

A Voting-Based Blockchain Interoperability Oracle

Michael Sober^{*†}, Giulia Scaffino^{*‡}, Christof Spanring[§], Stefan Schulte^{*†}
**Christian Doppler Laboratory for Blockchain Technologies for the Internet of Things*

†Institute of Data Engineering, TU Hamburg, Hamburg, Germany
{michael.sober, stefan.schulte}@tuhh.de

‡Institute of Logic and Computation, TU Wien, Vienna, Austria
giulia.scaffino@tuwien.ac.at

§Pantos GmbH, Vienna, Austria
christof.spanring@bitpanda.com

Abstract—Today’s blockchain landscape is severely fragmented as more and more heterogeneous blockchain platforms have been developed in recent years. These blockchain platforms are not able to interact with each other or with the outside world since only little emphasis is placed on the interoperability between them. Already proposed solutions for blockchain interoperability such as naive relay or oracle solutions are usually not broadly applicable since they are either too expensive to operate or very resource-intensive.

For that reason, we propose a blockchain interoperability oracle that follows a voting-based approach based on threshold signatures. The oracle nodes generate a distributed private key to execute an off-chain aggregation mechanism to collectively respond to requests. Compared to state-of-the-art relay schemes, our approach does not incur any ongoing costs and since the on-chain component only needs to verify a single signature, we can achieve remarkable cost savings compared to conventional oracle solutions.

Index Terms—blockchain interoperability, blockchain oracles, threshold signature, smart contracts

I. INTRODUCTION

In recent years, blockchain technology has become increasingly important not only as the underlying technology for cryptocurrencies [1] but also in many other application areas including supply chain management [2], healthcare [3], and others. This has led to the development of more and more heterogeneous blockchain platforms [4], [5]. These are often tailored to specific requirements, as there is not one blockchain that is capable of fulfilling the (often) disjunctive needs of different application areas.

Research and industry tend to not consider the interoperability between different blockchain platforms, turning these platforms into self-contained systems [4]. As a result, these platforms are not able to collaborate to benefit from novel features and properties of other or newly developed blockchain platforms. The problem of weak interoperability does not only exist within the world of blockchains but generally with systems that are outside the blockchain platform’s boundaries. As an example, we can use the smart contract and decentralized application platform Ethereum [6]. Smart contracts running in the Ethereum Virtual Machine (EVM) cannot interact with the outside world, which means that a smart contract is only able to read and modify the state of the hosting blockchain.

For many applications, however, it is important to be able to obtain data from the outside world. One can imagine, for example, a decentralized application that needs flight data from an airline to determine if a user has the right to compensation in case a flight is delayed or a cross-chain token transfer that requires the knowledge of whether the tokens were burned on the original blockchain. This problem is also known as the blockchain oracle problem [7].

Many attempts have been made to achieve interoperability between different blockchain platforms [5] and to solve the oracle problem [7], [8]. One possible approach to achieve interoperability between two blockchain platforms is the use of blockchain relays [9]. Such relays are usually provided as smart contracts running on a “target blockchain”, i.e., the blockchain which needs data from a “source blockchain”. With blockchain relays, the state of one blockchain is replicated within another blockchain, which enables callers of the relay contract to verify the existence of a transaction on the source blockchain.

Bitcoin and Ethereum use Merkle trees to store transactions in a block, whereby the root of the Merkle tree is stored in the block header. Therefore, a relay contract can use Simplified Payment Verification (SPV) which utilizes Merkle proofs to check whether a transaction is included in a block of a source blockchain [9]. Since block header validation is done by a smart contract on-chain, no trusted intermediary is needed to relay the block headers. However, this has the disadvantage that such relays also have considerable costs, since on-chain verification is very expensive and blocks have to be relayed continuously, even if they are perhaps not needed at all.

Another approach is to make use of blockchain oracles to get information from the outside world. Blockchain oracles are bridges between blockchain platforms and external data sources. The task of a blockchain oracle is to query data from external data sources and then to pass the data items to a smart contract. The problem here is to ensure the authenticity and integrity of the data because we have to trust the oracle that it behaves honestly [8].

One can differentiate between oracles that are based on a centralized or a decentralized approach [7]. Centralized oracles represent a single point of failure since trust in a single oracle node is required. Following the decentralized model, there is

no single point of failure and the trust assumption moves from one oracle node to multiple oracle nodes. Many decentralized oracles follow a voting-based approach to provide data, i.e., the votes are aggregated to determine the overall result. Unfortunately, the aggregation mechanism can become very expensive, if carried out on-chain.

To make a step towards blockchain interoperability and overcome the issues of current relay schemes and oracle solutions, we investigate the application of Boneh-Lynn-Shacham (BLS) threshold signatures to create an off-chain aggregation mechanism to reduce the operating costs of the oracle. Further, we examine the use of Distributed Key Generation (DKG) protocols to generate distributed private keys which are necessary for the creation of the threshold signatures while also preserving the decentralized nature of the blockchain. We provide the system design of a voting-based blockchain interoperability oracle that makes use of the aforementioned concepts and enables clients to verify that a transaction is included in another blockchain. Further, we deliver a prototypical implementation and show the applicability of our proposed solution by conducting a security and cost analysis.

The remainder of this paper is organized as follows: In Section II, we introduce some basic concepts needed in our approach. Subsequently, we present the design of the oracle in Section III. We follow up with implementation details of the prototype in Section IV and evaluate our solution in Section V. Afterward, we discuss related work in Section VI. Finally, Section VII concludes the paper.

II. BACKGROUND

In the course of this section, we explain the basics of BLS signatures, followed by a short description of Verifiable Secret Sharing (VSS) which leads us to a discussion of DKG protocols.

A. BLS Signatures

The BLS signature scheme [10] is based on elliptic curve pairings ($e : G_1 \times G_2 \rightarrow G_T$) respectively bilinear maps. Using BLS, a public key PK is generated by multiplying the selected private key SK with the generator G of a cyclic group. To create a signature σ one has to hash the message m on the curve $H(m)$. This can be accomplished by using a hashing algorithm like the Secure Hash Algorithm (SHA)-256, whereby the resulting hash is used as the x-coordinate of a point on the curve.

