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

Binbin Gu[1] and Faisal Nawab[1]

[1]Department of Computer Science, University of California, Irvine, 92697, California, USA.

Contributing authors: binbing@uci.edu; nawabf@uci.edu;

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. There is a challenge in using zero-knowledge proofs (zk-proof for short), which is the large proof generation time. We aim to overcome it with novel designs in zk-Oracle. zk-Oracle builds on zk-proofs 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 **550**× faster than the baseline method.

Keywords: Blockchain, IoT, DApps, zk-SNARKs

1 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 create new types of applications that were not previously possible. For example, DApps can be used to create decentralized markets [1], enable secure voting systems [2], or provide a platform for peer-to-peer lending [3].

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 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 [4]. Likewise, the cost of a smart contract operation/write is estimated to be around 3 dollars [5].

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, heavy-weight tasks like computation and data storage can be performed with off-chain nodes, reducing the monetary and performance overhead of performing actions on-chain.

However, this hybrid approach introduces 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 type of methods use authenticated data structures to provide trust in the outcome of off-chain nodes' processing [6–9]. However, the problem with these methods is that a trusted entity is needed to guarantee the integrity of such data structures. (2) The second type of methods relies on verifiable computing techniques [10–12]. However, these methods could be quite expensive for off-chain nodes. For example, the proving time of a CNN (Convolutional Neural Network) model on the dataset VGG16 [13] (around 568 MegaBytes) takes about 10 years [14] using state-of-the-art techniques such as zk-SNARKs [15].

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 processed with the verification program sequentially. Vertical batching, on the other hand,

breaks up the complete program into multiple small modules such that these modules can be performed sequentially or independently with the correct logic and outcome. We optimize the size of zk-proofs such that the proposed batching algorithm will not produce zk-proofs that are larger in size compared with that of 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 implemented as a layer on top of existing state-of-the-art zk-SARNK systems and tools, such as libsnark [16] and ZoKrates [17].

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 are small IoT devices that are not capable of compute/storage. (2) Gaming and social DApps, where users use small or mobile devices that are not available all the time, and may be limited in terms of compute due to 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 zk-proof generation. The proposed batching method can be easily implemented with 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 550× 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 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 related work and conclude with a discussion of future directions and challenges in Section 7.

2 Preliminaries

2.1 Blockchain and DApps

Blockchain technology is a decentralized and distributed ledger system that allows for the secure and transparent storage and transfer of data. It is best known as the underlying technology behind cryptocurrencies such as Bitcoin, but its potential applications go far beyond that. One of the most promising use cases for blockchain technology is decentralized applications (DApps). These are software applications that run on a blockchain network and operate in a decentralized manner, meaning they are not controlled by any single entity or authority. DApps have the potential to transform a variety of industries, from

finance and healthcare to supply chain management and social media, by providing more secure, transparent, and efficient ways of exchanging information and conducting transactions.

A major 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 a large amount of data on off-chain nodes. While zk-Oracle guarnatees the integrity of the computation and data, it significantly reduces the on-chain transaction fees.

2.2 zk-SNARK

zk-SNARK [18] 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, 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) [19]. 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

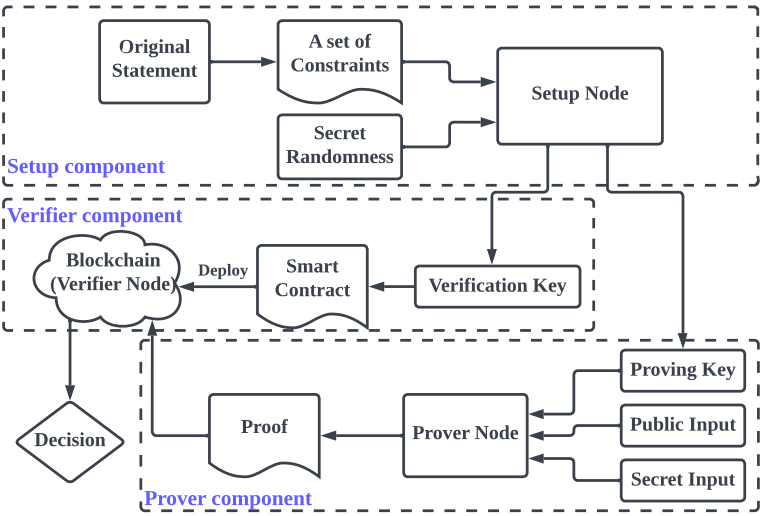


Fig. 1: The workflow of zk-SNARK for DApps.

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.

2.3 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 are then sent back to the original sender or to another party for verification.

In the IoT space, zk-SNARKs is used to verify the authenticity and integrity of sensor data without revealing the actual data being collected [20]. 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 can be used to verify the provenance of goods, ensuring that they have not been tampered with or counterfeited [21]. 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 can be used to verify the fairness of online games, ensuring that the game results are truly random and not influenced by any outside factors [22]. This could help to build trust and confidence among players and increase the overall enjoyment of the gaming experience.

