2.3), meaning that correct computations are accepted with high probability, while incorrect ones are accepted with a negligibly small probability. To save cost, computations are conducted off-chain while computation results are verified on-chain. However, the result verification scheme includes no statement about the quality of the result. This can be covered by the auction mechanism. Second, the auction-based mechanism allows that several task processors can contribute to a single task. Thereby, competition between task processors can be enabled. Furthermore, the possible greater variety of solutions can enhance the probability of an overall better solution. This is the reason why the system is truly suitable for offloading optimization problems.

With regard to cost, a trade-off between verifiable off-chain computation with zk-SNARKs and alternative variants exist. In Chapter 7, the zk-SNARKs-based variant has been compared to on-chain verification in the context of offloading TSP instances of different sizes. By on-chain verification we understand the verification by a smart contract only without the help of a cryptographic proof. We figured out, that solutions of TSP instances with less or equal than 29 cities should be submitted and verified by means of the on-chain variant, due to less cost. For instances with 30 or more or equal than 35 cities, the option with zk-SNARKs at constant is more cost-efficient. For instances of size 31 - 34 on-chain verification is cheaper (see Table 7.1).

Future work

In this section, possible extensions and improvements of our blockchain-based offloading approach with result verification are outlined.

Storage location of solutions

Storing data on a blockchain can be expensive. The more data is written on-chain, the more gas is needed in case of the Ethereum blockchain. In the evaluation of the proposed solution, it turned out that a major share of the total cost derives from storing submitted solutions on-chain. We assume that there is potential to further save cost through optimized solution delivery and storage processes. Thereby, the layout and composition of solutions can be reviewed. It is also questionable if its necessary to store entire solutions on a blockchain or whether a hash or the quality of a solution is also sufficient. In addition, alternative storage locations such as IPFS can be further investigated with the goal of reducing the cost of the offloading procedure. Apart from lowering cost, privacy can be considered when alternative storage locations and procedures are examined, so that only the task issuer can read a corresponding solution.

Cost-aware result verification
A trade-off between the result verification with zk-SNARKs and, e.g., on-chain verification exists when the TSP problem is offloaded. In the future, it can be beneficial if the result verification scheme can be selected in advance, e.g., by the task issuer or processor. Another possibility is that the auction smart contract automatically derives the most cost-efficient verification variant depending on, e.g., the number of requested cities at auction creation. However, this trade-off can differ between use-cases and has to be evaluated first. Nevertheless, such a feature can help to save cost.

This thesis presented an approach to realize computation offloading and result verification by means of blockchain technology and zk-SNARKs. Based on the evaluation, which outlines the overhead, cost drivers and the cost-efficiency of result verification with zk-SNARKS, we have shown that the result verification can be carried out at constant cost. Furthermore, we derived options to improve our blockchain-based computation offloading approach in future.

List of Figures

List of Tables

Listings

Acronyms

D2D device-to-device. 18, 19

DRL deep reinforcement learning. 18

DSL domain-specific-language. 23, 26–29, 42, 50, 51, 53–55, 57, 65, 69–71, 73

EVM Ethereum Virtual Machine. 8

GNT Golem Network Token. 14

IaaS Infrastructure-as-a-Service. 14, 15, 17

IoT Internet of Things. 1, 18

IPFS InterPlanetary File System. 15, 39, 83

P2P peer-to-peer. 5, 8, 14, 17

PaaS Platform-as-a-Service. 14, 15, 17

PoW proof-of-work. 8, 18

QAP Quadratic Arithmetic Program. 27

R1CS Rank-1-Constraint-System. 27

RLC Run on Lots of Computers. 15, 16

SaaS Software-as-a-Service. 14, 15

SGX Software Guard Extension. 16, 23, 24

SMPC Secure Multiparty Computation. 24, 25

TEE Trusted Execution Environments. 23, 24

TSP Traveling Sales Person. 2, 45, 46, 50, 51, 53, 58–61, 66, 69, 71, 74, 75

zk-SNARKs zero-knowledge non-interactive succinct argument of knowledge. 11, 12, 23–26, 28–30, 33, 41, 46, 49, 50, 61, 66, 69, 72, 74–77, 81

zk-STARKs Zero-Knowledge Scalable Transparent ARguments of Knowledge. 23, 25

ZKPs Zero-knowledge proofs. 3, 5, 9–12, 26

Bibliography

[1] Luca Aceto, Andrea Morichetta, and Francesco Tiezzi. Decision support for mobile cloud computing applications via model checking. In *3rd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering, MobileCloud 2015*, pages 199–204, 2015.

