zkBridge: Trustless Cross-chain Bridges Made Practical

Tiancheng Xie[1], Jiaheng Zhang[1], Zerui Cheng[2], Fan Zhang[3], Yupeng Zhang[4], Yongzheng Jia[7], Dan Boneh[5], Dawn Song[1,6]

ABSTRACT

Blockchains have seen growing traction with cryptocurrencies reaching a market cap of over 1 trillion dollars, major institution investors taking interests, and global impacts on governments, businesses, and individuals. Also growing significantly is the heterogeneity of the ecosystem where a variety of blockchains co-exist. Cross-chain bridge is a necessary building block in this multi-chain ecosystem. Existing solutions, however, either suffer from performance issues or rely on trust assumptions of committees that significantly lower the security. Recurring attacks against bridges have cost users more than 1.5 billion USD. In this paper, we introduce zkBridge, an efficient cross-chain bridge that guarantees strong security without external trust assumptions. With succinct proofs, zkBridge not only guarantees correctness, but also significantly reduces on-chain verification cost. We propose novel succinct proof protocols that are orders-of-magnitude faster than existing solutions for workload in zkBridge. With a modular design, zkBridge enables a broad spectrum of use cases and capabilities, including message passing, token transferring, and other computational logic operating on state changes from different chains. To demonstrate the practicality of zkBridge, we implemented a prototype bridge from Cosmos to Ethereum, a particularly challenging direction that involves large proof circuits that existing systems cannot efficiently handle. Our evaluation shows that zkBridge achieves practical performance: proof generation takes less than 20 seconds, while verifying proofs on-chain costs less than 230K gas. For completeness, we also implemented and evaluated the direction from Ethereum to other EVM-compatible chains (such as BSC) which involves smaller circuits and incurs much less overhead.

1 INTRODUCTION

Since the debut of Bitcoin, blockchains have evolved to an expansive ecosystem of various applications and communities. Cryptocurrencies like Bitcoin and Ethereum are gaining rapid traction with the market cap reaching over a trillion USD [12] and institutional investors [55, 70] taking interests. Decentralized Finance (DeFi) demonstrates that blockchains can enable finance instruments that are otherwise impossible (e.g., flash loans [62]). More recently, digital artists [32] and content creators [30] resort to blockchains for transparent and accountable circulation of their works.

Also growing significantly is the heterogeneity of the ecosystem. A wide range of blockchains have been proposed and deployed, ranging from ones leveraging computation (e.g., in Proof-of-Work [61]), to economic incentives (e.g., in Proof-of-Stake [38, 39, 47, 52, 56]), and various other resources such as storage [1, 35, 48, 63], and even time [2]. While it is rather unclear that one blockchain dominates others in all aspects, these protocols employ different techniques and achieve different security guarantees and performance. It has thus been envisioned that (e.g., in [20, 21, 29]) the ecosystem will grow to a *multi-chain* future where various protocols co-exist, and developers and users can choose the best blockchain based on their preferences, the cost, and the offered amenities.

A central challenge in the multi-chain universe is how to enable secure *cross-chain bridges* through which applications on different blockchains can communicate. An ecosystem with efficient and inexpensive bridges will enable assets held on one chain to effortlessly participate in marketplaces hosted on other chains. In effect, an efficient system of bridges will do for blockchains what the Internet did for siloed communication networks.

The core functionality of a bridge between blockchains $\mathcal{C}_1$ and $\mathcal{C}_2$ is to prove to applications on $\mathcal{C}_2$ that a certain event took place on $\mathcal{C}_1$, and vice versa. We use a generic notion of a bridge, namely one that can perform multiple functions: message passing, asset transfers, etc. In our modular design, the bridge itself neither involves nor is restricted to any application-specific logic.

The problem. While cross-chain bridges have been built in practice [3, 4, 9, 18], existing solutions either suffer from poor performance, or rely on central parties.

The operation of the bridge depends on the consensus protocols of both chains. If $\mathcal{C}_1$ runs Proof-of-Work, a natural idea is to use a light client protocol (e.g., SPV [61]). Specifically, a smart contract on $\mathcal{C}_2$, denoted by $\mathcal{SC}_2$, will keep track of block headers of $\mathcal{C}_1$, based on which transaction inclusion (and other events) can be verified with Merkle proofs. This approach, however, incurs a significant computation and storage overhead, since $\mathcal{SC}_2$ needs to verify all block headers and keep a long and ever-growing list of them. For non-PoW chains, the verification can be even more expensive. For example, for a bridge between a Proof-of-Stake chain (like Cosmos) and Ethereum, verifying a single block header on Ethereum would cost about 64 million gas [15] (about $6300 at time of writing), which is prohibitively high.

Currently, as an efficient alternative, many bridge protocols (PolyNetwork, Wormhole, Ronin, etc.) resort to a committee-based approach: a committee of validators are entrusted to sign off on state transfers. In these systems, the security boils down to, e.g., the honest majority assumption. This is problematic for two reasons. First, the extra trust assumption in the committee means the bridged asset is not as secure as native ones, complicating the security analysis of downstream applications. Second, relying on a small committee can lead to single point failures. Indeed, in a recent exploit of the Ronin bridge [27], the attackers were able to obtain five of the nine validator keys, through which they stole 624 million USD, making it the largest attack in the history of DeFi by Apr 2022[1]. Even the second and third largest attacks are also against bridges ($611m was stolen from PolyNetwork [6] and $326m was stolen from Wormhole [10]), and key compromise was suspected in the PolyNetwork attack.

[1]UC Berkeley
[2]Tsinghua University
[3]Yale University
[4]Texas A&M University
[5]Stanford University
[6]Oasis Labs
[7]Overeality Labs

[1]see the ranking at https://rekt.news/leaderboard

Our approach. We present zkBridge to enable an efficient cross-chain bridge without trusting a centralized committee. The main idea is to leverage zk-SNARK, which are succinct non-interactive proofs (arguments) of knowledge [19, 34, 36, 37, 41, 44, 45, 51, 64, 69, 73, 74, 77, 78]. A zk-SNARK enables a prover to efficiently convince $\mathcal{SC}_2$ that a certain state transition took place on $\mathcal{C}_1$. To do so, $\mathcal{SC}_2$ will keep track of a digest D of the latest tip of $\mathcal{C}_1$. To sync $\mathcal{SC}_2$ with new blocks in $\mathcal{C}_1$, anyone can generate and submit a zk-SNARK that proves to $\mathcal{SC}_2$ that the tip of $\mathcal{C}_1$ has advanced from D to D'.

This design offers three benefits. First, the soundness property of a zk-SNARK ensures the security of the bridge. Thus, we do not need additional security requirements beyond the security of the underlying blockchains. In particular zkBridge does not rely on a committee for security. Second, with a purpose-built zk-SNARK, $\mathcal{C}_2$ can verify a state transition of $\mathcal{C}_1$ far more efficiently than encoding the consensus logic of $\mathcal{C}_1$ in $\mathcal{SC}_2$. In this way, as an example for zkBridge from Cosmos to Ethereum, we reduce the proof verification cost from $\sim 80M$ gas to less than $230K$ gas on $\mathcal{C}_2$. The storage overhead of the bridge is reduced to constant. Third, by separating the bridge from application-specific logic, zkBridge makes it easy to enable additional applications on top of the bridge.

Technical challenges. To prove correctness of a given computation outcome using a zk-SNARK, one first needs to express the computation as an arithmetic circuit. While zk-SNARK verification is fast (logarithmic in the size of the circuit or even constant), proof generation time is at least linear, and in practice can be prohibitively expensive. Moreover, components used by real-world blockchains are not easily expressed as an arithmetic circuit. For example, the widely used EdDSA digital signature scheme is very efficient to verify on a CPU, but is expensive to express as an arithmetic circuit, requiring more than 2 million gates [13]. In a cross-chain bridge, each state transition could require the verification of hundreds of signatures depending on the chains, making it prohibitively expensive to generate the required zk-SNARK proof. In order to make zkBridge practical, we must reduce proof generation time.

To this end, we propose two novel ideas. First, we observe that the circuits used by cross-chain bridges are *data-parallel*, in that they contain multiple identical copies of a smaller sub-circuit. Specifically, the circuit for verifying N digital signatures contains N copies of the signature verification sub-circuit. To leverage the data-parallelism, we propose deVirgo, a novel distributed zero-knowledge proofs protocol based on Virgo [76]. deVirgo enjoys perfect linear scalability, namely, the proof generation time can be reduced by a factor of M if the generation is distributed over M machines. The protocol is of independent interest and might be useful in other scenarios. Other proof systems can be similarly parallelized [72].

While deVirgo significantly reduces the proof generation time, verifying deVirgo proofs on chain, especially for the billion-gate circuits in zkBridge, can be expensive for smart contracts where computational resources are extremely limited. To compress the proof size and the verification cost, we recursively prove the correctness of a (potentially large) deVirgo proof using a classic zk-SNARK due to Groth [54], hereafter denoted Groth16. The Groth16 prover outputs constant-size proofs that are fast to verify by a smart contract on an EVM blockchain. We stress that one cannot use Groth16 to generate the entire zkBridge proof because the circuits needed in zkBridge are too large for a Groth16 prover. Instead, our approach of compressing a deVirgo proof using Groth16 gives the best of both worlds: a fast deVirgo parallel prover for the bulk of the proof, where the resulting proof is compressed into a succinct Groth16 proof that is fast to verify. We elaborate on this technique in Section 5. This approach to compressing long proofs is also being adopted in commercial zk-SNARK systems such as [23, 24, 26].

Implementation and evaluation. To demonstrate the practicality of zkBridge, we implement an end-to-end prototype of zkBridge from Cosmos to Ethereum, given it is among the most challenging directions as it involves large circuits for correctness proofs. Our implementation includes the protocols of deVirgo and recursive proof with Groth16, and the transaction relay application. The experiments show that our system achieves practical performance. deVirgo can generate a block header relay proof within 20s, which is more than 100x faster than the original Virgo system with a single machine. Additionally, the on-chain verification cost decreases from ~80M gas (direct signature verification) to less than 230K gas, due to the recursive proofs. In addition, as a prototype example, we also implement zkBridge from Ethereum to other EVM-compatible chains such as BSC, which involves smaller circuits for proof generation and incurs much less overhead.

