zk-Oracle: Trusted Off-Chain Compute and Storage for Decentralized Applications

***Abstract*—Blockchain and Decentralized Applications (DApps) are increasingly important for creating trust and transparency in data storage and computation. However, on-chain transactions are often costly and slow. To overcome this challenge, off-chain nodes can be used to store and compute data. Unfortunately, this introduces the risk of untrusted nodes. To address this, authenticated data structures have been proposed, however, this ignores the compute of data from the raw data. We tackle this challenge by introducing zk-Oracle, which provides an efficient and trusted compute and storage off-chain. zk-Oracle builds on zero-knowledge proof (zk-proof for short) technologies to achieve two goals. First, the computation of data structures from raw data and the corresponding proof generation is improved in terms of performance. Second, the verification on-chain is inexpensive and fast. Our experiments show that we can speed up zk-proof generation by up to more than $550\times$ faster than the baseline method.**

***Index Terms*—zk-SNARK, blockchain, IoT**

I. Introduction

Blockchain is a distributed database that allows multiple parties to share and maintain a single, tamper-evident ledger of transactions. It is the technology underlying cryptocurrencies such as Bitcoin and Ethereum. DApps (decentralized applications) are applications that are built on top of blockchain. They are not controlled by any single authority, but rather operate on a decentralized network of computers. Blockchain and DApps are important because they offer a way to conduct transactions and exchange value without the need for a central authority. This not only has the potential to make transactions faster and more efficient, but also to create new types of applications that were not previously possible. For example, DApps could be used to create decentralized markets, enable secure voting systems, or provide a platform for peer-to-peer lending.

One of the challenges of blockchain-based DApps is the high cost and latency of transactions. Because all transactions on a blockchain must be processed by every node on the network, the more users a DApp has, the more computational power is required to process the transactions. This can lead to slow performance and high transaction fees. For example, writing to a blockchain smart contract can take tens of minutes or more to finalize [1]. And the cost of a smart contract operation is estimated that the average cost of a single smart contract operation is around 3 dollars [2].

To be practical for high-volume transactions, DApps now are built using a combination of on-chain and off-chain components to achieve the desired level of performance and cost efficiency. The on-chain component of a DApp typically consists of a smart contract that defines the rules and logic of the application, while the off-chain component consists of the user interface and other supporting services that interact with the smart contract. In this way, the heavy tasks like computation and data storage can be done with off-chain nodes, reducing the monetary and performance overhead of performing actions on-chain.

However, the security risks of utilizing off-chain nodes that are outside of the blockchain network and are thus not governed by the same security guarantees. For this reason, two kinds of techniques were often used to ensure that off-chain nodes will not act maliciously. (1) The first kind of methods use authenticated data structures to provide trust on the outcome of off-chain nodes' processing [3], [4], [5], [6], but the problem with those methods is that a trusted entity is needed to guarantee the integrity of such data structures. (2) The second type of methods rely on verifiable computing techniques [7], [8], [9]. However, these methods could be quite expensive for off-chain nodes. For example, the proving time takes about 10 years [10] in the state-of-the art zk-SNARKs [11] for the dataset VGG16 [12] (around 568 MegaBytes) using a CNN (Convolutional Neural Network) model.

In this work, we propose zk-Oracle, an on-chain/off-chain solution that enables efficient and cost-effective solutions for off-chain compute and storage. The main contribution is to study approaches to speed up zk-based proof generation. We propose a batching algorithm for zk-proof generation that utilizes two design patterns: (1) horizontal batching, and (2) vertical batching. Specifically, horizontal batching refers to splitting the whole input dataset (or workloads) into small ones, such that each batch of data can be performed with the algorithm program sequentially. Vertical batching, otherwise, breaks up the complete algorithm program into multiple small modules such that these modules can be performed sequentially with the correct logic and outcome. We optimize the size of zk-proofs such that the proposed batching algorithm will not produce larger size of zk-proofs compared with that the baseline solution. In addition, the proposed batching algorithm can be performed in parallel which further saves the zk-proof generation time. Lastly, the proposed batching method can be easily implemented with some state-of-the-art zk-SARNK systems and tools, such as lisnark [13] and ZoKrates [14].

Although zk-Oracle is applicable to general DApps, the focus in this paper is on two classes of applications: (1) IoT/supply chain applications where the data sources might be small IoT devices that are not capable of compute/storage. (2) Gaming and social DApps, where users might be using small or mobile devices that are not available all the time, and

may be limited in terms of compute for energy preservation.

The contributions of this paper are as follows:

- We propose zk-Oracle, an on-chain/off-chain solution that enables efficient and cost- effective solutions for off-chain compute and storage.
- We propose a batching algorithm that utilizes two design patterns—horizontal and vertical batching— to speed up the zk-proof generation. The proposed batching method can be easily implemented with some state-of-the-art zk-SARNK systems and tools.
- We conduct a comprehensive evaluation to study the effectiveness of our solution. Our experiments show that we can speed up zk-proof generation by up to more than $550\times$ faster than the baseline method.

The rest of the paper is organized as follows: We first present the preliminaries in Section 2. Then, we introduce the zk-Oracle design in Section 3 followed by the detailed techniques of accelerating zk-proof generation in Section 4. In Section 5, we show our experiments. In Section 6, we describe the related work and conclude with a discussion of future directions and challenges in Section 7.

II. Preliminaries

A. Blockchain and DApps

One challenge for DApps is the high cost of transactions on many blockchain platforms. This can make it expensive for users to interact with DApps especially when the DApps require heavy computation and large storage. Instead of storing and computing data on blockchain, zk-Oracle offloads the heavy computation processing and stores large amount of data to off-chain nodes. While zk-Oracle guarnatees the integrity of the computation and data, it significantly reduce the on-chain transaction fees.

