computers

MDPI

Article

Smart Contract Data Feed Framework for Privacy-Preserving Oracle System on Blockchain

Junhoo Park, Hyekjin Kim, Geunyoung Kim and Jaecheol Ryou *

Department of Computer Engineering, Chungnam National University, 99 Daehak-ro, Yuseong-gu, Daejeon 34134, Korea; junhpark@cnu.ac.kr (J.P.); kherootz@o.cnu.ac.kr (H.K.); gykim@cnu.ac.kr (G.K.)
* Correspondence: jcryou@cnu.ac.kr

Abstract: As blockchain-based applications and research such as cryptocurrency increase, an oracle problem to bring external data in the blockchain is emerging. Among the methods to solve the oracle problem, a method of configuring oracle based on TLS, an existing internet infrastructure, has been proposed. However, these methods currently have the disadvantage of not supporting privacy protection for external data, and there are limitations in configuring the process of a smart contract based on external data verification for automation. To solve this problem, we propose a framework consisting of middleware of external source server, data prover, and verification contract. The framework converts the data signed in the web server into a proof that the owner can prove with zk-SNARKs and provides a smart contract that can verify this. Through these procedures, data owners not only protect their privacy by proving themselves, but they can also automate on-chain processing through smart contract verification. For the proposed framework, we create a proof using libsnark for server data and show the performance and cost to verify with Solidity the smart contract language of the Ethereum platform.

Keywords: blockchain; smart contract; Oracle; zk-SNARKs

Citation: Park, J.; Kim, H.; Kim, G.; Ryou, J. Smart Contract Data Feed Framework for Privacy-Preserving Oracle System on Blockchain. *Computers* **2021**, *10*, 7. https://dx.doi.org/10.3390/computers10010007

Received: 30 November 2020
Accepted: 23 December 2020
Published: 28 December 2020

1. Introduction

Smart contract is a program that runs on a blockchain and is used to build various decentralized applications such as games, insurance, and finance. Ethereum, the most widely known smart contract platform [1], can be used to build decentralized applications. Ethereum supports Solidity [2] as a language for writing smart contracts, and Ethereum Virtual Machine (EVM), an execution environment for contracts.

In order to achieve decentralization, the blockchain verifies the validity of transactions by consensus (e.g., Bitcoin PoW [3]). Blockchain consensus is verified only for elements that must be verified by an internal system (e.g., the balance of accounts), and functions implemented in smart contracts guarantee the execution of input data, but do not guarantee the reliability of the data. In particular, when a function that implements a large amount of money transfer through input data is executed, the result derived from the input data becomes very important. However, the security provided by the blockchain or smart contract does not support the authenticity of the source of external data or the integrity of the inflow process.

Accordingly, connectivity with external systems is one of the challenges for the use of smart contracts. Since data generated outside the blockchain is information that cannot be verified inside the blockchain, it is necessary to secure the trust of external data. Oracle is a system that operates to make external data available on the blockchain.

Representatively, decentralized finance (De-Fi) is an area where oracle can be used. In order to implement staking and interest products necessary for decentralized finance, it is necessary to stable coins or tokens that are the basis of the product, and for this, it must be implemented based on current price exchange rate. For example, this is the current dollar exchange rate per price of Bitcoin. As representative websites that provide price per

coin information include CoinMarketCap [4] and exchanges [5], an oracle is required to utilize such information in the blockchain. Several proposals have been made to implement known Oracles.

Voting based schemes MakerDAO [6], one of the decentralized financial platforms, operates a committee to implement the Oracle system and exchanges tokens based on the prices they suggested. Oraclize [7] and Chainlink [8] deliver coin price information to Ethereum while operating nodes. In particular, Chainlink provides Link Tokens as a reward by introducing a reputation system for node operation. An approach such as Astraea [9] presents a game-theoretic approach for decentralized oracles. However, this method of determining data does not provide *authenticity* of the data itself.

TLS based schemes Research has been conducted to construct a protocol based on Transport Layers Security (TLS) [10] to provide cryptographic authentication of external data. There are methods of providing data based on hardware safety [11] and privacy preserving oracle to verify data with zero-knowledge proof [12]. An approach such as those in [11,12] requires a separate infrastructure that oracle can trust or participate as a verifier of data. Therefore, on-chain automation is impossible, and execution of smart contracts may be delayed due to external data feeds. A protocol [13] to support the verification of the server's signature has also been proposed, but it does not support privacy protection operations for user data.

As a solution for the confidentiality and transactional privacy of the blockchain, a method is to provide an anonymous cryptocurrency Zcash [14], Monero [15], or protocol modification [16–18]. This method requires a fork because it is done at the blockchain design stage. There are also methods [19,20] that verify calculations performed on the off-chain on the on-chain. This method should consider the cost for verification on-chain.