1.1 Our contribution

In this paper, we make the following contributions:

- In this paper, we propose zkBridge, a trustless, efficient, and secure cross-chain bridge whose security relies on succinct proofs (cryptographic assumptions) rather than a committee (external trust assumptions). Compared with existing cross-chain bridge projects in the wild, zkBridge is the first solution that achieves the following properties at the same time.
  - **Trustless and Secure:** The correctness of block headers on remote blockchains is proven by zk-SNARKs, and thus no external trust assumptions are introduced. Indeed, as long as the connected blockchains and the underlying light-client protocols are secure, and there exists at least one honest node in the block header relay network, zkBridge is secure.
  - **Permissionless and Decentralized:** Any node can freely join the network to relay block headers, generate proofs, and claim the rewards. Due to the elimination of the commonly-used central or Proof-of-Stake style committee for block header validation, zkBridge also enjoys better decentralization.
  - **Extensible:** Smart contracts using zkBridge enjoy maximum flexibility because they can invoke the updater contract to retrieve verified block headers, and then perform their application-specific verification and functionality (e.g., verifying transaction inclusion through auxiliary Merkle proofs). By separating the bridge from application-specific logic, zkBridge makes it easy to develop applications on top of the bridge.
  - **Universal:** The block header relay network and the underlying proof scheme in zkBridge is universal as long as the blockchain supports a light client protocol as in Definition 2.1.
  - **Efficient:** With our highly optimized recursive proof scheme, block headers can be relayed within a short time (usually tens of seconds for proof generation), and the relayed information

can be quickly finalized as soon as the proof is verified, thus supporting fast and flexible bridging of information.

In summary, zkBridge is a huge leap towards building a secure, trustless foundation for blockchain interoperability.

- We propose a novel 2-layer recursive proof system, which is of independent interest, as the underlying zk-SNARK protocol to achieve both reasonable proof generation time and on-chain verification cost. Through the coordination of deVirgo and Groth16, we achieve a desirable balance between efficiency and cost.
  - For the first layer, aiming at prompt proof generation, we introduce deVirgo, a distributed version of Virgo proof system. deVirgo combines distributed sumcheck and distributed polynomial commitment to achieve optimal parallelism, through which the proof generation phase is much more accelerated by running on distributed machines. deVirgo is more than 100x faster than Virgo for the workload in zkBridge.
  - For the second layer, aiming at acceptable on-chain verification cost, we use Groth16 to recursively prove that the previously generated proof by deVirgo indeed proves the validity of the corresponding remote block headers. Through the second layer, the verification gas cost is reduced from an estimated $\sim 80M$ to less than $230K$, making on-chain verification practical.
- We implement an end-to-end prototype of zkBridge and evaluate its performance in two scenarios: from Cosmos to Ethereum (which is the main focus since it involves large proof circuits that existing systems cannot efficiently handle), and from Ethereum to other EVM-compatible chains (which in comparison involves much smaller circuits). The experiment results show that zkBridge achieves practical performance and is the first practical cross-chain bridge that achieves cryptographic assurance of correctness.

2 BACKGROUND

In this section we cover the preliminaries, essential background on blockchains, and zero-knowledge proofs.

2.1 Notations

Let $\mathbb{F}$ be a finite field and λ be a security parameter. We use $f(),h()$ for polynomials, x,y for single variables, bold letters $\mathbf{x},\mathbf{y}$ for vectors of variables. Both $\mathbf{x}[i]$ and x_i denote the i-th element in $\mathbf{x}$. For $\mathbf{x}$, we use notation $\mathbf{x}[i:k]$ to denote slices of vector $\mathbf{x}$, namely $\mathbf{x}[i:k]=(x_i,x_{i+1},\cdots,x_k)$. We use $\mathbf{i}$ to denote the vector of the binary representation of some integer i.

Merkle Tree. Merkle tree [59] is a data structure widely used to build commitments to vectors because of its simplicity and efficiency. The prover time is linear in the size of the vector while the verifier time and proof size are logarithmic in the size of the vector. Given a vector of $\mathbf{x}=(x_0,\cdots,x_{N-1})$, it consists of three algorithms:

- $\mathsf{rt} \leftarrow \mathsf{MT.Commit}(\mathbf{x})$
- $(\mathbf{x}[i],\pi_i) \leftarrow \mathsf{MT.Open}(\mathbf{x},i)$
- $\{1,0\} \leftarrow \mathsf{MT.Verify}(\pi_i,\mathbf{x}[i],\mathsf{rt})$.

2.2 Blockchains

A blockchain is a distributed protocol where a group of nodes collectively maintains a *ledger* which consists of an ordered list of *blocks*. A block blk is a data-structure that stores a header blkH and a list of transactions, denoted by $\mathsf{blk}=\{\mathsf{blkH};\mathsf{trx}_1,\ldots,\mathsf{trx}_t\}$. A block header contains metadata about the block, including a pointer to the previous block, a compact representation of the transactions (typically a Merkle tree root), validity proofs such as solutions to cryptopuzzles in Proof-of-Work systems or validator signatures in Proof-of-Stake ones.

Security of blockchains. The security of blockchains has been studied extensively. Suppose the ledger in party i's local view is $\mathsf{LOG}_i^r=[\mathsf{blk}_1,\mathsf{blk}_2,\ldots,\mathsf{blk}_r]$ where r is the *height*. For any $2\le k\le r$ and the k-th block blk_k , $\mathsf{blk}_k.\mathsf{ptr}=\mathsf{blkH}_{k-1}$, so every single block is linked to the previous one. For the purpose of this paper, we care about two (informal) properties:

1. **Consistency**: For any honest nodes i and j, and for any rounds of r_0 and r_1, it must be satisfied that either $\mathsf{LOG}_i^{r_0}$ is a prefix of $\mathsf{LOG}_j^{r_1}$ or vice versa.
2. **Liveness**: If an honest node receives some transaction trx at some round r, then trx will be included into the blockchain of all honest nodes eventually.

Smart contracts and gas. In addition to reaching consensus over the content of the ledger, many blockchains support expressive user-defined programs called *smart contracts*, which are stateful programs with state persisted on a blockchain. Without loss of generality, smart contract states can be viewed a key-value store (and often implemented as such.) Users send transactions to interact with a smart contract, and potentially alter its state.

A key limitation of existing smart contract platforms is that computation and storage are scarce resources and can be considerably expensive. Typically smart contract platforms such as Ethereum charge a fee (sometimes called gas) for every step of computation. For instance, EdDSA signatures are extremely cheap to verify (a performant CPU can verify 71000 of them in a second [40]), but verifying a single EdDSA signature on Ethereum costs about 500K gas, which is about $49 at the time of writing. Storage is also expensive on Ethereum. Storing 1KB of data costs about 0.032 ETH, which can be converted to approximately $90 at the time of writing. This limitation is not unique to Ethereum but rather a reflection of the low capacity of permissionless blockchains in general. Therefore reducing on-chain computation and storage overhead is one of the key goals.

2.3 Light client protocol

In a blockchain network, there are full nodes as well as light ones. Full nodes store the entire history of the blockchain and verify all transactions in addition to verifying block headers. Light clients, on the other hand, only store the headers, and therefore can only verify a subset of correctness properties.

The workings of light clients depend on the underlying consensus protocol. The original Bitcoin paper contains a light client protocol (SPV [61]) that uses Merkle proofs to enable a light client who only stores recent headers to verify transaction inclusion. A number of improvements have been proposed ever since. For instance, in Proof-of-Stake, typically a light client needs to verify account balances in the whole blockchain history (or up to a snapshot), and considers the risk of long range attacks. For BFT-based consensus,

a light client needs to verify validator signatures and keeps track of validator rotation. We refer readers to [42] for a survey.

To abstract consensus-specific details away, we use

$$\mathsf{LightCC}(\mathsf{LCS}_{r-1},\mathsf{blkH}_{r-1},\mathsf{blkH}_r) \rightarrow \{\mathsf{true},\mathsf{false}\}$$

to denote the block validation rule of a light client: given a new block header blkH_r, LightCC determines if the header represents a valid next block after blkH_{r-1} given its current state LCS_{r-1}. We define the required properties of a light client protocol as follows:

Definition 2.1 (Light client protocol). A light client protocol enables a node to synchronize the block headers of the state of the blockchain. Suppose all block headers in party i's local view is $\mathsf{LOG}H_i^r = [\mathsf{blkH}_1,\mathsf{blkH}_2,...,\mathsf{blkH}_r]$, the light client protocol satisfies following properties:

1. **Succinctness**: For each state update, the light client protocol only takes $O(1)$ time to synchronize the state.
2. **Liveness**: If an honest full node receives some transaction trx at some round r, then trx must be included into the blockchain eventually. A light client protocol will eventually include a block header blkH_i such that the corresponding block includes the transaction trx.
3. **Consistency**: For any honest nodes i and j, and for any rounds of r_0 and r_1, it must be satisfied that either $\mathsf{LOG}H_i^{r_0}$ is a prefix of $\mathsf{LOG}H_j^{r_1}$ or vice versa.

2.4 Zero-knowledge proofs

An argument system for an NP relationship $\mathcal{R}$ is a protocol between a computationally-bounded prover $\mathcal{P}$ and a verifier $\mathcal{V}$. At the end of the protocol, $\mathcal{V}$ is convinced by $\mathcal{P}$ that there exists a witness $\mathbf{w}$ such that $(\mathbf{x};\mathbf{w}) \in \mathcal{R}$ for some input $\mathbf{x}$. We use $\mathcal{G}$ to represent the generation phase of the public parameters pp. Formally, consider the definition below, where we assume $\mathcal{R}$ is known to $\mathcal{P}$ and $\mathcal{V}$.

Definition 2.2. Let λ be a security parameter and $\mathcal{R}$ be an NP relation. A tuple of algorithm $(\mathcal{G},\mathcal{P},\mathcal{V})$ is a zero-knowledge argument of knowledge for $\mathcal{R}$ if the following holds.

