Blockchain-Based Result Verification for Computation Offloading

Benjamin Körbel[1,2], Marten Sigwart[1,2], Philip Frauenthaler[1,2], Michael Sober[1,2], and Stefan Schulte[1,2(✉)]

[1] Christian Doppler Laboratory for Blockchain Technologies for the Internet of Things, TU Hamburg, Hamburg, Germany
{michael.sober,stefan.schulte}@tuhh.de

[2] Christian Doppler Laboratory for Blockchain Technologies for the Internet of Things, TU Wien, Vienna, Austria
https://www.cdl-bot.at

Abstract. Offloading of computation, e.g., to the cloud, is today a major task in distributed systems. Usually, consumers which apply offloading have to trust that a particular functionality offered by a service provider is delivering correct results. While redundancy (i.e., offloading a task to more than one service provider) or (partial) reprocessing help to identify correct results, they also lead to significantly higher cost.

Hence, within this paper, we present an approach to verify the results of offchain computations via the blockchain. For this, we apply zero-knowledge proofs to provide evidence that results are correct. Using our approach, it is possible to establish trust between a service consumer and arbitrary service providers. We evaluate our approach using a very well-known example task, i.e., the Traveling Salesman Problem.

Keywords: Offloading · Verification · Blockchain

1 Introduction

Offloading of computational tasks has gained a lot of research attention in recent years [18]. The basic idea of offloading is that a client device outsources resource-intensive computational tasks to providers, often in exchange for a fee [13]. Hence, when offloading tasks, two parties are involved. Task issuers (i.e., *service consumers*) potentially have limited computational capabilities and therefore are interested in outsourcing particular tasks. Conversely, task processors (i.e., *service providers*) may have idle computational resources and offer their CPU-cycles and further computational resources to process these tasks. Typical examples are the offloading of data processing tasks from lightweight Internet of Things (IoT) or mobile devices in order to decrease processing time or to save energy [24]. For instance, machine learning, (combinatorial) optimization tasks, or the application of heuristics (e.g., genetic algorithms) to solve a complex problem require often resources not available to a potential service consumer. Apart from overcoming limited computational resources, scalability and fault tolerance are also major reasons why offloading is applied [10].

H. Hacid et al. (Eds.): ICSOC 2021, LNCS 13121, pp. 99–115, 2021.
https://doi.org/10.1007/978-3-030-91431-8_7

Offloading can be done to resources following the Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), or Software-as-a-Service (SaaS) models, depending on the needs of the service consumer. Traditionally, computation offloading leverages the cloud, e.g., [16], but more recently, offloading to resources at the edge of the network has also been widely discussed, e.g., [12].

Regardless of the technological setting, offloading requires a client to trust the service provider to deliver correct results. This is a major market entry barrier, since service consumers naturally trust well-known service providers more than new market participants.

In order to avoid reliance on a particular provider, the usage of blockchain technologies for task offloading has previously been discussed [23]. In such approaches (e.g., Golem or iExec—see Sect. 2), the blockchain is a service broker, which brings together consumers and providers, and often delivers further functionalities, e.g., automated settlement after the offloading task has been carried out. Also, the offloading results are delivered through the blockchain.

However, to the best of our knowledge, none of the existing approaches performs a check of the correctness of the delivered results. Ideally, before service consumers pay the service providers for their work, they have an assurance that the returned results can be fully trusted. Previous studies are aware of this issue and discuss solutions based on, e.g., redundant computing, reprocessing fractions of a task locally, or reputation-based systems in order to ensure correct results [3,6,20]. While these approaches may reduce the risk of receiving wrong results, they cannot proof that a result is correct [23]. In other, non-blockchain solutions, the user needs to trust a third party which provides functionalities ensuring trust in the offloading results, e.g., [17].

Furthermore, it should be noted that many offloading tasks do not deliver a deterministic result. For instance, if offloading machine learning tasks or heuristics to solve NP-complete problems, the computation results can differ. This further complicates checking the correctness of a result, since redundant computing or partial reprocessing may lead to different results.

Within the work at hand, we address this issue by conceptualizing, implementing, and evaluating a blockchain-based offloading approach that can prove the proper execution of a particular computation task. By using a blockchain, we dissolve the dependency on a trusted third party. Using a public blockchain also helps to achieve transparency, since information about the off-chaining procedure is publicly available. To prove the correctness of computational results, we apply zero-knowledge proofs (ZKPs). We evaluate our approach using the well-known Traveling Salesman Problem (TSP), showing in which use case areas the proposed solution is beneficial if compared to other alternative approaches, and assessing the cost and time overhead of our approach.

In brief, we provide the following contributions in this paper:

- We assess approaches to ensure trust in results provided by service providers.
- We discuss the utilization of ZKPs and blockchain technologies in order to verify the results for offloaded tasks.

- We design and implement a blockchain-based solution for computation offloading with result verification.
- We apply the TSP as a running example and in order to evaluate the overhead resulting from the presented approach.

The remainder of this paper is organized as follows: In Sect. 2, we discuss the related work. In Sect. 3, we assess different approaches to verify the results of offloaded computation tasks. Based on this, we present our design and implementation in Sect. 4. Section 5 shows the results of the evaluation of the presented work, and Sect. 6 concludes this paper.