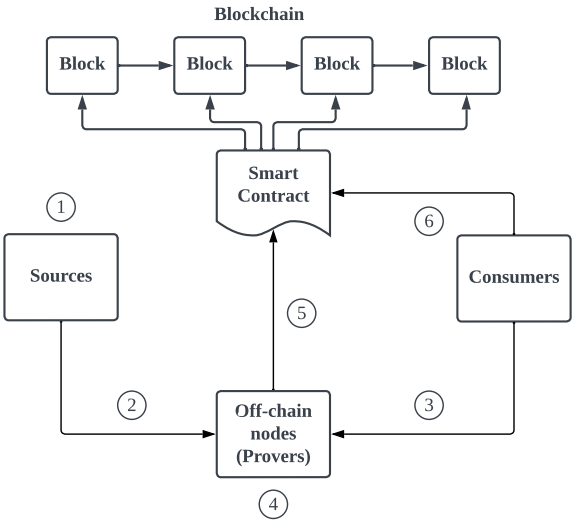


Fig. 2: The framework of zk-Oracle.

In the social network space, zk-SNARKs can be used to verify the authenticity of user accounts, ensuring that the person behind the account is who they claim to be [23**?** , 24]. This helps reduce the prevalence of fake accounts and increase trust among users. It can also be used to verify the authenticity of content posted on the network, helping to reduce the spread of misinformation and fake news.

3 zk-Oracle Design

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

3.1 System Model

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

- **Sources:** The sources collect or generate raw data. Examples are IoT devices that 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 contracts.
- **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 proven to be valid, then the smart contract punishes the off-chain prover by withdrawing funds from its escrow account.

Security model. The Off-chain prover is not trusted. It can deviate from the protocol in arbitrary ways, similar to byzantine failures [25]. 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. For safety we consider an asynchronous network model, meaning that we do not make assumptions about the network to guarantee safety. However, for liveness, since it cannot be guaranteed in asynchronous settings [26], we assume a partially-synchronous model.

In a partially synchronous model [27], the network model assumes that there is an upper bound on message delays and that the network switches between periods of synchrony and asynchrony. During the synchronous periods, all messages are delivered within the bound and the network behaves as if it were synchronous. During the asynchrony periods, messages may be delayed beyond the bound, and the network behaves as if it were asynchronous.

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

3.2 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. An example of r can be "What is the logistic regression model trained based on D given some pre-defined parameters?".
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 performs 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 the performance of zk-Oracle in Section 4.

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 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 π. We store the transformed data with auxiliary data structures. For example, we build the key-value pairs with a Merkle tree structure. When a consumer wants to read a specific value, they will receive the 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 without receiving a large data structure.

4 Accelerating zk-Proof Generation

While zk-proof generation takes a long 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.

zk-Oracle focuses on the method based on the Groth16 [15] schema. The proof size of Groth16 is relatively small which makes Groth16 popular for blockchain-based applications since verifying the Groth16 proof is cheap.

4.1 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 on 100 batches with each batch containing 100 data samples (the total number of samples is still 10000 in this case).

From this analysis, the state-of-the-art zk-SNARKs transform the computation of a circuit into an equivalent representation called a Quadratic Arithmetic Program (QAP) [28]. Assuming that a circuit with N wires and M gates is a computational structure that represents the sequence of logical operations to be performed. In the QAP approach, this circuit is transformed into a set of polynomials. Specifically, O(N) polynomials are generated, each with a degree of O(M). These polynomials represent the relationships and computations within the circuit. The complexity of evaluating these O(N) polynomials depends on both the number of polynomials and the degree of each polynomial. So the evaluation complexity is O(MN), where M represents the degree of the polynomials, and N represents the number of wires (variables) in the circuit.

While the more complex computation tasks have both larger M and N, they often need much more time to generate zk-proofs.

The experimental results and analysis motivate us to split large computation tasks into small ones which we refer to as batches in this paper. By using

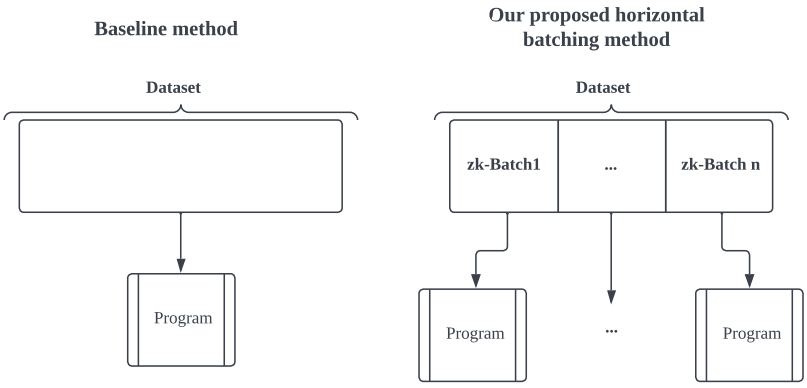


Fig. 3: The illustration of horizontal batching.

the batch techniques, the zk-proof generation time for each subtask becomes shorter and thus leads to a lower zk-proof generation time in total. In the following, we first present a formal definition of the batching approach used in this work. After that, we present two batching techniques that divide large computation tasks into small ones. Finally, we describe the methods for optimizing the size of zk-proofs.

Definition 1 (zk-batch) Given a set of datasets $D = \{D_1, D_2, ..., D_m\}$ and a set of programs $P = \{P_1, P_2, ..., P_n\}$, where D_i is a dataset and P_i is a program, a zk-batch $B(D_i, P_j)$ is a program where the program P_j is performed on the dataset D_i.

Example 1 Suppose we are training a logistic regression model on a dataset $D = \{D_1, D_2, ..., D_m\}$ with a training program $P = \{P_1, P_2, ..., P_n\}$. Denote P_1 as the program for the first epoch of training, then a zk-Batch $B(D_1, P_1)$ means that we train the logistic regression model for 1 epoch on the dataset D_1. And P_i training on D_j is the *i-th* epoch of training on D_j. We apply all $m \times n$ components to mimic training for n epochs on m datasets.