- **Completeness**. For every pp output by $\mathcal{G}(1^\lambda)$, $(\mathbf{x};\mathbf{w}) \in \mathcal{R}$ and $\pi \leftarrow \mathcal{P}(\mathbf{x},\mathbf{w},\mathsf{pp})$,
$$\Pr[\mathcal{V}(\mathbf{x},\pi,\mathsf{pp}) = 1] = 1$$
- **Knowledge Soundness**. For any PPT prover $\mathcal{P}^*$, there exists a PPT extractor $\mathcal{E}$ such that for any auxiliary string $\mathbf{z}$, $\mathsf{pp} \leftarrow \mathcal{G}(1^\lambda)$, $\pi^* \leftarrow \mathcal{P}^*(\mathbf{x},\mathbf{z},\mathsf{pp})$, $w \leftarrow \mathcal{E}^{\mathcal{P}^*(\cdot)}(\mathbf{x},\mathbf{z},\mathsf{pp})$, and
$$\Pr[(\mathbf{x};\mathbf{w}) \notin \mathcal{R} \wedge \mathcal{V}(\mathbf{x},\pi^*,\mathsf{pp}) = 1] \leq \mathsf{negl}(\lambda),$$
where $\mathcal{E}^{\mathcal{P}^*(\cdot)}$ represents that $\mathcal{E}$ can rewind $\mathcal{P}^*$,
- **Zero knowledge**. There exists a PPT simulator $\mathcal{S}$ such that for any PPT algorithm $\mathcal{V}^*$, $(\mathbf{x};\mathbf{w}) \in \mathcal{R}$, pp output by $\mathcal{G}(1^\lambda)$, it holds that
$$\mathsf{View}(\mathcal{V}^*(\mathsf{pp},\mathbf{x})) \approx \mathcal{S}^{\mathcal{V}^*}(\mathbf{x}),$$
where $\mathsf{View}(\mathcal{V}^*(\mathsf{pp},\mathbf{x}))$ denotes the view that the verifier sees during the execution of the interactive process with $\mathcal{P}$, $\mathcal{S}^{\mathcal{V}^*}(\mathbf{x})$ denotes the view generated by $\mathcal{S}$ given input $\mathbf{x}$ and transcript of $\mathcal{V}^*$, and $\approx$ denotes two perfectly indistinguishable distributions.

We say that $(\mathcal{G},\mathcal{P},\mathcal{V})$ is a **succinct** argument system[2] if the total communication (proof size) between $\mathcal{P}$ and $\mathcal{V}$, as well as $\mathcal{V}$'s running time, are $\mathsf{poly}(\lambda,|\mathbf{x}|,\log|\mathcal{R}|)$, where $|\mathcal{R}|$ is the size of the circuit that computes $\mathcal{R}$ as a function of λ.

3 ZKBRIDGE PROTOCOL

At a high level, a smart contract is a stateful program with states persisted on a blockchain. A bridge like zkBridge is a service that enables smart contracts on different blockchains to transfer states from one chain to another in a secure and verifiable fashion.

Below we first explain the design of zkBridge and its workflow through an example, then we specify the protocol in more detail. For ease of exposition, we focus on one direction of the bridge, but the operation of the opposite direction is symmetric.

3.1 Overview of zkBridge design

To make it easy for different applications to integrate with zkBridge, we adopt a modular design where we separate application-specific logic (e.g., verifying smart contract states) from the core bridge functionality (i.e., relaying block headers).

Figure 1 shows the architecture and workflow of zkBridge. The core bridge functionality is provided by a **block header relay network** (trusted only for liveness) that relays block headers of $\mathcal{C}_1$ along with correctness proofs, and an **updater contract** on $\mathcal{C}_2$ that verifies and accepts proofs submitted by relay nodes. The updater contract maintains a list of recent block headers, and updates it properly after verifying proofs submitted by relay nodes; it exposes a simple and application-agnostic API, from which application smart contracts can obtain the latest block headers of the sender blockchain and build application-specific logic on top of it.

Applications relying on zkBridge will typically deploy a pair of contracts, a sender contract and a receiver contract on $\mathcal{C}_1$ and $\mathcal{C}_2$, respectively. We refer to them collectively as application contracts or relying contracts. The receiver contract can call the updater contract to obtain block headers of $\mathcal{C}_1$, based on which they can perform application specific tasks. Depending on the application, receiver contracts might also need a user or a third party to provide application-specific proofs, such as Merkle proofs for smart contract states.

As an example, Fig. 1 shows the workflow of *cross-chain token transfer*, a common use case of bridges, facilitated by zkBridge. Suppose a user $\mathcal{U}$ wants to trade assets (tokens) she owns on blockchain $\mathcal{C}_1$ in an exchange residing on another blockchain $\mathcal{C}_2$ (presumably because $\mathcal{C}_2$ charges lower fees or has better liquidity), she needs to move her funds from $\mathcal{C}_1$ to $\mathcal{C}_2$. A pair of smart contracts $\mathcal{SC}_{\text{lock}}$ and $\mathcal{SC}_{\text{mint}}$ are deployed on blockchains $\mathcal{C}_1$ and $\mathcal{C}_2$ respectively. To move the funds, the user locks \$v tokens in $\mathcal{SC}_{\text{lock}}$ (Step ❶ in Fig. 1) and then requests \$v tokens to be issued by $\mathcal{SC}_{\text{mint}}$. To ensure solvency, $\mathcal{SC}_{\text{mint}}$ should only issue new tokens if and only if the user has locked tokens on $\mathcal{C}_1$. This requires $\mathcal{SC}_{\text{mint}}$ to read the states of $\mathcal{SC}_{\text{lock}}$ (the balance of $\mathcal{U}$, updated in step ❷) from a different blockchain, which it cannot do directly. zkBridge enables this by relaying the block headers of $\mathcal{C}_1$ to $\mathcal{C}_2$ along with proofs (step ❸

[2] In our construction, we only need a succinct non-interactive arguments of knowledge (SNARK) satisfying the first two properties and the succinctnes for validity. The zero knowledge property could be used to further achieve privacy.

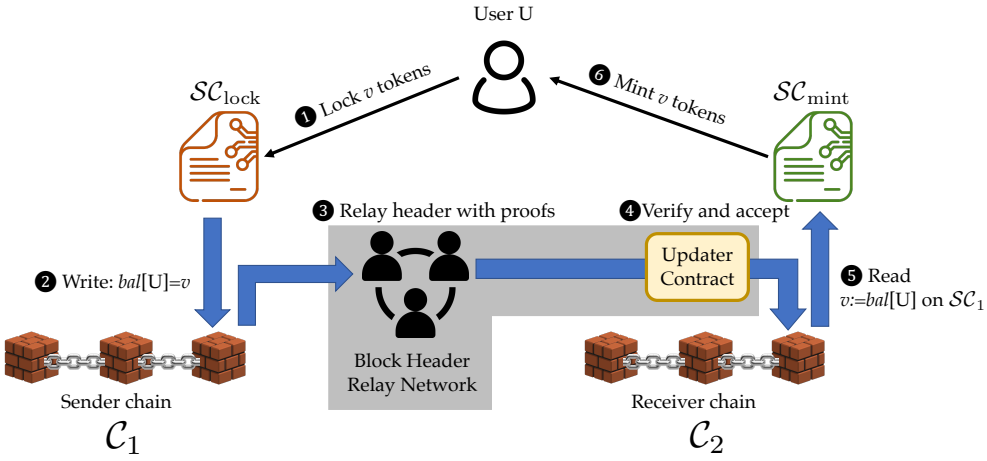

Figure 1: The design of zkBridge illustrated with the example of cross-chain token transfer. The components in shade belongs to zkBridge. For clarity we only show one direction of the bridge and the opposite direction is symmetric.

and ❹). $\mathcal{SC}_{\text{mint}}$ can retrieve the block headers from the smart contract frontend (the updater contract), check that the balance of user $\mathcal{U}$ is indeed \$v (step ❺), and only then mint \$v tokens (Step ❻).

Besides cross-chain token transfer, zkBridge can also enable various other applications such as cross-chain collateralized loans, general message passing, etc. We present three use cases in Section 3.3.

3.2 Protocol detail

Having presented the overview, in this section, we specify the protocol in more detail.

3.2.1 Security and system model. For the purpose of modeling bridges, we model a blockchain $\mathcal{C}$ as a block-number-indexed key-value store, denoted as $\mathcal{C}[t]:\mathcal{K}\rightarrow\mathcal{V}$ where t is the block number, $\mathcal{K}$ and $\mathcal{V}$ are key and value spaces respectively. In Ethereum, for example, $\mathcal{V}=\{0,1\}^{256}$ and keys are the concatenation of a smart contract identifier $\mathcal{SC}$ and a per-smart-contract storage address K. For a given contract $\mathcal{SC}$, we denote the value stored at address K at block number t as $\mathcal{SC}[t,K]$, and we call $\mathcal{SC}[t,\cdot]$ the *state* of $\mathcal{SC}$ at block number t. Again, for ease of exposition, we focus on the direction from $\mathcal{SC}_1$ to $\mathcal{SC}_2$, denoted as $\mathcal{BR}[\mathcal{SC}_1\rightarrow\mathcal{SC}_2]$.

Functional and security goals. We require the bridge $\mathcal{BR}[\mathcal{SC}_1\rightarrow\mathcal{SC}_2]$ to reflect states of $\mathcal{SC}_1$ correctly and timely:

1. **Correctness**: For all t,K, $\mathcal{SC}_2$ accepts a wrong state $V\neq\mathcal{SC}_1[t,K]$ with negligible probability.
2. **Liveness**: Suppose $\mathcal{SC}_2$ needs to verify $\mathcal{SC}_1$'s state at (t,K), the bridge will provide necessary information eventually.

Security assumptions. For correctness, zkBridge does not introduce extra trust assumptions besides those made by the underlying blockchains. Namely, we assume both the sender blockchain and the receiver blockchain are consistent and live (Section 2), and the sender chain has a light client protocol to enable fast block header verification. For both properties, we assume there is at least one honest node in the relay network, and that the zk-SNARK used is sound.

3.2.2 Construction of zkBridge. As described in Section 3, a bridge $\mathcal{BR}[\mathcal{SC}_1\rightarrow\mathcal{SC}_2]$ consists of three components: a block header relay network, a updater contract, and one or more application contracts. Below we specify the protocols for each component.

Block header relay network. We present the formal protocol of block header relay network in Protocol 1.

Protocol 1 Block header relay network

procedure RelayNextHeader(LCS_{r-1},blkH_{r-1})
Contact k different full nodes to get the block headers following blkH_{r-1}, namely blkH_r.
Generate a ZKP π proving

$$\mathsf{LightCC}(\mathsf{LCS}_{r-1},\mathsf{blkH}_{r-1},\mathsf{blkH}_r)\rightarrow\mathsf{true}.$$

Send $(\pi,\mathsf{blkH}_r,\mathsf{blkH}_{r-1})$ to the updater contract.
end procedure

Nodes in the block header relay network run RelayNextHeader with the current state of the updater contract $(\mathsf{LCS}_{r-1},\mathsf{blkH}_{r-1})$ as input. The exact definition of LCS_{r-1} is specific to light client protocols (see [42] for a survey). The relay node then connects to full nodes in $\mathcal{C}_1$ and gets the block header blkH_r following blkH_{r-1}. The relay node generates a ZKP π showing the correctness of blkH_r, by essentially proving that blkH_r is accepted by a light client of $\mathcal{C}_1$ after block blkH_{r-1}. It then sends (π,blkH_r) to the updater contract on $\mathcal{C}_2$. To avoid the wasted proof time due to collision (note that when multiple relay nodes send at the same time, only one proof can be accepted), relay nodes can coordinate using standard techniques (e.g., to send in a round robin fashion). While any zero-knowledge proofs protocol could be used, our highly optimized one will be presented later in Section 4.