2 Related Work

To the best of our knowledge, the field of blockchain-based, verifiable task offloading is still a novel research area, and not too many approaches have been presented so far.

Golem [20], iExec [6], and SONM [21] are three commercial solutions, aiming at decentralizing offloading to the cloud [23]. Their respective primary goal is to provide solutions to decrease market entry barriers, by allowing arbitrary providers to offer computational resources on a blockchain, and arbitrary consumers to use these resources. Notably, the intended providers of cloud resources are not large-scale data centers, but could be anyone with idle computational resources. Golem, iExec and SONM aim at providing marketplace and broker functionalities, and apply a pay-per-use model, i.e., the consumer has to pay for using computational infrastructure or for processing a particular task. Notably, in contrast to the work at hand, which focuses on a SaaS model, these solutions aim at providing computing power in general, i.e., on the IaaS level.

With regard to result verification, Golem supports redundant computation, but also allows to recompute fractions of an offloaded task locally (i.e., at the service consumer's side), and to subsequently compare the results. Also, Golem implements a reputation mechanism, which is based on consumer (e.g., late payments) and provider behavior (e.g., not delivering results in time), respectively [20]. iExec applies a similar approach, where the service consumer can define the needed reliability of the results. If this value is high, a higher degree of redundancy is applied when computing the offloading tasks, and more reliable providers, i.e., with a high reputation, are selected. Notably, iExec also allows to support Software Guard Extension (SGX), which is a kind of enclave-based off-chain computations (see Sect. 3) [6]. So far, SONM does not implement a verification mechanism, but names reputation management as a major enabler to provide reliable computation results [21].

None of the so-far discussed approaches provides a proof that the results of a computation are correct. Instead, redundant computations, recomputing fractions of tasks locally, and reputation-based methods only *decrease* the risk that the results are not correct. Especially redundant computations also increase the cost by quite some degree, since all involved service providers charge a fee for the computations. Reputation systems can be helpful, but provide market

entry barriers since new service providers need to build a reputation. Also, it remains unclear how these solutions handle results which are not deterministic, e.g., for machine learning or heuristic tasks.

FlopCoin [3] is a blockchain-based offloading framework with a decentralized incentive and reputation scheme. Among other metrics, the reputation of participants is used as input for the offloading decision, i.e., to which provider of computational resources a particular task is offloaded. EdgeChain [14] uses a blockchain and smart contracts to link computational resources at the edge and IoT devices which need to offload tasks. The blockchain is used to monitor the offloading procedure and to conduct payments. A mechanism to detect malicious nodes based on past behavior is also introduced. Qiu et al. [15] discuss a similar approach, but apply deep reinforcement learning to find an assignment of tasks and available edge resources. Very recently, another approach for offloading to the fog has been presented by Wu et al. [25]. The focus of this work is also on the actual decision making, i.e., where to place which offloaded task. In contrast, we aim primarily on proving that computed results are valid.

To the best of our knowledge, none of the discussed research papers directly address offloading result verification. Hence, the work at hand complements existing work, and could be used within existing solutions in order to proof that an offloaded computation provides valid results.

3 Result Verification for Offloaded Tasks

As discussed above, it is the goal of the work at hand to provide mechanisms that can verify results of offloaded computational tasks. In general, we focus on the SaaS model, but in fact, result verification could also be done for user-deployed services using the IaaS or PaaS model.

To achieve result verification, different schemes could be applied: *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 (here: the service provider). Subsequently, the validity of the computation can be verified on-chain using a smart contract [4].

Verifiable off-chain computation can be realized using ZKPs [7]. The basic idea behind ZKPs is to convince someone that a statement is true without revealing any underlying information needed to proof that the statement is true. This allows to hide the input data for a proof and therefore supports data privacy. Importantly, in the scenario at hand, this facilitates the verification that the results delivered by a service provider are correct, without the need that the provider reveals its applied service or algorithm. Hence, the computation performed by the service provider remains a blackbox from the perspective of result verification. This is even the case for non-deterministic computations, e.g., if a heuristic is applied. As long as it is possible to define rules which describe if an offloading result is valid, ZKPs can be applied successfully.

ZKPs can be realized in the form of Zero-Knowledge Non-Interactive Succinct Arguments of Knowledge (zk-SNARKs), Zero-Knowledge Scalable Transparent Arguments of Knowledge (zk-STARKs), and Bulletproofs [9].

zk-SNARKs are non-interactive and provide relatively cheap verification by their succinctness. Before generating a proof and performing the verification step, a one-time setup must be carried out by a trusted party. 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. Both zk-STARKs and Bulletproofs feature growing proof-size and on-chain verification, while zk-SNARKs are independent of the task complexity and provide compact proves [4]. Due to the succinctness of zk-SNARKs, very short proofs (i.e., in the range of bytes) can be provided, which is very beneficial when blockchain technology is involved. Therefore, we decided to apply zk-SNARKs for result verification.

We have also investigated other result verification schemes: For instance, *Secure Multiparty Computation* (SMPC) protocols enable the construction of privacy-preserving off-chain computation schemes, but are accompanied by high overhead [4]. *Enclave-based off-chain computation* relies on Trusted Execution Environments (TEEs) which enable code execution while preserving confidentiality and integrity. The enclave-based scheme allows universal computations but has potential security issues [8,19]. *Incentive-driven off-chain computing* rewards nodes which are doing verification work to check if a computation is correct. One implementation of this scheme is TrueBit [22]. A challenge when using the described scheme is to keep nodes motivated for performing verifications continuously. Also, the throughput of completed computation tasks and the general service can be hindered by malicious verifiers by marking each computation result as faulty [4].