While the batching methods are often used in many contexts, there are new challenges when applied to zero-knowledge proofs. These challenges include how to deal with the inputs and outputs of the batches so that the proof generation can be done in parallel and its cost can be significantly reduced without increasing the verification cost significantly.

4.2 Horizontal Batching for zk-Proof Generation

Horizontal batching for zk-proof generation aims to split the whole input dataset (or workloads) into smaller ones, such that each batch of data can be performed with the program sequentially. Figure 3 shows an illustration of horizontal batching. Essentially, the program should be able to process a batch of data either independently or sequentially without affecting the final outcome of the computation task. The zk-batch B_i can be processed for proof generation before B_{i-1} when the programs are processed independently.

Formally, given a dataset $D = \{D_1, D_2, ..., D_m\}$ and a program P, horizontal batching works when $P(D)$ is equivalent to the result of executing P in the order of $P(D_1), P(D_2), ..., P(D_m)$, where $P(D)$ represents the outcome obtained by executing P on D. The property also holds if $P(D_i)$ is independent of $P(D_j)$ where $i \neq j$. More precisely, horizontal batching requires that the computation task can be executed on a subset of D in a sequential or parallel way.

Example 2 Suppose that we are training a machine learning (ML for short) model on a dataset $D = \{D_1, D_2, ..., D_m\}$ with a training program P. And P uses the batch gradients to update the model. That is, the gradients are computed for a single batch to update the model each time. In the scenario, the whole process can be represented in the following order: $P(D_1), P(D_2), ..., P(D_m)$. Therefore, horizontal batching works naturally for such a task.

4.3 Vertical Batching for zk-Proof Generation

Vertical batching for zk-proof generation breaks up the complete program into multiple small modules such that they maintain the same correct logic and outcome. Figure 4 illustrates the vertical batching workflow. In principle, any program can be split into multiple small modules as long as the program has more than one computation operation.

Formally, given a dataset D and a program $P = \{P_1, P_2, ..., P_n\}$, vertical batching can work when $P(D)$ is equivalent to the result by executing $P_1, P_2, ..., P_n$ in the order of $P_1(D), P_2(P_1(D)), ..., P_n(P_{n-1}(\cdots(D)))$ where $P_i(D)$ represents the outcome obtained by executing P_i on D. The property also holds if $P_i(D)$ is independent of $P_j(D)$ where $i \neq j$. The intermediate states $P_{n-1}(P_{n-2}(\cdots(D)))$ can be precomputed without using zero-knowledge proofs as the cost of precomputing these intermediate states is negligible than generating zero-knowledge proofs.

To make the outcome of each zk-batch more interpretable, we make each zk-batch have the same programming structure. For example, in ML training tasks, a zk-batch can be the program that trains the ML model for one epoch. A complete ML algorithm involving 20 epochs for training a ML model would lead to 20 zk-batches using the vertical batching method. All 20 batches have the same programming structure and thus we are able to perform vertical batching.

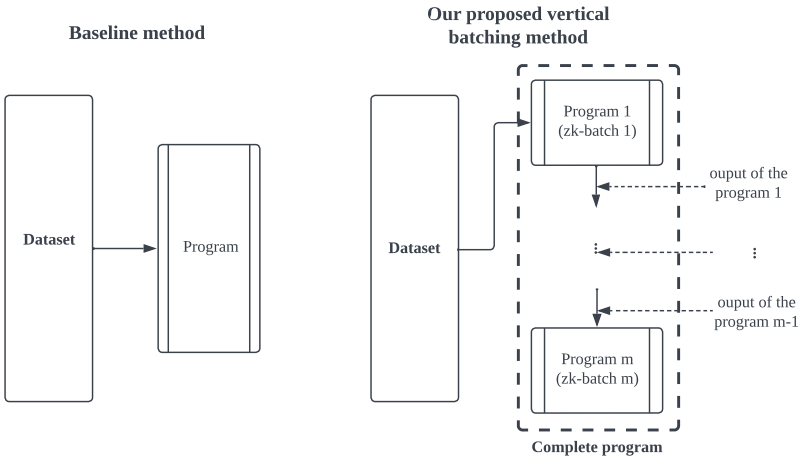


Fig. 4: The illustration of vertical batching.

While a zk-batch means a partial dataset in horizontal batching, it represents a partial program in vertical batching. We describe the representations of baseline, horizontal and vertical batching methods for comparison with the Definition 1. Denote $D = \{D_1, D_2, ..., D_m\}$ and $P = \{P_1, P_2, ..., P_n\}$ as the whole dataset and the complete program respectively. The task with the baseline method can be represented as $B(D, P)$, the task with horizontal batching can be represented as $B(D_1, P), B(D_2, P), ..., B(D_m, P)$ and the task with vertical batching can be denoted as $B(D, P_1), B(D, P_2), ..., B(D, P_n)$.

4.4 Limitations of Horizontal Batching and Vertical Batching

Horizontal batching only works for data-independent computations. One example is the machine learning algorithm with Stochastic Gradient Descent (SGD) [29]. SGD is an iterative optimization algorithm that updates the model's parameters based on the gradient of the loss function computed on a single randomly sampled training example at a time. Unlike batch gradient descent, which computes the gradient of the loss function for all training examples in the dataset, SGD updates the parameters on a small subset of the data at each iteration, which can lead to faster convergence. However, horizontal batching does not work for tasks that need to load the whole input dataset into memory for computation, such as the computation with batch gradient descent.