We intend to provide an environment in which data owners can directly prove it and smart contracts can verify it in order to implement Oracle on the blockchain. For this, the following requirements must be satisfied:

- Support for authenticity in the blockchain for external data
- Support on-chain automation for external data
- Support user privacy protection for external data

Solidity, the most used smart contract language, is executed deterministically. Smart contracts triggered by transactions are implemented to perform predetermined actions. ZoKrates, a tool that supports verification of off-chain operations generated outside the blockchain, guarantees the privacy of data executed by smart contracts based on zero-knowledge proof. Ethereum smart contract supports Solidity language conversion of verification operations so that operations performed off-chain can be verified. Zkay [21], which expanded the results of individual operations, developed a language that helps developers easily manage verification contracts. Language research like this supports verification of operations performed off-chain, but does not support authentication for external data. Therefore, the proposed framework should include the language function of the smart contract that can verify this.

Among the ways to solve data privacy in smart contracts, a new blockchain infrastructure can be designed. For example, Hawk [17] and Arbitrium [18] rely on a trusted manager or use hardware-based TEE (Trusted Execution Environment) [22] such as Ekiden [23]. Regarding Oracle, Town Crier [11] provides infrastructure based on reliable hardware.

In this paper, we present Ziraffe, a framework for automated smart contract execution on-chain. The main function of the framework is to enable smart contracts to automate off-chain operation verification while supporting data authentication and privacy protection. Ziraffe provides middleware that supports signatures in TLS-based servers, clients that provide verifiable data in smart contracts without compromising authentication by data providers (owners), and interfaces that support easy construction in applications. Table 1 shows the differences between the Ziraffe framework and other solutions.

Table 1. Comparison with the proposed oracle solution.

Supported Features	Voting Based	TLS Based	Ziraffe
Authenticity	×	○	○
On-chain automation	×	×	○
Privacy Protection	×	○	○

The contributions of this paper are as follows:

- Protocol supporting external data authentication: Introduce a signature system necessary to prevent repudiation of the data source required to authenticate externally provided data. Based on this, the blockchain can verify the origin of the data presented by the owner of the data and verify that the correct data has been verified. The data owner (certifier) cannot intermediately manipulate the data received from the data source.
- Proposal of Interface for Developer Convenience: Design for the privacy of the prover provides an oracle interface that can be written in Solidity so that a smart contract developer can verify a proof including authentication of external data to solve a problem difficult to implement.
- Provide an evaluation of the performance and cost of protocols and languages that can be referenced when implementing the proposed framework. The main performance corresponds to the cost of generating proof data and verifying the smart contract implemented in Solidity on Ethereum and the size of the stored data.

The structure of this paper is as follows. Section 2 explains the background information for understanding the proposed framework. Section 3 proposes the purpose and concept of the proposed framework. Section 4 deals with the details of the proposed framework. Section 5 proceeds with security and evaluation of the implementation. Through review of related studies in Section 6, we compare the differences with this study and conclude in Section 7.

2. Background

2.1. Blockchcin and Smart Contract

Blockchain is being used in cryptocurrencies such as Bitcoin. The important point of decentralization is that when participants participate in the protocol in the blockchain, they all assume equal authority and are not controlled by a single specific institution. Decentralization by consensus is not controlled by a specific minority or a single institution, so the blockchain has censorship resistance. Also, since the blockchain has an immutable character as a ledger, it makes it difficult to forge or alter past details. In the blockchain, irreversibility is proportional to the length of the chain accumulated over time, and integrity is ensured by connecting each block with a cryptographic one-way function.

In blockchain, transactions are basically publicly stored to form a shared ledger and verification. Pseudonymity is provided by using the hash value of the public key address, but the movement of funds and the sum of the amount can be tracked. Transaction disclosure changes the state of the smart contract, which is a problem in systems that require anonymity because prior information can be known.

Smart contracts are programs that run on the blockchain, and Ethereum's Solidity is a representative development language. Solidity runs within the Ethereum Virtual Machine (EVM), a Turing-Complete Languages virtual machine. The EVM is executed with the input value passed to the smart contract. In order to prevent Denial of Service (DoS), the concept of gas is introduced. Therefore, a smart contract platform such as Ethereum must present the cost of the operation executed in the blockchain to the transaction along with the fee. On the other hand, operations executed on the off-chain are not related to operations executed on the blockchain, so there is no cost. The method of verifying operations performed off-chain on-chain using verifiable operations is cost effective.

2.2. Oracle with TLS

In the blockchain, smart contract oracles deal with the problem of securing the reliability of external information. When the information is fetched from outside the blockchain, the oracle is responsible for the data feed. TLS can be applied to verify that the information is obtained from the data source. Currently modified TLS 1.3 supports public key cryptography, but since it does not provide a signature function to prevent repudiation of the web server, a separate web server must participate in the blockchain network or the protocol must be modified so that the server signs with a private key. TLS-N [13] constitutes Oracle by adding non-repudiation mechanisms to TLS. TLS-N is configured to be compatible with TLS1.3, and during the TLS handshake process, the requester and generator verify the result of signing the TLS records exchanged with each other on the blockchain and propose a mechanism that cannot deny the process of communicating with each other.