[2] Alexandre Adamski. Overview of intel sgx - part 1, sgx internals. `https://blog.quarkslab.com/overview-of-intel-sgx-part-1-sgx-internals.html`. Online, last visited at 2020-08-14.

[3] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolic, Sharon Weed Cocco, and Jason Yellick. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018*, pages 30:1–30:15, 2018.

[4] Carsten Baum, Ivan Damgård, and Claudio Orlandi. Publicly auditable secure multiparty computation. In *Security and Cryptography for Networks - 9th International Conference, SCN 2014*, pages 175–196, 2014.

[5] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. *IACR Cryptology ePrint Archive*, 2018:46, 2018.

[6] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. *IACR Cryptology ePrint Archive*, 2014:349, 2014.

[7] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In *Proceedings of the 23rd USENIX Security Symposium, 2014*, pages 781–796, 2014.

[8] Juan Benet. IPFS - content addressed, versioned, P2P file system. *CoRR*, abs/1407.3561, 2014.

[9] Manuel Blum, Paul Feldman, and Silvio Micali. Non-interactive zero-knowledge and its applications (extended abstract). In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, 1988*, pages 103–112, 1988.

[10] Manuel Blum, Alfredo De Santis, Silvio Micali, and Giuseppe Persiano. Noninteractive zero-knowledge. *SIAM Journal on Computing*, 20(6):1084–1118, 1991.

[11] Sean Bowe, Ariel Gabizon, and Matthew D. Green. A multi-party protocol for constructing the public parameters of the pinocchio zk-snark. In *Financial Cryptography and Data Security - FC 2018 International Workshops, BITCOIN, VOTING, and WTSC, Revised Selected Papers*, pages 64–77, 2018.

[12] Sean Bowe, Ariel Gabizon, and Ian Miers. Scalable multi-party computation for zk-snark parameters in the random beacon model. *IACR Cryptology ePrint Archive*, 2017:1050, 2017.

[13] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Gregory Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy, 2018*, pages 315–334, 2018.

[14] Vitalik Buterin. Ethereum: A next-generation smart contract and decentralized application platform. `https://github.com/ethereum/wiki/wiki/White-Paper`. Online, last visited at 2020-04-05.

[15] Dimitris Chatzopoulos, Mahdieh Ahmadi, Sokol Kosta, and Pan Hui. Flopcoin: A cryptocurrency for computation offloading. *IEEE Trans. Mob. Comput.*, 17(5):1062–1075, 2018.

[16] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah M. Johnson, Ari Juels, Andrew Miller, and Dawn Song. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In *IEEE European Symposium on Security and Privacy, EuroS&P 2019*, pages 185–200, 2019.

[17] Jens Clausen. Branch and bound algorithms - principles and examples. `https://janders.eecg.utoronto.ca/1387/readings/b_and_b.pdf`. Online, last visited at 2020-09-11.

[18] Jacob Eberhardt and Jonathan Heiss. Off-chaining models and approaches to off-chain computations. In *Proceedings of the 2nd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers, SERIAL@Middleware 2018*, pages 7–12, 2018.

[19] Jacob Eberhardt and Stefan Tai. Zokrates documenation. `https://zokrates.github.io/`. Online, last visited at 2019-12-15.

[20] Jacob Eberhardt and Stefan Tai. On or off the blockchain? insights on off-chaining computation and data. In *Service-Oriented and Cloud Computing - 6th IFIP WG 2.14 European Conference, ESOCC 2017, Proceedings*, pages 3–15, 2017.

[21] Jacob Eberhardt and Stefan Tai. Zokrates - scalable privacy-preserving off-chain computations. In *IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), iThings/GreenCom/CPSCom/SmartData 2018*, pages 1084–1091, 2018.

[22] Etherscan. Ethereum average gas limit chart. `https://etherscan.io/chart/gaslimit`. Online, last visited at 2020-04-05.