Vertical batching only works for program-divisible computations. Most programs can be divisible regardless of their expressiveness. For example, we can

take each line of code as a batch in the extreme case. In practice, it is better to make each zk-batch have the same programming structure to achieve good expressiveness in the types of statements we can prove as described in Section 4.3.

4.5 Optimizing the 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 zk-batch results in a large final zk-proof size. 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 zk-batch $B(D, P)$, denoted as $\pi_{B(D,P)}$, is made up of the public input dataset D, the output of the computation Out and the cryptographic commitment C. The size of $\pi_{B(D,P)}$ then equals the sum of the size of the input, output, and cryptographic commitment. The equation is shown below.

$$|\pi_{B(D,P)}| = |D| + |Out| + |C| \tag{1}$$

where $|\cdot|$ denotes the size of the enclosed set. The following pseudocode shows the function $F(\cdot)$ that we want to prove.

```
def F(public D, private w) -> output data type{
    output = P(D, w)
    return output
}
```

$F(\cdot)$ is the function to be proven rather than the one used to generate the proof. We denote $Proof(F(\cdot))$ as the proof generation function for generating a proof for the function $F(\cdot)$. At a high level, the function $F(\cdot)$ proceeds in three steps to generate a zk-proof. First, $F(\cdot)$ takes the public input D and private input w as inputs. Second, it performs the computation program $P(D, w)$ to get the output. Third, it returns the desired computation result obtained from $P(D, w)$. After running $F(D, w)$ with a zk-SNARK tool, a zk-proof $\pi_{B(D,P)}$ is generated which is used to prove that $F(D, w)$ is run correctly.

To reduce the size of the proofs, we relate and chain the proofs of different batches together. This creates three types of batches in a chain of batches: the head zk-batch, middle zk-batch, and tail zk-batch. The proposed three kinds of zk-batches work for both horizontal and vertical batching. The following pseudocode $F_{head}(\cdot)$ shows the function that we want to prove for the head (first) zk-batch:

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

where R_1 is the output obtained by pre-computing $P(D_1, w)$ (see the step ④(a) in Figure 2) without generating zk-proofs. R_1, like the function $P(\cdot)$, must be public to the other users to guarantee the validity of the zk-proof for $\pi_{B(D_1,w)}$. F_{head} only returns a bool type instead of the real output of $P(D_1, w)$ 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 $P(D_1, w)$ is large.

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

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

$F_{middle}(\cdot)$ takes no public inputs. Instead, it initializes an input inside $F_{middle}(\cdot)$ as R_{i-1} which is the output of the $(i-1)$-th zk-batch. Similar to the strategy in F_{head}, F_{middle} returns a bool type instead of the real output of $P(input, w)$ to reduce the size of its zk-proof. Also, R_{i-1} and R_i, like the function $P(\cdot)$, 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) zk-batch, we define the following function $F_{tail}(\cdot)$ that we want to prove for the tail batch.

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

$F_{tail}(\cdot)$ takes no public inputs. Instead, it initializes an input inside $F_{tail}(\cdot)$ as R_{n-1} which is the output of the $(k-1)$-th zk-batch. Different from $F_{head}(\cdot)$

and $F_{middle}(\cdot)$, $F_{tail}(\cdot)$ returns the desired output instead of a boolean value because the output is to be included in a zk-proof such that the other users can verify 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 users to verify the validity of these zk-proofs, the programs P, the set of datasets $D = \{D_1, D_2, ..., D_m\}$, the set of programs $P = \{P_1, P_2, ..., P_n\}$, and the set of generated outputs $R = \{R_1, R_2, ..., R_{mn-1}\}$ on $B(D_i, P_j)$ ($i \in [1, m]$, $j \in [1, n]$) are all public. The full algorithm generates a zk-proof for each zk-batch (lines $5 - 18$). For the first and last zk-batch, it adopts $F_{head}(\cdot)$ and $F_{tail}(\cdot)$, as described before, to generate zk-proofs (lines $7 - 11$). For the middle zk-batches, we generate the proofs for the computation tasks described in the functions $F_{middle}(\cdot)$ (lines $12 - 14$). The batching algorithm finally returns mn zk-proofs i.e $\pi_{B(D_1,P1)}$ to $\pi_{B(D_m,P_n)}$ (line 18).

In the Algorithm 1, we assume that both the horizontal and vertical batching methods are applied for zk-proof generation. However, this algorithm also works for the scenario when only one of the horizontal and vertical batching methods works.

Algorithm 1 Batching method for zk-proof generation

Require: (Public information)
1: The program P;
2: The set of datasets $D = \{D_1, D_2, ..., D_m\}$;
3: The set of programs $P = \{P_1, P_2, ..., P_n\}$;
4: The set of generated outputs $R = \{R_1, R_2, ..., R_{mn-1}\}$ on $B(D_i, P_j)$ ($i \in [1, m]$, $j \in [1, n]$).
Require: (Input) The private input x and public input D;
Ensure: (Output) zk-proofs for mn zk-batches;
5: **for** $i \leftarrow 1$ to n **do**
6: **for** $j \leftarrow 1$ to m **do**
7: **if** $i == 1$ and $j == 1$ **then**
8: $\pi_{B(D_1,P_1)} = Proof(F_{head}(D_i, w))$;
9: **else**
10: **if** $(i \neq 1$ and $j \neq 1)$ and $(i \neq n$ and $j \neq m)$ **then**
11: $\pi_{B(D_i,P_j)} = Proof(F_{middle}(w))$;
12: **else**
13: $\pi_{B(D_m,P_n)} = Proof(F_{tail}(w))$;
14: **end if**
15: **end if**
16: **end for**
17: **end for**
18: **return** all proofs;

