# Snarktor: A Decentralized Protocol for Scaling SNARKs Verification in Blockchains

Alberto Garoffolo alberto.garoffolo@gmail.com Telos Foundation Dmytro Kaidalov dmytro.kaidalov@iohk.io IOG

Roman Oliynykov roman.oliynykov@iohk.io IOG V.N.Karazin Kharkiv National University

January 2024

#### Abstract

The use of zero-knowledge Succinct Non-Interactive Arguments of Knowledge (zk-SNARK) and similar types of proofs has become increasingly popular as a solution for improving scalability, privacy, and interoperability of blockchain systems. However, even with the most advanced proving systems, verifying a single SNARK proof can require a significant amount of computational resources making it expensive to be performed on-chain. This becomes a noticeable bottleneck in scaling SNARK-based applications.

Further efficiency improvement to avoid this bottleneck lies in utilizing distributed recursive proof composition to aggregate multiple existing proofs into one that verifies all underlying proofs.

Building upon this concept, we present a new protocol for decentralized recursive proof aggregation allowing one unique proof to aggregate many input proofs to be efficiently verified on-chain, increasing the throughput and cost efficiency of SNARK-based blockchains. The protocol is designed for decentralized environments where independent actors (provers) can join and contribute to the proof generation process. We also present an incentive scheme for such actors. The protocol is abstract enough to be used with a variety of proving systems that support recursive aggregation.

# Contents

1	Introduction	<b>2</b>
	1.1 Related Work	3
<b>2</b>	Preliminaries	4
3	The General Scheme	7
	3.1 Model	8
	3.2 Merging Protocol	9
	3.3 Decentralized Merging Protocol	12
	3.4 Incentives Protocol	15
	$3.4.1$ Inclusion fee $\ldots$	16
	3.4.2 Aggregation fee	17
4	Aggregation as a Service	17
	4.1 Functional and Security Requirements	18
	4.2 Details of Recursive Aggregation	18
	4.2.1 Base SNARK	19
	4.2.2 Merge SNARK	19
	4.2.3 Final SNARK	21
	4.3 Aggregation Requests and User Accounts	22
	4.4 Aggregated Proof Submission	23
	4.5 Rewards Withdrawal	24
5	Further Scaling with Fast Proof Gossiping	<b>25</b>
6	Conclusions	26

# 1 Introduction

In recent years, the utilization of zk-SNARKS[7, 10, 21, 12] and STARKs[6, 26] has witnessed a significant upsurge across various domains. These innovative methods have addressed critical challenges and unlocked new possibilities for enhancing privacy, scalability, and interoperability in the realm of blockchain technology and beyond.

One notable application is the implementation of zk-EVM rollups [4, 33, 25, 2], which leverage zero-knowledge proofs to facilitate scalability. By compressing transaction data while retaining its validity, zk-EVM rollups enhance efficiency without compromising security. Another application is the SNARK-based sidechains [16, 17] and bridges [31] that enable seamless communication between disparate blockchain networks, fostering secure and trustless interoperability. Succinct blockchains with fully proven state also heavily rely on SNARKs [14, 9]. In addition to that, SNARKs can play a crucial role in enabling use cases where user data privacy is required while engaging in decentralized applications [3, 1].

Furthermore, the fusion of zero-knowledge principles with Artificial Intelligence (AI) holds immense potential. It allows AI models to be employed without the need for on-chain execution. Instead, only the verification of the executed result occurs on-chain, reducing computational overhead while maintaining the integrity of the AI processes. As regulatory compliance becomes increasingly important, digital identity will become essential in many use cases. Zero-knowledge proofs will be key in maintaining a balance between privacy and meeting new regulatory standards.

Given these developments, it is clear that blockchain networks must accommodate an everincreasing volume of transactions reliant on zero-knowledge proofs. The throughput of these networks will be heavily influenced by their efficiency to verify proofs at scale. However, the verification of a single zero-knowledge proof, even utilizing the most efficient proving systems, can consume a substantial portion of the computational budget for verifying the entire block, and it might exceed the computational limit for individual transaction verification [28, 29].

This situation is further exacerbated when a specific use case requires the usage of proving systems with efficient proof aggregation that generally are much less efficient in verification speed and proof size. For instance, consider ZK rollups, such as those employing zkEVM with STARKs<sup>[25]</sup> or sidechain bridges with fully proven states<sup>[17]</sup>. In these scenarios, achieving verification performance suitable for the target blockchain requires converting the proof by performing so-called proof wrapping. This process can take several hours, hindering many use cases. The situation becomes even worse considering the need to support many ZK-based applications such as rollups, cross-chain bridges, privacy preserving transactions, etc.

An approach that can address these challenges consists of constructing a system that leverages the advantages of efficient recursive proof composition[11, 14, 9]. Recursive proof composition involves creating new proofs that encapsulate and verify multiple existing proofs. By aggregating the proofs, it's possible to substantially reduce the verification cost and improve the scalability of a blockchain system supporting ZK-enabled transactions. The recursive approach can optimize the verification process and mitigate the issues introduced by expensive proof wrappings mentioned previously.

Building upon this concept, we present a scalable and robust protocol for decentralized recursive proof aggregation. It allows one unique proof aggregating many proofs for different transactions. The transactions can be totally unrelated (e.g., some transactions can use ZK for protecting user data privacy, some for compliance purposes, and others using ZK to validate a zk-rollup state update) and may even utilize different proving systems. The resulting proof could be verified more efficiently having a constant verification time independent from the number of aggregated proofs. This would not only enhance the scalability and efficiency of a blockchain system but also make it more feasible for use cases that require low latency, removing the need for expensive proof wrappings.

Such a protocol can be implemented as a service on top of an existing blockchain system. It is designed to work in a decentralized environment where independent actors (provers) can join and contribute to the recursive proof generation process. The protocol is designed to create competition between provers in order to incentivize fair cost of proof generation. As the proof aggregation process requires the creation and dissemination of many intermediate proofs, we also discuss how to avoid expensive proof verification during broadcasting, which further improves the efficiency of the aggregation process.

### 1.1 Related Work

A lot of research effort has been devoted to the problem of improving the efficiency of proving systems. Active use of SNARKs and STARKs in blockchain systems continues to attract attention to this field. The main improvement directions are:

- 1. Developing more efficient proving systems [27, 21, 6, 13, 22, 30].
- 2. Distributing generation of a single proof [15, 23].

3. Aggregation of several proofs for efficient verification [14, 32, 8].

Here we briefly discuss some of the relevant works.

[15] focuses on distributing the generation of a single SNARK proof by secret-sharing the witness and utilizing multi-party computation protocols to distribute proof generation. Contrary, we are focusing not on distributing the computational work of generating a particular proof but on distributing the recursive aggregation of many ZK proofs so that each server will work on its own proof. The framework presented in [15] can complement our work by helping generate base proofs that are aggregated using our protocol.

Mina [14, 9] introduced the notion of a succinct blockchain relying on recursive SNARKs to prove state transitions. Mina's approach to decentralized proof generation shares a few aspects with the goal we want to achieve. The main difference is that Mina produces aggregated proofs for proving state transitions and this implies coordinating the aggregation based on a predefined order of transactions. On the contrary, our objective is to aggregate independent proofs created by users or external systems. The absence of dependencies enables certain design possibilities for the protocol.

[32, 8] rigorously analyzes decentralized proof creation in the context of proving state transitions in sidechains. As with Mina, the main difference with our work is that we consider independent proofs while proving state transitions introduces dependency among aggregated proofs.

# 2 Preliminaries

Throughout the paper, whenever we use the term "proof" or "ZK proof" we essentially mean Succinct Non-Interactive Argument of Knowledge (SNARK). However, the presented algorithms and protocols are generic enough to be used with other similar proving systems, such as STARKs. In what follows we introduce some relevant definitions and notations

In what follows we introduce some relevant definitions and notations.

**Definition 2.1. Collision-Resistant Hash Function (CRH).** A hash function H is collision-resistant if the probability of finding two different input strings a and b such that H(a) = H(b) is negligible (a more formal definition can be found, e.g., in [18]).

Whenever we refer to a hash function, we suppose it is collision-resistant.

**Definition 2.2. Strict Binary Tree (SBT).** A strict binary tree is a tree in which every node is either a leaf or has exactly two children (left and right) (see Fig. 1).



Figure 1: Examples of strict binary trees.

Any mention of a binary tree assumes a strict binary tree.

**Definition 2.3. Merkle Hash Tree (MHT).** The Merkle Hash Tree, or simply Merkle Tree (MT), is a strict binary tree where the value of an internal node is computed as the hash of values of its children, and the value of a leaf node is the direct hash of a data block represented by this leaf (see Fig. 2) [24].



Figure 2: Merkle Hash Tree.

We call the top-level node  $(h_1 \text{ in Fig. 2})$  the root hash of the MHT. Given that a collisionresistant hash function is used to calculate tree nodes, we can consider root hash as a tree authenticator: it is impossible to tamper even a single bit of data in the tree without also changing the root hash.

An important feature of the Merkle tree structure is that it produces a concise proof of a particular data block's membership in a tree with the particular root hash. E.g., if one wants to prove that  $data_4$  (Fig. 2) is included in the MHT tree with the root hash  $h_1$ , they just need to provide a verifier with the data block along with a tuple of internal nodes  $(h_{43}, h_{31}, h_{22})$  that will allow recalculating the tree root and comparing it to the provided root  $h_1$ . We call it **Merkle proof**.