The selection of zk-SNARKs allows us to make use of the ZoKrates toolbox [5], which supports the entire process of specifying, integrating and deploying ZKPs on Ethereum-based blockchains. The toolbox consists of a Domain-specific Language (DSL), a compiler and generators for proofs as well as smart contracts for verification. In brief, ZoKrates can be used to execute a computational task off-chain. Afterwards, the result of the computation (here: of an offloading task) and the corresponding proof are written back to a blockchain. The proof that attests correct (or incorrect) computation can then be verified on-chain. Therefore, the computational effort on a blockchain is reduced, while privacy can be preserved due to the usage of ZKPs.

4 Design and Implementation

4.1 Overview

After having selected zk-SNARKs as the underlying approach to provide result verification for computation offloading, we are now able to design a solution.

As discussed before, we make use of a blockchain-based approach to offload tasks to service providers. While this has been proposed before (see Sect. 2), there

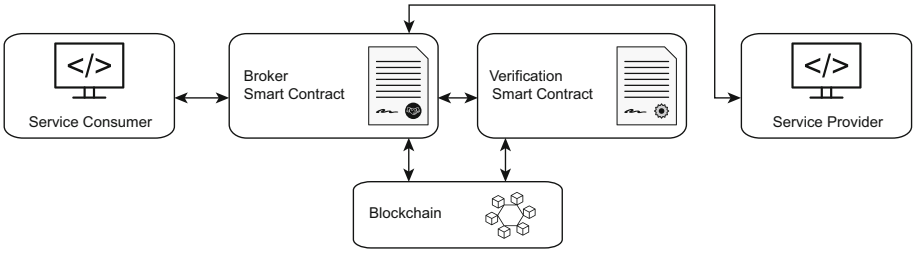

Fig. 1. Blockchain-based computation offloading with result verification

is lack of solutions which allow to verify the results delivered by the service providers. Due to space constraints, we focus on this particular functionality in the work at hand. However, we have in fact designed and implemented a framework which covers the necessary functionality stack, i.e., acts as a blockchain-based broker for service consumers (offloaders) and service providers, and implements an incentive structure, so that fees can be charged and are automatically paid if a result has been verified. Notably, while the implemented solution can be used by traditional cloud providers to offer their resources and services, it could also be used in fog and edge settings, or by private persons who want to offer spare computational resources.

In the case of non-deterministic results, e.g., since a heuristic is applied by a service provider (see Sect. 1), our framework allows to obtain results from different providers and to compare the result quality. The integration of methods to assess the result quality is part of our future work (see Sect. 6).

Figure 1 shows the components of the software solution. As it can be seen, the system consists of the service consumer (i.e., the client software for the task offloader), the service provider (i.e., the according client software), the broker smart contract and the result verification smart contract. In the following subsections, we discuss the core components with a focus on the verification functionalities.

4.2 Blockchain-based Brokering and Result Verification

The blockchain serves two major purposes in our scenario: First, it acts as a broker during the offloading process, i.e., facilitates the cooperation between a service consumer and a service provider. This includes provisioning of results to the service consumer and payment to the service provider. Second, the blockchain delivers the result verification (see Sect. 4.4). Following the approach presented in this work, no preexisting relationship and no position of trust between a service consumer and potential service providers need to exist.

Brokering functionalities and result verification are implemented using smart contracts. Within the work at hand, we use Ethereum for this, since it provides a broad acceptance in the research community as well as industry. It should be noted that the presented approach is per se protocol-agnostic, and could also be

implemented using a permissioned blockchain like Hyperledger Fabric. However, we opted to use a public blockchain in the work at hand.

4.3 Broker Smart Contract

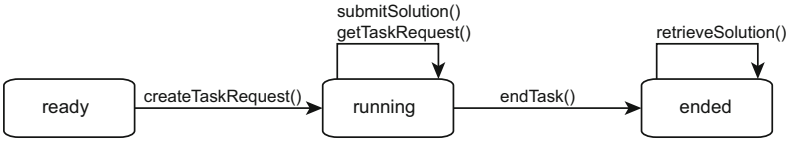

Fig. 2. Simplified state diagram of the broker smart contract

Figure 2 provides an overview of the states the broker smart contract passes through. As it can be seen in the state diagram, the contract may be in the states *ready*, *running*, and *ended*. For the transition between the states, particular functions must be called. Before any interaction is possible, i.e., any function is callable, the smart contract needs to be deployed on the blockchain. After the deployment, it is in state *ready*. At this point, a service consumer can create a new request for offloading, i.e., a new offloading task, using *createTaskRequest()*.

The necessary inputs for *createTaskRequest()* are a stake, which is used as a deposit for the later payment to the service provider, information about the offloaded task, and a boolean value if the result should be verified or not. Once this has been done, the state changes to *running*.

At this point of time, a potential service provider can retrieve all the necessary information to process the task by calling *getTaskRequest()*. Notably, the selection of the service provider could follow different patterns, e.g., based on reputation and/or load balancing as proposed in the related work (see Sect. 2), by applying a reverse auction so that potential service providers compete for the requests, or other allocation techniques.