Algorithm Properties. The batching algorithm for zk-proof generation owns two properties: (1) The algorithm can be performed in parallel. All the three functions $F_{head}(\cdot)$, $F_{middle}(\cdot)$ and $F_{tail}(\cdot)$ take some known inputs such as D and R, meaning that none of them will interact with each other. Therefore, all the three functions can be performed independently. This property is good for improving performance as it allows for parallel execution. (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 zk-batches and $|\cdot|$ denotes the size of the enclosed set. Because the size of each commitment of a zk-proof is constant (with the state-of-the-art and commonly used Groth16 schema [15]), we conclude that the sizes of the zk-proofs for the head, middle and tail zk-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 becomes close to that of the baseline $O(|D| + |Out| + |C|)$.

Generating a single zk-proof of zk-proofs. To further optimize the size of zk-Oracle's k zk-proofs, we can generate a zk-proof for proving that the k zk-proofs are valid. In this way, the number of cryptographic commitments is reduced to 1. And the optimized zk-proof size becomes $O(|D| + |Out| + |C|)$ which equals to that of the baseline. The following pseudocode shows how to do it.

```
def zk-G(public D, private w) -> desired data type{
    zk_proof = [zk-proofs for k zk-batches]
    for 1 to k:
        Verify(zk_proof, w)
    return output
    // output could be a boolean value to indicate
    // whether these zk-proofs are valid or not
}
```

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

The way to generate a single zk-proof of zk-proofs here is more like a proof aggregation method [30]. This method takes multiple zk-proofs as inputs and outputs a single proof.

Another way to generate a single proof is called recursive zero-knowledge proofs [31, 32]. While recursive zero-knowledge proofs allow chained logical reasoning, they allow more expressiveness in the types of statements we can prove. However, it is quite expensive to generate the zk proof in a recursive way. The reason is that verifying a zk proof still takes millions of gates that will enlarge the size of the circuits of the sub-programs except for the first one.

Due to the drawback of the aggregation methods, we verify all the proofs instead of the proof generated in the aggregation method to improve the

expressiveness of the types of statements we are proving. We show the extra cost of doing all the zk-proofs verification on-chain compared with a single proof verification in the experiment section. Also, we compare our method with the recursive method with respect to the proof generation overhead.

5 Experiment

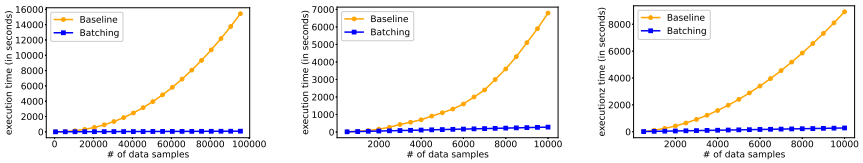
In this section, we perform an experimental evaluation of the performance of zk-Oracle.

5.1 Setup

Experimental setup. Our experiments are performed on the Ethereum Goerli test network, which has recently 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* [17], a toolbox for zkSNARKs on Ethereum. ZoKrates 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 that implements zk-SNARK schemes. And we use Groth16 [15] scheme to derive proofs with a small size with *ZoKrates*.
Datasets: We use the Yahoo! Cloud Serving Benchmark (YCSB) [33] to generate the workload for database experiments. The second dataset, 3D Road Network (Road for short) [34], includes 3D road network with highly accurate elevation information. It contains $430K$ data samples. We use this dataset for Logistic Regression training and Neural Network inference tasks.
Tasks: We show the effectiveness of our solution on three common tasks.

- Key-value updates. This task takes key-value pairs as the input and outputs the inclusion proof for the key-value pair and the sequence number where it is added. We use the Merkle tree structure [35] to construct the pairs. The inclusion proof includes the sibling node of every node in the path from the data item to the root of the Merkle tree. A client receiving the proof calculates the root of the Merkle tree using the provided hashes. If the calculated MMR root matches the original MMR root, then the client knows that the received item is correct. This task is common in databases, blockchain and many other areas. Horizontal batching is used to evaluate our method on this task. Specifically, we partition all the key-value pairs into k disjoint batches while preserving the order of the batches and the elements within each batch. And then we generate the proofs for each batch sequentially.

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



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

Fig. 5: zk-Proof generation time (each zk-batch contains a fixed number of instances/samples)

- Logistic Regression model training. We train a Logistic Regression model and send the model to the blockchain. We use vertical batching in this task. As the algorithm takes multiple epochs for training, we take the process of one epoch as a batch. And we adopt the batch gradient descent algorithm to update the model's parameters based on all the training dataset for each batch. We run the epochs of training sequentially so that the final parameters of the Logistic Regression model are not changed.
- Neural Network inference. We use a Neural Network model to do inference tasks and send the predictions to the blockchain. We use both horizontal and vertical batching in this task. Specifically, we partition the dataset into multiple subsets. Each time, we use the Neural Network model on a subset of the dataset to do inference. As the inference on the data samples is independent of each other, horizontal batching will not change the final inference results. For each subset, we take the computation of each layer as a batch using vertical batching.

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 ML 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 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 to 1500 dollars.

5.2 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

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

Table 1: zk-Proof generation time (in seconds) with different numbers of zk-batches.