If it is not possible to find such a point using this x-coordinate, it can simply be incremented until a valid point is found. This is an essential difference from other signature schemes in which the hash can be used directly. After that, the signature can be calculated by multiplying the point with the private key. To verify the signature, it comes down to checking the bilinear pairings as can be seen in Eq. 1.

$$e(\sigma, G) = e(H(m), PK) \quad (1)$$

This signature scheme has the advantage that it allows to generate particularly short signatures. Another very important aspect especially for this work is that it is possible to create threshold signatures [11] using a secret sharing scheme (see Section II-B), whereby multiple participants share a distributed private key and t out of n signature shares are required to create a valid signature.

Also, it enables the aggregation of multiple signatures [12], whereby we only need to verify two elliptic curve pairings to verify the aggregate signature. This is particularly interesting in the area of blockchain technology. For example, one could instead of verifying every single signature of the transactions in a block, only verify a single aggregate signature.

B. Verifiable Secret Sharing

With the help of a secret sharing scheme, it is possible to divide a secret S into n shares whereby each party gets a different share of S , but at least t shares need to be known to recover S . Otherwise, it is not possible to get any knowledge about S . These schemes are also known as (t, n) threshold schemes.

One of the first secret sharing schemes has been proposed by Shamir [13] in 1979. Using Shamir's scheme, a dealer picks a polynomial $f(x)$ (see Eq. 2) of degree $t - 1$ whereby the constant term a_0 of the polynomial is equal to S and the coefficients are chosen randomly.

$$f(x) = a_0 + a_1x + \dots + a_{t-1}x^{t-1} \quad (2)$$

The secret can be shared with n different parties by evaluating the polynomial at the positions $x_n = 1..n$ and distributing the values for $f(x_n)$ along with x_n to the respective parties. Since every polynomial of degree $t - 1$ is defined by exactly t points, we can make use of polynomial interpolation (e.g., using the Lagrange interpolation formula) to recover the original polynomial and to evaluate it at position 0 to get the secret.

One problem with this, however, is that Shamir's scheme does not take into account that a dealer could distribute incorrect or inconsistent shares. Furthermore, the shareholders could also return incorrect shares. We cannot assume that the dealer and the shareholders are trustworthy, which is why we need secret sharing schemes that also take this kind of behavior into consideration, as is the case with VSS schemes. By using such schemes, each party can verify that it has received the correct information from the dealer and that shareholders submitted the correct shares. Hereby, we limit ourselves mainly to non-interactive VSS schemes, in which only the dealer sends messages and no communication between the other participants is necessary. This reduces the communication overhead.

One commonly used example of such a scheme is Feldman's VSS [14]. Feldman's VSS is based on Shamir's secret sharing scheme but additionally makes use of homomorphic encryption to commit to the secret and coefficients of the random polynomial. The commitments are broadcast while the shares

are distributed through private channels. Each party can use the commitments to verify the validity of its share, due to the homomorphic property of the used encryption scheme.

A problem with this approach is that the commitment to the secret also leaks information about the secret. This is where Pedersen’s VSS [15], another very well-known scheme, constitutes a more secure solution. Like Feldman’s VSS, it is also based on Shamir’s secret sharing scheme, but it makes use of a different commitment scheme that ensures that no information about the secret is leaked unless one can find a solution to the discrete logarithm problem.

C. Distributed Key Generation

In the previous section, we discussed VSS, whereby such schemes ensure that a dealer distributes the secret correctly and the shareholders cannot provide incorrect shares of the secret. However, the dealer yet has to be trusted as it still knows the secret. This is, among other things, a big concern in the area of threshold cryptography, where we want that only certain subsets of $t \leq n$ participants can jointly encrypt data or create a signature. Since the dealer knows about the distributed private key, it has the opportunity to encrypt data or to create signatures without the consent of at least t out of n participants.

To avoid this, a DKG protocol that allows the generation of distributed private keys without the dependency on any trusted third party can be used. In such protocols, no party owns the private key and the private key is never reconstructed. Many such protocols have been proposed over time [16]–[19], whereby we limit ourselves to the first proposed DKG protocol by Pedersen [16] to describe the basic concept.

In Pedersen’s DKG protocol, every participant acts as a dealer during one of the n parallel executions of Feldman’s VSS scheme to share a randomly picked secret. Each participant publishes its commitments using a broadcast channel, such as a blockchain. After that, every participant sends the signed private shares to the other participants through private channels. When receiving a share, each participant verifies the share by using the previously published commitments. If it is an invalid share, a complaint including the share and the signature is broadcast. Each participant computes its private share by summing up all shares received from the other parties which shared their secret correctly and computes the public key through the published commitments.

III. SYSTEM DESIGN

In this section, we propose the design of a voting-based blockchain interoperability oracle. Initially, we give a brief overview of the basic concept. We follow up with a description of the architecture and finally provide a comprehensive definition of the functionality.

A. Overview

The proposed design for a voting-based blockchain interoperability oracle (see Figure 1) allows clients to verify that a transaction is included in another blockchain. It uses an

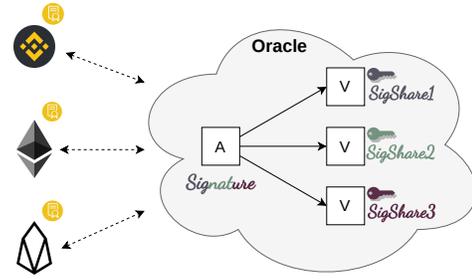


Fig. 1. Overview of the System

off-chain aggregation mechanism (see Section III-E) based on BLS threshold signatures. The oracle nodes are divided into one aggregator and multiple validators which can collectively respond to requests issued by clients (i.e., parties which are interested in the verification of a transaction from another blockchain). For this, the oracle nodes use a DKG protocol to generate a distributed private key (see Section III-D), whereby each node only knows its private key share and the shared public key. Validators obtain the data from the other blockchain and sign it with their private key share, while the aggregator collects the data and the signature shares from the validators to recover the final signature to submit it along with the data to the smart contract. If the aggregator is not able to collect at least the threshold of signature shares with the same result, it cannot generate a valid signature. To ensure reliability, the aggregator changes over time so that each oracle node takes over the task of the aggregator at some point.