Since this is not in the focus of the work at hand, we implement a simplified approach, i.e., once a service provider has computed a result, it can be published using *submitSolution()*. When calling this function, the service provider needs to deliver the actual solution to the request. At this point, network participants including the service consumer can see the submitted solution due to the public nature of the blockchain. To circumvent that a service consumer reads the result and does not pay the provider, the stake deposited by the consumer is used.

Any network participant (here: the service consumer or the service providers) may close the task by using *endTask()*, which also means that the state changes to *ended* and that the payment to the service provider is triggered. Notably, this is only possible once a minimum duration has passed, which is also defined by the service consumer. When the task is ended, the service consumer can collect the solution by calling the function *retrieveSolution()*. As written before, the consumer could also read the result simply from the blockchain. We explicitly

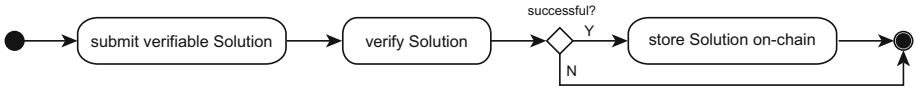

Fig. 3. High level process of the result verification

foresee *retrieveSolution()* as the possibility to implement a more sophisticated function here, e.g., to encrypt and decrypt the result in order to not release the solution publicly on the blockchain, or to provide the solution via a blockchain-external data storage like the InterPlanetary File System (IFPS) [11], in order to save gas cost.

4.4 Result Verification

Figure 3 gives a high level overview of result verification within our approach. As it can be seen, solutions of verifiable tasks are written to the blockchain only if the result verification is successful. Otherwise, the submission is discarded. The activity *verify Solution* is part of the result verification while all other activities in Fig. 3 belong to the broker smart contract discussed in Sect. 4.3. The verification is performed within a separate smart contract generated by ZoKrates. For this, the broker smart contract has to trigger the verification function *verifyTx()* in the verification smart contract. An example implementation of *verifyTx()* is discussed in Sect. 4.6.

The procedure of the result verification is illustrated in Fig. 4. First, a service provider decides to contribute to a particular offloading task, which is the start event for the result verification. For this, the provider retrieves all relevant input data. Based on that, the service provider computes the task locally (and therefore off-chain) with a service running on the provider's computational resources. After a solution has been found, a witness has to be computed. This means that a program specified in DSL code is run with the service provider's result as input; this DSL program is providing important input for the ZoKrates toolbox as discussed in more detail in Sect. 4.6. If the verification succeeds, the DSL program returns a witness that proves proper computation. Otherwise, it can be assumed that an error occurred during the computation phase or a wrong result has been entered. In this case, the computation is repeated. If the computation fails again, the procedure ends unsuccessfully (not shown in Fig. 4).

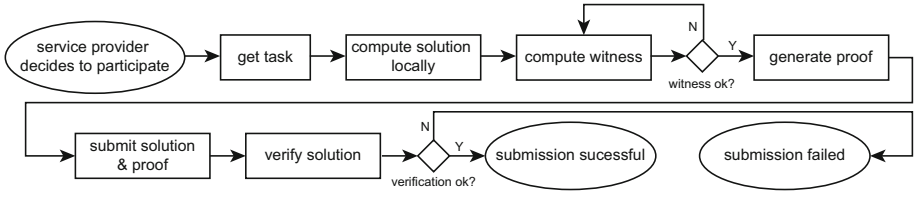

Fig. 4. Result verification procedure

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 service provider's hardware. Afterwards, the solution and the proof can be submitted to the broker smart contract. When a solution is submitted, the broker smart contract calls the verification function *verifyTx()* of the verification smart contract. If the verification function returns true, we can assume that the computation has been executed honestly. As a result, the solution is stored on-chain. Otherwise, if false is returned, the submission of the service provider is discarded entirely. In both cases, the procedure subsequently ends.

4.5 Implementing Result Verification for Specific Use Cases

Result verification is naturally tied to specific computation problems. This means that a certain part of the result verification, more specifically the DSL program implemented using the ZoKrates toolbox, is not generic and must be adapted whenever a different computation problem has to be served.

Accordingly, 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 service consumers for new computation problems can be met. The mandatory steps to add a new use case are depicted in Fig. 5.

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. Within the DSL program, some logic, e.g., conditions, has to be specified, which makes the solution for a specific problem true. The DSL program takes a number of inputs (depending on the use case) and verifies if all specified conditions are met. Consider the offloading task of calculating the sum of two integers. In this case, the equation $a + b = c$ must hold and has to be encoded in the DSL. In general, an arbitrary task can be verified, as long as it is possible to define a rule that the result has to follow. As described in Sect. 3, for this, the service provider does not have to disclose any information about the applied algorithm or method.

Then, keys and the verification smart contract have to be generated. For the keys, this means that a trusted setup is necessary. In the work at hand, we assume that the trusted setup is performed by the developers of the smart contracts presented in Sect. 4.2. Afterwards, the on-chain verification for the specific computation problem is ready for deployment. Alternatively, a multiparty computation protocol could be applied, e.g., [2], that prevents fake proofs as long as one participant is honest.