B. zk-SNARK

zk-SNARK stands for "Zero-Knowledge Succinct Non-Interactive Argument of Knowledge", and it refers to a proof construction where one can prove possession of certain information, without revealing that information. For instance, a zk-SNARK can be used to prove and verify this statement "Given a public predicate F and a public input x, I know a secret input w such that $F(x, w) = true$". Given a statement s, the zk-SNARK is used in the following way by utilizing three components: the setup component, the prover component, and the verifier component for DApps (Figure 1):

- In the *setup component*, a setup node generates a proving key Pk_s and a verification key Vk_s that will be used to generate and verify proofs. Although these two keys can be published, the computation work to generate these two keys should remain a secret. Therefore, for zk-SNARK, the setup—which is a one-time process before operation—must be performed by a trusted node or multiparty computation (MPC) [15]. After setup, there is no need for trusted nodes. The generation of the two keys is influenced by the type of computation that needs to be proven. The user provides the program to be proven/verified as well as the inputs to such computation. The user assigns which parts of the inputs are public and which parts are secret. In zk-Oracle, for example, the program to prove/verify is the one that updates the key-value pairs and produces a new state about the key-value pairs; and the inputs to the program are the previous state and its digest as well as the operations that are applied to the previous state to generate the new state.
- The prover node in the *prover component* is responsible for generating the correctness proof of the computation. It needs three parameters, the proving key Pk_s, the public information, Inf_{pub}, and the secret information, Inf_{secret} which is optional. After collecting these parameters, the prover node generates a proof π_s of the computation outcome.
- In the *verifier component*, the verifier uses three parameters: the verification key Vk_s, the public information Inf_{pub}, and the proof π_s to verify the proof π_s. After collecting these parameters, the verifier node generates a decision (True or False). In hybrid blockchains, the verifier can be a smart contract. Typical zk-SNARK protocols are designed so that verification is fast at the expense of a more lengthy proof generation process. This is suitable for hybrid blockchains, since generating proofs is performed by off-chain nodes that do not have the constraints of smart contracts, while verification is performed on-chain.

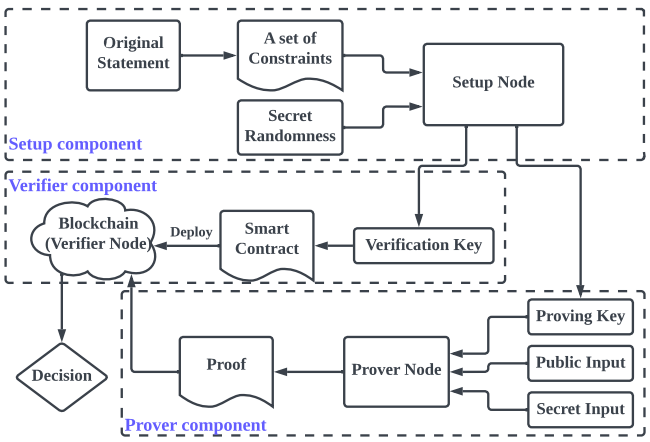


Fig. 1: The workflow of zk-SNARK for DApps.

C. Use Cases

To use zk-SNARK to do computation on raw data, the raw data would first need to be sent to an off-chain node. This node would then perform the necessary calculations to get the zk-proofs for such calculations, using the zk-SNARK proof construction. The zk-proofs could then be sent back to the original sender or to another party for verification. The exact process for transforming the data would depend on the specific application and the requirements of the parties involved. In general, however, the process would involve the prover generating a zk-SNARK proof that they have the

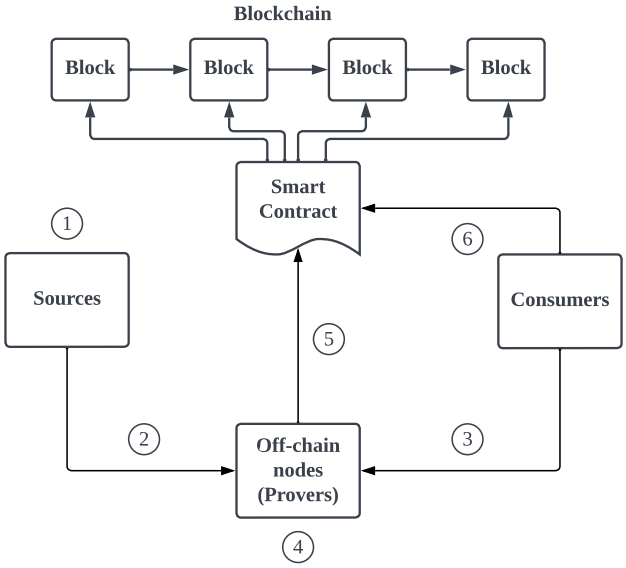


Fig. 2: The framework of zk-Oracle.

knowledge required to transform the data, and the verifier using the proof to verify that the transformation was performed correctly without revealing any information about the original data. This process could be repeated multiple times to ensure that the data has been transformed correctly and accurately.

In the IoT space, zk-SNARKs could be used to verify the authenticity and integrity of sensor data without revealing the actual data being collected. This could be especially useful in applications where sensitive information is being collected, such as in healthcare or financial services. In the supply chain space, zk-SNARKs could be used to verify the provenance of goods, ensuring that they have not been tampered with or counterfeited. This could be especially useful in industries where counterfeiting is a major concern, such as in the pharmaceutical or luxury goods industries.