To incentivize block header relay nodes, provers may be rewarded with fees after validating their proofs. We leave incentive design for future work. A prerequisite of any incentive scheme is unstealability [65], i.e., the guarantee that malicious nodes cannot steal others' proofs. To this end, provers will embed their identifiers

(public keys) in proofs, e.g., as input to the hash function in the Fiat-Shamir heuristic [49].

We note that this design relies on the security of the light client verifier of the sender chain. For example, the light client verifier must reject a valid block header that may eventually become orphaned and not part of the sender chain.

The updater contract. The protocol for the updater contract is specified in Protocol 2.

Protocol 2 The updater contract

```
headerDAG := ∅                                          ▷ DAG of headers
LCS := ⊥                                                ▷ light client state
procedure HeaderUpdate(π,blkH_r,blkH_{r−1})
    if blkH_{r−1} ∉ headerDAG then
        return False        ▷ skip if parent block is not in the DAG
    end if
    if π verifies against LCS,blkH_{r−1},blkH_r then
        Update LCS according to the light client protocol.
        Insert blkH_r into headerDAG.
    end if
end procedure
procedure GetHeader(t)    ▷ t is a unique identifier to a block header
    if t ∉ headerDAG then
        return ⊥                                        ▷ tell the caller to wait
    else
        return headerDAG[t],LCS   ▷ The LCS will help users to determine if t is on a fork.
    end if
end procedure
```

The updater contract maintains the light client's internal state including a list of block headers of $\mathcal{C}_1$ in headerDAG. It has two publicly exposed functions. The HeaderUpdate function can be invoked by any block header relay node, providing supposedly the next block header and a proof as input. If the proof verifies against the current light client state LCS and blkH_{r-1}, the contract will do further light-client checks, and then the state will be updated accordingly. Since the caller of this function must pay a fee, DoS attacks are naturally prevented.

The GetHeader function can be called by receiver contracts to get the block header at height t. Receiver contracts can use the obtained block header to finish application-specific verification, potentially with the help of a user or some third party.

Application contracts. zkBridge has a modular design in that the updater contract is application-agnostic. Therefore in $\mathcal{BR}[\mathcal{SC}_1 \rightarrow \mathcal{SC}_2]$, it is up to the application contracts $\mathcal{SC}_1$ and $\mathcal{SC}_2$ to decide what the information to bridge is. Generally, proving that $\mathcal{SC}_1[t,K] = V$ is straightforward: $\mathcal{SC}_2$ can request for a Merkle proof for the leaf of the state Trie Tree (at block number t) corresponding to address K. The receiver contract can obtain blkH_t from the updater contract by calling the function GetHeader(t). Then it can verify $\mathcal{SC}_1[t,K] = V$ against the Merkle root in blkH_t. Required Merkle proofs are application-specific, and are typically provided by the users of $\mathcal{SC}_2$, some third party, or the developer/maintainer of $\mathcal{SC}_2$.

Security arguments. The security of zkBridge is stated in the following theorem.

Theorem 3.1. *The bridge $\mathcal{BR}[\mathcal{SC}_1 \rightarrow \mathcal{SC}_2]$ implemented by protocols 1 and 2 satisfies both consistency and liveness, assuming the following holds:*

1. *there is at least one honest node in the block header relay network;*
2. *the sender chain is consistent and live;*
3. *the sender chain has a light-client verifier as in Def. 2.1; and*
4. *the succinct proof system is sound.*

Proof (sketch). To prove the consistency of DAG, we first need to convert the DAG into a list of blocks to match the definition of blockchain consistency. We define an algorithm *Longest* : DAG → List such that given a DAG, the algorithm will output a list MainChain representing the main chain. For example, if the sender chain is Ethereum, the algorithm *Longest* will first calculate the path with the maximum total difficulty in the DAG represented by L, and then output $\mathsf{MainChain} := \mathsf{L}[:-K]$. Here K is a security parameter. By assumption 1 and 2, there will be an honest node in our system running either a full node or a light node, which will be consistent with the sender chain. Also, according to assumption 1, at least one prover node is honestly proving the light client execution. By assumption 4 that the proof system is sound, the updater contract will correctly verify the light-client state. We argue that the updater contract is correctly running the light-client protocol. Therefore, by the consistency of the light-client protocol, MainChain will be consistent with any other honest node.

The liveness of our protocol directly follows from the liveness of $\mathcal{C}_1$ and its light client protocol. ∎

3.3 Application use cases

In this section, we present three examples of applications that zkBridge can support.

Transaction inclusion: a building block. A common building block of cross-chain applications is to verify transaction inclusion on another blockchain. Specifically, the goal is to enable a receiver contract $\mathcal{SC}_2$ on $\mathcal{C}_2$ to verify that a given transaction trx has been included in a block B_t on $\mathcal{C}_1$ at height t. To do so, the receiver contract $\mathcal{SC}_2$ needs a user or a third-party service to provide the Merkle proof for trx in B_t. Then, $\mathcal{SC}_2$ will call the updater contract to retrieve the block header of $\mathcal{C}_1$ at height t, and then verify the provided Merkle proof against the Merkle root contained in the header.

Next, we will present three use cases that extend the building block above.

1. Message passing and data sharing. Cross-chain message passing is another common building block useful for, e.g., sharing off-chain data cross blockchains.

Message passing can be realized as a simple extension of transaction inclusion, by embedding the message in a transaction. Specifically, to pass a message m from $\mathcal{C}_1$ to $\mathcal{C}_2$, a user can embed m in a transaction trx_m, send trx_m to $\mathcal{C}_1$, and then execute the above transaction inclusion proof.

2. Cross-chain assets transfer/swap. Bridging native assets is a common use case with growing demand. In this application, users

can stake a certain amount of token T_A on the sender blockchain $\mathcal{C}_1$, and get the same amount of token T_A (for native assets transfer if eligible) or a certain amount of token T_B of approximately the same value (for native assets swap) on the receiver blockchain $\mathcal{C}_2$. With the help of the transaction inclusion proof, native assets transfer/swap can be achieved, as illustrated at a high level in Section 3.1. Here we specify the protocol in more detail.

To set up, the developers will deploy a lock contract $\mathcal{SC}_{\text{lock}}$ on $\mathcal{C}_1$ and a mint contract $\mathcal{SC}_{\text{mint}}$ on $\mathcal{C}_2$. For a user who wants to exchange n_A of token T_A for an equal value in token T_B, she will first send a transaction trx_{lock} that transfers n_A of token T_A to $\mathcal{SC}_{\text{lock}}$, along with an address $addr_{\mathcal{C}_2}$ to receive token T_B on $\mathcal{C}_2$. After trx_{lock} is confirmed in a block B, the user will send a transaction trx_{mint} to $\mathcal{SC}_{\text{mint}}$, including sufficient information to verify the inclusion of trx_{lock}. Based on information in trx_{mint}, $\mathcal{SC}_{\text{mint}}$ will verify that trx_{lock} has been included on $\mathcal{C}_1$, and transfer the corresponding T_B tokens to the address $addr_{\mathcal{C}_2}$ specified in trx_{lock}. Finally, $\mathcal{SC}_{\text{mint}}$ will mark trx_{lock} as minted to conclude the transfer.

3. Interoperations for NFTs. In the application of Non-fungible Token (NFT) interoperations, users always lock/stake the NFT on the sender blockchain, and get minted NFT or NFT derivatives on the receiver blockchain. By designing the NFT derivatives, the cross-chain protocol can separate the ownership and utility of an NFT on two blockchain systems, thus supporting locking the ownership of the NFT on the sender blockchain and getting the utility on the receiver blockchain.

3.4 Efficient Proof Systems for zkBridge

The most computationally demanding part of zkBridge is the zero-knowledge proofs generation that relay nodes must do for every block. So far we have abstracted away the detail of proof generation, which we will address in Sections 4 and 5. Here, we present an overview of our solution.

For Proof-of-Stake chains, the proofs involve verifying hundreds of signatures. A major source of overhead is field transformation between different elliptic curves when the sender and receiver chains use different cryptography implementation, which is quite common in practice. For example, Cosmos uses EdDSA on Curve25519 whereas Ethereum natively supports a different curve BN254. The circuit for verifying a single Cosmos signature in the field supported by Ethereum involves around 2 million gates, thus verifying a block (typically containing 32 signatures) will involve over 64 million gates, which is too big for existing zero-knowledge proofs schemes.

To make zkBridge practical, we propose two ideas.

Reducing proof time with deVirgo We observe that the ZKP circuit for verifying multiple signatures is composed of multiple copies of one sub-circuit. Our first idea is to take advantage of this special structure and distribute proof generation across multiple servers. We propose a novel *distributed ZKP protocol* dubbed deVirgo, which carefully parallelizes the Virgo [76] protocol, one of the fastest ZKP systems (in terms of prover time) without a trusted setup. With deVirgo, we can accelerate proof generation in zkBridge with perfect linear scalability. We will dive into the detail of deVirgo in Section 4.

Reducing on-chain cost by recursive verification. While verifying deVirgo proofs on ordinary CPUs is very efficient, on-chain verification is still costly. To further reduce the on-chain verification cost (computation and storage), we use *recursive verification*: the prover recursively proves the correctness of a (potentially large) Virgo proof using a smart-contract-friendly zero-knowledge protocol to get a small and verifier-efficient proof. At a high level, we trade slightly increased proof generation time for much reduced on-chain verification cost: the proof size reduces from 200+KB to 131 bytes, and the required computation reduces from infeasible amount of gas to 210K gas. We will present more detail of recursive verification in Section 5.

4 DISTRIBUTED PROOF GENERATION

As observed previously, the opportunity for fast prover time stems from the fact that the circuit for verifying N signatures consists of N copies of identical sub-circuits. This type of circuits is called data-parallel [67]. The advantage of data-parallel circuits is that there is no connection among different sub-copies. Therefore, each copy can be handled separately. We consider accelerating the proof generation on such huge circuits by dealing with each sub-circuit in parallel. In this section, we propose a distributed zk-SNARK protocol on data-parallel circuits.