#### Definition 2.4. Succinct Non-Interactive Argument of Knowledge (SNARK).

A SNARK is a proving system consisting of a triplet of algorithms (Setup, Prove, Verify) that allows proving satisfiability of a set of inputs to an arithmetic constraint system. An arithmetic constraint system C is a set of constraints for a specific computation. We indicate a satisfying assignment as C(a, w), where a is a public input and w is a witness (see, e.g., [30] for more formal definition).

The algorithms (Setup, Prove, Verify) are defined such that

- 1.  $(pk, vk) \leftarrow Setup(C, 1^{\lambda})$  bootstraps SNARK for a constraint system C under security parameter  $\lambda$ . The bootstrapped SNARK is specified by a pair of keys (pk, vk) which are a proving key and a verification key correspondingly.
- 2.  $\pi \leftarrow Prove(pk, a, w)$  evaluates a proof  $\pi$ , which confirms that (a, w) is a satisfying assignment for C.
- 3.  $true/false \leftarrow Verify(vk, a, \pi)$  verifies that  $\pi$  is a valid proof attesting to the satisfying assignment (a, w) for the constraint system C.

Algorithms (Setup, Prove, Verify) satisfy the following properties:

1. Completeness. For any constraint system C and (a, w), if  $\pi \leftarrow Prove(pk, a, w)$  is a valid proof, then  $Verify(vk, a, \pi)$  is always true.

- 2. Knowledge soundness. If a pair (a, w) is not a satisfying assignment for C, then the probability of obtaining  $\pi$  such that  $Verify(vk, a, \pi) = true$  is negligible.
- 3. Succinctness. For every constraint system C bootstrapped with (pk, vk) and every  $a \in F^r$ , the size of a proof and verification time is polynomial in  $\lambda$ .

**Definition 2.5. Recursive proof aggregation.** Let assume we have two independent SNARK proofs represented by tuples  $\mathbf{p}_1 := (vk^A, (a_1^A, \pi_1^A))$  and  $\mathbf{p}_2 := (vk^B, (a_2^B, \pi_2^B))$ , where  $vk^A$  and  $vk^B$  are verification keys of some arbitrary SNARK systems A and B, and pairs  $(a_1^A, \pi_1^A)$  and  $(a_2^B, \pi_2^B)$  are concrete public inputs and proof evaluations for them. The recursive proof aggregation is realized by a special *Merge* **SNARK** that proves the following statement:

"given verification keys  $vk^A$  and  $vk^B$  and public inputs  $a_1^A$  and  $a_2^B$ , there exist  $(\pi_1^A, \pi_2^B)$  such that  $Verify(vk^A, a_1^A, \pi_1^A)$  and  $Verify(vk^B, a_2^B, \pi_2^B)$  evaluates to true".

The algorithms (Setup, Prove, Verify) for the MergeSNARK are defined as follows:

- 1.  $(pk^{Merge}, vk^{Merge}) \leftarrow Setup(C_{Merge}, 1^{\lambda})$ , where  $C_{Merge}$  implements verification of aggregated proofs.
- 2.  $\pi^{Merge} \leftarrow Prove(pk^{Merge}, a^{Merge}, w^{Merge})$ , where
  - public input  $a^{Merge}$  commits to public inputs  $a_1^A, a_2^B$  and verification keys  $vk^A, vk^B$  of the aggregated proofs,
  - witness  $w^{Merge}$  contains aggregated proofs  $\mathbf{p}_1$  and  $\mathbf{p}_2$ .
- 3.  $true/false \leftarrow Verify(vk^{Merge}, a^{Merge}, \pi^{Merge})$  attests the statement defined above.

Note that the public input  $a^{Merge}$  and witness  $w^{Merge}$  may contain some additional information not related to the aggregated proofs.

By applying the *MergeSNARK* recursively not only to some arbitrary input proofs, but also to Merge proofs themselves, it is possible to construct a single SNARK proof that attests to many base input proofs.

Note that the terms "aggregated proof" and "merged proof" are used interchangeably throughout the paper.

The notation  $\mathbf{p}_i$  is used to denote some abstract proof when the concrete details are not important.  $\mathbf{p}_i^{base}$  denotes some base input proof that is to be aggregated.  $\mathbf{p}_{i;j}^{agg}$  denotes an aggregated proof obtained using Merge SNARK, where i:j identifies the range of aggregated base proofs (e.g.,  $\mathbf{p}_{1:4}^{agg}$  denotes an aggregated proof that verifies 4 input proofs  $\mathbf{p}_{1}^{base}, \mathbf{p}_{3}^{base}, \mathbf{p}_{3}^{base}, \mathbf{p}_{4}^{base}, \mathbf{p}_{4}^{base},$ 

The aggregated proofs can be represented as strict binary trees where leaves are base input proofs and all intermediate nodes are themselves aggregated proofs (see example in Fig. 6).

We define the weight  $w(\mathbf{p}_i)$  of the proof  $\mathbf{p}_i$  as the number of base proofs aggregated by it (i.e., the number of leaves in its tree representation). By convention, the weight of a base proof equals one:  $w(\mathbf{p}_i^{base}) = 1$ .

# 3 The General Scheme

In the proposed design, we introduce aggregation as a service, where a user delegates the task of aggregating a ZK proof for his transaction along with other proofs submitted by other users. The result of such aggregation is going to be submitted on-chain on a regular basis, eliminating the need to directly include the original users' proofs.

The aggregation service consists of on-chain and off-chain components. The off-chain component primarily manages the aggregation process: it handles user requests and continuously schedules proofs merging. The on-chain component manages submission and verification of the aggregated proof, charges users and pays out rewards for the aggregation work done by provers and other involved actors.

The basic flow is shown in Fig. 3. In order to participate in the protocol, all actors (e.g., provers) should be registered on-chain. The on-chain component can be implemented, for instance, as a smart contract, or integrated natively in the consensus protocol of a blockchain system.

In order to describe the flow, let's take as an example an on-chain application that leverages ZK proofs to protect users' data privacy. In such a scenario, in order to interact with the application, a user should provide a ZK proof along with the transaction. In our model, the user instead of directly submitting the transaction with the proof, will first submit an off-chain request to aggregate it. When the aggregation is done, the final aggregated proof is submitted on-chain. At this point the user will be automatically charged for the successful aggregation work and will be able to submit his transaction on-chain referring to the aggregated proof instead of including the initial base proof directly. This process can be made totally transparent to the user by implementing a well-designed user interface.



Figure 3: Basic flow of the aggregation service.

This section focuses mostly on the aggregation process defining the necessary protocols. The next section will provide details on how this can be realized in a blockchain setting. We identify the following core elements of the recursive proof aggregation:

1. Merging protocol. It defines the strategy for building the aggregation proof tree. There can be many strategies, our goal is to define the optimal one which provides the best

throughput, resilience to malicious actions, and balanced utilization of the blockchain for submissions.

2. Incentives protocol. It defines the incentives system for the involved actors (e.g., provers) such that it facilitates honest and cost-optimized execution of the merging protocol.

### 3.1 Model

We identify the following actors participating in the proof aggregation:

- 1. **Submitters.** They submit final aggregated proofs on-chain. From the merging protocol perspective their task is to pick up an aggregated proof and include it into a block or submit to the smart contract (depending on the implementation).
- 2. Schedulers. Special entities that coordinate the proof aggregation process. Specifically, their task is to maintain a sequence of proofs and provide a schedule defining who, how, and when make the computational work of merging proofs.
- 3. **Provers.** The actual workers who perform the task of merging proofs according to the schedule provided by schedulers.
- 4. Users. Users submit requests for the ZK proofs they want to be aggregated. When the corresponding ZK proof is aggregated and submitted on-chain, the user can submit a transaction referring to the aggregated proof.

Let proof submitters, schedulers, and provers be represented by the corresponding sets **B**, **S**, and **P**. Let  $N_{\mathbf{B}} = |\mathbf{B}|$ ,  $N_{\mathbf{S}} = |\mathbf{S}|$ , and  $N_{\mathbf{P}} = |\mathbf{P}|$  be, correspondingly, the number of submitters, schedulers, and provers in the system. We assume that the honest majority assumption holds for the sets of proof submitters and schedulers.

Let  $\mathbf{Q}_{zkp} = (\mathbf{p}_i)_{i\geq 1}$  be a sequence of ZK proofs to be aggregated. It consists of both newly requested base proofs and already merged proofs that have not been submitted on-chain and wait for further aggregation.

The main goal of the protocol is to merge many base proofs  $\mathbf{p}_{isg}^{base}$  so that they can be included and verified on-chain as part of a single aggregated proof  $\mathbf{p}_{i:j}^{agg}$ . We assume that sets **B**, **S**, and **P** constitute a global state, so that all actors in the system have a consistent view of them. On the other hand, each actor has its own view of  $\mathbf{Q}_{zkp}$  (it is never committed on-chain). We denote the local state of party p as  $\mathbf{Q}_{zkp}^{p}$ . In many cases there will be no difference in local views  $\mathbf{Q}_{zkp}$  of different parties, so we will not explicitly differentiate them.

The protocol operates in the environment where time is divided into slots  $sl_i$  of constant duration. The time of the slots is synchronized among all actors.

We assume a model where messages are delivered to all network participants by the end of the slot where they have been issued. E.g., if a proof  $\mathbf{p}_{1:2}^{agg} \leftarrow merge(\mathbf{p}_1, \mathbf{p}_2)$  has been scheduled and produced in slot  $sl_0$ , then all actors will receive  $\mathbf{p}_{1:2}^{agg}$  until the end of  $sl_0$ .