In the gaming industry, zk-SNARKs could be used to verify the fairness of online games, ensuring that the game results are truly random and not influenced by any outside factors. This could help to build trust and confidence among players and increase the overall enjoyment of the gaming experience. In the social network space, zk-SNARKs could be used to verify the authenticity of user accounts, ensuring that the person behind the account is who they claim to be. This could help to reduce the prevalence of fake accounts and increase trust among users. It could also be used to verify the authenticity of content posted on the network, helping to reduce the spread of misinformation and fake news.

III. zk-Oracle Design

In this section, we describe the design of zk-Oracle

A. System Model

zk-Oracle consists of the following components (Figure 2):

- **Sources:** The sources collect the raw data from their accessible resources. Examples are IoT devices which use sensors to collect data from their environment.
- **Off-chain Provers:** The off-chain provers compute the data from the raw data and perform zk-SNARK computation to generate proofs of their computation.
- **Consumers:** The consumers send read and write requests to smart contracts and get the response from smart contract.
- **Smart contracts:** On-chain smart contracts handle the verification and maintenance of digests related to the computation results and zk-proof data. Also, the smart contract handles the punishment strategy by verifying whether the zk-proof is valid. If the zk-proof can not be proved to be valid, then the smart contract punishes the off-chain prover by withdrawing funds from its escrow account.

Security model. Off-chain prover are not trusted. They can deviate from the protocol in arbitrary ways, similar to byzantine failures [16]. Off-chain provers can collude together and with consumers. The smart contract logic executes correctly—without deviating from the protocol—due to running on blockchain. Write requests are assumed to be authenticated by consumers, which prevents off-chain provers from fabricating clients requests.

Network model. Similar to prior systems, we assume a synchronous model for liveness. That is, during each time step, all components of the system are expected to execute their assigned tasks and update their internal states.

System assumptions. We assume that the sources (see Figure 2) have low compute/ storage capabilities that they can not computing the data from the raw data. The off-chain provers have high compute/storage capabilities but are not trusted instead.

B. Overview

We now provide a description of zk-Oracle's core design. We will describe the end-to-end life-cycle of the zk-Oracle workflow.

Step ①: A source s creates or collects the raw data D from its environment.

Step ②: The source s sends the raw data D to an off-chain prover (node) p.

Step ③: A consumer sends a request r to an off-chain prover (node) p.

Step ④: After the prover p receives the raw data D and the request r, it performs two steps to complete the computation task.

- Step ④(a): The prover p first does the computation on D according to the requirement of the request r. After the computation finishes, the prover p gets the final output of the computation (possibly with many intermediate outputs).
- Step ④(b): Next, the prover p performs zk-SNARK computation to get the corresponding zk-proof π for the computation. Although generating the corresponding zk-proof π also provides the prover p with the final output

of the computation, we will show why Step ④(a) is necessary for zk-Oracle in Section IV.

Step ⑤: The prover p sends the corresponding zk-proof to the smart contract sc on blockchain. The sc verifies whether the zk-proof π is valid. If the π is not valid, the prover will be punished.

Step ⑥: The consumer reads the output and the transformed data after the smart contract successfully verifies the π. We store the transformed data with tailored structures. For example, we build the key-values pairs with a Merkle tree structure. When a consumer wants to read a specific value, he will receive the some hash values of the Merkle tree nodes instead of the whole Merkle tree structure. In this way, the consumer can verify the integrity of the value he reads without receiving a large data structure.

IV. Accelerating zk-Proof Generation

While the zk-proof generation takes enormous time with the zk-SNARK baseline method, we propose a solution to speed up the zk-proof generation process. Our method works for any zk-SNARK-based method since it does not rely on specific zk-SNARK constructions.

A. Motivation

We observe that the zk-proof generation time significantly increases when the complexity of the computation task grows. For example, the zk-proof generation time for training a logistic regression model is 1 second with 100 training data samples; however, the zk-proof generation time becomes more than 6000 seconds when training with 10000 data samples. We notice that the total time for zk-proof generation is only 100 seconds when we train a logistic regression model with 100 training data samples 100 times.

From the theoretical analysis, the state-of-the-art zk-SNARKs transform the computation of a circuit into an equivalent representation called a Quadratic Arithmetic Program [17]: a circuit with N wires and M gates is transformed into a satisfaction problem about $O(N)$ polynomials of degree $O(M)$. The complexity of evaluating these polynomials yields $O(MN)$. While the more complex computation tasks have both larger M and N, they often need much more time for generating zk-proofs.

The above experimental results and theoretical analysis motivate us to split large computation tasks into small ones such that the zk-proof generation time for each subtask become lightweight. In the following, we first talk about two batching techniques that divides the large computation task into small ones. After that, we introduce the methods for optimizing the size of zk-proofs.

B. Horizontal batching for zk-proof generation

Horizontal batching for zk-proof generation aims to split the whole input dataset (or workloads) into small ones, such that each batch of data can be performed with the algorithm program sequentially. Figure 3 shows the illustration of the horizontal batching. Essentially, the algorithm program should be able to process a batch of data independently without affecting the outcome of the computation task. For example, the task involving the write operations on a database can use the horizontal batching method as the write operations are processed one by one. Nonetheless, the horizontal zk-proof scaling does not apply for the tasks which need to load the whole input dataset into memory for computation. Examples are those ML algorithms which do not compute the batch graidents for training.

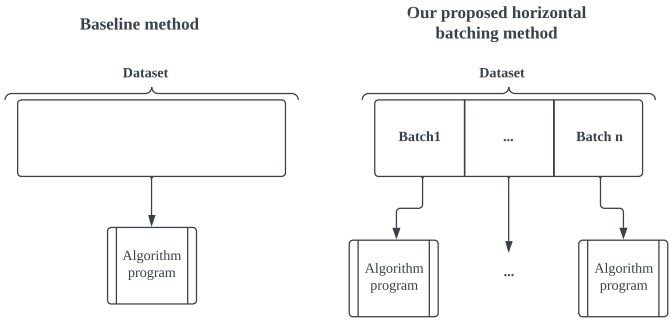


Fig. 3: The illustration of horizontal batching.