When the result verification scheme is integrated into the broker 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 broker smart contract for each use case. In this case, the basic functions of the broker smart contract,

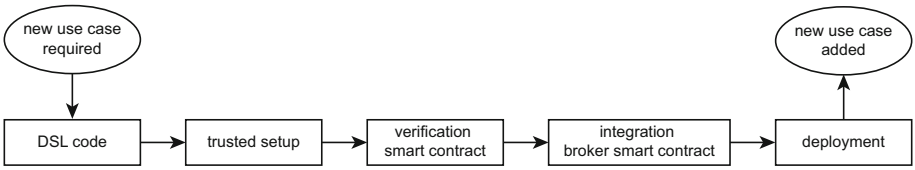

Fig. 5. Process for adding result verification for additional use cases

e.g., *getTaskRequest()* can be copied, but use case-specific data, e.g., necessary fields for the problem instance and solutions, must be considered and adapted as well. Furthermore, the function *submitSolution()* that also calls *verifyTx()* of the verification smart contract has to be tailored. A separation per use case would lead to more compact code artifacts and better maintenance. However, to simplify the description of our solution approach, we only discuss the usage of one broker smart contract in this paper.

When a new use case is added, the main effort consists of rewriting the DSL program and integrating the verification smart contract into the broker smart contract. The aforementioned trusted setup and creation of the verification smart contract is mainly handled by the ZoKrates toolbox. The deployment of new smart contracts is tool-supported as well. This means that merely predefined commands have to be executed.

4.6 Example Implementation

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 based on the TSP.

The TSP is selected since it is very well-known among computer scientists, and easy to understand, but actually hard to compute. In brief, the TSP describes the problem to find the shortest path to travel a predefined number of cities and to return back to the origin city, but visiting any other city only once. As input, a list of cities and the distances between each pair of cities are given [1]. Once a solution to the TSP has been found, its validity can be verified in little time, e.g., by traversing the path of a solution. Interestingly, it is also easy to assess the quality of the solution, i.e., by comparing the computed path length.

The TSP is an NP-hard problem, which also opens interesting future research direction (see Sect. 6). Notably, the TSP is merely used as an exemplary use case in the work at hand. The presented approach explicitly allows to verify other tasks as well. The according implementation including the broker smart contract can be found at Github[1].

There are some restrictions which have to be taken into account because of the applied ZoKrates toolbox. First, the toolbox only supports numbers, i.e.,

[1] https://github.com/ben2048/blockchainBasedComputationOffloading.

Alg. 1. Main function of the DSL program

```
1   def main(
2   private field[10] path, private field mapnumber, field sum,
3   private field[10] cities, field[2] hashOfCities, field[2] hashOfPath
4   ) -> (field):
5
6   1 == basicInputCheck(path, cities, mapnumber)
7   1 == checkCities(path, cities, mapnumber)
8   sum == calculateSum(path, mapnumber)
9
10  field[2] hashedPath = hash(concat(path))
11  hashOfPath[0] == hashedPath[0]
12  hashOfPath[1] == hashedPath[1]
13
14  field[2] hashedCities = hash(concat(cities))
15  hashOfCities[0] == hashedCities[0]
16  hashOfCities[1] == hashedCities[1]
17
18  return 1
```

cities in the TSP have to be represented by numbers, not by their names. Second, ZoKrates does not support any dynamic fields. Accordingly, the size of an array needs to be defined at compile time. Both constraints complicate the implementation a little bit, but do not lead to any significant restrictions. We discuss the impact of the fixed array sizes also in the evaluation in Sect. 5.

As described in Sect. 4.5, the result verification is based on a DSL which proves program inputs against defined checks. With regard to the TSP, the result verification has to verify whether a solution has been computed properly. In other words, service providers need to prove that they have found a valid solution for an TSP instance. To accomplish that, the produced path has to be Hamiltonian (i.e., each city appears exactly once in the path) and the path length must correspond to the sum of the connections between the cities on the basis of the path and the given distances [1]. As input, the map of cities for the TSP has to be defined, including the cities (represented by numbers) and the distances between the cities, i.e., a complete graph made up from vertices (cities) and edges (distances) between the vertices. This data structure can be stored as an array within the DSL program.

In the following paragraphs, we discuss the example given in Algorithm 1. To verify that a service provider has computed a correct result, the following information needs to be provided (lines 2–3): (i) The computed `path`, consisting of a sequence of numbers representing cities, (ii) an ID `mapnumber` for the map which has been used for solving the TSP instance (this allows to use different maps with the TSP), (iii) the computed length of the path `sum`, (iv) the `cities` for which the minimal distance has been computed, and (v) the hashed path `hashOfPath` and the hashed cities `hashOfCities`. As it can be seen, the example allows a maximum of ten cities on a path, but other lengths are also possible.