	Key-value updating (100K)		Logistic Regression Training (10K)		Neural Network Inference (10K)	
Baseline	15893		6792		8937	
	Key-value updating (100K)		Logistic Regression Training (10K)		Neural Network Inference (10K)	
(# of zk-batches)	Recursive	Batching	Recursive	Batching	Recursive	Batching
5	4712	3210	2639	924	4717	2660
10	4967	1310	3678	410	4974	1160
15	5426	930	4971	317	5481	333
20	6358	580	6349	280	6291	124
25	7845	75	7882	197	7839	30
30	9177	**67**	9201	151	9297	18
35	10621	69	10628	116	10664	**16**
40	12066	73	12109	99	12178	17
45	13574	75	13609	68	13971	**16**
50	15198	75	15143	**46**	15162	17

computing for the task without zk-proof generation). While the time for key generation and witness computation is less than one second, the executing time is dominated by compiling a circuit. Because state-of-the-art zkSNARK systems [16] 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 $100K$ as the maximal amount of workload for the key-value updates, $10K$ for Logistic Regression model training and $10K$ for Neural Network inference tasks. We calculate the average executing time of the above-specified number of data samples or operations in the evaluation.

Figure 5(a), (b) and (c) illustrate the executing time for the three tasks. For all three tasks, each zk-batch contains 500 data samples or operations. Compared with the Baseline method, the batching method performs similarly in terms of executing time for all three tasks when the amount of workload is small (i.e. with a small number of zk-batches). However, the batching method reduces the execution time significantly when the size of the workload increases. Specifically, the batching method saves more than 4 hours, 2 hours, and 2 hours for the key-value updates, Logistic Regression model training, and Neural Network inference tasks. This also implies that the batching method can save even more time when the workload increases, which is the case for DApps that would generate large amounts of data continuously.

We evaluate the zk-proof generation time by varying the number of zk-batches for a fixed amount of workloads (and datasets). The experimental results of the baseline, batching and recursive [32] methods are shown in

Table 1. The fewer the number of zk-batches, the larger the size of each zk-batch as the total number of data samples is fixed. For the recursive method, the proof generation time of the recursive method increases when the number of batches increases. This is because each proof, except for the first proof, involves the circuit for verifying the previous proof. More batches lead to more proof generation time to verify the previous proof. For the batching method, the results show that using the smaller size of a zk-batch usually saves more time than that of a zk-batch with a larger size. The best choice of the number of zk-batches is not always the larger number of zk-batches. 30 is the best one for the key-value updates task. 50 is the best choice for the Logistic Regression training. 35 and 45 are the best choices for the Neural Network inference task. The reason is that the zk-proof generation involves (constant) "preparing" time for each zk-batch. In addition, we observe from Table 1 that the zk-proof generation time fluctuates in only a small range when the number of zk-batches is larger than a threshold. However, the larger number of zk-batches will produce extra time for verifying more zk-proofs. Therefore, choosing the largest number of zk-batches is not the best choice.

5.3 On-chain Cost Evaluation

The generated zk-proofs need to be sent to the smart contract for verification. Figure 6 illustrates the on-chain cost with different numbers of zk-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 the parameters of the Logistic Regression model rather than the verification computing. Therefore, the extra on-chain cost with more numbers of zk-batches is not linear with the number of zk-batches.

We show the on-chain cost with different numbers of zk-batches for the key-value updating and Neural Network inference tasks in Table 2. 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 relative to both the key-value updates and Neural Network inference tasks.

zk-Oracle makes trade-offs between the proof generation time and the on-chain cost via the batching method. The more number of batches, the higher on-chain cost. However, the proof generation time may not decrease when the number of batches increases as shown in Table 1.

5.4 Scalability Evaluation

We evaluate the scalability of the Batching method with multiple off-chain provers (nodes). As can be seen from Table 3, the zk-proof generation time

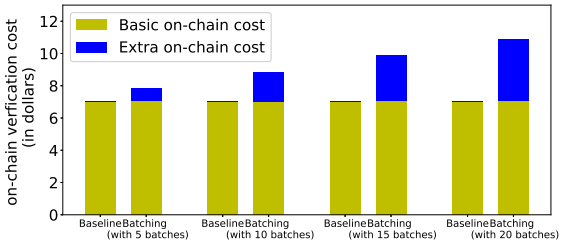


Fig. 6: On-chain cost (in dollars) with different numbers of zk-batches for the Logistic Regression model training task.

Table 2: On-chain cost (in dollars) with different numbers of batches for the key-value updating and Neural Network inference tasks.

	Key-value updating (100K)		Neural Network Inference (100)	
(# of zk-batches)	Baseline	Batching	Baseline	Batching
5	3.015	4.020	30.725	31.730
10	3.015	5.025	30.725	32.735
15	3.015	6.030	30.725	33.740
20	3.015	7.035	30.725	34.745
25	3.015	8.040	30.725	35.750
30	3.015	9.045	30.725	36.755
35	3.015	10.050	30.725	37.760
40	3.015	11.055	30.725	38.765
45	3.015	12.060	30.725	39.770
50	3.015	13.065	30.725	40.775

is reduced when the number of off-chain provers increases. The zk-proof generation time can be scaled quasilinearly by adding more off-chain provers, indicating the parallel nature and scalability of our proposed batching method.