Every slot  $sl_i$  is assigned with a single proof scheduler  $s_i \in \mathbf{S}$ . Moreover, all slots are grouped into submission epochs of length L (L is a system parameter). Every submission epoch  $E_i$  is assigned with a single submitter  $sb_k \in \mathbf{B}$ . The slot  $sl_i$  belongs to the epoch  $E_j$  iff  $j = \lfloor \frac{i}{L} \rfloor$  (see Fig. 4).

Note that the exact assignment protocol is out of scope for this research and depends on the particular platform where the aggregation service is implemented. We formalize the assignment procedure through the functions *scheduler* :  $N \to \mathbf{S}$  and *submitter* :  $N \to \mathbf{B}$  which, given a slot

Slot	sl <sub>o</sub>	sl1	sl <sub>2</sub>	sl3	sl4	sl <sub>5</sub>	sl <sub>6</sub>	sl7	sl <sub>8</sub>	sl <sub>9</sub>	sl <sub>10</sub>	sl <sub>11</sub>	sl <sub>12</sub>	sl <sub>13</sub>	sl <sub>14</sub>	<b>sl</b> <sub>15</sub>
Scheduler	<b>S</b> <sub>1</sub>	s3	<b>s</b> <sub>7</sub>	<b>s</b> <sub>5</sub>	<b>s</b> <sub>7</sub>	<b>S</b> <sub>11</sub>	s <sub>12</sub>	s <sub>15</sub>	<b>S</b> <sub>3</sub>	<b>S</b> <sub>4</sub>	s <sub>9</sub>	s <sub>8</sub>	s <sub>13</sub>	<b>S</b> <sub>10</sub>	<b>s</b> <sub>1</sub>	<b>s</b> <sub>2</sub>
<u></u>																
Submission epoch $E_0$								Submission epoch $E_1$								
Submitter sb <sub>1</sub>								Submitter sb <sub>2</sub>								

Figure 4: Example of slots and submission epochs. There are two epochs of length L = 8. Submitters  $sb_1$  and  $sb_2$  are assigned to epochs  $E_0$  and  $E_1$  correspondingly. Moreover, every slot is assigned with a single scheduler.

or epoch, returns correspondingly the scheduler  $s_i$  assigned to the slot or submitter  $sb_i$  assigned to the epoch (e.g., in Fig. 4 scheduler $(sl_{10}) = s_8$ , submitter $(E_1) = sb_2$ ).

We assume that at most one proof from  $\mathbf{Q}_{zkp}$  is extracted and submitted on-chain during every submission epoch. The submitter is allowed (but not obliged) to submit on-chain exactly one aggregated proof during his epoch. Note that for doing this the submission epochs should be synchronized on-chain such that it is possible to explicitly define the submission period. For instance, it can be done by attaching epoch boundaries to slots or blocks in the underlying blockchain system. The exact mechanism is out of scope for this research and highly depends on the blockchain system where the protocol will be implemented.

Every slot the following actions are performed:

- 1. A proof scheduler assigned for slot  $sl_i$  issues a schedule  $sch_i$  at the beginning of the slot. The schedule defines which provers should merge what ZK proofs.
- 2. A prover  $pr_k \in \mathbf{P}$ , if assigned to merge two proofs  $\mathbf{p}_i$  and  $\mathbf{p}_j$ , does it and disseminates the resulting merged proof  $\mathbf{p}_{i,j}^{agg} \leftarrow merge(\mathbf{p}_i, \mathbf{p}_j)$  to other network participants.

We assume that dissemination of the schedule and merged proofs are done within a single slot.

Every submission epoch  $E_e$  the following action is performed:

• The submitter of epoch  $E_e$  takes one aggregated proof, scheduled by some  $sch_i \in E_e$ , and submits it on-chain.

Without loss of generality, let's assume that schedulers do it at a particular slot during their epoch. For instance, the aggregated proof is picked up from  $\mathbf{Q}_{zkp}$  and submitted on-chain in the last slot of the epoch. Then, the submission cadence will be strictly  $c_{subm} = L$  slots. We will call such a slot a submission slot.

# 3.2 Merging Protocol

In this section we put aside the schedulers and discuss what will be the most efficient strategy to build an aggregated proof. We define efficiency by two metrics:

- the average number of ZK proofs that can be aggregated and submitted on-chain per unit of time,
- balanced distribution of the number of base proofs aggregated by submitted proofs.

Let's define the following theorem which will help us to select an efficient merging protocol.

**Theorem 1.** Given a set of provers  $\mathbf{P}$ , aggregated proof submission period  $c_{subm} = L$  (in slots), and assuming a prover can generate only one proof per slot, the maximal average number of base proofs that can be aggregated by submitted proofs is  $|\mathbf{P}| \cdot c_{subm} + 1$ .

*Proof (sketch).* Any merging strategy assumes building some aggregated proof which can be represented as a strict binary tree where internal nodes are merged proofs and leaves are base proofs. A strict binary tree with  $N_l$  leaves contains  $2N_l - 1$  total nodes of which  $N_l - 1$  are non-leaf nodes representing merged proofs. So, independently of the strategy used to build the tree, it requires  $N_l - 1$  mergings to aggregate  $N_l$  base proofs.

Given  $|\mathbf{P}|$  provers and  $c_{subm}$  slots per submission, the overall computational capacity of provers is  $|\mathbf{P}| \cdot c_{subm}$  proofs per submission, thus the maximal average number of base proofs that can be aggregated per submission is  $|\mathbf{P}| \cdot c_{subm} + 1$ .

Now let's define a concrete merging protocol.

# Algorithm 1: Merging Protocol

#### **Definitions:**

- $\mathbf{Q}_{zkp}$  is a sequence of elements indexed from 1 (i.e.,  $\mathbf{Q}_{zkp}[1]$  is a head element)
- $\mathbf{p}_x \leftarrow remove(\mathbf{Q}_{zkp}, j)$  removes j's element from the sequence  $\mathbf{Q}_{zkp}$  and stores it in the variable  $\mathbf{p}_x$
- $-insert(\mathbf{Q}_{zkp}, j) \leftarrow \mathbf{p}_x$  inserts proof  $\mathbf{p}_x$  at position j in  $\mathbf{Q}_{zkp}$

#### Initialization:

 $\mathbf{Q}_{zkp} = [\mathbf{p}_1^{base}, \mathbf{p}_2^{base}, \mathbf{p}_3^{base}, \dots] - \text{initial infinite sequence of base proofs}$  $\mathbf{P} = \{pr_1, pr_2, \dots, pr_k\} - \text{fixed set of provers}$ 

for every slot  $sl_i$  do

 $\begin{array}{c|c} \mathbf{if} \ sl_i \ is \ a \ proof \ submission \ slot \ \mathbf{then} \\ & \mathbf{p}^{\{base, agg\}} \leftarrow remove(\mathbf{Q}_{zkp}, 1) \\ & \text{Submit} \ \mathbf{p}^{\{base, agg\}} \ \text{on-chain as an aggregated proof} \\ \textbf{for} \ j \leftarrow 1 \ \mathbf{to} \ |\mathbf{P}| \ \mathbf{do} \\ & \mathbf{p}_x \leftarrow remove(\mathbf{Q}_{zkp}, j) \\ & \mathbf{p}_y \leftarrow remove(\mathbf{Q}_{zkp}, j+1) \\ & \mathbf{p}_{x,y}^{agg} \leftarrow merge_{pr_j}(\mathbf{p}_x, \mathbf{p}_y) \\ & \text{insert}(\mathbf{Q}_{zkp}, j) \leftarrow \mathbf{p}_{x,y}^{agg} \end{array} \right) // \text{designated pair of proofs for every prover } pr_j \\ \end{array}$ 

The example execution is represented in Fig. 5. It shows how the sequence  $\mathbf{Q}_{zkp}$  evolves as slots pass. Fig. 6 represents binary trees of merged proofs that are submitted on-chain.

The described algorithm defines the merging strategy. Note that it is not adopted to work in a decentralized environment yet. Later we will see how to turn this protocol to work in a decentralized setting with help of schedulers. For now our goal is to analyze whether it satisfies necessary efficiency properties.

**Lemma 2.** The protocol described by Algorithm 1 provides throughput of  $|\mathbf{P}| \cdot c_{subm} + 1$  aggregated base proofs per one submitted on-chain proof.



Figure 5: Merging protocol execution example in a setting with  $|\mathbf{P}| = 4$  provers and  $c_{subm} = L = 2$  aggregation submission cadence.



Figure 6: Continuation of example in Fig. 5: tree representation of aggregated proofs submitted at slots 2,4, and 6.

*Proof (sketch).* After the warm-up period the submitted aggregated proofs will always represent strict binary trees with a certain repeated structure containing  $|\mathbf{P}| \cdot c_{subm} + 1$  leaves. Given that  $|\mathbf{P}| \cdot c_{subm} + 1$  per proof is a maximal achievable aggregation throughput due to Theorem 1, thus Algorithm 1 is optimal. The detailed analyses of proof trees generation has been done in [32]. We refer an interested reader to Section 4 in [32] which analyses a similar merging protocol.

Now let's see how to turn Algorithm 1 to work in a decentralized environment.

#### 3.3 Decentralized Merging Protocol

Algorithm 1 defines a merging protocol. But it is not adopted for the decentralized environment: it does not address how provers are assigned to work on proofs from  $\mathbf{Q}_{zkp}$  and how they are incentivized.