There are many zero knowledge proofs protocols [19, 34, 36, 37, 45, 51, 64, 69, 73, 74, 76] supporting our computation. We choose Virgo as the underlying ZKP protocols for two reasons: 1. Virgo does not need a trusted setup and is plausibly post-quantum secure. 2. Virgo is one of the fastest protocols with succinct verification time and succinct proof size for problems in large scale. We present a new distributed version of Virgo for data-parallel arithmetic circuits achieving optimal scalability without any overhead on the proof size. Specifically, our protocol of deVirgo on data-parallel circuits with N copies using N parallel machines is N times faster than the original Virgo while the proof size remains the same. Our scheme is of independent interest and is possible to be used in other Virgo-based systems to improve the efficiency.

We provide the overall description of deVirgo as follows. Suppose the prover has N machines in total, labeled from $\mathcal{P}_0$ to $\mathcal{P}_{N-1}$. Assume $\mathcal{P}_0$ is the master node while other machines are ordinary nodes. Assume $\mathcal{V}$ is the verifier. Given a data-parallel arithmetic circuit consisting of N identical structures, the naïve algorithm of the distributed Virgo is to assign each sub-circuit to a separate node. Then each node runs Virgo to generate the proof separately. The concatenation of N proofs is the final proof. Unfortunately, the proof size in this naive algorithm scales linearly in the number of sub-circuits, which can be prohibitively large for data-parallel circuits with many sub-copies. To address the problem, our approach removes the additional factor of N in the proof size by aggregating messages and proofs among distributed machines. Specifically, the original protocol of Virgo consists of two major building blocks. One is the GKR protocol [53], which consists of d sumcheck protocols [58] for a circuit of depth d. The other is the polynomial commitment (PC) scheme. We design distributed schemes for each of the sumcheck and the polynomial commitment (PC). In our distributed sumcheck protocol, a master node $\mathcal{P}_0$ aggregates messages from all machines, then sends the aggregated message to $\mathcal{V}$ in every round, instead of sending messages from all machines directly to $\mathcal{V}$. Our protocol for distributed sumcheck has exactly the same proof

size as the original sumcheck protocol, thus saving a factor N over the naïve distributed protocol. Additionally, in our distributed PC protocol, we optimize the commitment phase and make $\mathcal{P}_0$ aggregate N commitments into one instead of sending N commitments directly to $\mathcal{V}$. During the opening phase, the proof can also be aggregated, which improves the proof size by a logarithmic factor in the size of the polynomial.

We present preliminaries in Section 4.1, the detail of the distributed sumcheck protocol in Section 4.2 and the detail of the distributed PC protocol in Section 4.3. We combine them all together to build deVirgo in Section 4.4.

4.1 Preliminaries

Multi-linear extension/polynomial. Let $V : \{0,1\}^\ell \to \mathbb{F}$ be a function. The *multi-linear extension/polynomial* of V is the unique polynomial $\tilde{V} : \mathbb{F}^\ell \to \mathbb{F}$ such that $\tilde{V}(\mathbf{x}) = V(\mathbf{x})$ for all $\mathbf{x} \in \{0,1\}^\ell$. $\tilde{V}$ can be expressed as:

$$\tilde{V}(\mathbf{x}) = \sum_{\mathbf{b}\in\{0,1\}^\ell} \prod_{i=1}^{\ell} ((1-x_i)(1-b_i)+x_i b_i)) \cdot V(\mathbf{b}),$$

where b_i is i-th bit of b.

Identity function. Let $\beta : \{0,1\}^\ell \times \{0,1\}^\ell \to \{0,1\}$ be the identity function such that $\beta(\mathbf{x},\mathbf{y}) = 1$ if $\mathbf{x} = \mathbf{y}$, and $\beta(\mathbf{x},\mathbf{y}) = 0$ otherwise. Suppose $\tilde{\beta}$ is the multilinear extension of β. Then $\tilde{\beta}$ can be expressed as: $\tilde{\beta}(\mathbf{x},\mathbf{y}) = \prod_{i=1}^{\ell} ((1-x_i)(1-y_i)+x_i y_i)$.

4.2 Distributed sumcheck

Background: the sumcheck protocol. The sumcheck problem is to sum a multivariate polynomial $f : \mathbb{F}^\ell \to \mathbb{F}$ over all binary inputs: $\sum_{b_1,\cdots,b_\ell\in\{0,1\}} f(b_1,\cdots,b_\ell)$. The sumcheck protocol allows the prover $\mathcal{P}$ to convince the verifier $\mathcal{V}$ that the summation is H via a sequence of interactions, and the formal protocol is presented in Protocol 3 in Appendix A. The high-level idea of the sumcheck protocol is to divide the verification into ℓ rounds. In each round, the prover only sends a univariate polynomial to the verifier. The verifier checks the correctness of the polynomial by a single equation. Then this variable will be replaced by a random point sampled by the verifier. As there are totally ℓ variables in f, after ℓ rounds, the claim about the summation will be reduced to a claim about f on a random vector $\mathbf{r}$. Given the oracle access to f on a random vector, the verifier can check the last claim.

Background: the sumcheck equation in the GKR protocol. In the GKR protocol working for a layered arithmetic circuit, both parties build a sumcheck equation to describe wire connections between the i-th layer and the $(i+1)$-th layer. Without loss of generality, we suppose there are 2^ℓ gates in each layer. We define a polynomial $V_i : \{0,1\}^\ell \to \mathbb{F}$ such that $V_i(\mathbf{b})$ represents the value of gate $\mathbf{b}$ in layer i, where $\mathbf{b}$ is the binary representation of integer b. We use $\tilde{V}_i$ as the multi-linear polynomial of V_i. Then we can write a sumcheck equation

$$\tilde{V}_i(\mathbf{g}) = \sum_{\mathbf{x}\in\{0,1\}^\ell} f(\mathbf{x}, \tilde{V}_{i+1}(\mathbf{x})), \tag{1}$$

where f is some polynomial from $\mathbb{F}^\ell$ to $\mathbb{F}$ and $\mathbf{g}$ is a random vector in $\mathbb{F}^\ell$. By invoking the sumcheck protocol, the prover reduces a claim about the i-th layer to a claim about the $(i+1)$-th layer. Suppose the circuit depth is d, after running d sumcheck protocols, the prover reduces the claim about the output layer to the input layer, which the verifier itself can verify. Due to the space limitation, we present the formal GKR protocol in Protocol 5 in Appendix C.

We treat Equation 1 as the sumcheck equation in the GKR protocol and give the complexity of the sumcheck protocol running on Equation 1 as follows.

Complexity of the sumcheck protocol. For the multivariate polynomial of f defined in Equation 1, the prover time in Protocol 3 is $O(2^\ell)$. The proof size is $O(\ell)$ and the verifier time is $O(\ell)$.

In the setting of data-parallel circuits, we distribute the sumcheck polynomial f among parallel machines. Suppose the data-parallel circuit C consists of N identical sub-circuits of $\mathrm{C}_0,\cdots,\mathrm{C}_{N-1}$ and $N = 2^n$ for some integer n without loss of generality. The polynomial $f : \mathbb{F}^\ell \to \mathbb{F}$ is defined on C by Equation 1.

The idea of our distributed sumcheck protocol is to treat each sub-copy as a new circuit as there is no wiring connections across different sub-circuits. We define polynomials of $f^{(0)},\cdots,f^{(N-1)}$ on $\mathrm{C}_0,\cdots,\mathrm{C}_{N-1} : \mathbb{F}^{\ell-n} \to \mathbb{F}$ respectively by Equation 1 in the GKR protocol, which have the same form as f defined on C. The naïve approach is running the sumcheck protocol on these polynomials separately. As there are N proofs in total and each size is $O(\ell-n)$, the total proof size will be $O(N(\ell-n))$. To reduce the proof size back to ℓ, the prover needs to aggregate N proofs to generate a single proof on f. We observe that the sumcheck protocol on data-parallel circuits satisfies $f^{(i)}(\mathbf{x}) = f(\mathbf{x},\mathbf{i})$. As shown in Protocol 3, the protocol proceeds for ℓ variables round by round. We first run the sumcheck protocol on variables that are irrelevant to the index of sub-copies in the circuit. In the first $(\ell-n)$ rounds, each prover $\mathcal{P}_i$ generates the univariate polynomial of $f_j^{(i)}(x_j)$ for $f^{(i)}(\mathbf{x})$ and sends it to $\mathcal{P}_0$. $\mathcal{P}_0$ constructs the univariate polynomial for $f_j(x_j)$ by summing $f_j^{(i)}(x_j)$ altogether since $f_j(x_j) = \sum_{i=0}^{N} f_j^{(i)}(x_j)$, and sends $f_j(x_j)$ to $\mathcal{V}$ in the j-th round. The aggregation among parallel machines reduces the proof size to constant in each round. Hence the final proof size is only $O(\ell)$. A similar approach has appeared in [68]. The main focus of [68] was improving the prover time of the sumcheck protocol in the GKR protocol to $O(2^\ell(\ell-n))$ for data-parallel circuits, which was later subsumed by [73] with a prover running in $O(2^\ell)$ time. Instead, our scheme is focused on improving the prover time by N times with distributed computing on N machines without any overhead on the proof size.

With this idea in mind, we rewrite the sumcheck equation on f as follows.

$$H = \sum_{\mathbf{b}\in\{0,1\}^\ell} f(\mathbf{b}) = \sum_{i=0}^{N-1} \sum_{\mathbf{b}\in\{0,1\}^{\ell-n}} f^{(i)}(\mathbf{b}).$$

Then we divide the original sumcheck protocol on f into 3 phases naturally in the setting of distributed computing. We present the formal protocol of distributed sumcheck in Protocol 4 in Appendix B.

1. From round 1 to round $(\ell - n)$ (step 1. in Protocol 4), $\mathcal{P}_i$ runs the sumcheck protocol on $f^{(i)}$ and sends the univariate polynomial to $\mathcal{P}_0$. After receiving all univariate polynomials from other machines, $\mathcal{P}_0$ aggregates these univariate polynomials by summing them together and sends the aggregated

univariate polynomial to the verifier. When $\mathcal{P}_0$ receives a random query from the verifier, $\mathcal{P}_0$ relays the random challenge to all nodes as the random query of the current round.