6 Related Work

We discuss existing systems for scaling DApps and the efficient 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 [36–38] are built on top of the main (or layer-1) blockchain. State-of-the-art layer-2 solutions include Plasma [39] sidechain [40] state channels [41] Rollups [42] and TrueBit [43]. 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

Table 3: zk-Proof generation time (in seconds) with different number of off-chain provers (nodes). We set the number of zk-batches to 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

proof systems [44], we propose an efficient solution to speeding up the 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 [45], Neural Network inference [46–49], fairness degree of a ML model [50] and boolean circuits [51] 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 [52–54] or using proof bootstrapping to recursively composing proofs: proving statements about [55–57] 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 [32, 57, 58] 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 or better gadgets using a modular approach [58–60]. 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 [16] and ZoKrates [17], for general computation tasks in DApps.

7 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× faster than the baseline method.

8 Acknowledgments

This research is partly supported by the NSF under grants CNS1815212 and SaTC-2245372.

References

[1] Morstyn, T., Teytelboym, A., McCulloch, M.D.: Designing decentralized markets for distribution system flexibility. IEEE Transactions on Power Systems **34**(3), 2128–2139 (2018)

[2] Hjálmarsson, F.., Hreiarsson, G.K., Hamdaqa, M., Hjálmtỳsson, G.: Blockchain-based e-voting system. In: 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), pp. 983–986 (2018). IEEE

[3] Uriawan, W., Hasan, O., Badr, Y., Brunie, L.: Collateral-free trustworthiness-based personal lending on a decentralized application (dapp). In: SECRYPT, pp. 839–844 (2021)

[4] Croman, K., Decker, C., Eyal, I., Gencer, A.E., Juels, A., Kosba, A., Miller, A., Saxena, P., Shi, E., Gün Sirer, E., *et al.*: On scaling decentralized blockchains. In: International Conference on Financial Cryptography and Data Security, pp. 106–125 (2016). Springer

[5] Ethereum Charts and Statistics. https://etherscan.io/charts (2022)

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

[7] Xu, C., Zhang, C., Xu, J., Pei, J.: Slimchain: scaling blockchain transactions through off-chain storage and parallel processing. Proceedings of the VLDB Endowment **14**(11), 2314–2326 (2021)

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

[9] Zhang, C., Xu, C., Xu, J., Tang, Y., Choi, B.: Gemˆ 2-tree: A gas-efficient structure for authenticated range queries in blockchain. In: 2019 IEEE 35th International Conference on Data Engineering (ICDE), pp. 842–853 (2019). IEEE

[10] Gennaro, R., Gentry, C., Parno, B.: Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In: Annual Cryptology

Conference, pp. 465–482 (2010). Springer

[11] Eberhardt, J., Heiss, J.: Off-chaining models and approaches to off-chain computations. In: Proceedings of the 2nd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers, pp. 7–12 (2018)

[12] Eberhardt, J.: Scalable and privacy-preserving off-chain computations (2021)

[13] Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556 (2014)

[14] Lee, S., Ko, H., Kim, J., Oh, H.: vcnn: Verifiable convolutional neural network based on zk-snarks. Cryptology ePrint Archive (2020)

[15] Groth, J.: On the size of pairing-based non-interactive arguments. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques, pp. 305–326 (2016). Springer

[16] LAB, S.: libsnark: a C++ library for zk-SNARK proofs. https://github.com/scipr-lab/libsnark (2017)

[17] Eberhardt, J., Tai, S.: 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), pp. 1084–1091 (2018). IEEE

[18] Petkus, M.: Why and how zk-snark works. arXiv preprint arXiv:1906.07221 (2019)

[19] Cramer, R., Damgård, I., Nielsen, J.B.: Multiparty computation from threshold homomorphic encryption. In: International Conference on the Theory and Applications of Cryptographic Techniques, pp. 280–300 (2001). Springer

[20] Wu, W., Liu, E., Gong, X., Wang, R.: Blockchain based zero-knowledge proof of location in iot. In: ICC 2020-2020 IEEE International Conference on Communications (ICC), pp. 1–7 (2020). IEEE

[21] Sahai, S., Singh, N., Dayama, P.: Enabling privacy and traceability in supply chains using blockchain and zero knowledge proofs. In: 2020 IEEE International Conference on Blockchain (Blockchain), pp. 134–143 (2020). IEEE

[22] Blum, M., Feldman, P., Micali, S.: Non-interactive zero-knowledge and its applications. In: Providing Sound Foundations for Cryptography: On the

Work of Shafi Goldwasser and Silvio Micali, pp. 329–349 (2019)

[23] Sánchez, D.C.: Zero-knowledge proof-of-identity: Sybil-resistant, anonymous authentication on permissionless blockchains and incentive compatible, strictly dominant cryptocurrencies. arXiv preprint arXiv:1905.09093 (2019)

[24] Gu, B., Li, Z., Liu, A., Xu, J., Zhao, L., Zhou, X.: Improving the quality of web-based data imputation with crowd intervention. IEEE Transactions on Knowledge and Data Engineering **33**(6), 2534–2547 (2019)

[25] LAMPORT, L., SHOSTAK, R., PEASE, M.: The byzantine generals problem. ACM Transactions on Programming Languages and Systems **4**(3), 382–401 (1982)

[26] Lamport, L.: Lower bounds for asynchronous consensus. Future Directions in Distributed Computing: Research and Position Papers, 22–23 (2003)

[27] Drăgoi, C., Henzinger, T.A., Zufferey, D.: Psync: a partially synchronous language for fault-tolerant distributed algorithms. ACM SIGPLAN Notices **51**(1), 400–415 (2016)