We address these issues in the following way:

- 1. The aggregation process is coordinated by schedulers which assign provers to merge particular proofs according to their local views  $\mathbf{Q}_{zkp}^{s_i}$ . At the beginning of each slot an assigned scheduler issues a schedule and shares it with all network participants.
- 2. The newly generated aggregated proofs are then propagated through the network till the end of the slot where the schedule appeared. Every scheduler and prover update their local views of  $\mathbf{Q}_{zkp}$  with newly generated proofs.
- 3. Every submission epoch, an assigned submitter extracts one proof from his local  $\mathbf{Q}_{zkp}^{sb_i}$  and submits it on-chain.

Note that under the assumption of messages dissemination till the end of the slot, all participants should have the same view of  $\mathbf{Q}_{zkp}$ . In what follows we may omit referring to local views  $\mathbf{Q}_{zkp}^{s_i}$  of particular parties and instead refer to  $\mathbf{Q}_{zkp}$  assuming local views are consistent.

As we will see later in more details, we can also assume that schedulers do not need to know when and what proof from  $\mathbf{Q}_{zkp}$  is going to be submitted on-chain, so they can continue scheduling proofs aggregation without having to coordinate with the submission process. Moreover, as we will see in Section 3.4, both schedulers and provers are incentivized to aggregate further also the proofs that might already be included on-chain. This means that the submitted aggregated proofs might partially overlap in terms of base input proofs they aggregate. For now, we just assume that the on-chain aggregation service logic handles this by disabling the overlapped subtree. We will discuss it in more detail in the following sections.

Now let's define the decentralized version of the merging protocol more formally. We start by defining a schedule:

$$sch_n := (se_1, se_2, \dots, se_m),$$

which is a sequence of schedule entries  $se_i$  that assign a particular prover  $pr_j \in \mathbf{P}$  to merge 2 proofs  $\mathbf{p}_x$  and  $\mathbf{p}_y$  from  $\mathbf{Q}_{zkp}$ :

$$se_i := (pr_j, (\mathbf{p}_x, \mathbf{p}_y)).$$

Then the decentralized merging protocol is defined by Algorithm 2.

#### Algorithm 2: Decentralized Merging Protocol

#### **Definitions:**

- scheduler :  $N \to \mathbf{S}$  an assignment procedure such that for every slot  $sl_i$  there is a single scheduler  $s_i \leftarrow scheduler(sl_i)$
- submitter :  $N \to \mathbf{B}$  an assignment procedure such that for every submission epoch  $E_i$ there is a single submitter  $sb_j \leftarrow submitter(E_i)$

#### Initialization:

$\mathbf{Q}_{zkp} = [\mathbf{p}_1^{base}, \mathbf{p}_2^{base},, \mathbf{p}_p^{base}]$	- initial sequence of base proofs
$\mathbf{P} = \{pr_1, pr_2, \dots, pr_k\}$	– fixed set of provers
$\mathbf{S} = \{s_1, s_2,, s_c\}$	– fixed set of schedulers
$\mathbf{B} = \{sb_1, sb_2, \dots, sb_b\}$	– fixed set of submitters

for every slot  $sl_i$  every scheduler  $s_i \in \mathbf{S}$  do

if  $s_j = scheduler(sl_i)$  is an assigned scheduler for  $sl_i$  then At the beginning of the slot,  $s_j$  issues a  $sch_i := (se_1, se_2, ..., se_m)$ , assigning, at his discretion<sup>*a*</sup>, the provers from **P** to the pairs of proofs from  $\mathbf{Q}_{zkp}^{s_j}$ . The pairs of proofs for merging are selected as in Algorithm 1.

for every slot  $sl_i$  every submitter  $sb_i \in \mathbf{B}$  do

if  $sl_i \in E_k$  such that  $sb_j = submitter(E_k)$  and  $sl_i$  is a proof submission slot for  $sb_j$ then

 $sb_j$  takes the head proof from  $\mathbf{Q}_{zkp}^{sb_j}$  and submits it on-chain

for every slot  $sl_i$  every prover  $pr_i \in \mathbf{P}$  do

 $pr_i$  observes  $sch_i$  created by  $s_i = scheduler(sl_i)$ 

**if** there is  $se_n \in sch_i$  such that  $se_n = (pr_j, (\mathbf{p}_x, \mathbf{p}_y))$  **then**   $\[ pr_j \text{ computes } \mathbf{p}_{x,y}^{agg} \leftarrow merge(\mathbf{p}_x, \mathbf{p}_y) \text{ and broadcasts } \mathbf{p}_{x,y}^{agg} \text{ to the network} \]$ 

for every slot  $sl_i$  every actor  $p \in {\mathbf{S} \cup \mathbf{P} \cup \mathbf{B}}$  do

Observe the network and update its local view  $\mathbf{Q}_{zkp}^{p}$  as follows:

if there is a new valid proof  $\mathbf{p}_{x,y}^{agg} \notin \mathbf{Q}_{zkp}^{p}$  that has been generated by  $pr_{j}$  according to some  $se_n = (pr_j, (\mathbf{p}_x, \mathbf{p}_y)) \in sch_i$  then

 $\[ \]$  Remove  $\mathbf{p}_x$  and  $\mathbf{p}_y$  from  $\mathbf{Q}_{zkp}^p$  and insert  $\mathbf{p}_{x,y}^{agg}$ 

if there is a new valid proof  $\mathbf{p}_v^{base} \notin \mathbf{Q}_{zkp}^p$  that has been requested for aggregation

then

| Insert  $\mathbf{p}_v^{base}$  in  $\mathbf{Q}_{zkp}^p$ 

Example in Fig. 7 shows how the proofs are merged and how the sequence  $\mathbf{Q}_{zkp}$  evolves as slots pass. Every slot has its own assigned scheduler which issues a schedule according to his local view  $\mathbf{Q}_{zkp}^{s_i}$ . The submission epoch length is L = 2, so that every second slot the

<sup>&</sup>lt;sup>a</sup>It may be based on the prover cost and other observed factors

assigned submitter takes one aggregated proof and submits it on-chain. Note that submitted proofs continue being aggregated so that schedulers do not have to observe the chain to avoid aggregation of proofs already appeared on-chain. This is an important feature as it keeps two processes of scheduling and submitting completely independent.



Figure 7: Example of the decentralized merging protocol execution in a setting with  $|\mathbf{P}| = 4$  provers and  $c_{subm} = L = 2$  submission cadence. The proofs are aggregated independently whether they are submitted or not.

Fig. 8 continues the example on Fig. 7 and shows the binary tree representing the aggregated proof  $\mathbf{p}_{1:16}^{agg}$  that has been submitted on-chain at slot  $sl_6$ . As seen, the subtree of the proofs with the root  $\mathbf{p}_{1:8}^{agg}$  (in red) has already been submitted on-chain at slot  $sl_4$ . It means that  $\mathbf{p}_{1:16}^{agg}$  validate only base input proofs  $(\mathbf{p}_i^{base})_{9\leq i\leq 16}$  while the subtree under  $\mathbf{p}_{1:8}^{agg}$  is disabled.



Figure 8: Aggregation tree of the proof  $\mathbf{p}_{1:16}^{agg}$  from example on Fig. 7.

As can be seen, Algorithm 2 differs from Algorithm 1 in one important aspect: it also aggregates proofs that have already been submitted on-chain. We argue that it will have a little impact on the throughput compared to Algorithm 1 (detailed analyses can be found in [32]).

Note that Algorithm 2 does not impose strict rules on what proofs from  $\mathbf{Q}_{zkp}$  should be picked up for merging and to which provers they should be assigned. The schedulers do it on their own discretion. But we would like the aggregation process to follow the protocol described

in Algorithm 1, which has been shown efficient. This implies that schedulers should prioritize proofs with higher weight. Enforcing required behavior at the consensus level is challenging and inefficient, thus, we rely on the incentive scheme to encourage schedulers to follow the merging strategy defined by Algorithm 1. In such a way, the aggregation process will produce a structure where at each slot, a subtree of constant size will be merged to an existing tree of proofs (see example in Fig. 9).



Figure 9: Example of the aggregation process in the case of 2 provers. The submission cadence is every second slot, the black left arrow denotes the proof being submitted on-chain. Dotted circles represent base input proofs obtained from users. Solid circles represent aggregated proofs.

Regarding the selection of provers, we assume that schedulers do it on their own discretion (for instance, the most cost efficient and reliable provers are selected). The following section describes an incentive scheme adjusted to encourage the desired behavior.

### 3.4 Incentives Protocol

The protocol from Section 3.2 defines the set of actors and merging strategy that provides maximal throughput. The crucial component to realize this protocol is how to incentivize actors to follow the merging strategy.

We assume that the only incentive to participate in the protocol is the direct fees collected from users wanting their base proofs being aggregated. Thus, we need a protocol on how to distribute the fees collected from users.

An important observation here is that the number of proofs that is generated during the aggregation process is approximately the same as the number of initial base proofs (e.g., see example in Fig. 6), but their "importance" is different. Moreover, we need to incentivize not only provers, who do the actual job, but also schedulers, who coordinate provers, and submitters, who submit the final aggregated proof on-chain.

We assume that every request for aggregation pays a self-set aggregation fee  $fee_{agg}(\mathbf{p}_i^{base})$ . Moreover, we define  $fee_{agg}(\mathbf{p}^{agg})$  for the aggregated proofs as a remaining fee after deducting the cost required to produce  $\mathbf{p}^{agg}$ . These remaining fees provide incentives for further aggregation.