Since the web server does not directly communicate with the blockchain, the data provided must be directly verified by the data owner on the blockchain. To this end, it must be configured so that the user who owns the data can be issued a signed form of his/her data provided on the web and provided it on the blockchain for verification. In order to provide the data format that can be sent and received in the client–server structure in JSON type and to be provided in a verifiable format so that it can be saved as a smart contract stored in the blockchain, the server middleware must be added separately. We have constructed the Oracle framework with this approach.

2.3. zk-SNARKs for Smart Contract

A zero-knowledge proof is a cryptographic protocol constructed in the presence of a verifier to prove and a verifier to verify when a certain fact exists. For example, when applying zero-knowledge proof in the authentication system, it can be configured in a way that proves that the value is known without revealing the password or personal information required for authentication. It can be implemented as a protocol by defining what facts to prove and verify with zero-knowledge proof and what secret values to hide.

Zero-knowledge proof verifies only whether the statement about the fact that the verifier wants to prove is true/false without revealing the secret value in the protocol between the prover and the verifier. When a function given to a prover and a verifier exists, if the verifier generates a proof value by inputting a sentence and a secret value as the input of the function, the verifier must not know any value related to the secret during the verification process. Summarizing these characteristics, the proof of zero knowledge must satisfy the following three conditions:

- Completeness: If a sentence is true, the verifier must be able to understand this fact
- Soundness: If a sentence is false, it shouldn't convince the verifier
- Zero Knowledgeness: During the verification process, the verifier must know nothing but the true and false of the sentence.

NIZK (Non-Interactive Zero-Knowledge) converts the prover's statement to be verifiable and does not leak information other than the truth of the statement, so it is possible to prove personal data. zk-SNARK (zero knowledge Succinct Non-Interactive Argument of Knowledge) [24] does not interact in a non-interactive manner from short proofs. Thus, the prover can persuade the verifier with one message. The prover can use personal information during the creation of the proof, and the verifier has no knowledge of that information.

In off-chain computation, zk-SNARKs can perform calculations after a validator with a weak calculation outsources the calculation to an unreliable validator and return a result with evidence that the result is correct. In particular, in zk-SNARK, since the verification cost is not related to calculation, it is cost-effective in platforms that require calculation costs such as Ethereum. After the introduction of zk-SNARKs, methods such as optimization were also studied [25].

Since zk-SNARK, zero-knowledge proof has been developed from various perspectives. The approaches of these studies complement the initial trusted-setup or support

homomorphic computation through range proofs. However, among the zero-knowledge proof implementations studied so far, zk-SNARKs has the lowest verification complexity and can utilize less on-chain verification cost in the blockchain. Therefore, we designed the framework for zk-SNARKs.

In order to enable smart contracts to verify data outside the blockchain, we utilize the ZoKrates toolbox. ZoKrates uses the Rust implementation of zk-SNARKs' Bellman library. ZoKrates provides a verification contract implemented in Solidity for verification on Ethereum. The language proposed by Zkay utilizes ZoKrates to help developers construct and distribute applications. Since this is supported at the level of the development language, developers can more easily manage transactions that deploy and operate verification contracts. Similar to zkay, Ziraffe framework creates a contract written in Solidity using Zokrates, but another interface is needed to achieve the purpose of verifying external signatures. Currently, Zokrates is the most widely known tool that enables solidity verification within Ethereum, so it was designed to support a new language type for Oracle to verify external data based on Zokrates.

3. Framework Overview

This section describes the composition of the Ziraffe framework. The Ziraffe framework consists of three parts as shown in Figure 1. First, it can help distribute smart contracts on the blockchain. The second is to enable users to access external data sources and transfer the imported values to the blockchain. Finally, when a user fetches data from a data source, it allows the server to verify the external origin by signing the issue. This section will explain each part in detail.