The path and the map are needed to calculate the distance and to compare it with the stated length (i.e., the result) from the service provider. Path and cities are used to determine if all cities are covered exactly once in a path. The hashed

Alg. 2. verifyTx()

```
1   function verifyTx(
2   uint[2] memory a, uint[2][2] memory b,
3   uint[2] memory c, uint[6] memory input
4   ) public returns (bool r) {
5       Proof memory proof;
6       proof.a = Pairing.G1Point(a[0], a[1]);
7       proof.b = Pairing.G2Point([b[0][0], b[0][1]], [b[1][0], b[1][1]]);
8       proof.c = Pairing.G1Point(c[0], c[1]);
9       uint[] memory inputValues = new uint[](input.length);
10      for(uint i = 0; i < input.length; i++){
11          inputValues[i] = input[i];
12      }
13      if (verify(inputValues, proof) == 0) {
14          return true;
15      } else {
16          return false;
17      }
18  }
```

path is necessary to prevent malicious behavior originating from the service provider when submitting a solution. Without the hash, it would be possible to decouple the proof from the path. In other words, if the service provider submits a valid proof but an invalid path, it possibly cannot be detected on-chain, i.e., the result verification would succeed even though an incorrect path would be stored on the blockchain. To recognize and prevent such scenarios, we compare the hash and the path within the DSL program and in the broker smart contract.

As it can be seen, the main function first performs an input check (line 6). This is done in order to sort out solutions with invalid indices or map numbers. Next, it is checked if the stated path contains each city exactly once (line 7), i.e., if the path is Hamiltonian. Afterwards, it is checked if the stated path length (field `sum` in line 8) is equal to the sum resulting from the distances of the stated path based on the distances between cities, i.e., the map. Then, it is necessary to embed the hash of the path in the verification procedure (lines 10–12). 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, to prevent the aforementioned malicious action. For this, we utilize the implementation of SHA256 provided by ZoKrates.

As discussed before, the DSL program shown in Algorithm 1 runs off-chain, while the verification function *verifyTx()* is carried out on-chain. Notably, the number of inputs of *verifyTx()* is a cost factor. With regard to our use case and the specification of the input parameters in the DSL program, the number of cities scales with the instance size. This becomes a crucial cost factor, since the gas cost rise linearly with the number of public inputs provided. Hence, we make use of the hashed cities instead of the number of cities. This allows to make the cities a private input, but also makes it necessary to check the hash of the cities (lines 14–16), analogue to lines 10–12.

Based on the DSL program, ZoKrates is able to define *verifyTx()* as depicted in Algorithm 2. The input array consists of the path length, hash values for

the path and the cities plus the expected return value of the DSL program (lines 2–3). Afterwards, a new proof is instantiated (line 5). Then, *verifyTx()* requires three elliptic curve points in form of the arrays `a`, `b`, `c` (lines 6–8). These elliptic curve points actually make the zk-SNARKs proof and are delivered via the DSL program by the ZoKrates toolbox. Hence, the developer who integrates *verifyTx()* does not have to take care of the actual proofs, which is a major reason for using the ZoKrates toolbox.

The array `input` depicts the public inputs and the expected return value of the DSL code, and is used to fill the `inputValues` (lines 9–12). Afterwards, the actual verification is carried out (line 13). In the case of a successful verification, the boolean true is returned (line 14), else, the boolean false is returned (line 16). Thus, a service consumer can be sure that a result is valid (or not), and the broker smart contract could carry out the payment. Notably, since only the proof is published on the blockchain, no conclusions regarding the computation and concrete results are possible. This ensures the privacy property regarding the proof. However, as written above, the solution is still available on the chain. To avoid this, a solution could be encrypted (see Sect. 4.3).

5 Evaluation

5.1 Evaluation Setup

In order to evaluate the presented approach, we measure the overhead (regarding time and cost) occurring because of result verification.

As mentioned in Sect. 4, the computation offloading consists of an on-chain and an off-chain part. To evaluate the on-chain activities, the smart contracts have been deployed in a local Ethereum blockchain (using Truffle), while for the off-chain activities, ZoKrates has been installed locally in a Docker container.

5.2 Overhead Analysis

The result verification is part of the solution submission process as presented in Sects. 4.3 and 4.4. Hence, in order to evaluate the overhead with regard to gas cost and time, we implemented a second *submitSolution()* function in the broker smart contract. However, this version of the function does *not* verify the submitted solution. This allows us to compare the gas and time consumption of a benchmark with the according values of our zk-SNARKs-based verification approach. As a second benchmark, we conduct an on-chain result verification, i.e., a solution for a TSP instance is verified by a separate smart contract deployed on the blockchain. We apply two maps with size 30 and 70, respectively, in order to see how the map size (i.e., number of cities) influences the results. To get a complete picture of the cost overhead, TSP instance sizes between 3 and 30 (*map 30*) respectively 3 and 60 (*map 70*) are used.

The results regarding the cost for map 70 are shown in Fig. 6. Not surprisingly, the gas consumption with verification is higher than without verification. If

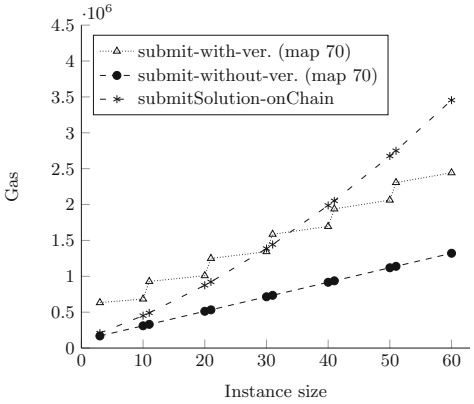

Fig. 6. Gas consumption

no verification is done, there is no difference between the two maps. Hence, there is only one plot for the offloading without verification. Actually, this benchmark also indicates how big the cost become if redundant computation is used (see Sect. 2). In that case, based on the level of redundancy, each submitted solution leads to the same cost. In addition, some overhead for the comparison (e.g., majority voting) of the results needs to be taken into account. This shows that redundant computing is not really an option with regard to the gas consumption for the solution submission.