2. In round $(\ell-n)$ (step 2. in Protocol 4), the polynomials of $f^{(0)},\cdots,f^{(N-1)}$ have been condensed to one evaluation on a random vector $\mathbf{r}\in\mathbb{F}^{\ell-n}$. $\mathcal{P}_0$ uses these N points as an array to construct the multi-linear polynomial $f':\mathbb{F}^n\to\mathbb{F}$ such that $f'(\mathbf{x})=f(\mathbf{r},\mathbf{x}[1:n])$.[3]
3. After round $(\ell-n)$ (step 3. in Protocol 4), $\mathcal{P}_0$ continues to run the sumcheck protocol on f' with $\mathcal{V}$ in last n rounds.

In this way, the computation of $\mathcal{P}_i$ is equivalent to running the sumcheck protocol in Virgo on C_i. It accelerates the sumcheck protocol in Virgo by N times without any overhead on the proof size using N distributed machines, which is optimal for distributed algorithms both in asymptotic complexity and in practice. We give the complexity of Protocol 4 in the following.

Complexity of the distributed sumcheck protocol. For the multivariate polynomial of f defined in Equation 1, The total prover work is $O(2^\ell)$ while the prover work for each machine is $O(\frac{2^\ell}{N})$. The communication between N machines is $O(N\ell)$. The proof size and the verifier time are both $O(\ell)$.

4.3 Distributed polynomial commitment

In the last step of the sumcheck phase, the prover needs to prove to the verifier $y=f(r_1,\cdots,r_\ell)$ for some value y. In Virgo, The prover convinces $\mathcal{V}$ of the evaluation by invoking the PC scheme. We present the PC scheme in Virgo and the complexity of the scheme in the following.

Background: the polynomial commitment in Virgo. Let $\mathcal{F}$ be a family of ℓ-variate multi-linear polynomial over $\mathbb{F}$. Let $\mathbb{H},\mathbb{L}$ be two disjoint multiplicative subgroups of $\mathbb{F}$ such that $|\mathbb{H}|=2^\ell$ and $|\mathbb{L}|=\rho|\mathbb{H}|$, where ρ is a power of 2. The polynomial commitment (PC) in Virgo for $f\in\mathcal{F}$ and $\mathbf{r}\in\mathbb{F}^\ell$ consists of the following algorithms:

- $\mathsf{pp}\leftarrow\mathsf{PC.KeyGen}(1^\lambda)$: Given the security parameter λ, the algorithm samples a collision resistant hash function from a hash family as pp.
- $\mathsf{com}_f\leftarrow\mathsf{PC.Commit}(f,\mathsf{pp})$: Given a multi-linear polynomial f, the prover treats 2^ℓ coefficients of f as evaluations of a univariate polynomial f_U on $\mathbb{H}$. The prover uses the inverse fast Fourier transform (IFFT) to compute f_U. Then the prover computes $\mathbf{f}_\mathbb{L}$ as evaluations of f_U on $\mathbb{L}$ via the fast Fourier transform (FFT). Let $\mathsf{com}_f=\mathsf{MT.Commit}(\mathbf{f}_\mathbb{L})$.
- $(y,\pi_f)\leftarrow\mathsf{PC.Open}(f,\mathbf{r},\mathsf{pp})$: The prover computes $y=f(\mathbf{r})$. Given $c=O(\lambda)$ random indexes $(k_1,\cdots,k_c)$, the prover computes $(\mathbf{f}_\mathbb{L}[k_1],\pi_{k_1})=\mathsf{MT.Open}(\mathbf{f}_\mathbb{L},k_1)$, $\cdots$, $(\mathbf{f}_\mathbb{L}[k_c],\pi_{k_c})=\mathsf{MT.Open}(\mathbf{f}_\mathbb{L},k_c)$. Let $\pi_f=(\mathbf{f}_\mathbb{L}[k_1],\pi_{k_1},\cdots,\mathbf{f}_\mathbb{L}[k_c],\pi_{k_c})$.[4]
- $\{1,\emptyset\}\leftarrow\mathsf{PC.Verify}(\mathsf{com}_f,\mathbf{r},y,\pi_f,\mathsf{pp})$: The verifier parses $\pi_f=(\mathbf{q}_\mathbb{L}[k_1],\pi_{k_1},\cdots,\mathbf{q}_\mathbb{L}[k_c],\pi_{k_c})$, then checks that $\mathbf{q}_\mathbb{L}[k_1]$, $\cdots$, $\mathbf{q}_\mathbb{L}[k_c]$ are consistent with y by a certain equation $p(f_\mathbb{L}[k_1],\cdots,f_\mathbb{L}[k_c],y)=0$, [5] and checks that $\mathbf{f}_\mathbb{L}[k_1]$, $\cdots$, $\mathbf{f}_\mathbb{L}[k_c]$ are consistent with com_f by $\mathsf{MT.Verify}(\pi_{k_1},\mathbf{f}_\mathbb{L}[k_1],\mathsf{com}_f)$, $\cdots$, $\mathsf{MT.Verify}(\pi_{k_c},\mathbf{f}_\mathbb{L}[k_c],\mathsf{com}_f)$. If all checks pass, the verifier outputs 1, otherwise the verifier outputs $\emptyset$.

Complexity of PC in Virgo. The prover time is $O(\ell\cdot 2^\ell)$. The proof size is $O(\lambda\ell^2)$ and the verifier time is $O(\lambda\ell^2)$.

In the setting of distributed PC, $\mathcal{P}_i$ knows $f^{(i)}$. With the help of $\tilde{\beta}$ function, we have

$$f(\mathbf{r})=\sum_{i=0}^{N-1}\tilde{\beta}(\mathbf{r}[\ell-n+1:\ell],\mathbf{i})f^{(i)}(\mathbf{r}[1:\ell-n]). \tag{2}$$

A straightforward way for distributed PC is that $\mathcal{P}_i$ runs the PC scheme on $f^{(i)}$ separately. In particular, $\mathcal{P}_i$ invokes PC.Commit to commit $f^{(i)}$ in the beginning of the sumcheck protocol. In the last round, $\mathcal{P}_i$ runs PC.Open to compute $f^{(i)}(\mathbf{r}[1:\ell-n])$ and sends the proof to $\mathcal{V}$. After receiving all $f^{(i)}(\mathbf{r}[1:\ell-n])$ from $\mathcal{P}_i$, $\mathcal{V}$ invokes PC.Verify to validate N polynomial commitments separately. Then $\mathcal{V}$ computes $\tilde{\beta}(\mathbf{r}[\ell-n+1:\ell],\mathbf{i})$ for each i. Finally, $\mathcal{V}$ checks $f(\mathbf{r})=\sum_{i=0}^{N-1}\tilde{\beta}(\mathbf{r}[\ell-n+1:\ell],\mathbf{i})f^{(i)}(\mathbf{r}[1:\ell-n])$.

Although the aforementioned naïve distributed protocol achieves $O(2^\ell(\ell-n))$ in computation time for each machine, the total proof size is $O(\lambda N(\ell-n)^2)$ as the individual proof size for each $\mathcal{P}_i$ is $O(\lambda(\ell-n)^2)$. To reduce the proof size, we optimize the algorithm by aggregating N commitments and N proofs altogether. For simplicity, we assume $\rho=1$ without loss of generality in the multi-linear polynomial commitment[6]. We present the formal protocol of distributed PC in Protocol 6 in Appendix D.

The idea of our scheme is that each $\mathcal{P}_i$ exchanges data with other machines immediately after computing $\mathbf{f}_\mathbb{L}^{(i)}$ instead of invoking MT.Commit on $\mathbf{f}_\mathbb{L}^{(i)}$ directly. The advantage of such arrangement is that the prover aggregates evaluation on the same index into one branch and can open them together by a single Merkle tree proof for this branch. As described in the polynomial commitment of Virgo, the prover needs to open $f_\mathbb{L}$ on some random indexes depending on $\mathbf{r}$ in PC.Open. As $\mathbf{r}$ is identical to each $f^{(i)}$, the prover would open each $f_\mathbb{L}^{(i)}$ at same indexes. If the prover aggregates $f_\mathbb{L}^{(i)}$ by the indexes, she can open N values in one shot by providing only one Merkle tree path instead of naïvely providing N Merkle tree paths, which helps her to save the total proof size by a logarithmic factor in the size of the polynomial.

Specifically, $\mathcal{P}_i$ collects evaluations of $\mathbf{f}_\mathbb{L}^{(0)}[i+1],\cdots,\mathbf{f}_\mathbb{L}^{(N-1)}[i+1]$ with identical index of $(i+1)$ in $\mathbb{L}$ from other machines (step 1. and step 2.). Then $\mathcal{P}_i$ invokes MT.Commit to get a commitment, $com_{h^{(i)}}$, for these values, and submits $com_{h^{(i)}}$ to $\mathcal{P}_0$ (step 3.). $\mathcal{P}_0$ invokes MT.Commit on $com_{h^{(0)}},\cdots,com_{h^{(N-1)}}$ to compute the aggregated commitment, com, and $\mathcal{P}_0$ sends com to $\mathcal{V}$ (step 4.). In the PC.Open phase, given a random index k_j from $\mathcal{V}$, $\mathcal{P}_0$ retrieves $\mathbf{f}_\mathbb{L}^{(N-1)}[k_j],\cdots,\mathbf{f}_\mathbb{L}^{(N-1)}[k_j]$ from $\mathcal{P}_{k_j-1}$, computes $(\mathsf{com}_{h^{(k_j-1)}},\pi_{k_j})=\mathsf{MT.Open}(\mathsf{com},k_j)$, and sends these messages to $\mathcal{V}$ (step 5. and step 6.). $\mathcal{V}$ can validate N evaluations by invoking MT.Verify only once (step 7.).

[3] The approach can extend to the product of two multi-linear polynomials, which matches the case in Virgo.

[4] The prover also computes $\log|\mathbb{L}|$ polynomials of $f_1,\cdots,f_{\log|\mathbb{L}|}$ depending on f. But sizes of these polynomials are $\frac{|\mathbb{L}|}{2},\cdots,1$ respectively. The prover commits these polynomial and opens them on at most c locations correspondingly. Our techniques on distributed commitment and opening can apply to these smaller polynomials easily. We omit the process for simplicity. It brings a logarithmic factor in the size of the polynomial on the proof size and the verification time.

[5] p also takes all openings on polynomials of $f_1,\cdots,f_{\log|\mathbb{L}|}$ (at most c for each polynomial) as input, we omit them for simplicity.

[6] In Virgo, $\rho=32$ for security requirements. Our scheme can extend to $\rho=32$ easily.

With this approach, we reduce the proof size to $O(\lambda(N+\ell^2))$. And the complexity of Protocol 6 is shown in the following.