As an integral part of the system, we also apply an incentive mechanism (see Section III-F) to encourage oracle nodes to engage as part of the decentralized oracle and to behave honestly. For this, the client must provide compensation for the transaction fees that arise from submitting the result using the oracle contract and also offer two additional rewards. These additional rewards consist of the aggregation reward and the validation reward. Without these rewards, the aggregator, as well as the validators, would have no interest in participating as they would only be providing their resources without getting anything back.

B. Architecture

In our architecture, we can differentiate between the components that are on the blockchain, i.e., the smart contracts, and the components that are off-chain, i.e., the oracle nodes.

For the on-chain components, we make use of three different smart contracts. The first of these is a registry contract, which is responsible for managing all oracle nodes (see Section III-C) and selecting the current aggregator (see Section III-E). Via this smart contract, an oracle node discovers the other oracle nodes and checks if they got selected as the current aggregator.

The next component is the oracle contract, which receives requests from clients and sends them to all oracle nodes. Furthermore, the oracle contract is also responsible for receiving and verifying the responses from the aggregator (see Sec-

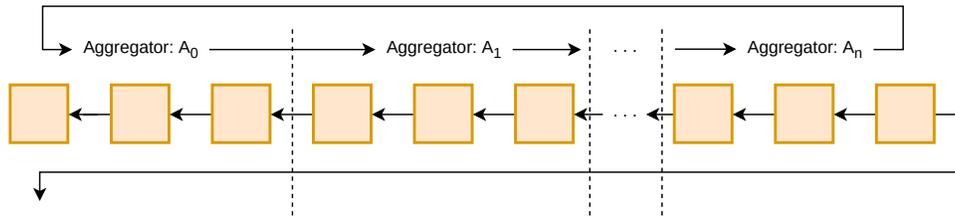


Fig. 2. Aggregator selection mechanism

tion III-E) and transferring the rewards. Further, it also stores all responses and makes them available to the clients.

The last smart contract is the key contract, which is responsible for managing the public key and initiating the generation of new keys (see Section III-D). The off-chain component consists of oracle nodes that can take on the tasks of the aggregator as well as the validator.

C. Oracle Registration

The first step of an oracle node to join the decentralized oracle is to register using the registry contract. During registration, the oracle nodes must provide their host address and BLS public key. Furthermore, a stake must be deposited which is used as a measure against Sybil attacks (see Section V-A3). After an oracle node has registered, it is eligible to take part in the run of the DKG protocol.

Another important point is that oracle nodes must also be able to deregister. Furthermore, oracle nodes can also be kicked out through a majority vote if they behave incorrectly and lose their stake.

Particular care is taken to ensure that the system remains fully operational. A distinction is made between the signature threshold and the threshold of qualified validators. The validator threshold is required to ensure that the system continues to work even if several validators fail or are misbehaving. Should the number of validators fall below the validator threshold, a new key generation process is triggered.

D. Distributed Key Generation

Each time a new oracle node registers, the registry contract checks whether a new run of the DKG protocol should be initiated. For this, we set a certain number, which indicates how many new registrations are necessary. Should the number be reached, the registry contract calls the key contract to trigger the start of the DKG protocol. The key contract then broadcasts a generation event containing the threshold which should be used.

On receiving the generation event, oracle nodes need to wait a certain amount of time (e.g., measured in blocks) to ensure that every oracle node had a chance to receive the event since it can take some time for a block to get propagated through the network. Then, the oracle nodes execute the DKG protocol with all currently registered oracle nodes. In our approach, we make use of Pedersen's DKG protocol (see Section II-C). However, other DKG protocols could also be used, as long as

the run of the protocol is made transparent through the usage of blockchain technology.

The entire run of the DKG protocol takes place off-chain, which means that we need to get the generated public key as well as the number of qualified validators into the blockchain, i.e., we already have the oracle problem in our proposed solution. One possibility of getting the public key into the blockchain is to use an on-chain aggregation mechanism. However, this approach has the disadvantage that it gets more expensive the higher the number of oracle nodes becomes. These costs could be neglected if the frequency of generating new keys is very low. The same applies to the possibility of using a dispute mechanism, which is used by our prototype whereby only one oracle node submits the shared public key and the other oracle nodes can dispute the key. Finally, one could also assume that several other oracles already exist that can be used for this task. These could be oracles that may be pursuing a completely different approach, or it is already another instance of the proposed oracle solution. With the latter, however, we have a bootstrapping problem where the first instance cannot call an existing oracle. Therefore, one would have to use one of the first two approaches or a centralized solution and change it later on.

E. Off-chain Aggregation

The task of aggregating the results and the signature shares is taken over by an aggregator, which is one of the oracle nodes. The aggregator changes in a cycle of n blocks based on a round-robin mechanism (see Figure 2). This has several advantages over an approach in which anyone can aggregate and submit the voting results: It prevents multiple simultaneous submissions of which only the first one would be successful, while all the other oracle nodes which also try to submit an aggregation result have to pay the costs for the failed transaction without getting compensated. This would lead to the problem that oracle nodes are not incentivized to submit a result for which it is not certain whether the submission would be successful or not. Since this is relatively difficult to determine, nodes could become reluctant to act as an aggregator.

Another benefit is that this approach also consumes less bandwidth since not all oracle nodes try to aggregate a result, which reduces the number of exchanged messages. The fact that the aggregator changes over time, still ensures that if an

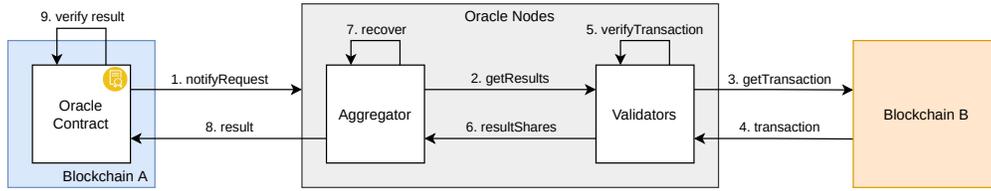


Fig. 3. Aggregation mechanism

aggregator should fail, the next aggregator will take its place and normal operation can be continued.