While not shown in the figure in order not to overload the plot, the submission of solutions based on the smaller map (with 30 cities) is marginally cheaper than solutions based on the larger map (with 70 cities).

It can also be seen that verifying TSP solutions on-chain (*submitSolution-onChain*) is cheaper than the approach presented in this work for small instances, but more expensive for large ones. Up to an instance size of 29, the on-chain verification is cheaper than applying zk-SNARKs. For instances of size 30, the zk-SNARKs-based 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 zk-SNARKs-based variant at instance size 40. 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 zk-SNARKs-based variant are visible. To determine the exact break-even point(s) between 29 and 40, we have performed further (not-depicted) measurements showing that the on-chain variant is more cost-efficient for instances of size 31–34. From instance size 35, the zk-SNARKs-based variant should be preferred.

These results show that it is necessary to discuss the line course in more detail: It becomes clear that the gas demand of all three variants shown in Fig. 6 increases with the instance size. It is also noticeable that the cost levels of *submit-without-ver. (map 70)* and *submitSolution-onChain* rise continuously,

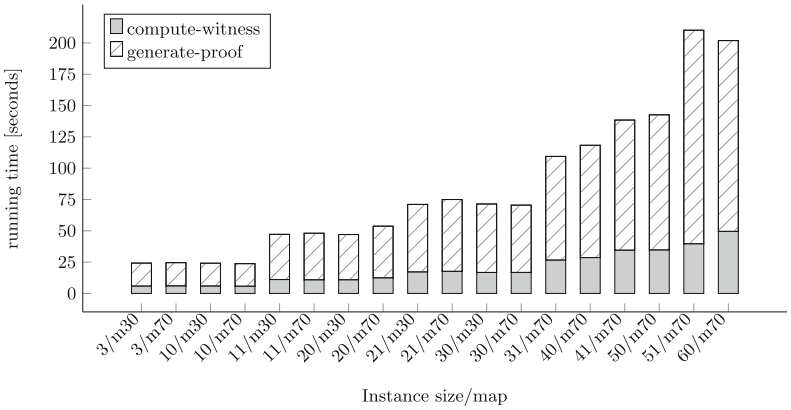

Fig. 7. Time overhead

while the levels of *submit-with-ver. (map 70)* increase step-wise. This is caused by the partitioned verification, due to the lack of dynamic fields and the subsequent fixed array sizes within the DSL of ZoKrates. For solutions of size 3 to 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 DSL solution for 11 cities is computed, the resulting path has to be padded, according to the expected number of inputs of the DSL program. Due to the fact that such a padding is not necessary when no verification is performed or the verification is done on-chain, the gas consumption merely rises continuously for these options, but for the zk-SNARKs-based approach, "jumps" in the plot can be seen. A solution to circumvent this would be to have different smart contracts for different TSP instance sizes.

Second, we observe the time overhead. Figure 7 depicts the time needed to execute the *compute-witness* and *generate-proof* step for TSP instances of sizes 3 to 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 overall run-time increases with the size of instances. The step *generate-proof* takes on average 3.2 (standard dev.: 0.28) times longer than the *compute-witness* step. Considering the increasing runtime of the proof generation and the computation of the witness, it becomes clear that the presented result verification approach should primarily be used in scenarios which are not very time-critical.

6 Conclusion

In order to provide solutions for fully decentralized task offloading, a number of blockchain-based solutions have already been proposed. However, to the best of our knowledge, none of these solutions is able to verify that an offloaded task is computed correctly. Instead, redundant computing or trust models are applied, which however cannot guarantee that a computation is valid.

We have therefore presented an approach which supports the verification of computation results, applying zk-SNARKs. Especially, this allows to compute results and proofs off-chain, and to only verify the proofs on-chain. While our solution could be integrated into existing blockchain-based offloading frameworks, we have also provided a simplified broker solution as part of this paper.

In our future work, we want to further extend the presented approach. Especially, we want to replace the brokering functionality by a more sophisticated one which also allows the broker to take into account quality requirements (e.g., a particular quality of a result), and to select based on this the best result from a number of provided solutions. In fact, selecting the TSP as our evaluation use case already lays the foundations for this, since service providers could deliver different solution qualities with different algorithms and at different cost to this NP-hard problem. In other scenarios, the assessment of the result quality is a more complex task and therefore an interesting direction of future work.

While currently our reference implementation applies a simple pricing scheme, i.e., the service provider is paid with the consumer's stake, more complex pricing might be useful. For instance, a dynamic pricing scheme based on the complexity of an offloaded computational task and the number of available service providers might be helpful. Last but not least, as has been shown in the evaluation, an on-chain verification is sometimes cheaper to conduct than the proposed off-chain result verification. Therefore, we will further investigate in which cases which of these two approaches should be preferred.

Acknowledgments. The financial support by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development and the Christian Doppler Research Association is gratefully acknowledged.