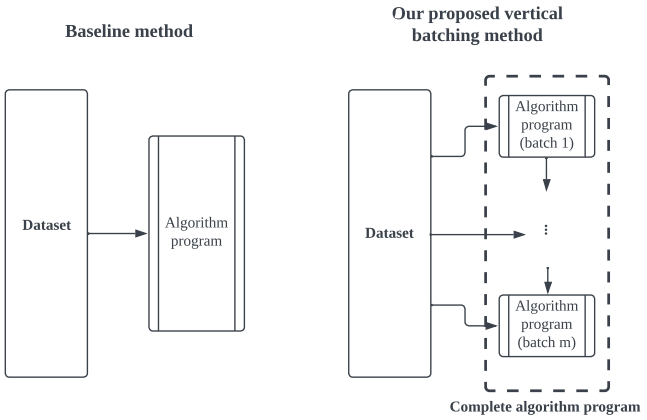


Fig. 4: The illustration of vertical batching.

C. Vertical batching for zk-proof generation

Vertical batching for zk-proof generation breaks up the complete algorithm program into multiple small modules such that these modules can be performed with the correct logic and outcome. Figure 4 illustrates the vertical batching workflow. In principle, any algorithm program can be split into multiple small modules, which we call batches here, as long as the algorithm program has more than one computation operation. To make the outcome of each batch more interpretable to the other users, a good way is to make each batch have the same functionality. For example, in ML training tasks, a good batch can be the the algorithm program that trains the ML model for one epoch. An complete ML algorithm involving 20 epochs for training a ML model would lead to 20 batches.

While a batch means a partial dataset in horizontal batching, it represents a partial algorithm program in vertical batching. We now make a formal definition of a batch used in the rest of the paper.

Definition 4.1 (Batch): Given a set of datasets $D = \{D_1, D_2, ..., D_m\}$ and a set of algorithm programs $A = \{A_1, A_2, ..., A_n\}$, where D_i is a dataset and A_i is an algorithm program, a batch $B(D_i, A_j)$ is a program where the algorithm program A_j performs with the dataset D_i.

With the above definition, the task with the baseline method can be represented as $B(D, A)$, where D and A are the whole dataset and the complete algorithm program respectively. And the maximal number of batches is $m \times n$, given a set of dataset $D = \{D_1, D_2, ..., D_m\}$ (assuming that D can be split for independent processing) and a set of algorithm programs $A = \{A_1, A_2, ..., A_n\}$.

D. Optimizing proof size

The solution to generating zk-proofs with horizontal and vertical batching can significantly reduce the zk-proof generation time. However, producing a zk-proof for each batch results in the large zk-proof size finally. To solve this problem, we propose a way to reduce the redundant content in the zk-proofs. Before introducing our solution, we first review the components of a zk-proof. A zk-proof for a batch $B(D, A)$, denoted as $\pi_{B(D,A)}$, is made up of the public input dataset A, the output of the computation and the cryptographic commitment. The size of $\pi_{B(D,A)}$ then equals to the sum of the size of the input, output and cryptographic commitment.

The following pseudocode shows how a zk-proof is generated with the help of a main function $M(\cdot)$.

```
def M(public D, private w) ->
                  desired data type{
    output = A(D, w)
    return output
}
```

At a high level, the main function $M(\cdot)$ proceeds three steps to generate a zk-proof. First, $M(\cdot)$ takes the public input D and private input w as inputs. Second, it performs the computation program $A(D, w)$ to get the output. Third, it returns the desired computation result obtained from $A(D, w)$. After running $M(D, w)$ with a zk-SNARK tool, a zk-proof $\pi_{B(D,A)}$ is generated.

In the following, we introduce our strategies for dealing with three types of batches: the head batch, middle batch and tail batch. The following pseudocode $M_{head}(\cdot)$ shows the main function to generate a zk-proof for the head (first) batch:

```
def M_head(public D_1, private w) -> bool{
    output = A(D_1, w)
    res = R_1
    return (output == res)
}
```

where R_1 is the output obtained by pre-computing $A(D_1, w)$ (see the step ④(a) in Figure 2) without generating zk-proofs. R_1, like the function $A(\cdot)$, should be public to the other users to guarantee the validity of the zk-proof for $\pi_{B(D_1,w)}$. M_{head} only returns a bool type instead of the real output of $A(D_1, w)$ because this makes the $\pi_{B(D_1,w)}$ much smaller (recall the components of a zk-proof at the beginning of this section) when the size of the output of $A(D_1, w)$ is large.

Assuming that there are k batches, to generate the zk-proofs for the 2-nd to $(k-1)$-th batches, i.e. middle batches, we illustrate with the following main function $M_{middle}(\cdot)$.

```
def M_middle(private w) -> bool{
    // i denotes the i-th batch
    input = R_{i-1}
    output = A(input, w)
    res = R_i
    return (output == res)
}
```

$M_{middle}(\cdot)$ takes no public inputs. Instead, it initializes an input inside $M_{middle}(\cdot)$ as R_{i-1} which is the output of the $(i-1)$-th batch. Similar to the strategy in M_{head}, M_{middle} returns a bool type instead of the real output of $A(input, w)$ to reduce the size of its zk-proof. Also, R_{i-1} and R_i, like the function $A(\cdot)$, should be public to the other users to verify the validity of the generated zk-proofs.