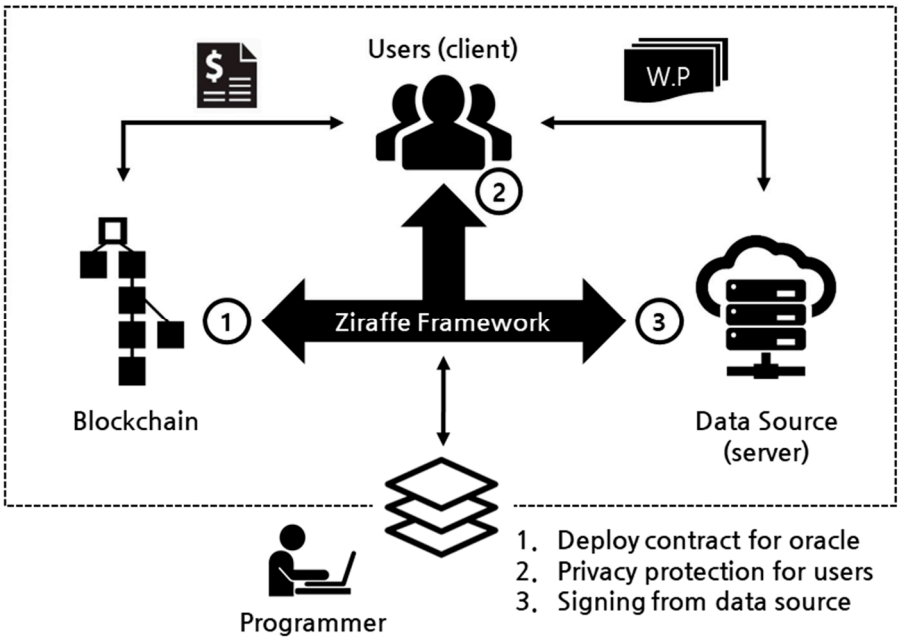

Figure 1. Ziraffe framework overview. This figure is an overview of the Ziraffe framework. Programmers can use the framework to develop and distribute smart contracts. Users can be provided with privacy protection through the framework in their credentials. The server of the data source supports signing through middleware.

3.1. Deploy Contract for Oracle

Smart contract developers write contracts using high-level languages. Various applications have been developed using Solidity, the most widely used language. Solidity can be compiled and used in various platforms such as Klaytn [26], RSK [27], TRON [28], and Hyperledger [29], as well as Ethereum. However, in order to implement Oracle in Solidity, it is

developed in a way that utilizes it after the Oracle infrastructure is implemented, so the Oracle interface to Solidity itself is not supported.

From the developer's point of view, when creating an application that requires Oracle, there is a problem that it must be developed in Solidity and linked to platforms such as Chainlink [8] or Provable [7]. This is because Oracle is not an interface supported by Solidity, but an interface supported by the platform. Therefore, a problem of selecting an Oracle platform operator arises from the developer's point of view. This soon leads to commissioning problems for Oracle costs.

Another issue is the privacy that occurs when external data is imported from smart contracts. Storing external data on the blockchain or submitting it to request verification causes unwanted information disclosure from users. By using a method such as off-chain computation, calculations related to personal information can be performed off-chain, and only verification can be performed with smart contracts. However, in order for a developer to design with this in mind, it is necessary to be able to directly develop a verification contract, and cryptographic knowledge is required. As a result, without a simple interface, it is difficult to solve the requirements for privacy issues.

The purpose of our framework is to allow users acting dependently within the application to verify external data. Therefore, in applications developed using the Ziraffe framework, the user can directly transmit external data if necessary. Developers only need to develop an application using the provided interface. Our solution is to introduce a type for smart contract language and a preprocessor that supports it to support an interface that can replace Oracle for developers.

The role of the preprocessor can be largely divided into two. The first is to support Oracle types that developers can use. Any developer who knows and can develop the Solidity language supports the implementation of Oracle using types. The second is to support a privacy protection type of verification contract. We use systems like ZoKrates to support creating contracts that require Oracle verification. Through the framework, developers can implement verification contracts written in Solidity by using interfaces without knowing about tools.

By utilizing the interface, not only does the developer's contract development become easy, but also the user's privacy can be protected. For this purpose, we implemented the type supported by the Solidity language and applied it to the development and distribution stage of the blockchain. The contract required for proof of external data is compiled by preprocessing the result of implementing the interface, and distributed as bytecode for it.

3.2. Privacy Protection for Users

Data that the user can certify by importing data from an external source may include proof of the balance of the bank or the issued qualification, proof of the details of the current server situation (for example, the current situation for insurance payment). In order for a user to directly provide a source for external data, it must first be possible to verify the presented data. Another important point is that privacy protection must be possible while generating a provable proof.

We use zk-SNARK to construct this framework. The proof generated in zk-SNARKs is included in the blockchain transaction, propagated, and configured to be verifiable on the blockchain. In the blockchain, it is possible to verify the origin of data and necessary facts from the transaction submitted by the user. At this time, the details of the user's personal information are not revealed.

Zk-SNARKs need a trusted setup for initial configuration. Therefore, in the smart contract development stage, the transaction fields that the user intends to present and the data provision details of the server must be defined in advance. When the fact to be proved is confirmed, the contract is distributed, and the user can prove the fact by submitting the result of the signed data received from the server.

3.3. Signing from Data Source

In the Ziraffe framework, the signature of the data source is essential. The ECDH protocol used in the TLS protocol is used for key exchange for HTTSP but does not provide signed data, so it does not provide users with non-repudiation of data sources. Therefore, in order to authenticate that the user's data came from the correct source, it is necessary to authenticate the source of the data. For this, we configured the framework to sign the server using the cryptographic key exchanged with ECDH.

In order for the server to sign data details to the user, data exchange of a specified standard is required between the user and the server. In order to generate proof of the data that the user wants to prove with zk-SNARK, parameters for input data required during the proof and verification process must be defined. We do this by putting middleware on the server and configuring it to communicate with the user.