The aggregation mechanism (see Figure 3) starts when a client sends a request to the oracle contract. After that, all oracle nodes are notified about the request (Step 1). Subsequently, the aggregator begins to collect results from the validators (Step 2), whereby the request only needs to contain the request number so that the validators know which request should be fulfilled. On receiving a request from the aggregator, a validator retrieves the transaction from the target blockchain (Step 3 and 4), verifies the transaction (Step 5), and returns the response consisting of the response to the request and the signature share (Step 6).

The aggregator collects results until it has at least threshold t identical results with valid signature shares. If the aggregator does not receive at least t results with valid signature shares, it will not be able to recover the signature. This can happen, as there can be temporary inconsistencies in blockchain systems, validators fail or behave incorrectly.

Regarding the first problem, however, one can assume that consistency will be achieved at some point (depending on the source blockchain) and that the threshold of the validators will agree. This is one of the advantages that is given by the fact that this oracle is limited to providing data from other blockchains. In this case, the aggregator can try again after a while, or if the aggregator changes in the meantime, the new aggregator will take over the request and repeat the voting.

If the aggregator has received enough shares, it recovers the full BLS signature using Lagrange interpolation (Step 7). After recovering the full BLS signature, the aggregator can submit the result to the oracle contract by providing the result and the signature (Step 8). The oracle contract then hashes the result to a point on the curve and checks the elliptic curve pairing (Step 9). If the elliptic curve pairing is successful, the smart contract calculates the reward (see Section III-F) and transfers it to the aggregator. Finally, the oracle contract notifies the client that the result is available. Here, however, one could also pursue a different approach in which the client also defines a callback function which is to be called when the result gets submitted. In this case, other problems would have to be considered, such as the costs caused by the callback function. Furthermore, the result is then not easy to obtain for other clients.

F. Incentive Mechanism

Since the aggregator only submits a response including the data and the threshold signature, the oracle contract does not

have any information about the validators who were contacted, which makes it difficult to reward validators directly. We cannot rely on the aggregator to provide us with this information either, as it cannot be verified whether it is correct, as the entire DKG protocol is carried out off-chain. Furthermore, it is not particularly practical to reward each validator individually, as this would result in very high fees for the client, not only because the client has to pay each validator individually but also because each transfer causes additional costs during the execution of the smart contract.

Therefore, we propose to only reward the aggregator for submitting the result. The aggregator gets compensated for the transaction costs, receives an aggregation reward, and additionally has the chance to win the validation reward with a certain probability. The probability is scaled super-linearly based on the deposited stake to encourage the creation of only one identity [20]. If the aggregator is not lucky and does not win the validation reward, the reward is maintained by the oracle contract until an aggregator gets lucky enough to win the accumulated validation rewards.

It is particularly important to ensure that the validators are not able to predict whether the aggregator will win the validation reward. Otherwise, validators would benefit from not providing the aggregator with an answer to be able to increase their own chances of winning when they become aggregators themselves (see Section III-E). Therefore, we leverage the unpredictable randomness provided by the signature which is recovered by the aggregator to decide whether the aggregator receives the validation reward. In the case of the aggregator, it does not matter whether it knows if it will receive the validation reward before submitting, since the aggregator is still encouraged to submit the result, as to at least receive the aggregation reward. As for the validators, they are still incentivized to answer the current aggregator with the assumption that it will not receive the reward and thus increase their winnable reward when it is their turn to be an aggregator.

Since a round-robin mechanism is used for aggregator selection, care should be taken that the chance of winning the validation reward is not too high. Validators who were recently selected as the aggregator have a reason to assume that another aggregator will win the validation reward during the time they will have to wait to be selected again. They may be tempted to stop validating until they believe otherwise. Therefore, the chance should be small enough such that the validators can still assume that no one will win the reward in the current aggregation round.

IV. IMPLEMENTATION

After defining the system design of our solution, we created a prototypical implementation. Our prototype enables Ethereum-based blockchains to exchange data with each other. However, the solution can be implemented to work with other blockchains. For this, the target blockchain needs to have smart contract capabilities and enable elliptic curve pairing checks. In this section, we discuss the used technologies as well as the implementation of the smart contracts and the oracle node which is available as open-source software on GitHub¹.

A. Smart Contracts

For the implementation of the prototype, we decided in favor of Ethereum because it is one of the most popular second-generation blockchains and accordingly a wide range of tools is available. Another important factor was that Ethereum already provides a precompiled contract that enables elliptic curve pairing checks on the `alt_bn128` curve.

The registry contract stores all oracle nodes in an iterable mapping and provides clients with the necessary functions to retrieve them. For the aggregator selection mechanism, we defined a threshold of six blocks after which the aggregator will be switched. Further, after every third registration, the registry contract calls the key contract to trigger a new execution of the DKG protocol.

On receiving the call from the registry contract, the key contract calculates the threshold based on the number of currently registered oracle nodes and emits the key generation event. For the threshold, we defined that the majority of all registered nodes is necessary to produce a valid signature. For submitting the key, we provide a function in the prototype via which the public key can be set by one of the oracle nodes. The public key can be disputed if the majority of oracle nodes agree.

When implementing the oracle contract, we especially paid attention to implementing the contract in a gas-efficient way. Since storage operations are among the most expensive, we have tried to keep the number of these low. If a client sends a request to the oracle contract, the request is only emitted as an event and not saved in storage. This is possible because only oracle nodes need to be able to read the requests. However, this approach involves more work for the oracle nodes since they have to filter the blockchain for past events in case they missed some of them.

As has already been mentioned before, we use a precompiled contract that allows pairing checks for the `alt_bn128` curve to verify the BLS signatures submitted by the aggregator. These precompiled contracts are already existing contracts that run outside the EVM and perform more complex tasks. One of the advantages of these contracts is that they are usually cheaper. Further, we make use of the try and increment approach (see Section II) to hash the response on the curve.