To generate the zk-proof for the tail (last) batch, we explain with the following main function $M_{tail}(\cdot)$.

```
def M_tail(private w) -> desired data type{
    input = R_{k-1}
    output = A(input, w)
    return output
}
```

$M_{tail}(\cdot)$ takes no public inputs. Instead, it initializes an input inside $M_{tail}(\cdot)$ as R_{n-1} which is the output of the $(k-1)$-th batch. Different with $M_{head}(\cdot)$ and $M_{middle}(\cdot)$, $M_{tail}(\cdot)$ returns the desired output instead of a boolean value because the output should be included in a zk-proof such that the other users can see the final output of the computation task.

Algorithm 1 shows the full algorithm of how the zk-proofs are generated with the proposed batching method. To enable anyone else to verify the validity of these zk-proofs, the algorithm programs A, the set of datasets $D = \{D_1, D_2, ..., D_m\}$, the set of algorithm programs $A = \{A_1, A_2, ..., A_n\}$, and the set of generated outputs $R = \{R_1, R_2, ..., R_{mn-1}\}$ on $B(D_i, A_j)$ $(i \in [1, m],\ j \in [1, n])$ are public. The full algorithm generates a zk-proof for each batch (lines $1-15$). For the first and last batch, it adopts $M_{head}(\cdot)$ and $M_{tail}(\cdot)$, as described before, to generate zk-proofs (lines $3-5$ and lines $7-9$). For the middle batches, we use $M_{middle}(\cdot)$ to generate the zk-proofs (lines $10-12$). The batching algorithm finally returns mn zk-proofs i.e $\pi_{B(D_1,A1)}$ to $\pi_{B(D_m,A_n)}$ (line 16).

Algorithm Properties. The batching algorithm for zk-proof generation owns two properties: (1) The algorithm can be performed in parallel. All the three functions $M_{head}(\cdot)$, $M_{middle}(\cdot)$ and $M_{tail}(\cdot)$ take some known inputs such as D and R, meaning that none of them will interact with each other.

Algorithm 1: Batching method for zk-proof generation

Public information:
The algorithm program A;
The set of datasets $D = \{D_1, D_2, ..., D_m\}$;
The set of algorithm programs $A = \{A_1, A_2, ..., A_n\}$;
The set of generated outputs $R = \{R_1, R_2, ..., R_{mn-1}\}$ on $B(D_i, A_j)$ ($i \in [1, m]$, $j \in [1, n]$).
Input : The private input x and public input D;
Output : zk-proofs for mn batches

for $i \leftarrow 1$ *to* m **do**
  for $j \leftarrow 1$ *to* n **do**
    if $i == 1$ **then**
      $\pi_{B(D_1,A_1)} = M_{head}(D_i, w)$;
    end
    else
      if $i == m$ *and* $j == n$ **then**
        $\pi_{B(D_m,A_n)} = M_{tail}(w)$;;
      end
      else
        $\pi_{B(D_i,A_j)} = M_{middle}(w)$;
      end
    end
  end
end
return $\pi_{B(D_1,A1)}$ to $\pi_{B(D_m,A_n)}$;

Therefore, all the three functions can be performed independently. (2) The size of zk-proofs is $O(|D| + |Out| + k|C|)$ where Out and C are the final output and the cryptographic commitment for each zk-proof respectively, k is the number of batches and $|\cdot|$ is a size calculation function. Because the size of each commitment of a zk-proof is constant (with the state-of-the-art and commonly used Groth schema [11]), we conclude that the sizes of the zk-proofs for the head, middle and tail batches are $O(|D|+|C|)$, $O(|C|)$ and $O(|Out|+|C|)$ respectively. Consequently, the total size of these zk-proofs is $O(|D| + |Out| + k|C|)$. The size of C is often quite small. When the sizes of D and Out are large, the zk-proof size of our batching algorithm become close to that of the baseline $O(|D| + |Out| + |C|)$.

Generating a single zk-proof of zk-proofs. To further optimize the size of these k zk-proofs, we can generate a zk-proof for proving that the k zk-proofs are valid. The following pseudocode shows how to do it.

```
def zk-G(public D, private w) ->
                    desired data type{
    zk_proof = [zk-proofs for k batches]
    for 1 to k:
        Verify(zk_proof, w)
    return output
}
```

In this way, verifying the k zk-proofs can be done with off-chain machines, and only a single zk-proof will be verified on blockchain.

V. Experiment

In this section, we perform an experimental evaluation of the performance of zk-Oracle.

A. Setup

Experimental setup. Our experiments are performed on the Ethereum Goerli test network, which now has switched to proof-of-stake (PoS). We implement the on-chain components using solidity smart contracts, and implement off-chain components using Javascript and Python. Software and libraries that we use for specific approaches are mentioned later in the section. The experimental environment is a computer with a Quad-Core Intel Core i5 processor, 8GB memory, running macOS Catalina. We use *ZoKrates* [14], which supports automatically generating the verifier smart contract in solidity, to implement a zkSNARKs-based approach. The implementation of *ZoKrates* is based on *libsnark* [1], a cryptographic library which implements zk-SNARK schemes. And we use Groth16 [11] scheme to derive proofs with a small size with *ZoKrates*.

Datasets: We use the Yahoo! Cloud Serving Benchmark (YCSB) [18] to generate the workload for database updating experiments. The second dataset, 3D Road Network (Road for short) [19], includes 3D road network with highly accurate elevation information. It contains $430K$ data samples. We use this dataset for the Logistic Regression training and Neural Network inference tasks.

Tasks: We show the effectiveness of our solution on three common tasks.