The server's middleware communicates with the user, not with the blockchain. Therefore, there is no need to directly connect to the blockchain or register an account on the blockchain. The server is only responsible for communicating with the user to sign and return the data the user needs. This is an extension of the HTTPS protocol. However, in order to verify the result of the data signed by the server with the private key, the developer must be able to register the smart contract public key in the application development stage.

When the middleware issues signed data, the user takes the signed data as input and creates a proof that does not reveal the secret value. In the contract for verifying the proof, the user's public key is used to verify that the proof submitted by the user comes from the correct data source, and thus the accuracy of the fact to be proved is guaranteed.

4. Implementation

Building an application using the Ziraffe framework consists of the following procedures. Developers create contracts using Solidity language, and use the framework's API to create Oracle functions for data requested by the application. In addition, the application registers the public key of the source corresponding to the data source. The developer compiles the created contract and distributes it to the blockchain network. When running an application and verifying external data such as user credentials, the user requests the data source to perform HTTPS communication. The data source responds to the user by signing the data requested by the user in the middleware. The user generates a proof that can be verified by calculating the data received from the data source with zk-SNARKs. The user sends the transaction including the proof to the Oracle contract. Oracle contract verifies the submitted proof to verify the user's credentials.

4.1. Pre-Processing of Smart Contracts

In order to program a contract to fetch external information, we provide a Solidity-based contract interface. The process that is processed to convert to Solidity contract is shown in Figure 2, and shows the process of preprocessing and compiling the ZIRF file written in Solidity using the interface provided by Ziraffe framework. ZIRF is translated through Lexer, Parser, and Rewritter for the written language. Here, Lexer converts keywords (URL, JSON, function) or operators specified in Ziraffe and existing solidity into tokens. The parser creates an Abstract Syntax Tree with tokens according to Ziraffe's grammar rules. After that, Rewritter traverses the AST tree, checks the type, and rewrites the code.

The format of data fetched from an external source from the user's client is composed of JSON as shown in Figure 3a,b. JSON means a type to specify the value to be received as a result of an https request. It can be used in the same way as a structure, and the hash function is calculated according to the order of the values inside when verifying the signature. It is stored as a struct in Ethereum. Here, in the case of the format shown in Figure 3a, the Hash (balance | dataOfInquiry) value is saved.

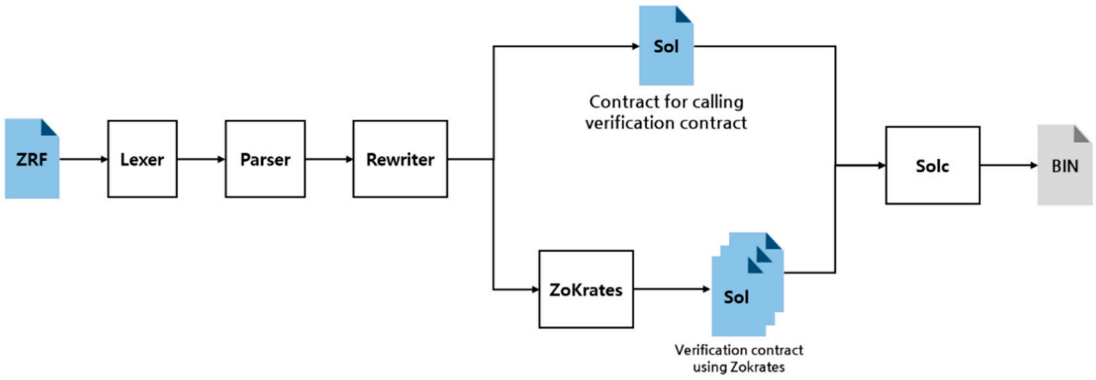

Figure 2. This figure shows the pretreatment steps of the Ziraffe framework. The ZRF file created by the developer is converted to Solidity code through pre-processing. After that, it is stored in the blockchain as a bytecode through a compilation process.

(a) JSON type Interface

```
JSON Bank {
    uint balance;
    uint dateOfInquiry;
}
```

(b) URL type Interface

```
struct URL {
    string url;
    bytes32 pk;
}
```

(c) An example of get<Bank> request

```
{
  "message": {
      "balance": 10000,
      "dateOfInquiry": 20201111
    },
    "signature": 0x…
}
```

(d) An example of ziraffe Interface based ZRF contract code

```
contract Bank {
    mapping(address => Bank) bank_balance;
    URL url;

    constructor(URL _url) {
    url = _url;
    }

    function save_date_of_inquiry(ReqParam string id, ReqParam password) public {
        Bank balance = get<Bank>(url, {id, password});
        bank_balance[msg.sender] = balance;
    }

    function check_balance(ReqParam string id, ReqParam password, uint limit) public returns (bool) {
        Bank balance = get<Bank>(url, {id, password});
        return balance.balance > limit;
    }
…
}
```

Figure 3. This figure is an example of the Ziraffe language used for smart contract development. (**a**) is an example of the JSON type used to prove the balance in the case of banks, and (**b**) shows the specification of the URL of the data source. (**c**) is the data format including signature. (**d**) shows an example of a contract created using the interface supported by the Ziraffe framework before pre-processing.