B. Oracle Node

The oracle node is implemented in the Go programming language. For the implementation, we used the advanced crypto library `Kyber`². We adapted the library to work with the `alt_bn128` curve used by Ethereum. The library provides the necessary packages for threshold BLS signatures and the implementation of a DKG protocol which is based on the protocol proposed by Pedersen (see Section II-C). For the DKG protocol, we needed a broadcast channel, for which we opted for the IOTA tangle [21], as it enables fee-less and publicly verifiable data exchange. Other broadcast channels (e.g., Ethereum) can also be used, but they may incur additional costs. The oracle nodes use a Go client to create transactions and send them to an IOTA node. These are zero-value transactions that are only used to exchange messages which are necessary to execute the DKG protocol.

Furthermore, the oracle nodes must be able to communicate directly with one another. The aggregator must be able to collect the results from the validators and all oracle nodes need a private channel to each other to distribute the private shares during the execution of the DKG protocol. Therefore, we make use of gRPC Remote Procedure Calls (gRPC) to connect the oracle nodes.

To interact with the smart contracts and retrieve data from an Ethereum blockchain, the oracle nodes need access to an Ethereum node whereby we make use of the Go Ethereum client to connect to the Remote Procedure Call (RPC) server. Further, we use a Go binding generator to create the counterparts of the smart contracts in Go to produce as little boilerplate code as possible.

V. EVALUATION

In this section, we analyze the security and the costs of the proposed solution. This provides insight into if the solution is applicable, what problems can arise and what needs to be considered.

A. Security Analysis

To analyze the security of the oracle, we look at various attack scenarios and the consequences that can result from them. In particular, we are looking at lazy voting, free loading, Sybil attacks, and the key submission.

We can categorize the oracle nodes based on the Byzantine-Altruistic-Rational (BAR) model proposed in [22], which has already found application in other works for security analysis in the area of blockchain technology, e.g., [23]: *Rational* oracle nodes deviate from the protocol as long as they will be able to increase their benefits by doing so. *Byzantine* oracle nodes, however, can unexpectedly deviate from the protocol for unknown reasons, whereby it does not matter if it is intentional or unintentional misbehavior and whether they gain a benefit from it. *Altruistic* oracle nodes will always adhere to the protocol no matter if the rational choice would provide them with additional benefits.

¹<https://github.com/pantos-io/ioporacle>

²<https://github.com/dedis/kyber>

The proposed oracle solution applies an incentive mechanism (see Section III-F) to encourage all oracle nodes to follow the protocol. However, it should be mentioned that even if all incentives are aligned properly, rational oracle nodes may deviate from the protocol. The reason for this is that also actors outside the oracle have to be taken into account. Hence, it may seem rational for oracle nodes to follow the protocol solely based on the applied incentive mechanism, but external factors can influence their decision by providing better benefits for arbitrary reasons.

1) *Lazy Voting*: In the lazy voting problem, rational oracle nodes do not deliver the correct result but always provide the same response to maximize their benefits. If e.g., it is the case that a certain result occurs particularly often, it can be more beneficial to always return the same result directly instead of executing the request. This could lead to an incorrect result being successfully submitted. In our proposed solution, this would mean that a lazy validator would always respond to the request “Is transaction tx included in blockchain B and confirmed by at least n blocks?” with *true*. Since it is more important for many use cases that a certain transaction is included and confirmed, it can be assumed that requests will be answered more frequently with *true* than with *false*. However, this problem can be circumvented by modifying the request. We expand the usual request and ask “In which block on blockchain B is transaction tx included and is it confirmed by at least n blocks?” instead. This question forces lazy validators to read from blockchain B since otherwise, they cannot know in which block the transaction is contained. The lazy voting problem is also discussed in related work (see Section VI-C) about decentralized oracles.

2) *Free Loading*: For the creation of a valid signature, the aggregator only needs to collect as many valid signature shares as the threshold t specifies. This means that in the best case only t validators have to execute the request. However, all other validators who have not contributed to the creation of the signature also have the chance to win the validation reward, when selected as an aggregator.

The problem here is that some validators may never respond for the aforementioned reason. Even though rational oracle nodes may be able to increase their benefits by not responding, they should still be encouraged to respond. The reason for this is that they minimize the risk of the aggregator not being able to get enough responses, which leads to a lower possible validation reward. Even if more than t validators behave altruistically, it may be the case that no signature can be created due to possible inconsistencies. This indicates that it is more beneficial for the validators to always respond to requests. Above all, validators do not have to perform any particularly resource-intensive tasks, which means that possible resource savings are low.

3) *Sybil Attacks*: Another important aspect that must be considered is to what extent Sybil attacks are possible and what the consequences are. A Sybil attack describes the threat that single faulty participants can control multiple identities, which enables them to compromise larger parts of a system.

Douceur [24] has shown that it is not possible to prevent such attacks without a central authority except for conditions that are not practicable for large-scale distributed systems. Since we are in the area of blockchain technology where we usually do not have a central authority, we have to consider such attacks. Further, in permissionless blockchains, everyone is allowed to join the system, whereby it is an easy task to create new identities by simply generating a new key pair. The same applies to the proposed oracle solution where any number of oracle nodes can register.

We have already mentioned in Section III that oracle nodes have to deposit a stake to make Sybil attacks more difficult. The intention is to make the creation of new identities expensive so that it becomes more difficult to control larger parts of the oracle. On the other hand, we also make it more difficult for honest oracle nodes to join the oracle. Therefore, the trade-off for a more Sybil-resistant oracle is less decentralization considering only oracle nodes that can provide the stake can participate. Nevertheless, an attacker is still able to register multiple oracle nodes to be able to create more signature shares, which enables the attacker to gain more voting power.

In the worst case, an attacker could register threshold t or more oracle nodes and decide on the result alone if it can provide the necessary stake. To provide better Sybil resistance, we further encourage the creation of only one identity, by increasing the chance of winning the validation reward super-linearly based on the deposited stake. As a result, it is more beneficial for oracle nodes to create a single identity with a higher stake than to split the stake between multiple identities, as this gives them a greater chance of winning the validation reward. Therefore, if one is only interested in gaining more benefits, this approach resembles the rational choice.