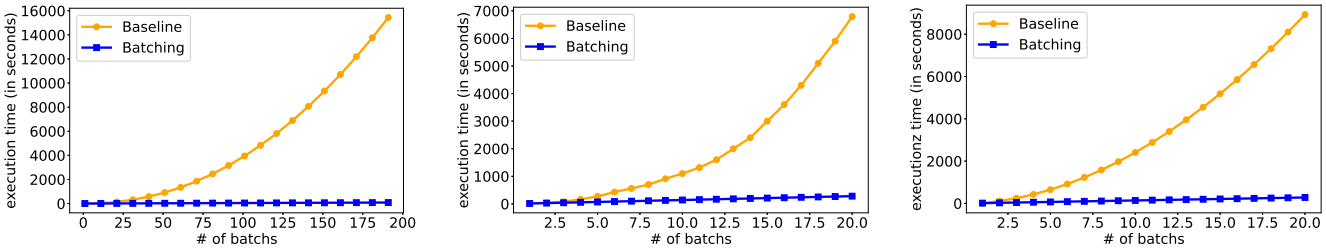
- Key-value updating. In this task, we update the key-value pairs and use the Merkle tree structure to construct the pairs. Instead of sending the whole Merkle tree struture to blockchain, we only send some necessary hash values of the Merkle tree to the blockchain to guarantee the integrity of the data structure.
- Logistic Regression model training. We train a Logistic Regression model and send the model to the blockchain.
- Neural Netowrk inference. We use a Neurual Network model to do inference tasks and send the predictions to the blockchain.

We call the method proposed to speed up zk-proof generation as the **Batching** method and the method for generating a zk-proof for a monolithic task as the **Baseline** method in the experiment.

Default parameters. Unless we mention otherwise, the number of features is set to 10 for the logistic regression model training. The neural network used for Machine Learning (ML for short) inference task has three layers.

Cost. In Ethereum, on-chain execution and verification cost is calculated in a unit called *gas*. For ease of exposition in the rest of this section, we also present the cost in dollars. Because the

[1] https://github.com/scipr-lab/libsnark



(a) Key-value updating (b) Logistic Regression training (c) Neural Network inference

Fig. 5: zk-Proof generation time

TABLE I: zk-Proof generation time (in seconds) with different numbers of batches.

Tasks (# of data samples or operations)	**Methods**	**5**	**10**	**15**	**20**	**25**	**30**	**35**	**40**	**45**	**50**
Key-value updating (100K)	Baseline	15893	15893	15893	15893	15893	15893	15893	15893	15893	15893
	Batching	3210	1310	930	580	75	67	69	73	75	75
Logistic Regression Training (10K)	Baseline	6792	6792	6792	6792	6792	6792	6792	6792	6792	6792
	Batching	924	410	317	280	197	151	116	99	68	46
Neural Network Inference (10K)	Baseline	8937	8937	8937	8937	8937	8937	8937	8937	8937	8937
	Batching	2660	1160	21	19	19	18	16	17	16	17

gas-dollar conversion rate fluctuates, we make the following assumption about the price of gas. We assume the base gas price as 20 Gwei[2] according to recent approximate pricing on Ethereum Mainnet at the time of writing this paper. We also assume that the price of one ether is equal 1500 dollars.

B. zk-Proof generation time evaluation

The executing time for generating zk-proofs mainly includes the time of compiling a circuit, key generation and witness computation (i.e. the normal computing for the task without zk-proof generation). While the time for key generation and witness is less than one second, the executing time is dominated by compiling a circuit. Because state-of-the-art zkSNARK systems [13] can only support statements of up to 10-20 million gates, we can not generate the zk-proof for the whole YCSB and Road datasets. Therefore, in each round of zk-proof generation, we select $100K$, $10K$ and $10K$ as the maximal amount of workload for the key-value updating, Logistic Regression model training and Neural Network inference tasks. And we calculate the their average executing time on the above specified number of data samples or operations for evaluation.

Figure 5(a), (b) and (c) illustrate the executing time for the three tasks. For all the three tasks, each batch contains 500 data samples or operations. Compared with the Baseline method, the batching method does not appear much superiority in terms of executing time for all the three tasks when the amount of workload is small (i.e. with a small number of batches). However, the batching method saves a immense amount of time when the workloads of the task increase. Specifically, the batching method saves more than 4 hours, 2 hours and 2 hours for the key-value updating, Logistic Regression model training and Neural Network inference tasks. This also implies that the batching method can save more and more time when the workload increases, which makes a lot of sense for real DApps.

We also evaluate the zk-proof generation time by varying the number of batches for a fixed amount of workloads (and datasets). The less the number of batches is, the larger the size of each batch. Table I shows that using the smaller size of a batch usually saves more time than that of a batch with larger size. Nevertheless, the best choice of the number of batches is not always the larger number of batches. 30 is the best one for the key-value updating task. 35 and 45 are the best choices for the Neural Network inference task. The reason is that the zk-proof generation involves some (constant) "preparing" time for each batch. In addition, we observe from Table I that the zk-proof generation time fluctuates in only a small range when the number of batches is larger than a threshold. However, the larger number of batches will produce more extra time for verifying more zk-proofs. Therefore, choosing the largest number of batches is not the best choice.

C. On-chain cost evaluation

The generated zk-proofs need to be send to the smart contract for verification. Figure 6 illustrates the on-chain cost with different numbers of batches for the Logistic Regression model training task. The basic on-chain cost for verifying a zk-proof is around 7 dollars and the extra cost for verifying the 5 to 20 zk-proofs generated by the batching method is around 1 to 4 dollars. The main on-chain cost is for storing

[2]Gwei is a denomination of Ethereum's ether (ETH). A gwei is one-billionth of one ETH.