Randomly added type and variable JSON and URL are changed to Solidity type and variable. JSON and URL are each converted to a struct used in Ethereum. The ReqParam variable is expressed in ZRF, but in the rewritten Solidity contract, it is removed so that the value does not increase in the on-chain. For newly added functions, get and post, a value is requested in the URL, and the requested value is converted into a contract that can verify signature verification into zero-knowledge proof.

When creating a contract for verification, it is created based on ZoKrates. Verification contract receives proof as an input and can prove without revealing the value (e.g., balance) of the entered information. The URL type defines the URL string value and the public key pk required to verify the signature of the value transmitted from the URL, and is stored in Ethereum in the form of a struct. ReqParm receives a string value as a value specifying the parameter used in the https request. The parameters used in the request are only used for https requests and are not stored in the blockchain. When requested,

it enters the form as {id, password} as the second argument of the function that makes the https request.

Get<Bank> is a function that sends a get request to the URL received as the first argument. It receives the JSON value received in the form of a bank and verifies the signature with the pk corresponding to the URL. The request function supports get and post among https requests. When requesting get<Bank>, the JSON value of the request result is shown as in Figure 3c. The message value is used as the bank value, and the signature is used by reading the signature value. An example for developing a contract to prove the balance of a bank that has implemented such a function is shown in Figure 3d.

4.2. Middleware

TLS 1.3 exchanges keys to be used for https via ECDHE. It is possible to check whether the currently connected server is an authenticated server by using the server's certificate received through key exchange. After that, the server performs encrypted communication to request data and receive a response. However, since the TLS 1.3 protocol does not support application level signing in the server, there is a problem that the client cannot prove to a third party that the data is issued by the server. Ziraffe is configured to support signature by applying middleware that can be used in the server.

After normal key exchange, the server responds to the client's data request. The middleware responds with the JSON standard defined for the data the server responds, and is configured to respond by signing with the server's private key. As shown in Figure 3c, it adds the server signature according to the JSON requested by the client. The signed data may include a data value (e.g., balance) to be responded, a timestamp, a Blocknumber, a user's address, and an ID value used for server authentication.

The role of the server is simply to sign the issued data to prevent non-repudiation. Therefore, when developing using the Ziraffe framework, the programmer must define the specification between the user (client) and the external source (server) at the stage of application construction. The specified standard is required during the setup process in zk-SNARK.

4.3. Proof Generation

The user creates a Proof that can be verified with a contract as a prover to prove data along with the role of a client communicating with an external source. For example, a prover can submit "How much I have" to the blockchain to verify it, and create a proof so that the amount is not revealed.

Since zk-SNARKs require an initial trusted setup, the developer defines the facts that the user should prove in advance and defines the operation while simultaneously distributing JSON and a verifiable contract for this. The main contents of the operation are (1) verification of the pk corresponding to the source and URL of the server, (2) verification of the signature of data issued from the server, and (3) verification of the private data value among JSON values without revealing it.

The generated proof composes and submits a transaction that calls the verification contract and submits the input value used in proof creation and the server's signature together. In verification contract, the proof operation is verified, and the operation result is used to determine true/false.

5. Evaluation

We evaluate the implementation of the Ziraffe framework described earlier. In the blockchain, smart contract platforms such as Ethereum have to pay the cost of operations. In addition, if you write values to storage data that is permanently stored in Ethereum, you have to pay for it. On-chain operations require cost, whereas off-chain operations do not require cost, so the main performance can be measured in time. Therefore, the main performance indicators are the time to generate a proof in the off-chain with zk-SNARKs

from the server issued data, the gas cost consumed to verify the proof in the contract deployed on Ethereum, and the size required to store the proof.

The Ethereum network used for the evaluation was based on the Rapsten Test network, and the client that generated the proof in off-chain operation has an Intel Core i7-8850 CPU @ 2.60 GHz and 16 GB RAM. The data used to generate the proof can vary depending on how many times the Hash value required to fix the length of JSON is executed. Therefore, the test was conducted for the proof that the SHA-256 hash function was executed and the signature was verified. ALT_BN1208 EdDSA was used to generate the signature.

Table 2 shows the result of JSON input targeting JSON input using the Ziraffe framework, and the hashed value of the JSON data is signed, and the time to generate the proof is 12,413 msec. At this time, 256,112 gas is displayed as the required value to verify the proof. The value stored in the storage of the contract is 256 bytes, and in the case of zk-SNRAKs, since the complexity of verification cost is constant, it can be seen that the value reflects only the price of the network state regardless of the proof creation time.