We also assume that prover  $cost cost(pr_k)$  (the payment for which a prover is willing to do a computational job) is constant and specified upon prover registration in the system<sup>1</sup>.

 $<sup>^{1}</sup>$ Of course nothing prevents from implementing a flexible update mechanism, but the point is that proving

Then, the distribution of fees is done according to the following rules:

#### $. Fee \ distribution \ rules_$

- 1. Every two proofs  $\mathbf{p}_x$  and  $\mathbf{p}_y$  that are scheduled in slot  $sl_i$  by scheduler  $s_j$  to be recursively merged into a single proof  $\mathbf{p}_{x,y}^{agg}$  by a prover  $pr_k$  collectively possess  $fee_{total}(\mathbf{p}_{x,y}^{agg}) = fee_{agg}(\mathbf{p}_x) + fee_{agg}(\mathbf{p}_y)$
- 2.  $fee_{total}$  is splitted into 3 parts defined by system parameters  $CUR, INC, AGG \in (0, 1)$  such that  $AGG \ge 0.5$  and CUR + INC + AGG = 1:
  - (a)  $fee_{cur}(\mathbf{p}_{x,y}^{agg}) = CUR \cdot fee_{total}(\mathbf{p}_{x,y}^{agg})$  is paid to the to the scheduler and prover at the current level for generating  $\mathbf{p}_{x,y}^{agg}$ , it is divided as follows:
    - i. Prover reward:  $r_{pr_k}(\mathbf{p}_{x,y}^{agg}) = cost(pr_k) \leq fee_{cur}(\mathbf{p}_{x,y}^{agg})$  (the prover cost cannot be larger than the amount of fees available for this proof).
    - ii. Scheduler reward:  $r_{s_i}(\mathbf{p}_{x,y}^{agg}) = fee_{cur}(\mathbf{p}_{x,y}^{agg}) r_{pr_k}(\mathbf{p}_{x,y}^{agg}).$
  - (b)  $fee_{inc}(\mathbf{p}_{x,y}^{agg}) = INC \cdot fee_{total}(\mathbf{p}_{x,y}^{agg})$  is reserved for the inclusion of the final aggregated proof on-chain.
  - (c)  $fee_{agg}(\mathbf{p}_{x,y}^{agg}) = AGG \cdot fee_{total}(\mathbf{p}_{x,y}^{agg})$  is reserved for further aggregation (i.e., it becomes the fee of  $\mathbf{p}_{x,y}^{agg}$  at the next level of recursion).
- 3. System parameters CUR, INC, AGG can be defined differently depending on the required properties.

Example in Fig. 10 represents distribution of fees while aggregating 8 base proofs. The system parameters CUR, INC, AGG are set such that at each level 40% is paid to the scheduler and prover, 5% is reserved for the submitter, and 55% always goes to the next level to pay for further aggregation.

#### 3.4.1 Inclusion fee

The inclusion fee is paid to the submitter of the aggregated proof (we also call it submitter's reward), it should at least compensate for the transaction fee on the target blockchain system. The inclusion fees from all the proofs constituting the submitted aggregated proof, except those which have already been submitted, are summed up and paid to the submitter:

$$r_{sb_k}(\mathbf{p}_x^{agg}) = \sum_{\mathbf{p}_y^{agg} \in V} fee_{inc}(\mathbf{p}_y^{agg}),\tag{1}$$

where  $\mathbf{p}_x^{agg}$  is the submitted proof and set V contains all underlying aggregated proofs that has not been submitted yet:

 $V = \{ \mathbf{p}_{y}^{agg} \mid \mathbf{p}_{y}^{agg} \subseteq \mathbf{p}_{x}^{agg} \text{ AND } \mathbf{p}_{y}^{agg} \text{ has not been submitted previously} \}.$ 

Considering example on Fig. 10 let assume that proofs  $\mathbf{p}_{1:4}^{agg}$  and  $\mathbf{p}_{1:8}^{agg}$  has been submitted on chain by  $sb_1$  and  $sb_2$  correspondingly. Then, the reward of  $sb_1$  is the sum of inclusion fees constituting  $\mathbf{p}_{1:4}^{agg}$  (marked with red) and reward of  $sb_2$  is the sum of inclusion fees constituting  $\mathbf{p}_{1:8}^{agg}$  except those underlying  $\mathbf{p}_{1:4}^{agg}$  (marked with green):

$$r_{sb_1}(\mathbf{p}_{1:4}^{agg}) = fee_{inc}(\mathbf{p}_{1:4}^{agg}) + fee_{inc}(\mathbf{p}_{1:2}^{agg}) + fee_{inc}(\mathbf{p}_{3:4}^{agg}) = 31,$$

costs of different provers should be known before the work is scheduled, so that schedulers can decide which provers to choose based on this information



Figure 10: Example of fees distribution assuming the setup CUR = 0.4, INC = 0.05, AGG = 0.55. The inclusion fees from red proofs will be paid to the submitter of  $\mathbf{p}_{1:4}^{agg}$ , while inclusion fees from green proofs will be paid to the submitter of  $\mathbf{p}_{1:8}^{agg}$ .

$$r_{sb_2}(\mathbf{p}_{1:8}^{agg}) = fee_{inc}(\mathbf{p}_{1:8}^{agg}) + fee_{inc}(\mathbf{p}_{5:8}^{agg}) + fee_{inc}(\mathbf{p}_{5:6}^{agg}) + fee_{inc}(\mathbf{p}_{7:8}^{agg}) = 37.54$$

The basic idea behind accumulating inclusion fees is that it can accommodate the changing on-chain fees. The submitter may choose to skip submission if inclusion fees do not cover his expenses, in this case the aggregation would keep going and inclusion fees would accumulate until there is enough to be submitted on-chain.

#### 3.4.2 Aggregation fee

Note that AGG > 0.5 provides us an important feature: the fee paid for merging proofs grows with the weight of the proofs being merged (though, it has an upper bound dependent on the AGG parameter). This creates an incentive for the schedulers to prioritize proofs with larger weight facilitating the merging strategy specified by Algorithm 1.

# 4 Aggregation as a Service

As discussed earlier, aggregation is a service, where a client delegates the task of aggregating a ZK proof for its transaction with proofs submitted by other users thus eliminating the need to include it directly alongside the actual transaction.

So far in Section 3 we established the core elements of the aggregation process. Even though it is designed to work in a decentralized network, it does not address possible malicious behavior of different actors.

This section establishes additional elements required to make the aggregation process secure and reliable in the environment with malicious actors. We also discuss some technical aspects of the recursive proof aggregation in more detail.

# 4.1 Functional and Security Requirements

The aggregation service must satisfy the following requirements:

- 1. Non-repudiation of aggregation requests. Once a user requested aggregation and the off-chain component did the work which ended up in the on-chain submission, the user is obliged to pay for this work. This is an important requirement given that the request and aggregation process are performed off-chain. As we cannot enforce payments off-chain, we have to make sure that the user will eventually pay once the aggregation is done. The violation of this requirement would provide a possibility to overwhelm the service with many junk requests wasting resources and disrupting normal operation.
- 2. Guarantee of rewards. The provers and schedulers should be guaranteed to receive their rewards for the aggregation job once the aggregated proof is submitted on-chain. Such guarantee should not rely on trust to any specific party.

We satisfy these requirements by utilizing the capabilities of recursive ZK proofs. In a nutshell, the idea is to embed the information about user fees and rewards into the aggregation process, such that the submitted aggregated proof, besides verifying underlying base proofs, commits also to the fee and reward information.

The following subsections provide more details on how this can be achieved. Note that we omit many details and in some cases the description may be simplified and incomplete. Our goal is to describe basic principles of the aggregation service.

### 4.2 Details of Recursive Aggregation

In what follows we assume that the type of ZK proofs we are dealing with is SNARK (Succinct Non-interactive Argument of Knowledge). Though, the protocol can also be applied to other types of ZK proofs having similar properties (e.g., STARKs). The idea of recursive proofs has been discussed, e.g., in [14, 9, 11, 16].

In a nutshell, we would like to define a SNARK that attests to several underlying SNARK proofs provided by users. It can be done by applying proof composition techniques. Then we will be able to construct a single SNARK proof that attests to many base proofs.

**Definition 4.1. Recursive SNARKs for the aggregation service.** We define recursive SNARKs composition as a triple of SNARKs (*Base*, *Merge*, *Final*) such that:

- 1. **Base** is a SNARK that wraps arbitrary user SNARK into a unified one that is suitable for further aggregation with *Merge* SNARK.
- 2. Merge is a SNARK that merges two other SNARKs (either *Base* or *Merge*) proving the validity of underlying proofs and additionally committing to the information about the prover, the scheduler, and fees.
- 3. Final is a SNARK that wraps a *Merge* proof before its submission on-chain. The *Merge* proof is not suitable for direct submission, so *Final* SNARK does final preparation and adds additional information.

We detail each of them in the following subsections. In general, the *Base* SNARK is an initial wrap made by a user, the *Final* SNARK is the final wrap made by a submitter, and the *Merge* SNARK is the main aggregation mechanism used by provers and schedulers. Note that the *Merge* SNARK is conceptually the one introduced by Definition 2.5 amended with some additional logic related to fee management. Figure 11 represents different types of SNARKs used in the aggregation process.



Figure 11: SNARKs used in the aggregation process. B - *Base* SNARK, M - *Merge* SNARK, F - *Final* SNARK.

#### 4.2.1 Base SNARK

The Base SNARK is required to unify the representation of arbitrary user SNARKs, so users have to wrap their proofs using Base SNARK before requesting aggregation.