References

1. Applegate, D.L., Bixby, R.E., Chátal, V., Cook, W.J.: The Traveling Salesman Problem - A Computational Study. Princeton University Press, Princeton (2007)
2. Ben-Sasson, E., Chiesa, A., Tromer, E., Virza, M.: Succinct non-interactive zero knowledge for a von Neumann architecture. In: 23rd USENIX Security Symposium, pp. 781–796. USENIX Association (2014)
3. Chatzopoulos, D., Ahmadi, M., Kosta, S., Hui, P.: FlopCoin: a cryptocurrency for computation offloading. IEEE Trans. Mobile Comput. **17**(5), 1062–1075 (2018)
4. Eberhardt, J., Heiss, J.: Off-chaining models and approaches to off-chain computations. In: 2nd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers, pp. 7–12. ACM (2018)
5. Eberhardt, J., Tai, S.: ZoKrates - scalable privacy-preserving off-chain computations. In: 1st IEEE International Conference on Blockchain, pp. 1084–1091. IEEE (2018)
6. Fedak, G., Bendella, W., Alves, E.: Blockchain-Based Decentralized Cloud Computing. https://iex.ec/wp-content/uploads/pdf/iExec-WPv3.0-English.pdf (2017)
7. Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game or a completeness theorem for protocols with honest majority. In: 19th Annual ACM Symposium on Theory of Computing, pp. 218–229. ACM (1987)

8. Greenberg, A.: Hackers can mess with voltages to steal intel chips' secrets. https://www.wired.com/story/plundervolt-intel-chips-sgx-hack
9. Kosba, A., Papadopoulos, D., Papamanthou, C., Song, D.: MIRAGE: succinct arguments for randomized algorithms with applications to universal zk-SNARKs. In: 29th USENIX Security Symposium, pp. 2129–2146. USENIX Association (2020)
10. Kosta, S., Aucinas, A., Hui, P., Mortier, R., Zhang, X.: ThinkAir: dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In: 31st IEEE International Conference on Computer Communications, pp. 945–953. IEEE (2012)
11. Krejci, S., Sigwart, M., Schulte, S.: Blockchain- and IPFS-based data distribution for the internet of things. In: Brogi, A., Zimmermann, W., Kritikos, K. (eds.) ESOCC 2020. LNCS, vol. 12054, pp. 177–191. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-44769-4_14
12. Mach, P., Becvar, Z.: Mobile edge computing: a survey on architecture and computation offloading. IEEE Commun. Surv. Tutorials **19**(3), 1628–1656 (2017)
13. Marston, S., Li, Z., Bandyopadhyay, S., Zhang, J., Ghalsasi, A.: Cloud computing - the business perspective. Decis. Support Syst. **51**, 176–189 (2011)
14. Pan, J., Wang, J., Hester, A., AlQerm, I., Liu, Y., Zhao, Y.: EdgeChain: an edge-IoT framework and prototype based on blockchain and smart contracts. IEEE Internet of Things J. **6**(3), 4719–4732 (2019)
15. Qiu, X., Liu, L., Chen, W., Hong, Z., Zheng, Z.: Online deep reinforcement learning for computation offloading in blockchain-empowered mobile edge computing. IEEE Trans. Veh. Technol. **68**(8), 8050–8062 (2019)
16. ur Rehman Khan, A., Othman, M., Madani, S.A., Khan, S.U.: A survey of mobile cloud computing application models. IEEE Commun. Surv. Tutorials **16**(1), 393–413 (2014)
17. Santos, N., Gummadi, K.P., Rodrigues, R.: Towards trusted cloud computing. In: 2009 Conference on Hot Topics in Computing. USENIX Association, Article No. 3 (2009)
18. Satyanarayanan, M.: A brief history of cloud offload: a personal journey from odyssey through cyber foraging to cloudlets. GetMobile Mob. Comput. Commun. **18**(4), 19–23 (2014)
19. Schwarz, M., Weiser, S., Gruss, D., Maurice, C., Mangard, S.: Malware guard extension: using SGX to conceal cache attacks. In: Polychronakis, M., Meier, M. (eds.) DIMVA 2017. LNCS, vol. 10327, pp. 3–24. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-60876-1_1
20. Skrzypczak, A.: Golem Architecture. https://blog.golemproject.net/golem-architecture/ (2017). Accessed 20 May 2021
21. Sonm Pte. Ltd.: SONM - Supercomputer Organized by Network Mining. https://whitepaper.io/document/326/sonm-whitepaper
22. Teutsch, J., Reitwießner, C.: A scalable verification solution for blockchains. CoRR abs/1908.04756 (2019). http://arxiv.org/abs/1908.04756
23. Uriarte, R.B., De Nicola, R.: Blockchain-based decentralized cloud/fog solutions: challenges, opportunities, and standards. IEEE Commun. Stand. Mag. **2**(3), 22–28 (2018)
24. Wu, H., Sun, Y., Wolter, K.: Energy-efficient decision making for mobile cloud offloading. IEEE Trans. Cloud Comput. **8**(2), 570–584 (2020)
25. Wu, H., Wolter, K., Jiao, P., Deng, Y., Zhao, Y., Xu, M.: EEDTO: an energy-efficient dynamic task offloading algorithm for blockchain-enabled IoT-edge-cloud orchestrated computing. IEEE Internet Things J. **8**(4), 2163–2176 (2021)