Table 2. Proof generation and verification results of external data using zk-SNARK in Ziraffe framework.

Key Value	Proving Time (ms)	Verification Cost (gas)	Proof Size (bytes)
Signature with 1 hash function	5354	246,640	256
Signature with 2 hash function	12,413	258,112	256
Signature with 3 hash function	25,125	258,176	256

6. Related Work

The Ziraffe framework aims to support an environment for Oracle contract programmers to easily build Oracle. This section explains the differences between the existing research and the proposed framework.

Blockchain and Privacy Since a decentralized block chain verifies and stores the validity of transactions, privacy issues can arise. As a solution to this, there are studies applying zero-knowledge proof to payment such as Zcash [14] or Monero [15] to ensure the anonymity of transactions. However, these studies do not support data source authentication because they are designed for verification without revealing the facts that can be verified on-chain.

Oracle with TLS There are studies that attempt to vote in a decentralized environment to implement Oracle, but it does not support the authentication process for users to bring their own data. On the other hand, Town Crier [11], TLS-N [13], and Deco [12] support data authentication by presenting a protocol for connecting TLS to the blockchain. In particular, in the case of Deco, in TLS 1.3 version, the protocol was configured so that the attestor can verify to the oracle (verifier) through a third-party handshake method without modifying the server for authentication to external sources. In this study, Oracle is configured to operate without a separate operation, eliminating platform dependence and supporting a framework for developers to develop such environments.

Zero knowledge Proof for Smart Contract In order to apply zero knowledge proof to smart contract, Hawk constructed a protocol using manager. ZoKrates [20] and Zkay [21] proposed a language that supports developers to implement off-chain operations and verify them with contracts on-chain. The Ziraffe framework proposes a framework that can solve Oracle problems by using ZoKrates to authenticate external data sources and provide interfaces to them.

7. Conclusions

In this paper, we showed that the Ziraffe framework proposed by us can support authentication of origin for external data and protection of user privacy. In addition, it is configured to support the interface so that the programmer can easily implement the contract for verification. The proposed process makes it easy to implement zk-SNARKs verification when building applications that require data from external sources, and can bring usefulness to the development stage. Reflecting this, developers can build the necessary environment according to the requirements of the application, and users are guaranteed to hide private data when submitting data to the blockchain. We tested this only on Ethereum, but now Solidity is being used on smart contract platforms such as Klaytn, RSK, and Hyperledger. Therefore, we believe that our solution will be practical, as it can be enabled in Solidity to build applications in a decentralized environment.

Currently, we have proposed a method based on the ZoKrates tool that utilizes zk-SNARKs, but the zero-knowledge proof scheme is developing. We use zk-SNARKs, which are cost-effective due to low verification complexity, but they have a disadvantage that they require initial trusted-setup and an additional study of interfaces for applying zero-knowledge proofs may be required for various applications with homomorphic computation support. Further, since the middleware of the server is operated by a third party separately from the blockchain, problems such as censorship resistance or abusing may occur. This part is expected to be better researched if data sources are diversified and implemented in a trusted environment such as Intel SGX. We will develop the framework in the future, including these researches.

Author Contributions: J.P. and J.R. contributed ideas and manuscripts. H.K. and G.K. supplemented the manuscript and participated in its implementation. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (2019-0-00108, Developing Oracle Technology to connect Blockchain Smart Contracts to the authenticity proof external data).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Buterin, V. Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform. 2014. Available online: https://blockchainlab.com/pdf/Ethereum_white_paper-a_next_generation_smart_contract_and_decentralized_application_platform-vitalik-buterin.pdf (accessed on 30 November 2020).
2. Ethereum Foundation. 2018. Solidity Documentation. Available online: https://docs.soliditylang.org/en/v0.4.24/ (accessed on 30 November 2020).
3. Nakamoto, S. Bitcoin: A Peer-to-peer Electronic Cash System; White Paper; 2008. Available online: https://git.dhimmel.com/bitcoin-whitepaper/ (accessed on 30 November 2020).
4. CointMarketCap. 2020. Available online: https://coinmarketcap.com/ (accessed on 30 November 2020).
5. Binance. 2017. Available online: https://www.binance.com/ (accessed on 30 November 2020).
6. The Maker Procotocol: MakerDAO' Multi-Collateral Dai (MCD) System. 2019. Available online: https://makerdao.com/en/whitepaper/ (accessed on 30 November 2020).
7. Bernani, T. Oraclize; London, UK. 2016. Available online: http://www.oraclize.it (accessed on 30 November 2020).
8. Ellis, A.S.; Juels, S.N. Chainlink: A Decentralized Oracle Network. *Retrieved March* **2017**, *11*, 2018.
9. Adler, J.; Berryhill, R.; Veneris, A.; Poulos, Z.; Veira, N.; Kastania, A. Astraea: A decentralized blockchain oracle. In Proceedings of the 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), Halifax, NS, Canada, 30 July–3 August 2018; pp. 1145–1152.
10. The Transport Layer Security (TLS) Protocol Version 1.3 RFC8446. Available online: https://tools.ietf.org/html/rfc8446 (accessed on 30 November 2020).
11. Zhang, F.; Cecchetti, E.; Croman, K.; Juels, A.; Shi, E. Town crier: An authenticated data feed for smart contracts. In Proceedings of the 2016 aCM sIGSAC Conference on Computer and Communications Security, Vienna, Austria, 24–28 October 2016; pp. 270–282.