**Definition 4.2. Base SNARK for the aggregation service.** The Base SNARK is defined by a triplet (*Setup*, *Prove*, *Verify*) such that:

- $(pk^{Base}, vk^{Base}) \leftarrow Setup(1^{\lambda})$  bootstraps Base SNARK;
- $-\pi^{Base} \leftarrow Prove(pk^{Base}, a^{Base}, w^{Base})$  where  $a^{Base}$  is a public input that commits to user information, paid fee, and arbitrary user SNARK. The witness  $w^{Base}$  contains data required to verify user request;
- $true/false \leftarrow Verify(vk^{Base}, a^{Base}, \pi^{Base})$  verifies that Base is a valid proof attesting to the validity of an arbitrary user proof.

To request aggregation a user provides a pair  $(a^{Base}, \pi^{Base})$ , where  $a^{Base} = H(user\_info, a^u, vk^u)$  is a public input which commits to user information  $user\_info = (pub\_key, fee, nonce)$  and an arbitrary SNARK proof represented by a verification key  $vk^u$  and public input  $a^u$ .

The witness  $w^{Base} = (\pi^u, sig)$  contains the user SNARK proof and a signature of the message  $m = H(fee, nonce, a^u, vk^u)$ .  $\pi^{Base}$  attests that  $true \leftarrow Verify(vk^u, a^u, \pi^u)$  and that the signature sig is valid.

#### 4.2.2 Merge SNARK

The Merge SNARK underpins the recursive aggregation process. At each level the public input commits to the information about the fees, the prover and scheduler, as well as information about underlying levels.

**Definition 4.3. Merge SNARK for the aggregation service.** The Merge SNARK is defined by a triplet (Setup, Prove, Verify) such that:

- $(pk^{Merge}, vk^{Merge}) \leftarrow Setup(1^{\lambda})$  bootstraps Merge SNARK;
- $-\pi^{Merge} \leftarrow Prove(pk^{Merge}, a^{Merge}, w^{Merge})$  evaluates proof  $\pi^{Merge}$  confirming underlying proofs  $\pi_1^{bm}$  and  $\pi_1^{bm}$  ( $\pi_{\{1,2\}}^{bm} \in w^{Merge}$ ,  $bm \in \{Base, Merge\}$ ) are valid and additionally verifies information about the prover, scheduler, and fees committed in public input  $a^{Merge}$ ;

$$- true/false \leftarrow Verify(vk^{Merge}, a^{Merge}, \pi^{Merge})$$
 verifies that  $\pi^{Merge}$  is a valid proof.

The public input  $a^{Merge}$  is defined as follows:

1 0

$$a^{Merge} \stackrel{\text{def}}{=} (H(aggr\_info), P_{com}, S_{com}), \qquad (2)$$

$$aggr_info \stackrel{\text{def}}{=} (pk_{pr}, fee_{cur}, H(sch_c), aggr_info_1, aggr_info_2),$$

where

$pk_{pr}$	– prover public key,
cost	– prover registered cost of work,
$fee_{cur}$	- current level aggregation fee,
$H(sch_c)$	- schedule hash,
$aggr\_info\_1$	$- aggr_info$ of the 1st underlying Merge proof (or $user_info$ if it is a Base
	proof),
$aggr\_info\_2$	$-  aggr\_info$ of the 2nd underlying Merge proof (or $user\_info$ if it is a Base
	proof),
$P_{com}$	– provers commitment (e.g., Merkle root hash of the prover's list),
$S_{com}$	- schedulers commitment (e.g., Merkle root hash of the scheduler's list).

The Merge SNARK enforces the following rules:

#### Merge SNARK Statement\_\_\_\_

- 1. The proof has been generated by an assigned prover according to the valid schedule.
- 2. The two underlying proofs are valid:

$$true \leftarrow Verify(vk^{\{Merge, Base\}}, a_1, \pi_1),$$

$$true \leftarrow Verify(vk^{\{Merge, Base\}}, a_2, \pi_2),$$

where

$$a_i = a_i^{Merge} = (H(aggr\_info\_i), P_{com}, S_{com}) - \text{if } \pi_i \text{ is a Merge proof, or } a_i = a_i^{Base} = (H(aggr\_info\_i), a^u, vk^u) - \text{if } \pi_i \text{ is a Base proof.}$$

- 3. The current level aggregation fee is the sum of fees passed to the next level from two underlying proofs (as defined in Section 3.4).
- 4. Schedule hash  $H(sch_c)$  represents a valid schedule, which has been issued by an assigned scheduler.

Note that the  $aggr_info$  argument of the public input basically constitutes a root of a Merkle tree with leaves representing base proofs and each node containing additional metadata about the aggregation process. So it commits to the whole aggregation process such that one can

easily show that a certain node is present in the tree by providing a node and corresponding Merkle path (e.g., a prover can easily show that he generated a certain proof and is entitled for a reward for that).

#### 4.2.3 Final SNARK

The Final SNARK is generated by a submitter, it wraps the Merge SNARK. Note that the Merge SNARK represents an aggregated proof for many base proofs. As defined by Algorithm 2 in Section 3.2, the base proofs are aggregated continuously so a certain subtree of the Merge proof could have already been submitted. As previously described such subtree should be disabled and this operation is done by the Final SNARK. The Final SNARK can also allow the usage of another proving system that is more efficient for on-chain verification compared to the one used in Merge SNARK.

**Definition 4.4. Final SNARK for the aggregation service.** The Final SNARK is defined by a triplet (*Setup*, *Prove*, *Verify*) such that:

- $(pk^{Final}, vk^{Final}) \leftarrow Setup(1^{\lambda})$  bootstraps Final SNARK;
- $-\pi^{Final} \leftarrow Prove(pk^{Final}, a^{Final}, w^{Final})$  evaluates a proof  $\pi^{Final}$  that confirms  $\pi^{Merge} \in w^{Final}$  is valid and additionally verifies the submitter info and information about all validated base proofs committed in public input  $a^{Final}$ ;
- $-true/false \leftarrow Verify(vk^{Final}, a^{Final}, \pi^{Final})$  verifies that  $\pi^{Final}$  is a valid proof.

The public input  $a^{Final}$  is defined as follows:

 $a^{Final} \stackrel{\text{def}}{=} (proven\_data[], disabled\_nodes[], H(aggr\_info\_final), P_{com}, S_{com}, sb\_info), \quad (3)$ 

$$base\_data \stackrel{\text{def}}{=} (a^u, vk^u, pub\_key, fee, nonce),$$

where

$proven_data[]$	– an array of <i>base_data</i> entries containing information about the
	aggregated base proofs (including information about the users who
	requested aggregation and fees - it is the same information committed in
	the corresponding $a^{Base}$ ),
$disabled\_nodes[]$	– an array of root nodes of the disabled subtrees,
$aggr\_info\_final$	– a modified aggr_info from the underlying Merge SNARK, where the
	disabled nodes are nullified,
$P_{com}$	– provers commitment (e.g., Merkle root hash of the prover's list),
$S_{com}$	- schedulers commitment (e.g., Merkle root hash of the scheduler's list),
$sb_info$	- information about the submitter.

The Final SNARK enforces the following rules:

#### \_Final SNARK Statement.

- 1. The proof has been submitted by an assigned submitter and there is a valid signature  $sig \in w^{Final}$  made with the private key of the submitter on the message  $m = H(proven\_data[], disabled\_nodes[], aggr\_info\_final, sb\_info).$
- 2. The underlying Merge proof is valid:

 $true \leftarrow Verify(vk^{Merge}, a^{Merge}, \pi^{Merge}),$ 

where  $a^{Merge} = (aggr_info, P_{com}, S_{com}).$ 

- 3. The *aggr\_info\_final* is exactly the same as *aggr\_info* except the nodes provided in *disabled\_nodes* which are nullified.
- 4. The submitter fee is the sum of the inclusion fees of all nodes in the aggregation tree except disabled.
- 5. proven\_data corresponds to actually aggregated base proofs excluding disabled ones.

The Final proof  $\pi^{Final}$  is part of a transaction that is submitted and verified on-chain along with the public input  $a^{Final}$ .  $P_{com}$  and  $S_{com}$  are provided by the on-chain verifier. Once the Final proof is included on chain, the users, which proofs were aggregated, are charged with fees stated in *proven\_data*.

Note that we omitted many details to ease explanation. What is most important is the recursive accumulation of the fee information. Recall from the incentive scheme defined in Section 3.4 that at each level of aggregation, the fees collected from the underlying proofs are splitted among prover, scheduler, submitter, and next level fees. To be able later to provide the rewards to the corresponding actors, the information is embedded into the aggregation information.  $aggr_info_final$  is basically a hash commitment of this information.

### 4.3 Aggregation Requests and User Accounts

As discussed in Section 3, the aggregation request from a user is done off-chain and the aggregation work is performed before the user gets charged for it. The user will be charged only when the transaction with the final aggregated proof is submitted on-chain. This creates a challenge: **how to ensure that users will eventually pay for their aggregation requests?** If it is not adequately addressed then the system will be vulnerable to the denial-of-service attacks.

We address this issue by introducing the following rules:

- 1. The aggregation service maintains an account for each user on-chain. The account must be created and topped up by the user before any off-chain request is submitted. The balance of accounts is controlled by the aggregation service, the user is not able to withdraw assets freely from his account.
- 2. The transaction with the final aggregated proof is accepted on-chain if and only if corresponding user accounts have enough balance to pay for every underlying base proof that has been aggregated.
- 3. The schedulers and provers are incentivized to pick up for aggregation only those requests that come from users with enough on-chain balance. Otherwise they might end up doing work that will not be accepted by the on-chain component and they will not be rewarded.