4) *Key Submission*: The selected key submission mechanism also imposes some security risks. As has already been discussed, the public key can be submitted by using an on-chain aggregation mechanism, dispute mechanism, or another oracle solution for the key submission (see Section III-D). While the use of an oracle solution appears to be more cost-effective, we must be aware of the security risks that arise from this approach.

A central oracle solution represents a single point of failure, which can submit arbitrary public keys to then be able to create valid signatures. As a result, a single participant could gain full control over the oracle. However, if one uses a decentralized approach, the risk is lower.

It must be ensured that both oracles, i.e., the oracle used for the key submission and the interoperability oracle, are independent of each other as possible. Otherwise, oracle nodes that participate in both oracles would be able to change the result in their favor. In the worst case, this would mean that a certain subset of oracle nodes could have complete control over the oracle used for key submission and can thus select the public key. This means that even if the attacker is unable to create more than threshold t oracle nodes in the interoperability oracle, it can get control over the interoperability oracle if it can control the oracle for key submission.

B. Cost Analysis

We analyze the costs of the proposed solution by comparing the implemented prototype with two alternative approaches which do not make use of BLS threshold signatures. Accordingly, we implement two additional oracle contracts (*On-chain Oracle*, *ECDSA Oracle*) which follow different aggregation mechanisms. Furthermore, we compare the costs of our approach with the costs incurred by ETH Relay [23], a novel relay scheme (see Section VI-C), to examine how well our approach performs compared to state-of-the-art schemes. To ensure repeatability, the implementations, as well as the evaluation scripts, are also included in the open-source project on GitHub (see Section IV).

The *On-chain Oracle* implements an on-chain aggregation mechanism whereas each oracle node calls the oracle contract to submit a result. The *ECDSA Oracle* makes use of Elliptic Curve Digital Signature Algorithm (ECDSA) signatures to verify the result. In contrast to our proposed scheme, an aggregator does not submit a single BLS signature but rather submits several ECDSA signatures that are verified by the oracle contract. Since there is no reasonable source of randomness for either of these variants, the reward is paid out to each oracle node that is part of the majority.

For the experiment, we use a private Ethereum blockchain based on the Muir Glacier hard fork, on which we deploy all smart contracts. In this experiment, we request the verification of a transaction with every type of the aforementioned aggregation mechanisms. Besides, we are changing the number of participating oracle nodes to be able to determine how the costs develop with an increasing amount of oracle nodes.

By comparing the different mechanisms (see Figure 4), it can be seen that the costs for the on-chain aggregation mechanism are considerably higher than those of the other two. This is due to the reason that for the on-chain mechanism more storage space is needed and each participant has to create a transaction to submit its vote. In comparison, the other mechanism in which an aggregator only submits several ECDSA signatures, is more cost-efficient. The problem here, however, is that the costs continue to rise with the number of oracle nodes, even though the verification of ECDSA signatures is a relatively cheap operation on the Ethereum platform.

Our proposed solution based on BLS threshold signatures, on the other hand, causes almost constant costs that are independent of the number of oracle nodes, since only one signature has to be verified. The costs are only almost constant as the try and increment approach is used for hashing. As can be seen in Figure 5, submitting the result consumes on average 257,607 gas with a standard deviation of 21,671 gas. Verifying a BLS signature is an expensive operation, but it is considerably more cost-efficient as the number of oracle nodes increases. With more than three oracle nodes, it is already cheaper than the on-chain mechanism and with more than 15 nodes, it is also cheaper than the ECDSA mechanism. Therefore, by using this approach we can achieve a higher

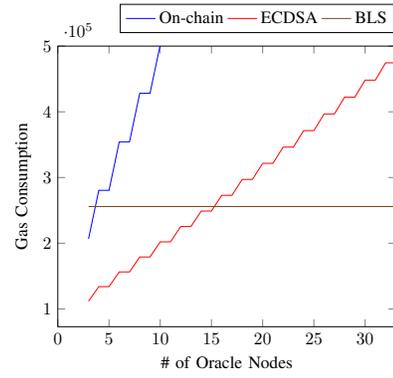


Fig. 4. Gas consumption of the different aggregation mechanisms

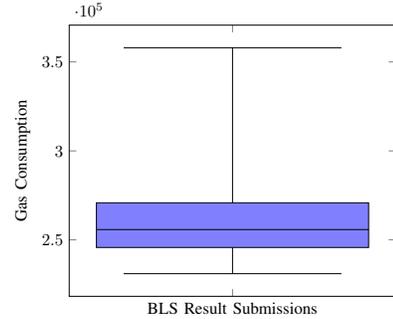


Fig. 5. Gas consumption of the result submission with BLS signatures

degree of decentralization without increasing the costs.

Now that we have compared our approach with two different oracle solutions, we examine how the costs differ compared to a relay solution. To conduct this analysis, we choose ETH Relay because it is an advanced relay solution that is specifically designed to be more cost-effective. For the comparison, we assume a period of 100 blocks. Every block header submission in ETH Relay consumes 284,041 gas with a standard deviation of 3,679 gas. As a result, submitting 100 block headers consumes around 28,404,100 gas. Using our presented approach, we can submit 110 results that incur roughly the same gas costs as the 100 block header submissions of ETH Relay. The costs for the actual request must also be taken into account. In the case of ETH Relay, the relay contract needs to carry out a SPV and check the membership of the block in the longest chain, which means that the costs increase as the search depth increases. In the case of our oracle-based relay, however, a request only ever consists of an event being emitted, whereby the result can simply be accessed directly, which results in constant costs.

At the moment, the application of interoperability solutions is a rather infrequent occurrence, which presumably suggests that not many requests are made. In the worst case, this could mean that not a single transaction has to be verified for 100 blocks. With ETH Relay, the blocks still have to be submitted and thus the costs must be sustained. Hence, keeping the relay alive is a huge burden since these submitted block headers do not yield any profit for the submitter. In contrast, our presented

oracle-based solution does not incur any costs in this scenario, as every request is fulfilled on demand. However, if the number of requests increases drastically such that it is more than 110, ETH Relay would be the more cost-efficient solution. One must also note that in this case further adjustments can be made to the oracle such that every request enables the verification of all the transactions within a block, rather than a single one, by providing the Merkle root of such a block.