Complexity of distributed PC. Given that f is a multi-linear polynomial with ℓ variables, the total communication among N machines is $O(2^\ell)$. The total prover work is $O(2^\ell \cdot \ell)$ while the prover work for each device is $(\frac{2^\ell}{N} \cdot \ell)$. The proof size is $O(\lambda(N+\ell^2))$. The verification cost is $O(\lambda(N+\ell^2))$.

4.4 Combining everything together

In this section, we combine the distributed sumcheck and the distributed PC altogether to build deVirgo.

For a data-parallel layered arithmetic circuit C with N copies and d layers, following the workflow of Virgo in Protocol 7 in Appendix E, our distributed prover replaces d sumcheck schemes in Virgo by d distributed sumcheck schemes, and replaces the PC scheme in Virgo by our distributed PC scheme to generate the proof. We present the formal protocol of deVirgo in Protocol 8 in Appendix F. And we have the theorem as follows.

Theorem 4.1. *Protocol 8 is an argument of knowledge satisfying the completeness and knowledge soundness in Definition 2.2 for the relation $C(\mathbf{x},\mathbf{w}) = \mathbf{1}$, where C consists of N identical copies of $C_0,\cdots,C_{N-1}$.*

Proof (sketch). **Completeness.** The completeness is straightforward.

Knowledge soundness. deVirgo generates the same proof as Virgo for d sumcheck protocols. So we only need to consider the knowledge soundness of distributed PC scheme. If the commitment of f is inconsistent with the opening of $f(\mathbf{r})$ in the distributed PC scheme, there must exist at least one $f^{(i)}(\mathbf{r}[1:\ell-n])$ being inconsistent with the commitment f by Equation 2. Otherwise, when all $f^{(i)}(\mathbf{r}[1:\ell-n])$ are consistent with the commitment of f, $f(\mathbf{r})$ must be consistent with the commitment of f. As shown in Protocol 6, com_f is equivalent to $com_{f^{(i)}}$ with additional dummy messages in each element of the vector in the Merkle tree commitment. It does not affect the soundness of the PC in Virgo in the random oracle model [75, 76]. The verifier outputs 0 in the PC.Verify phase with the probability of $(1-\mathsf{negl}(\lambda))$. Therefore, deVirgo still satisfies knowledge soundness.

The zero-knowledge property is not necessary as there is no private witness in the setting of zkbridge. However, we can achieve zero-knowledge for deVirgo by adding some hiding polynomials. Virgo uses the same method to achieve zero-knowledge. ∎

Additionally, Fiore and Nitulescu [50] introduced the notion of O-SNARK for SNARK over authenticated data such as cryptographic signatures. Protocol 8 is an O-SNARK for any oracle family, albeit in the random oracle model. To see this, Virgo relies on the construction of computationally sound proofs of Micali [60] to achieve non-interactive proof and knowledge soundness in the random oracle model, which has been proven to be O-SNARK in [50]. Hence Virgo is an O-SNARK, and so is deVirgo because deVirgo also relies on the same model.

Protocol 8 achieves optimal linear scalability on data-parallel circuits without significant overhead on the proof size. In particular, our protocol accelerates Virgo by N times given N distributed machines. Additionally, the proof size in our scheme is reduced by a factor of N compared to the naïve solution of running each subcopy of data-parallel circuits separately and generating N proofs. The complexity of Protocol 8 is shown in the following.

Complexity of distributed Virgo. Given a data-parallel layered arithmetic circuit C with N sub-copies, each having d layers and m inputs, the total prover work of Protocol 8 is $O(|C|+Nm\log m)$. The prover work for a single machine is $O(|C|/N+m\log m)$, and the total communication among machines is $O(Nm+Nd\log|C|)$. The proof size is $O(d\log|C|+\lambda(N+\log^2 m))$. The verification cost is $O(d\log|C|+\lambda(N+\log^2 m))$.

5 REDUCING PROOF SIZE AND VERIFIER TIME

Although deVirgo improves the prover time by orders of magnitude, we want to further reduce the cost of the verification time and the proof size. As mentioned in the above section, the circuit which validates over 100 signatures is giant due to non-compatible instructions on different curves across different blockchains. Additionally, Virgo's proof size, which is around 210KB for a circuit with 10 million gates, is large in practice. Thus we cannot post deVirgo's proof on-chain and validate the proof directly. Aiming at smaller proof size and simpler verification on-chain, we propose to further compress the proof by recursive proofs with two layers. Intuitively, for a large-scale statement $(\mathbf{x},\mathbf{w}) \in \mathcal{R}$ in Definition 2.2, the prover generates the proof π_1 by a protocol with fast prover time in the first layer. If the length of π_1 is not as short as desired, then the prover can produce a shorter proof π_2 by invoking another protocol for $(\mathbf{x},\pi_1) \in \mathcal{R}'$ in the second layer, where $\mathcal{R}'$ represents that π_1 is a valid proof for $(\mathbf{x},\mathbf{w}) \in \mathcal{R}$. To shrink the proof size and simplify the verification as much as possible, we choose Groth16 as the second layer ZKP protocol since Groth16 has constant proof size and fast verification time. Moreover, the curve in Groth16 is natively supported by Ethereum, which is beneficial for saving on-chain cost on Ethereum. In our approach, the prover invokes deVirgo to generate π_1 on the initial circuit in the first layer. In the second layer, the prover invokes Groth16 to generate π_2 on the circuit implementing the verification algorithm of deVirgo where $|\pi_2| \ll |\pi_1|$. The prover only needs to submit π_2 on-chain for verification. The recursion helps cross-chain bridges to reduce gas cost on blockchains because of simple verification on the compatible curve. The security of recursive proofs relies on random oracle assumption, which can be instantiated by a cryptographic hash function in practice [45].

Performance gains. We use the signature validation circuit for Cosmos [11] as an example to show concrete numbers of the verification circuit of deVirgo in Table 1. We record the size of the whole verification circuit in the 2^{nd} column, the size for the GKR part in the 3^{rd} column, and the size for the PC part in the 4^{th} column, as the number of signatures in data-parallel circuits increases from 1 to 128 in the 1^{st} column. The number of gates in the 2^{nd} column equals the sum of numbers of gates in the 3^{rd} column and the 4^{th} column. As shown in Table 1, although the data-parallel circuit size expands,

# of sigs	Total circuit size	Circuit size for GKR part	Circuit size for PC part
1	1.2×10^7 gates	8.4×10^6 gates	3.3×10^6 gates
4	1.2×10^7 gates	8.4×10^6 gates	4.0×10^6 gates
32	1.3×10^7 gates	8.4×10^6 gates	4.7×10^6 gates
128	1.4×10^7 gates	8.4×10^6 gates	5.4×10^6 gates

Table 1: The verification circuit size of deVirgo

the size for the sumcheck part in deVirgo's verification circuit does not change. That is because the verification for the GKR part is only based on the structure of the sub-circuit, which is identical among different copies. However, the size for the PC part in deVirgo's verification circuit up-scales sub-linearly in the number of copies due to the growth of the polynomial size. Even given 128 copies of the signature validation circuit, the bottleneck of deVirgo's verification circuit is the sumcheck part. Therefore, the recursive proof size and the recursive verification cost are independent of the number of signatures to validate in our instance. In addition, the prover time of Groth16 on the verification circuit of deVirgo is only 25% of the prover time of deVirgo in practice. Therefore, our recursive proof scheme reduces the on-chain proof verification cost from $\sim 8\times10^7$ gas (an estimation) to less than 2.3×10^5 gas.

6 IMPLEMENTATION AND EVALUATION

To demonstrate the practicality of zkBridge, we implement a prototype from Cosmos [11] (a PoS blockchain built on top of the Tendermint [57] protocol) to Ethereum, and from Ethereum to other EVM-compatible chains such as BSC. Supports for other blockchains can be similarly implemented with additional engineering effort, as long as they support light client protocols defined in Definition 2.1. In this section, we discuss implementation detail, its performance, as well as operational cost.

The bridge from Cosmos to Ethereum is realized with the full blown zkBridge protocol presented so far to achieve practical performance. In comparison, the direction from Ethereum to other EVM-compatible chains incurs much less overhead for proof generation and does not require deVirgo. Therefore, in what follows, we mainly focus on the direction from Cosmos to Ethereum.

6.1 Implementation details

The bridge from Cosmos to Ethereum consists of four components: a relayer that fetches Cosmos block headers and sends them to Ethereum (implemented in 300+ lines of Python), deVirgo (implemented in 10000+ lines of C++) for distributed proof generation, a handcrafted recursive verification circuit, and an updater contract on Ethereum (implemented in 600+ lines of Solidity). Our signature verification circuit is based on the optimized signature verification circuit [14]. However, we use Gnark instead of Circom as in [14] for better efficiency for proof generation.

6.1.1 Generating correctness proofs. Relay nodes submit Cosmos block headers to the updater contract on Ethereum along with correctness proofs, which proves that the block is properly signed by the Cosmos validator committee appointed by the previous block. (In Cosmos a hash of the validator committee members is included in the previous block.)

In Cosmos, each block header contains about 128 EdDSA signatures (on Curve25519), Merkle roots for transactions and states, along with other metadata, where 32 top signatures are required to achieve super-majority stakes. However, the most efficient curve supported by the Ethereum Virtual Machine (EVM) is BN254. To verify Cosmos digital signatures in EVM, one must simulate Curve25519 on curve BN254, which will lead to large circuits. Concretely, to verify a Cosmos block header (mainly, to verify about 32 signatures), we need about 64 million gates. We implement deVirgo (Section 4) and recursive verification (Section 5) to accelerate proof generation and verification.

Moreover, in practical deployment, multiple relayers can form a pipeline to increase the throughput. Looking ahead, based on the evaluation results, our implementation can handle 1 second block time in Cosmos with 120+ capable relayers in the network.

For proof verification, we build an outer circuit that verifies Virgo proofs and use Gnark [16] to generate the final Groth16 proof that can be efficiently verified by the updater contract on Ethereum.

6.1.2 The updater contract. We implement the updater contract on Ethereum in Solidity that verifies Groth16 proofs and keeps a list of the Cosmos block headers in its persistent storage. The cost of verifying a Groth16 proof on-chain is less than $230K$ gas.

The updater contract exposes a simple API which takes block height as its input, and returns the corresponding block header. The receiver contracts can then use the block header to complete application-specific verification.