More formally, the aggregation service maintains an account  $user\_acc$  for every registered user:

$$user\_acc \stackrel{\text{def}}{=} (pub\_key, balance, nonce),$$
 (4)

where

 $pub_key - public key of the user$ 

balance – prepaid amount that has been transferred by the user to the aggregation service

*nonce* – user nonce that is increased with every aggregation request, it is needed to prevent requests replay.

The aggregation request itself is a signed transaction disseminated off-chain:

$$user\_request \stackrel{\text{def}}{=} (pub\_key, fee, nonce, a^u, vk^u, \pi^{Base}), \tag{5}$$

where

- $a^u$ ,  $vk^u$  public input and verification key of an arbitrary user SNARK that is being aggregated
- $\pi^{Base}$  Base proof that verifies underlying user proof and user signature for the message  $msg = H(fee, nonce, a^u, vk^u).$

Upon receiving the request, a scheduler picks it up for processing if and only if there is enough balance in the on-chain user account to pay fees stated in the request and the nonce is higher than the one stored in the account. If there are several requests from the same user being aggregated at the same time, the on-chain balance should be enough to cover all of them.

The prepaid user accounts satisfy an important functional requirement of our aggregation service - **non-repudiation of aggregation requests**.

# 4.4 Aggregated Proof Submission

The aggregated proof can be submitted on-chain once per submission epoch. Each submission epoch has an assigned submitter which can do this. The on-chain component must verify the validity of the submitter.

The aggregated proof submission transaction has the following structure:

$$aggregated\_proof\_transaction \stackrel{\text{def}}{=} (proven\_data[], \ disabled\_nodes[], \\ aggr\_info\_final, \ sb\_info, \ \pi^{Final}), \ (6)$$

where

*proven\_data*[] – an array of *user\_request\_data* entries which contain information about the user and proof that has been aggregated:

$$user\_request\_data \stackrel{\text{def}}{=} (pub\_key, fee, nonce, a^u, vk^u)$$

 $\begin{array}{ll} disabled\_nodes & - \text{ an array of nodes representing the disabled subtrees in the aggregation} \\ & \text{tree} \\ aggr\_info\_final - \text{ a hash of the Merkle tree root of the aggregation info (as defined by [3])} \\ sb\_info & - \text{ information about the submitter (public key and submitter rewards)} \\ & \pi^{Final} & - \text{ the final proof.} \end{array}$ 

# On-chain aggregated\_proof\_transaction processing\_ Transaction Verification:

- 1. Verify that the submitter is a valid one for the current submission epoch.
- 2. Verify final proof:  $true \leftarrow Verify(vk^{Final}, a^{Final}, \pi^{Final})$ , where

 $a^{Final} = (proven\_data[], disabled\_nodes[], aggr\_info\_final, P_{com}, S_{com}, sb\_info).$ 

Note that  $P_{com}$  and  $S_{com}$  are provided directly by the on-chain verifier.

- 3. Verify *disabled\_nodes*[]: the entries must point only to the subtrees that have already been submitted on-chain and that have not been disabled before.
- 4. Verify for each user\_request\_data in proven\_data[] that the corresponding user has enough balance in his on-chain account and that the account nonce is less than the one from user\_request\_data.

If the transaction verification succeeds, the on-chain state is updated as follows:

- 1. For each user\_request\_data in proven\_data[] update the corresponding user account:
  - (a) Decrease the user balance by fee defined in *user\_request\_data*,
  - (b) Update the user nonce with nonce defined in *user\_request\_data*.
- 2. Transfer all collected user fees to the reward pool of the aggregation service.
- 3. Save aggr\_info\_final as a commitment of the aggregation process.

### 4.5 Rewards Withdrawal

The rewards are paid out from the collected fees that are kept by the aggregation service. The onchain component maintains a reward pool from which every prover and scheduler can withdraw their rewards. Recall that every  $aggr_proof\_transaction$  (see (6)) contains an  $aggr\_info\_final$ which commits to the aggregation tree. Each node in the aggregation tree represents a Merge proof and contains information about the prover, scheduler and corresponding fees (e.g., as in Fig. 10). A prover/scheduler can create a withdrawal transaction providing a proof of his work. The proof is basically a reference to a particular  $aggr\_info$  (2) value and corresponding Merkle path.

Doing such a withdrawal for every generated proof or issued schedule will be very expensive to process on-chain. Fortunately, we can rely on SNARKs to provide a single short proof for many withdrawals.

Thus, the aggregation service implements an additional SNARK for withdrawals.

#### Definition 4.5. Reward withdrawal SNARK for the aggregation service.

We define the Reward SNARK such that:

- $(pk^{Rew}, vk^{Rew}) \leftarrow Setup(1^{\lambda})$  bootstraps Reward SNARK;
- $-\pi^{Rew} \leftarrow Prove(pk^{Rew}, a^{Rew}, w^{Rew})$  where  $a^{Rew}$  is a public input containing the information about the withdrawing actor (i.e., prover or scheduler), the reward sum, and an

array of  $aggr_info$  values which commits to the actor's work. The witness  $w^{Rew}$  contains corresponding Merkle paths and aggregation info needed for verification;

-  $true/false \leftarrow Verify(vk^{Rew}, a^{Rew}, \pi^{Rew})$  verifies that  $\pi^{Rew}$  is a valid proof attesting to the validity of the provided data.

Having such a SNARK it is possible to make many withdrawals within a single transaction. Though, the on-chain logic should keep track of what has been already withdrawn by a particular actor to prevent double withdrawal.

Note that here we omit a lot of details, providing just a general idea of how withdrawals can be implemented.

The withdrawal mechanism satisfies one of the main functional requirements of our aggregation service - **guarantee of rewards**. If a prover/scheduler performed a work (e.g., generated some intermediate proof during the aggregation process) that was submitted on-chain, he would always have a possibility to get rewards for this work.

# 5 Further Scaling with Fast Proof Gossiping

Considering that the off-chain component of the aggregation service will rely on a trustless and decentralized p2p network with similar characteristics of blockchain networks, the aggregation protocol throughput will depend on computational complexity of the operations needed for proofs broadcasting. More specifically, in order to protect the network against DDoS attacks, proof broadcasting procedure requires that every node verifies each ZK proof before gossiping it to its neighbors.

In order to further improve the protocol's throughput, we propose an alternative approach for the broadcasting verification process that removes the need for every node to verify each proof. More specifically, we restrict the proof verification requirement only to the prover assigned to merge that specific proof. In other words, proofs are broadcasted into the network without the need to verify them, schedulers will assign the proofs to be merged to the selected provers and only the provers will be required to verify the validity of the assigned proofs. This will significantly speed up broadcasting of proofs and, consequently, the throughput of the aggregation service.

In order to prevent DDoS attacks, we need to make sure that an adversary cannot flood the network with invalid proofs. To prevent this, we introduce a cryptographic proof of misbehavior. Specifically, the cryptographic proof of misbehavior consists of a SNARK named **ProofOfIn-validProof**, proving the invalidity of another proof given some public inputs and a verification key. The main idea is to use such proof to prove the misbehavior and consequently charge the attacker for the misbehavior proving costs in a trustless way. Such cost will be locked by the user in the aggregation request and will be released once the aggregation proof proving the validity/invalidity of the Base proof is included on-chain or after a timeout. Such cost will be calculated applying a multiplier *invalid\_proof\_fee\_mult* (a system parameter) to the self set fee. This system parameter will be calculated based on the ratio between the merge cost and the proof of misbehavior cost.

In order to also improve gossiping of merged proofs but at the same time protecting the system from malicious provers spreading invalid proofs through the network, the same protocol can be applied to the merged proofs broadcasting. In this case each prover must have an aggregation account and merge proofs, similarly to base proofs, will be shared through the network along with the lock needed to cover the cost of invalid proof creation. In such a way the system will be able to charge misbehaving provers but also leverage the fast gossiping protocol for intermediate proofs.

Going into more details, in order to charge only the actual misbehaving actors, it is important to enforce that a specific actor submitted an invalid proof. For this purpose it is necessary to provide the possibility to prove that the invalid proof was submitted by that specific user.

Doing a step back, in the model without this additional scaling improvement, in the merging process it was sufficient to prove that there was a valid proof given the public inputs and the verification key. There was no need to prove the proof was issued by a specific actor, but it was just needed to ensure the user paid for the aggregation work.

In the new setting, in order to charge the actor for misbehavior, we need to ensure there is no possibility for him to repudiate a submitted proof. Along this direction, in order to provide the possibility to the prover to prove that an invalid proof was submitted by a specific actor, the aggregation request must include a signature *non\_rep\_sig* of a message composed by the hash of the proof, the public input and the verification key: *non\_rep\_msg* =  $H(a^u, vk^u, \pi^{Base})$ . In such a way the actor can't repudiate the proof and the prover creating the ProofOfInvalidProof, will also be able to prove that the user submitted an invalid proof. On the other hand such enforcement should be seen also as an important protection for honest users and provers in order to prevent dishonest provers from charging honest participants for invalid proofs that they did not submit.

Going further, after having included in the aggregation request the signature of the hash of the proof, we need to modify the aggregation request broadcast procedure in order to verify the correctness of the signature. Obviously, in order to obtain scaling improvements, the computational cost of signature verification and message hash computation should be lower than the proof verification. In this regard, the analysis of the level of the scaling improvements is strictly related to the specific proving system adopted.