[23] Gilles Fedak, Wassim Bendella, and Eduardo Alves. Blockchain-based decentralized cloud computing. `https://iex.ec/wp-content/uploads/pdf/iExec-WPv3.0-English.pdf`. Online, last visited at 2019-12-15.

[24] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct nizks without pcps. *IACR Cryptology ePrint Archive*, 2012:215, 2012.

[25] Golem Factory GmbH. The golem project. `https://golem.network/crowdfunding/Golemwhitepaper.pdf`. Online, last visited at 2019-12-15.

[26] Oded Goldreich. Zero-knowledge twenty years after its invention. *IACR Cryptology ePrint Archive*, 2002:186, 2002.

[27] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987*, pages 218–229, 1987.

[28] Oded Goldreich, Silvio Micali, and Avi Wigderson. Proofs that yield nothing but their validity for all languages in NP have zero-knowledge proof systems. *J. ACM*, 38(3):691–729, 1991.

[29] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18(1):186–208, 1989.

[30] Andy Greenberg. Hackers can mess with voltages to steal intel chips' secrets. `https://www.wired.com/story/plundervolt-intel-chips-sgx-hack`. Online, last visited at 2020-01-18.

[31] Jens Groth. On the size of pairing-based non-interactive arguments. In *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, 2016, Proceedings, Part II*, pages 305–326, 2016.

[32] Synergy Research Group. Amazon, microsoft, google and alibaba strengthen their grip on the public cloud market. `https://www.srgresearch.com/articles/amazon-microsoft-google-and-alibaba-strengthen-their-grip-public-cloud-market`. Online, last visited at 2020-04-09.

[33] Synergy Research Group. Big four still dominate in q1 as cloud market growth exceeds 50https://www.srgresearch.com/articles/gang-four-still-racing-away-cloud-market. Online, last visited at 2020-04-09.

[34] Vinay Gupta. A brief history of blockchain. https://hbr.org/2017/02/a-brief-history-of-blockchain. Online, last visited at 2020-04-15.

[35] Galal Hassan and Khalid Elgazzar. The case of face recognition on mobile devices. In *IEEE Wireless Communications and Networking Conference, WCNC 2016*, pages 1–6, 2016.

[36] Robby Houben and Alexander Snyers. Cryptocurrencies and blockchain. https://www.europarl.europa.eu/cmsdata/150761/TAX3%20Study%20on%20cryptocurrencies%20and%20blockchain.pdf. Online, last visited at 2020-09-25.

[37] Intel. Intel sgx - take control of protecting your data. https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions.html. Online, last visited at 2020-08-14.

[38] Sanjay Jain, Prateek Saxena, Frank Stephan, and Jason Teutsch. How to verify computation with a rational network. *CoRR*, abs/1606.05917, 2016.

[39] Bernadette Johnson. How amazon echo works. https://electronics.howstuffworks.com/gadgets/high-tech-gadgets/amazon-echo.htm. Online, last visited at 2019-12-15.

[40] Ahmed E. Kosba, Charalampos Papamanthou, and Elaine Shi. xjsnark: A framework for efficient verifiable computation. In *2018 IEEE Symposium on Security and Privacy, 2018*, pages 944–961, 2018.

[41] Mengting Liu, F. Richard Yu, Yinglei Teng, Victor C. M. Leung, and Mei Song. Distributed resource allocation in blockchain-based video streaming systems with mobile edge computing. *IEEE Trans. Wirel. Commun.*, 18(1):695–708, 2019.

[42] Sonm Pte. Ltd. Sonm - supercomputer organized by network mining. https://whitepaper.io/document/326/sonm-whitepaper. Online, last visited at 2020-08-20.

[43] Sean Marston, Zhi Li, Subhajyoti Bandyopadhyay, and Anand Ghalsasi. Cloud computing - the business perspective. In *44th Hawaii International International Conference on Systems Science (HICSS-44 2011)*, pages 1–11, 2011.

[44] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/en/bitcoin-paper. Online, last visited at 2020-04-03.

[45] Arvind Narayanan, Joseph Bonneau, Edward Felten, Andrew Miller, and Steven Goldfeder. *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction*. 2016.

[46] Jianli Pan, Jianyu Wang, Austin Hester, Ismail AlQerm, Yuanni Liu, and Ying Zhao. Edgechain: An edge-iot framework and prototype based on blockchain and smart contracts. *IEEE Internet Things J.*, 6(3):4719–4732, 2019.