[28] Parno, B., Howell, J., Gentry, C., Raykova, M.: Pinocchio: Nearly practical verifiable computation. Communications of the ACM **59**(2), 103–112 (2016)

[29] Ketkar, N., Ketkar, N.: Stochastic gradient descent. Deep learning with Python: A hands-on introduction, 113–132 (2017)

[30] Gailly, N., Maller, M., Nitulescu, A.: Snarkpack: Practical snark aggregation. In: International Conference on Financial Cryptography and Data Security, pp. 203–229 (2022). Springer

[31] Bitansky, N., Canetti, R., Chiesa, A., Tromer, E.: Recursive composition and bootstrapping for snarks and proof-carrying data. In: Proceedings of the Forty-fifth Annual ACM Symposium on Theory of Computing, pp. 111–120 (2013)

[32] Ben-Sasson, E., Chiesa, A., Tromer, E., Virza, M.: Scalable zero knowledge via cycles of elliptic curves. Algorithmica **79**(4), 1102–1160 (2017)

[33] Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with ycsb. In: Proceedings of the 1st ACM Symposium on Cloud Computing, pp. 143–154 (2010)

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

[35] Merkle, R.C.: A digital signature based on a conventional encryption function. In: Conference on the Theory and Application of Cryptographic Techniques, pp. 369–378 (1987). Springer

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

[37] Lin, Q., Gu, B., Nawab, F.: Rollstore: Hybrid onchain-offchain data indexing for blockchain applications. IEEE Transactions on Knowledge & Data Engineering (01), 1–16 (2024)

[38] Gu, B., Kargar, S., Nawab, F.: Efficient dynamic clustering: Capturing patterns from historical cluster evolution. arXiv preprint arXiv:2203.00812 (2022)

[39] Poon, J., Buterin, V.: Plasma: Scalable autonomous smart contracts. White paper, 1–47 (2017)

[40] Garoffolo, A., Viglione, R.: Sidechains: Decoupled consensus between chains. arXiv preprint arXiv:1812.05441 (2018)

[41] Miller, A., Bentov, I., Kumaresan, R., McCorry, P.: Sprites: Payment channels that go faster than lightning. CoRR, abs/1702.05812 (2017)

[42] Sguanci, C., Spatafora, R., Vergani, A.M.: Layer 2 blockchain scaling: A survey. arXiv preprint arXiv:2107.10881 (2021)

[43] Teutsch, J., Reitwießner, C.: A scalable verification solution for blockchains. arXiv preprint arXiv:1908.04756 (2019)

[44] Pinto, A.M.: An introduction to the use of zk-snarks in blockchains. In: Mathematical Research for Blockchain Economy, pp. 233–249. Springer, ??? (2020)

[45] Zhang, J., Fang, Z., Zhang, Y., Song, D.: Zero knowledge proofs for decision tree predictions and accuracy. In: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, pp. 2039–2053 (2020)

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

[47] Zhao, L., Wang, Q., Wang, C., Li, Q., Shen, C., Feng, B.: Veriml: Enabling integrity assurances and fair payments for machine learning as a service. IEEE Transactions on Parallel and Distributed Systems **32**(10), 2524–2540 (2021)

[48] Feng, B., Qin, L., Zhang, Z., Ding, Y., Chu, S.: Zen: Efficient zero-knowledge proofs for neural networks. IACR Cryptol. ePrint Arch. **2021**, 87 (2021)

[49] Gu, B., Singh, A., Zhou, Y., Fang, J., Nawab, F.: Ml on chain: The case and taxonomy of machine learning on blockchain. In: 2023 IEEE International Conference on Blockchain and Cryptocurrency (ICBC), pp. 1–18 (2023). IEEE

[50] Segal, S., Adi, Y., Pinkas, B., Baum, C., Ganesh, C., Keshet, J.: Fairness in the eyes of the data: Certifying machine-learning models. In: Proceedings of the 2021 AAAI/ACM Conference on AI, Ethics, and Society, pp. 926–935 (2021)

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

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

[53] Groth, J., Kohlweiss, M., Maller, M., Meiklejohn, S., Miers, I.: Updatable and universal common reference strings with applications to zk-snarks. In: Annual International Cryptology Conference, pp. 698–728 (2018). Springer

[54] Ozdemir, A., Wahby, R.S., Whitehat, B., Boneh, D.: Scaling verifiable computation using efficient set accumulators. In: Proceedings of the 29th USENIX Conference on Security Symposium, pp. 2075–2092 (2020)

[55] Goldwasser, S., Kalai, Y.T., Rothblum, G.N.: Delegating computation: interactive proofs for muggles. Journal of the ACM (JACM) **62**(4), 1–64 (2015)

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

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

[58] Campanelli, M., Fiore, D., Querol, A.: Legosnark: Modular design and composition of succinct zero-knowledge proofs. In: Proceedings of the 2019

ACM SIGSAC Conference on Computer and Communications Security, pp. 2075–2092 (2019)

[59] Kosba, A., Papamanthou, C., Shi, E.: xjsnark: A framework for efficient verifiable computation. In: 2018 IEEE Symposium on Security and Privacy (SP), pp. 944–961 (2018). IEEE

[60] Singh, N., Dayama, P., Pandit, V.: Zero knowledge proofs towards verifiable decentralized ai pipelines. In: Financial Cryptography and Data Security: 26th International Conference, FC 2022, Grenada, May 2–6, 2022, Revised Selected Papers, pp. 248–275 (2022). Springer