Just as an example, if we consider FRI[5] based recursive efficient proving systems like Plonky2[27] using Poseidon[20] as the hash function, the hash computation is a preponderant part of the proof verification process. Assuming that its needed to hash the same order of magnitude of data in the proof verification process and in the *non\_rep\_msg* computation, in order to produce an improvement in the broadcasting verification process, we should utilize for the *non\_rep\_msg* computation a hash function that is faster in primitive than the one used by the proving system. On the other hand we should also take into consideration that in case of need of creating the ProofOfInvalidProof, the hash of *non\_rep\_msg* should be proven and consequently the proving efficiency of such hash function will affect the *invalid\_proof\_fee\_mult* factor. Considering these tradeoffs, the Monolith hash function[19] seems a good candidate for the *non\_rep\_msg* computation as it provides primitive performances comparable to SHA3 (much faster than Poseidon) and still acceptable proving costs.

# 6 Conclusions

Zero-knowledge techniques play an increasingly important role in the blockchain system allowing a wide range of different applications, such as ZK rollups, privacy-preserving transactions, trustless interoperability between chains, and many more. Nevertheless, despite all developments in optimization of ZK proving systems, their execution on-chain is still expensive.

In the paper we propose the solution to optimize proof verification. It relies on aggregation of many initial proofs from different users into a single one that verifies all of them. By aggregating the proofs, it is possible to substantially reduce the verification cost and improve scalability of a blockchain system. We presented a system for decentralized recursive proof aggregation that can be implemented as a service on top of an existing blockchain system. It is designed to work in a decentralized environment where independent actors (provers) can join and contribute to the recursive proof generation process. The research subject is still under ongoing research. In future work, we plan to uncover more details about specific components and analysis of the proposed construction.

Leveraging the efficiency characteristics of recursive proof composition and constructing a decentralized protocol that embraces these principles holds the potential to revolutionize the verification process for zero-knowledge proofs. As the blockchain landscape evolves to accommodate ever-expanding transaction volumes and diverse use cases, developing such protocols become increasingly important to realize the vision of decentralized, scalable, and privacy-preserving blockchain networks.

# References

- [1] Midnight: new data protection blockchain. https://midnight.network/.
- [2] zkSync: Ethereum zk rollup. https://docs.zksync.io/userdocs/.
- [3] Zcash: Privacy-protecting digital currency, 2016. https://z.cash/.
- [4] All about Polygon zkEVM and zkEVM Rollup, 2022. https://mirror.xyz/msfew.eth/ JJudP\_Kf-IS6VhbF-qU0BUor1Ap6SFEb0TzY0HZ34Rc.
- [5] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Fast reed-solomon interactive oracle proofs of proximity. In Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella, editors, 45th International Colloquium on Automata, Languages, and Programming, ICALP 2018, July 9-13, 2018, Prague, Czech Republic, volume 107 of LIPIcs, pages 14:1–14:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.
- [6] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable Zero Knowledge with No Trusted Setup. In Advances in Cryptology – CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part III, page 701–732, Berlin, Heidelberg, 2019. Springer-Verlag. https: //doi.org/10.1007/978-3-030-26954-8\_23.
- [7] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture. Cryptology ePrint Archive, Paper 2013/879, 2013. https://eprint.iacr.org/2013/879.
- [8] Yuri Bespalov, Alberto Garoffolo, Lyudmila Kovalchuk, Hanna Nelasa, and Roman Oliynykov. Probability models of distributed proof generation for zk-snark-based blockchains. *Mathematics*, 9(23), 2021. https://www.mdpi.com/2227-7390/9/23/3016.
- Joseph Bonneau, Izaak Meckler, Vanishree Rao, and Evan Shapiro. Coda: Decentralized cryptocurrency at scale. Cryptology ePrint Archive, Paper 2020/352, 2020. https:// eprint.iacr.org/2020/352.
- [10] Sean Bowe and Ariel Gabizon. Making Groth's zk-SNARK Simulation Extractable in the Random Oracle Model. Cryptology ePrint Archive, Paper 2018/187, 2018. https: //eprint.iacr.org/2018/187.
- [11] Sean Bowe, Jack Grigg, and Daira Hopwood. Halo: Recursive Proof Composition without a Trusted Setup. Cryptology ePrint Archive, Report 2019/1021, 2019. https://eprint. iacr.org/2019/1021.

- [12] Parno Bryan, Howell Jon, Gentry Craig, and Raykova Mariana. Pinocchio: Nearly practical verifiable computation. In 2013 IEEE Symposium on Security and Privacy, pages 238–252, 2013. https://ieeexplore.ieee.org/document/6547113.
- [13] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Psi Vesely, and Nicholas Ward. Marlin: Preprocessing zkSNARKs with Universal and Updatable SRS. Cryptology ePrint Archive, Paper 2019/1047, 2019. https://eprint.iacr.org/2019/1047.
- [14] Brad Cohn, Evan Shapiro, and Emre Tekişalp. Mina: Economics and Monetary Policy, 2020. https://minaprotocol.com/wp-content/uploads/economicsWhitepaper.pdf.
- [15] Sanjam Garg, Aarushi Goel, Abhishek Jain, Guru-Vamsi Policharla, and Sruthi Sekar. zkSaaS: Zero-Knowledge SNARKs as a Service. Cryptology ePrint Archive, Paper 2023/905, 2023. https://eprint.iacr.org/2023/905.
- [16] Alberto Garoffolo, Dmytro Kaidalov, and Roman Oliynykov. Zendoo: a zk-SNARK Verifiable Cross-Chain Transfer Protocol Enabling Decoupled and Decentralized Sidechains. In 2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS), pages 1257–1262, 2020. https://ieeexplore.ieee.org/document/9355752.
- [17] Alberto Garoffolo, Dmytro Kaidalov, and Roman Oliynykov. Zendoo: a zk-SNARK Verifiable Cross-Chain Transfer Protocol Enabling Decoupled and Decentralized Sidechains. Cryptology ePrint Archive, Paper 2020/123, 2020. https://eprint.iacr.org/2020/123.
- [18] Oded Goldreich. The Foundations of Cryptography Volume 1, Basic Techniques. Cambridge University Press, 2001.
- [19] Lorenzo Grassi, Dmitry Khovratovich, Reinhard Lüftenegger, Christian Rechberger, Markus Schofnegger, and Roman Walch. Monolith: Circuit-friendly hash functions with new nonlinear layers for fast and constant-time implementations. Cryptology ePrint Archive, Paper 2023/1025, 2023. https://eprint.iacr.org/2023/1025.
- [20] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. Poseidon: A new hash function for Zero-Knowledge proof systems. In 30th USENIX Security Symposium (USENIX Security 21), pages 519–535. USENIX Association, August 2021.
- [21] Jens Groth. On the Size of Pairing-based Non-interactive Arguments. Cryptology ePrint Archive, Paper 2016/260, 2016. https://eprint.iacr.org/2016/260.
- [22] Ulrich Haböck, Alberto Garoffolo, and Daniele Di Benedetto. Darlin: Recursive Proofs using Marlin. Cryptology ePrint Archive, Paper 2021/930, 2021. https://eprint.iacr. org/2021/930.
- [23] Alex Ozdemir and Dan Boneh. Experimenting with collaborative zk-SNARKs: Zero-Knowledge proofs for distributed secrets. In 31st USENIX Security Symposium (USENIX Security 22), pages 4291–4308, Boston, MA, August 2022. USENIX Association.
- [24] Merkle R.C. A digital signature based on a conventional encryption function. Advances in Cryptology — CRYPTO 1987. Lecture Notes in Computer Science, vol 293. Springer, Berlin, Heidelberg, 1988.
- [25] StarkWare. Starknet: permissionless, decentralized zero-knowledge rollup for Ethereum. https://docs.starknet.io/documentation/.

- [26] StarkWare. ethSTARK Documentation. Cryptology ePrint Archive, Paper 2021/582, 2021. https://eprint.iacr.org/2021/582.
- [27] Polygon Zero Team. Plonky2: Fast Recursive Arguments with PLONK and FRI, 2022. https://github.com/0xPolygonZero/plonky2/blob/main/plonky2/plonky2.pdf.
- [28] Justin Thaler. Measuring SNARK performance: Frontends, backends, and the future. https://a16zcrypto.com/posts/article/ measuring-snark-performance-frontends-backends-and-the-future/.
- [29] Justin Thaler. SNARK Security and Performance. https://a16zcrypto.com/posts/ article/snark-security-and-performance/.
- [30] Justin Thaler. Proofs, Arguments, and Zero-Knowledge. Foundations and Trends® in Privacy and Security, 4(2-4):117-660, 2022. http://dx.doi.org/10.1561/330000030.
- [31] Tiancheng Xie, Jiaheng Zhang, Zerui Cheng, Fan Zhang, Yupeng Zhang, Yongzheng Jia, Dan Boneh, and Dawn Song. zkBridge: Trustless Cross-chain Bridges Made Practical, 2022. https://arxiv.org/abs/2210.00264.
- [32] Bespalov Yuri, Kovalchuk Lyudmila, Nelasa Hanna, Oliynykov Roman, and Viglione Rob. Models for Generation of Proof Forest in zk-SNARK Based Sidechains. *Cryptography*, 7(1), 2023. https://www.mdpi.com/2410-387X/7/1/14.
- [33] Ye Zhang. Decentralized zk-Rollup, 2021. https://hackmd.io/@yezhang/SkmyXzWMY.