VI. RELATED WORK

So far, different blockchain interoperability solutions have been proposed. These include hash-locks, relay solutions, and oracles. In the course of this section, we examine solutions that are related to our work.

A. Hash-Locks

Hash-locks are a well-known technique to enable a basic form of blockchain interoperability without oracles and relays, e.g., to realize atomic cross-chain swaps [25]. Atomic swaps allow multiple parties to exchange their assets across multiple blockchains. The involved parties make use of Hashed Timelock Contracts (HTLCs) to escrow their assets with a hashlock h and timelock t . Ownership of the asset is only transferred if the receiver can provide the secret s such that $h(s) = h$ before t expires. However, attention must be paid to specify the right timelock values and use the correct deployment order of the contracts.

B. Relays

Frauenthaler et al. [23] propose ETH Relay, a novel relay scheme for Ethereum-based blockchains. A validation-on-demand pattern is used to keep the operating costs low by reducing the number of expensive full block header validations. Instead of validating every block header when it is submitted, off-chain clients have a certain time frame in which they can dispute submitted block headers. To make SPVs more efficient, the authors also optimize the traversal of the blockchain, by jumping from branching point to branching point instead of iterating over the whole data structure. While the authors achieve a remarkable cost reduction over traditional relay solutions, the costs remain quite high.

In [26], the authors present zkRelay, which is a relay solution that utilizes off-chain computations to validate batches of block headers through the usage of Zero-Knowledge Succinct Non-Interactive Argument of Knowledge (zkSNARK) proofs. Since these proofs are generated off-chain, the smart contract only needs to be able to verify the proof, removing the necessity of storing and validating every submitted block header, whereby only the last block header of a batch is stored. Although the solution offers improvements in the form of scalability and cost optimization, there are tradeoffs in terms of delay and hardware resources. Our solution, however, can immediately retrieve data from the source blockchain and has no excessive RAM consumption since no complex proofs are generated.

C. Oracles

Provable [27] (formerly known as Oraclize) is a centralized oracle service for various blockchain platforms, e.g., Ethereum and EOS. The Provable blockchain oracle utilizes TLSnotary authenticity proofs [28] to attest the authenticity of the data retrieved from the originating source. Within TLSnotary, an auditee can prove to an auditor the authenticity of information retrieved from a Web server that is using the Hypertext Transfer Protocol Secure (HTTPS) protocol, by utilizing the features of the underlying Transport Layer Security (TLS) protocols 1.0 and 1.1.

With Town Crier, the authors of [29] propose another centralized blockchain oracle, which uses the HTTPS/TLS protocol and additionally utilizes trusted hardware to ensure the authenticity of the data. The implementation uses Intel's Software Guard Extensions (SGX) which allows the execution of a process in a protected address space which guards the process against malicious software running outside of the enclave but also from various hardware attacks. While both of the aforementioned oracle services offer a solution to the oracle problem, the high level of centralization poses a major problem regarding scalability and single points of failure.

In [30], the authors present ChainLink, a decentralized oracle network. ChainLink offers a reputation-based voting system whereby users can issue queries to the ChainLink smart contracts. Queries are executed by the selected oracle nodes which retrieve the results from different or overlapping sets of data sources. These results are aggregated by a smart contract which is also responsible for the calculation of the outcome. Breidenbach et al. [31] further introduce a new off-chain reporting protocol for ChainLink, which however follows a different approach compared to our solution.

Peterson et al. [32] propose a decentralized oracle and prediction market platform called Augur. Within Augur, users can create prediction markets to get information that is external to the system. Market participants trade shares of those markets and reporters can vote by staking their REP tokens (Augur's native token) on one possible outcome. The reached consensus of reporters is considered as the outcome. Depending on the result, reporters receive a reporting fee from the markets. Augur's incentive mechanism encourages participants to behave honestly to maximize their profits, while misbehaving participants get penalized.

The authors of [33] propose ASTRAEA, another decentralized voting-based blockchain oracle. Submitters, voters, and certifiers play a voting game to decide on the truth value of boolean propositions. These propositions are added to the system by submitters, who pay fees to receive an answer to the submitted proposition. Voters play a low-risk/low-reward game by depositing a stake to answer a random proposition. Certifiers on the other hand play a high-risk/high-reward game whereby they can choose a proposition to certify but have to place a high stake.

Merlini et al. [34] propose an extension to this protocol to solve the lazy equilibrium problem, whereby all voters report

the same answer on all propositions. The authors describe a paired-question protocol, in which submitters add queries with two antithetic questions, whereby the oracle additionally needs to check if both answers converge to different outcomes. With this approach, the protocol ensures that honest voters receive higher rewards than lazy voters.

While the approaches discussed above are interesting solutions to the oracle problem, they are not specifically aiming at providing blockchain interoperability and also implement their mechanisms on-chain, which results in high costs.

VII. CONCLUSION

Research on closing the gaps between different blockchains has already led to several concepts and solutions. However, these are usually too expensive or very resource-intensive.

To overcome these issues, we propose a voting-based blockchain interoperability oracle that uses an off-chain aggregation mechanism based on BLS threshold signatures. The oracle nodes are divided into one aggregator and multiple validators and generate a distributed private key to collectively decide on the result of a request. Validators read the data from the other blockchain and sign it with their private key share. The selected aggregator collects the results and the signature shares from the validators to create a valid signature which is submitted to and verified by the oracle contract. Our evaluation shows that the proposed solution is more cost-efficient than other oracle solutions and also incurs lower costs than state-of-the-art relay schemes depending on the request rate.

In future work, we will investigate how we can improve Sybil resistance and the submission of the shared public key. We will also examine if there is still potential to further reduce the costs by applying other signature schemes and enabling requests such that multiple transactions can be verified.