[47] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *2013 IEEE Symposium on Security and Privacy, 2013*, pages 238–252, 2013.

[48] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: nearly practical verifiable computation. *Commun. ACM*, 59(2):103–112, 2016.

[49] Enigma Project. New to enigma? start here. `https://blog.enigma.co/welcome-to-enigma-start-here-e65c8c9125ef`. Online, last visited at 2020-08-14.

[50] Xiaoyu Qiu, Luobin Liu, Wuhui Chen, Zicong Hong, and Zibin Zheng. Online deep reinforcement learning for computation offloading in blockchain-empowered mobile edge computing. *IEEE Trans. Veh. Technol.*, 68(8):8050–8062, 2019.

[51] Jean-Jacques Quisquater, Myriam Quisquater, Muriel Quisquater, Michaël Quisquater, Louis C. Guillou, Marie Annick Guillou, Gaïd Guillou, Anna Guillou, Gwenolé Guillou, Soazig Guillou, and Thomas A. Berson. How to explain zero-knowledge protocols to your children. In *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, 1989, Proceedings*, pages 628–631, 1989.

[52] Christian Reitwiessner. zksnarks in a nutshell. `https://blog.ethereum.org/2016/12/05/zksnarks-in-a-nutshell/`. Online, last visited at 2020-04-20.

[53] Daniel J. Rosenkrantz, Richard E. Stearns, and Philip M. Lewis, II. An analysis of several heuristics for the traveling salesman problem. *SIAM Journal on Computing*, 6(3):563–581, 1977.

[54] Mahadev Satyanarayanan. A brief history of cloud offload: A personal journey from odyssey through cyber foraging to cloudlets. *GetMobile*, 18(4):19–23, 2014.

[55] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using SGX to conceal cache attacks. *CoRR*, abs/1702.08719, 2017.

[56] Ricky J. Sethi. Traveling salesman problem: An overview. `http://www.sethi.org/misc/Final-tsp-paper.html`. Online, last visited at 2020-03-18.

[57] Aleksandra Skrzypczak. Golem architecture. https://blog.golemproject.net/golem-architecture/. Online, last visited at 2020-09-05.

[58] Nick Szabo. Smart contracts. http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html. Online, last visited at 2020-04-05.

[59] Stefan Tai. Not acid, not base, but salt - A transaction processing perspective on blockchains. In *CLOSER 2017 - Proceedings of the 7th International Conference on Cloud Computing and Services Science, 2017*, page 5, 2017.

[60] Wenda Tang, Xuan Zhao, Wajid Rafique, and Wanchun Dou. A blockchain-based offloading approach in fog computing environment. In *IEEE International Conference on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications, ISPA/IUCC/BDCloud/SocialCom/SustainCom 2018*, pages 308–315, 2018.

[61] Jason Teutsch and Christian Reitwießner. A scalable verification solution for blockchains. https://people.cs.uchicago.edu/~teutsch/papers/truebit.pdf. Online, last visited at 2019-12-15.

[62] Jason Teutsch and Christian Reitwießner. A scalable verification solution for blockchains. *CoRR*, abs/1908.04756, 2019.

[63] Rafael Brundo Uriarte and Rocco De Nicola. Blockchain-based decentralized cloud/-fog solutions: Challenges, opportunities, and standards. *IEEE Communications Standards Magazine*, 2(3):22–28, 2018.

[64] Gavin Wood. Ethereum: A secure decentralized generalised transaction ledger. https://ethereum.github.io/yellowpaper/paper.pdf. Online, last visited at 2020-04-05.

[65] Qi Xia, Emmanuel Boateng Sifah, Kwame Omono Asamoah, Jianbin Gao, Xiaojiang Du, and Mohsen Guizani. Medshare: Trust-less medical data sharing among cloud service providers via blockchain. *IEEE Access*, 5:14757–14767, 2017.

[66] Zehui Xiong, Yang Zhang, Dusit Niyato, Ping Wang, and Zhu Han. When mobile blockchain meets edge computing. *IEEE Commun. Mag.*, 56(8):33–39, 2018.

[67] Guy Zyskind, Oz Nathan, and Alex Pentland. Enigma: Decentralized computation platform with guaranteed privacy. *CoRR*, abs/1506.03471, 2015.