12. Zhang, F.; Maram, D.; Malvai, H.; Goldfeder, S.; Juels, A. Deco: Liberating web data using decentralized oracles for tls. In Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, Online, 9–13 November 2020; pp. 1919–1938.
13. Ritzdorf, H.; Wüst, K.; Gervais, A.; Felley, G.; Capkun, S. TLS-N: Non-repudiation over TLS enablign ubiquitous content signing. In Proceedings of the NDSS Symposium 2018, San Diego, CA, USA, 18–21 February 2018.
14. Zcash. Available online: https://z.cash/ (accessed on 30 November 2020).
15. Monero. Available online: https://www.getmonero.org/ (accessed on 30 November 2020).
16. Sasson, E.B.; Chiesa, A.; Garman, C.; Green, M.; Miers, I.; Tromer, E.; Virza, M. Zerocash: Decentralized anonymous payments from bitcoin. In Proceedings of the 2014 IEEE Symposium on Security and Privacy, San Jose, CA, USA, 18–21 May 2014; pp. 459–474.
17. Kosba, A.; Miller, A.; Shi, E.; Wen, Z.; Papamanthou, C. Hawk: The blockchain model of cryptography and privacy preserving smart contracts. In Proceedings of the 2016 IEEE Symposium on Security and Privacy (SP), San Jose, CA, USA, 22–26 May 2016; pp. 839–858.
18. Kalodner, H.; Goldfeder, S.; Chen, X.; Weinberg, S.M.; Felten, E.W. Arbitrium: Scalable, private smart contracts. In Proceedings of the 27th {USENIX} Security Symposium, Baltimore, MD, USA, 15–17 August 2018; pp. 1353–1370.
19. Eberhardt, J.; Tai, S. On or Off the Blockchain? Insights on Off-Chaining Computation and Data. In *European Conference on Service-Oriented and Cloud Computing*; Springer: Cham, Swizerland, 2017; pp. 3–15.
20. Eberhardt, J.; Tai, S. ZoKrates-Scalable Privacy-Preserving Off-Chain Computations. In Proceedings of the 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), Halifax, NS, Canada, 30 July–3 August 2018; pp. 1084–1091.
21. Steffen, S.; Bichsel, B.; Gersbach, M.; Melchior, N.; Tsankov, P.; Vechev, M. zkay: Specifying and enforcing data privacy in smart contracts. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, London, UK, 11–15 November 2019; pp. 1759–1776.
22. Costan, V.; Devadas, S. Intel SGX Explained. *IACR Cryptol. EPrint Arch.* **2016**, *2016*, 1–118.
23. Cheng, R.; Zhang, F.; Kos, J.; He, W.; Hynes, N.; Johnson, N.; Song, D. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In Proceedings of the 2019 IEEE European Symposium on Security and Privacy (EuroS & P), Stockholm, Sweden, 17–19 June 2019; pp. 185–200.
24. Gennaro, R.; Gentry, C.; Parno, B.; Raykova, M. Quadratic span programs and succinct nizks without pcps. In Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, 26–30 May 2013; Springer: Berlin/Heidelberg, Germany, 2013; pp. 626–646.
25. Parno, B.; Howell, J.; Gentry, C.; Raykova, M. Pinocchio: Nearly practical verifiable computation. In Proceedings of the 2013 IEEE Symposium on Security and Privacy, San Francisco, CA, USA, 19–22 May 2019; pp. 238–252.
26. Klaytn, Position Paper. 2019. Available online: https://www.klaytn.com/Klaytn_PositionPaper_V2.1.0.pdf (accessed on 30 November 2020).
27. Sergio Demian Lerner, RSK: Bitcoin Powered Smart Contracts. 2019. White Paper. Available online: https://www.rsk.co/Whitepapers/RSK-White-Paper-Updated.pdf (accessed on 30 November 2020).
28. TRON Foundation, TRON Advanced Decentralized Blockchain Platform. 2018. Available online: https://tron.network/static/doc/white_paper_v_2_0.pdf (accessed on 30 November 2020).
29. Androulaki, E.; Barger, A.; Bortnikov, V.; Cachin, C.; Christidis, K.; De Caro, A.; Enyeart, D.; Ferris, C.; Laventman, G.; Manevich, Y.; et al. Hyperledger fabric: A distributed operating system for permissioned blockchains. In Proceedings of the Thirteenth EuroSys Conference, Porto, Portugal, 23–26 April 2018; pp. 1–15.