ACKNOWLEDGMENT

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

REFERENCES

- [1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *Decentralized Business Review*, p. 21260, 2008.
- [2] S. Saberi, M. Kouhizadeh, J. Sarkis, and L. Shen, "Blockchain technology and its relationships to sustainable supply chain management," *International Journal of Production Research*, vol. 57, no. 7, pp. 2117–2135, 2019.
- [3] E. J. De Aguiar, B. S. Faiçal, B. Krishnamachari, and J. Ueyama, "A survey of blockchain-based strategies for healthcare," *ACM Computing Surveys*, vol. 53, no. 2, 2020.
- [4] S. Schulte, M. Sigwart, P. Frauenthaler, and M. Borkowski, "Towards blockchain interoperability," in *International Conference on Business Process Management*. Springer, 2019, pp. 3–10.
- [5] R. Belchior, A. Vasconcelos, S. Guerreiro, and M. Correia, "A Survey on Blockchain Interoperability: Past, Present, and Future Trends," *arXiv:2005.14282*, 2020.
- [6] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, 2014.
- [7] H. Al-Breiki, M. H. U. Rehman, K. Salah, and D. Svetinovic, "Trustworthy blockchain oracles: review, comparison, and open research challenges," *IEEE Access*, vol. 8, pp. 85 675–85 685, 2020.

- [8] J. Heiss, J. Eberhardt, and S. Tai, "From oracles to trustworthy data on-chaining systems," in *2019 IEEE International Conference on Blockchain*. IEEE, 2019, pp. 496–503.
- [9] V. Buterin, "Chain interoperability," *R3 Research Paper*, 2016.
- [10] D. Boneh, B. Lynn, and H. Shacham, "Short signatures from the weil pairing," *Journal of Cryptology*, vol. 17, no. 4, pp. 297–319, 2004.
- [11] A. Boldyreva, "Threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme," in *6th International Workshop on Theory and Practice in Public Key Cryptography*. Springer, 2002, pp. 31–46.
- [12] D. Boneh, C. Gentry, B. Lynn, and H. Shacham, "Aggregate and verifiably encrypted signatures from bilinear maps," in *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2003, pp. 416–432.
- [13] A. Shamir, "How to share a secret," *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, 1979.
- [14] P. Feldman, "A practical scheme for non-interactive verifiable secret sharing," in *28th Annual Symposium on Foundations of Computer Science*. IEEE, 1987, pp. 427–438.
- [15] T. P. Pedersen, "Non-interactive and information-theoretic secure verifiable secret sharing," in *Annual international cryptography conference*. Springer, 1991, pp. 129–140.
- [16] —, "A threshold cryptosystem without a trusted party," in *Workshop on the Theory and Application of Cryptographic Techniques*. Springer, 1991, pp. 522–526.
- [17] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin, "Secure distributed key generation for discrete-log based cryptosystems," in *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 1999, pp. 295–310.
- [18] A. Kate and I. Goldberg, "Distributed key generation for the internet," in *2009 29th IEEE International Conference on Distributed Computing Systems*. IEEE, 2009, pp. 119–128.
- [19] E. Kokoris Kogias, D. Malkhi, and A. Spiegelman, "Asynchronous distributed key generation for computationally-secure randomness, consensus, and threshold signatures," in *2020 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2020, pp. 1751–1767.
- [20] Y. Cai, G. Fragkos, E. E. Tsiropoulou, and A. Veneris, "A truth-inducing sybil resistant decentralized blockchain oracle," in *2020 2nd Conference on Blockchain Research Applications for Innovative Networks and Services (BRAINS)*. IEEE, 2020, pp. 128–135.
- [21] S. Popov, "The tangle," *White paper*, 2018.
- [22] A. S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J.-P. Martin, and C. Porth, "Bar fault tolerance for cooperative services," in *Twentieth ACM Symposium on Operating Systems Principles*. ACM, 2005, pp. 45–58.
- [23] P. Frauenthaler, M. Sigwart, C. Spanring, M. Sober, and S. Schulte, "ETH relay: A cost-efficient relay for ethereum-based blockchains," in *2020 IEEE International Conference on Blockchain*. IEEE, 2020, pp. 204–213.
- [24] J. R. Douceur, "The sybil attack," in *International Workshop on Peer-to-Peer Systems*. Springer, 2002, pp. 251–260.
- [25] M. Herlihy, "Atomic cross-chain swaps," in *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*. Association for Computing Machinery, 2018, p. 245–254.
- [26] M. Westerkamp and J. Eberhardt, "zkRelay: Facilitating Sidechains using zkSNARK-based Chain-Relays," in *2020 IEEE European Symposium on Security and Privacy Workshops*. IEEE, 2020, pp. 378–386.
- [27] Provable. The provable blockchain oracle for modern dapps. Accessed: 2021-02-24. [Online]. Available: <https://provable.xyz>
- [28] TLSnotary. (2014) Tlsnotary - a mechanism for independently audited https sessions. Accessed: 2021-03-01. [Online]. Available: <https://tlsnotary.org/TLSNotary.pdf>
- [29] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi, "Town crier: An authenticated data feed for smart contracts," in *2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, p. 270–282.
- [30] S. Ellis, A. Juels, and S. Nazarov. (2017) Chainlink: A decentralized oracle network. Accessed: 2021-02-24. [Online]. Available: <https://link.smartcontract.com/whitepaper>
- [31] L. Breidenbach, C. Cachin, A. Coventry, A. Juels, and A. Miller. (2021) Chainlink off-chain reporting protocol. Accessed: 2021-07-16. [Online]. Available: <https://research.chain.link/ocr/pdf>
- [32] J. Peterson, J. Krug, M. Zoltu, A. K. Williams, and S. Alexander. (2019) Augur: a decentralized oracle and prediction market platform

(v2.0). Accessed: 2021-02-24. [Online]. Available: <https://augur.net/whitepaper.pdf>

- [33] J. Adler, R. Berryhill, A. Veneris, Z. Poulos, N. Veira, and A. Kastania, "Astraea: A decentralized blockchain oracle," in *2018 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)*. IEEE, 2018, pp. 1145–1152.
- [34] M. Merlini, N. Veira, R. Berryhill, and A. Veneris, "On public decentralized ledger oracles via a paired-question protocol," in *2019 IEEE International Conference on Blockchain and Cryptocurrency*. IEEE, 2019, pp. 337–344.