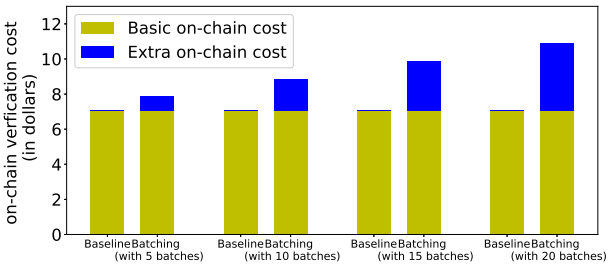


Fig. 6: On-chain cost for Logistic Regression model training.

TABLE II: On-chain cost (in dollars) with different numbers of batches for the key-value updating and Neural Network inference tasks

Tasks (# of data samples or operations)	**Methods**	**5**	**10**	**15**	**20**
Key-value updating (100K)	Baseline	3.015	3.015	3.015	3.015
	Batching	4.020	5.025	6.030	7.035
Neural Network Inference (100)	Baseline	30.725	30.725	30.725	30.725
	Batching	31.730	32.735	33.740	34.745

the parameters of the Logistic Regression model rather than the verification computing. Therefore, the extra on-chain cost with more numbers of batches is not linear with the number of batches.

We show the on-chain cost with different numbers of batches for the key-value updating and Neural Network inference tasks in Table II. The on-chain cost of the Baseline method for Neural Network inference is high because it is expensive to store 100 (note that it is not $10K$) predictions on blockchain. One possible way to reduce this cost is to choose a more efficient structure to store these predictions. The computing cost for verifying the zk-proofs is small relatively for both the key-value updating and Neural Network inference tasks.

D. Scalability Evaluation

We evaluate the scalability of the Batching method with multiple off-chain provers (nodes). As can be seen from Table III, the zk-proof generation time is reduced when the number of off-chain proves increases. And the zk-proof generation time can be scaled quasilinearly by adding more off-chain provers, indicating the good scalability of our proposed batching method.

VI. Related work

We discuss the existing systems for scaling DApps and the efficent zk-SNARK-based systems.

Existing systems for scaling DApps. Layer-2 solutions are methods for increasing the capacity of a blockchain beyond its current limits. Layer-2 solutions [20] are built on top of the main (or layer-1) blockchain. Typical layer-2 solutions consist of Plasma [21] sidechain [22] state channels [23]

TABLE III: zk-Proof generation time (in seconds) with different number of off-chain provers (nodes). We set the number of batches as 30 for this evaluation.

# of off-chain provers (nodes)	**1**	**3**	**5**	**7**	**9**
Key-value updating (100K)	67	24	15	11	9
Logistic Regression training (10K)	151	53	32	23	18
Neural Network Inference (10K)	18	8	5	4	3

Rollups [24] and TrueBit [25]. While each of these is solving a different problem, these layer-2 solutions combine both off-chain state and off-chain computations in arbitrary ways. While zk-Oracle builds on advances in zk-SNARK proof systems [26], we propose an efficient solution to speeding up zk-proof generation process.

Efficient zk-SNARK-based systems. There are mainly two kinds of methods for improving the efficiency of the zk-SNARK-based systems. The first one focuses on customizing zk-SNARK constructions for specific tasks and/or structures. Examples include the methods for decision trees [27], Neural Network inference [28], [29], and boolean circuits [30] tasks. The second one aims to make zk-SNARK constructions distributed and/or incremental. They concentrate on singling out basic computational tasks for achieving efficient distributed realizations [31], [32], [33] or using proof bootstrapping to recursively composing proofs: proving statements about [34], [35], [36] acceptance of the correctness of the latest step of the program. However, these systems are not easy to be implemented for general computations tasks due to their high complexity or they still suffer from enormous computational cost.

Although some work [36], [37] also proposes to break up the generic computation into sub-computations while proving each correct, they focus more on finding a pair of elliptic curves that provide larger bits of security. Also, it is not clear how these methods can be easily implemented for general computations tasks in DApps. Our work, however, builds an effective, economic and trusted system zk-Oracle that can be easily implemented with existing zk-SARNK systems and tools, such as lisnark [13] and ZoKrates [14], for general computation tasks in DApps.

VII. Conclusion

In this paper, we builds zk-Oracle, an efficient and trusted compute and storage off-chain for DApps. zk-Oracle is built on zk-SNARK systems and is compatible with existing state-of-the-art zk-SARNK systems. To speed up the zk-proof generation process, we propose two batching patterns, namely horizontal and vertical batching, for efficient zk-proof generation scaling. Our solution optimizes the size of zk-proofs so that the on-chain cost for verifying the zk-proofs can be minimized. Our experiments show that we can speed up zk-proof generation by up to more than $550\times$ faster than the baseline method.

REFERENCES

[1] K. Croman, C. Decker, I. Eyal, A. E. Gencer, A. Juels, A. Kosba, A. Miller, P. Saxena, E. Shi, E. Gün Sirer *et al.*, "On scaling decentralized blockchains," in *International conference on financial cryptography and data security*. Springer, 2016, pp. 106–125.

[2] "Ethereum charts and statistics," https://etherscan.io/charts, . [Online]. Available: https://etherscan.io/charts

[3] C. Zhang, C. Xu, H. Wang, J. Xu, and B. Choi, "Authenticated keyword search in scalable hybrid-storage blockchains," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2021, pp. 996–1007.

[4] C. Xu, C. Zhang, J. Xu, and J. Pei, "Slimchain: scaling blockchain transactions through off-chain storage and parallel processing," *Proceedings of the VLDB Endowment*, vol. 14, no. 11, pp. 2314–2326, 2021.

[5] H. Wang, C. Xu, C. Zhang, J. Xu, Z. Peng, and J. Pei, "vchain+: Optimizing verifiable blockchain boolean range queries," in *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 2022, pp. 1927–1940.

[6] C. Zhang, C. Xu, J. Xu, Y. Tang, and B. Choi, "Gemˆ 2-tree: A gas-efficient structure for authenticated range queries in blockchain," in *2019 IEEE 35th international conference on data engineering (ICDE)*. IEEE, 2019, pp. 842–853.

[7] R. Gennaro, C. Gentry, and B. Parno, "Non-interactive verifiable computing: Outsourcing computation to untrusted workers," in *Annual Cryptology Conference*. Springer, 2010, pp. 465–482.

[8] J. Eberhardt and J. Heiss, "Off-chaining models and approaches to off-chain computations," in *Proceedings of the 2nd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*, 2018, pp. 7–12.

[9] J. Eberhardt, "Scalable and privacy-preserving off-chain computations," 2021.

[10] S. Lee, H. Ko, J. Kim, and H. Oh, "vcnn: Verifiable convolutional neural network based on zk-snarks," *Cryptology ePrint Archive*, 2020.

[11] J. Groth, "On the size of pairing-based non-interactive arguments," in *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 2016, pp. 305–326.

[12] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[13] S. LAB, "libnsark: a c++ library for zk-snark proofs," https://github.com/scipr-lab/libsnark, 2017.

[14] J. Eberhardt and S. Tai, "Zokrates-scalable privacy-preserving off-chain computations," 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. 1084–1091.

[15] R. Cramer, I. Damgård, and J. B. Nielsen, "Multiparty computation from threshold homomorphic encryption," in *International conference on the theory and applications of cryptographic techniques*. Springer, 2001, pp. 280–300.

[16] L. LAMPORT, R. SHOSTAK, and M. PEASE, "The byzantine generals problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, pp. 382–401, 1982.

[17] B. Parno, J. Howell, C. Gentry, and M. Raykova, "Pinocchio: Nearly practical verifiable computation," *Communications of the ACM*, vol. 59, no. 2, pp. 103–112, 2016.

[18] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010, pp. 143–154.

[19] https://archive.ics.uci.edu/ml/datasets/3D+Road+Network+(North+Jutland,+Denmark).

[20] J. Stark, "Making sense of ethereum's layer 2 scaling solutions: State channels, plasma, and truebit," *Medium. com*, 2018.

[21] J. Poon and V. Buterin, "Plasma: Scalable autonomous smart contracts," *White paper*, pp. 1–47, 2017.

[22] A. Garoffolo and R. Viglione, "Sidechains: Decoupled consensus between chains," *arXiv preprint arXiv:1812.05441*, 2018.

[23] A. Miller, I. Bentov, R. Kumaresan, and P. McCorry, "Sprites: Payment channels that go faster than lightning," *CoRR, abs/1702.05812*, 2017.

[24] C. Sguanci, R. Spatafora, and A. M. Vergani, "Layer 2 blockchain scaling: A survey," *arXiv preprint arXiv:2107.10881*, 2021.

[25] J. Teutsch and C. Reitwießner, "A scalable verification solution for blockchains," *arXiv preprint arXiv:1908.04756*, 2019.

[26] A. M. Pinto, "An introduction to the use of zk-snarks in blockchains," in *Mathematical Research for Blockchain Economy*. Springer, 2020, pp. 233–249.

[27] J. Zhang, Z. Fang, Y. Zhang, and D. Song, "Zero knowledge proofs for decision tree predictions and accuracy," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 2039–2053.

[28] Z. Ghodsi, T. Gu, and S. Garg, "Safetynets: Verifiable execution of deep neural networks on an untrusted cloud," *Advances in Neural Information Processing Systems*, vol. 30, 2017.

[29] L. Zhao, Q. Wang, C. Wang, Q. Li, C. Shen, and B. Feng, "Veriml: Enabling integrity assurances and fair payments for machine learning as a service," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 10, pp. 2524–2540, 2021.

[30] I. Giacomelli, J. Madsen, and C. Orlandi, "{ZKBoo}: Faster {Zero-Knowledge} for boolean circuits," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 1069–1083.

[31] H. Wu, W. Zheng, A. Chiesa, R. A. Popa, and I. Stoica, "{DIZK}: A distributed zero knowledge proof system," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 675–692.

[32] J. Groth, M. Kohlweiss, M. Maller, S. Meiklejohn, and I. Miers, "Updatable and universal common reference strings with applications to zk-snarks," in *Annual International Cryptology Conference*. Springer, 2018, pp. 698–728.

[33] A. Ozdemir, R. S. Wahby, B. Whitehat, and D. Boneh, "Scaling verifiable computation using efficient set accumulators," in *Proceedings of the 29th USENIX Conference on Security Symposium*, 2020, pp. 2075–2092.

[34] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum, "Delegating computation: interactive proofs for muggles," *Journal of the ACM (JACM)*, vol. 62, no. 4, pp. 1–64, 2015.

[35] S. Setty, S. Angel, T. Gupta, and J. Lee, "Proving the correct execution of concurrent services in zero-knowledge," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 339–356.

[36] C. Costello, C. Fournet, J. Howell, M. Kohlweiss, B. Kreuter, M. Naehrig, B. Parno, and S. Zahur, "Geppetto: Versatile verifiable computation," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 253–270.

[37] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza, "Scalable zero knowledge via cycles of elliptic curves," *Algorithmica*, vol. 79, no. 4, pp. 1102–1160, 2017.