Batching. Instead of calling the updater contract on every new block header, we implemented *batching* where the updater contract stores Merkle roots of batches of B consecutive block headers. The prover will first collect B consecutive blocks, and then makes a unified proof for all B blocks. The updater contract will only need to verify one proof for the batch of B blocks. After the verification, the updater contract checks the difficulty, stores the block headers, and updates the light-client state. Storing one Merkle root every B blocks also reduces storage cost. Thus B can be set to balance user experience and cost: With a larger B, users need to wait longer, but the cost of running the system is lower.

We implement the aforementioned batched proof verification and show the experimental results in Section 6.2. In addition, we propose a more complex batching optimization presented in Appendix G for further optimization.

With batching, the cost for storing block headers and maintaining light-client states is amortized across B blocks. The bulk of the cost incurred by the updater contract is SNARK proof verification, which is the focus of our evaluation below.

6.2 Evaluation

We evaluate the performance of zkBridge (from Cosmos to Ethereum) from four aspects: proof generation time, proof generation communication cost, proof size, and on-chain verification cost.

6.2.1 Experiment setup. We envision that a relayer node in zkBridge will be deployed as a service in a managed network, therefore we evaluate zkBridge in a data-center-like environment. Specifically,

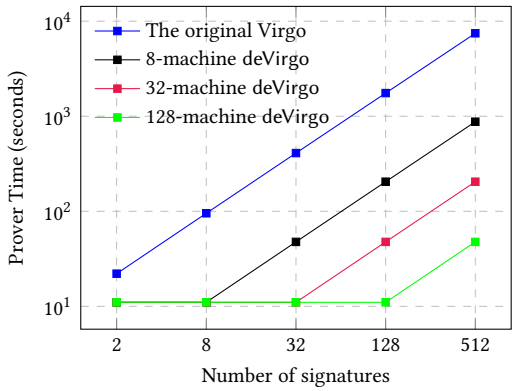

Figure 2: Prover time of deVirgo and the original Virgo for Cosmos block header verification.

we run all the experiments on 128 AWS EC2 c5.24xlarge instances with the Intel(R) Xeon(R) Platinum 8275CL CPU @ 3.00GHz and 192GB of RAM. Our implementation for the proof generation is parallelized with at most 128 machines. We report the average running time of 10 executions. Whenever applicable, we report costs both in terms of running time and monetary expenses.

6.2.2 Proof generation time of deVirgo. We first evaluate the main cryptographic building block—deVirgo—and compare its performance with the original Virgo [76]. The source code of the original Virgo is obtained at https://github.com/sunblaze-ucb/Virgo. We run both protocols on the same circuit for correctness proofs, which mainly consists of N invocation of EdDSA signature verification.

Figure 2 shows the prover time (in seconds) against different N. For deVirgo, we repeat the experiment with 8, 32, 128 distributed machines. According to Fig. 2, the prover time of the original Virgo increases linearly in the number of signatures N, while the prover time of deVirgo is almost independent of N until N is greater than the number of servers when computation becomes an bottleneck. The linear scalability suggests that the workload of each machine only depends on its own sub-circuit and the communication overhead is small. Table 2 reports the communication cost among parallel machines. The total communication cost is linear in the number of machines, consistent with the analysis in Section 4.4, with each machine sending and receiving around 1 GB of data. Since we envision a relayer node in zkBridge to be deployed in a data-center-like environment, the amount of traffic is reasonable.

In practice, the Cosmos block headers typically have $N = 128$ signatures while 32 top signatures are sufficient to achieve supermajority. Therefore, generating a correctness proof for a Cosmos block header would take more than 400 seconds with the original Virgo, but it decreases to 13.28 seconds with deVirgo, implying a 30x speedup. In general, as is consistent with the analysis in Section 4, deVirgo accelerates the proof generation on data-parallel circuits with N copies by a factor of almost N, which is optimal for distributed algorithms.

6.2.3 Proof size and verification time. To reduce on-chain verification cost, we use the recursive verification technique presented in Section 5. Now we report on its efficacy.

Recursive proof generation time. We implement recursive verification by invoking Groth16 (constructed using gnark [16]) on the verification circuit. We report the proof time in deVirgo, the generation time of recursive proofs (the column marked RV), and the sum, in Table 2, for various numbers of signatures. The RV time almost remains constant in the number of signatures verified by the deVirgo proofs. That is because of the data-parallel structure of the state transition proof circuit: the size of Groth16 verification circuit is only a function of the size of a sub-circuit.

The main benefit of recursive verification is a reduction in both proof size and verification cost.

Reduced proof size. Table 2 shows the proof size both with and without recursive verification. For the practical scenario where $N = 32$, the proof size is reduced from 1.9 MB to 131 Bytes. Overall, for $N = 32$, with an increase of about 25% in prover time, we get a reduction of around 14000x in proof size.

Reduced on-chain verification cost. The final proof is 131 Bytes while the final verification only costs 3 pairings. As shown in Table 2, the on-chain verification cost is constant (227K). In comparison, without recursive verification, directly verifying Virgo proofs on-chain would be infeasible. (Our estimation of the gas cost is 78M, which far exceeds the single block gas limit 30M).

6.2.4 Comparison with optimistic bridges. With batching, the confirmation latency of zkBridge is under 2 minutes, including 3×32 seconds for waiting for all blocks in the batch and another 20 seconds for proof generation. While this is not blazing fast, in comparison, optimistic bridges have much longer confirmation time. E.g., NEAR's Rainbow bridge has a challenge window of 4 hours [15] before which the transfer cannot be confirmed.

6.3 Cost analysis

In this section, we analyze the operational cost of zkBridge, which consists of off-chain cost (generating proofs) and on-chain cost (storing headers and verifying proofs).

Off-chain cost. Off-chain cost can vary significantly based on the deployment. While we use AWS in our performance benchmark, it may not be the best option for practical deployment. AWS service is expensive due to its high margin, elastic scaling capability, and high reliability, which isn't necessary for our proof generation process. To show a representative range, we consider two deployment options: cloud-based and self-hosted. For cloud-based deployment, we search for reputable and economical dedicated server rental services and choose Hetzner[17] as an example. For self-hosted options, we calculate the cost to purchase the hardware and the on-going cost (mainly the electricity).

On AWS c5.24xlarge, it takes 18 seconds to generate a proof with 32 machines. Renting a server with a similar spec as AWS c5.24xlarge from Hetzner costs $253.12 per month, thus the cost of cloud-based deployment with Hetzner will be around $8100 per month for all 32 machines. It translates to $0.02 per block.

	Proof Gen. Time (seconds)			Proof Gen. Comm. (GB)		Proof Size (Bytes)		On-chain Ver. Cost (gas)	
# of sigs	deVirgo	RV	total	total	per-machine	w/o RV	w/ RV	w/o RV	w/ RV
8	12.52	4.90	17.42	7.34	0.92	1946476	131	78M	227K
32	12.80	5.41	18.21	32.24	1.01	1952492	131	78M	227K
128	13.28	5.49	18.77	131.89	1.03	1958508	131	79M	227K

Table 2: Evaluation results. RV is the shorthand for recursive verification.

To estimate the cost for self-hosted deployment, we use online tools to configure a machine with a comparable spec to that in AWS. Table 3 in Appendix H reports the configuration and each machine costs around \$4.5k. The total setup cost is thus around \$4.5k $\times 32 =$ \$144k. For self-hosted servers, the main on-going cost is electricity. With each machine consuming 657W power, a 32-machine cluster consumes 0.105 kWh per block. Assuming US average electricity rate \$0.12/KWh [8], the electricity cost is \$0.012 per block, or \$5184 per month.

On-chain cost. On-chain cost refers to the total gas used for on-chain operation, and we report the equivalent USD cost based on the gas price (about 20 gwei) and ETH price (about 1600 USD) at the time of writing (August 2022). If we use efficient batched proofs, for a batch of N headers, the bulk of the verification cost is that of verifying one Groth16 proof, which costs less than $230K$ gas, roughly \$7.36. If we choose $N = 32$ for example, the on-chain cost will be \$0.23 per block. Moreover, if we adopt the optimization mentioned in appendix G, we can further reduce the on-chain cost and offload the cost to users if the number of users is large.

6.4 Ethereum to other EVM-compatible chains

So far we have focused on the bridge from Cosmos to Ethereum because generating and verifying correctness proofs for that direction is challenging. We also implement a prototype of a bridge from Ethereum to other EVM-compatible blockchains.

The high level idea is simple: upon receiving a block header, the updater contract on the receiver chain verifies the PoW and appends it to the list of headers if the verification is passed. However, a wrinkle to the implementation is that Ethereum uses a memory hard hash function, EthHash [71], which is prohibitively inefficient to run on-chain. Basically, EthHash involves randomly accessing elements in a 1 gigabyte dataset (called a DAG) derived from a public seed and the block height. Generating the DAGs on-chain is prohibitively expensive.

Our idea is to pre-compute many DAGs off-chain and store their hashes on-chain. Specifically, as part of zkBridge setup, we pre-compute 2,048 DAGs , build a Merkle tree for each DAG using MiMC [33], and store the Merkle roots on-chain. Per EthHash specification, a new DAG is generated every 30,000 blocks, so 2,048 of them can last for 10 years; the off-chain pre-computation process takes no more than 4 days. Then, the correctness proofs will show that a given EthHash PoW is correct with respect to the Merkle root of the DAG corresponding to the block in question. We emphasize that the setup process is verifiable and anyone can verify the published Merkle roots on their own before using the service. The circuit for verifying EthHash PoW has around 2 million gates.

The rest of the protocol is the same as a regular light client, which involves storing the headers, following the longest chain by computing accumulated difficulty, resolving forks, etc.

Cost analysis. Since EthHash PoW verification circuit has only around 2 million constraints, a single machine with the configuration in Appendix H can generate a proof within 10 seconds. As long as the receiver chain is EVM-compatible, the on-chain cost will be close to that presented in Section 6.3, since the updater contract only verifies Groth16 proofs in all cases.

7 RELATED WORK

In this section, we compare zkBridge to existing cross-chain bridge systems and the line of work on zk-rollups which also uses ZKPs for scalability and security.

Cross-chain bridges in the wild and security issues. Cross-chain systems are widely deployed and used. Below we briefly survey the representative ones. The list is not meant to be exhaustive. PolyNetwork [3] is an interoperability protocol using a side-chain as the relay with a two-phase commitment protocol. Wormhole [5] is a generic message-passing protocol secured by a network of guardian nodes, and its security relies on 2/3 of the committee being honest. Ronin operates in a similar model. While relying on decentralized committees for security, practical deployment usually opts for relatively small ones for efficiency (e.g., 9 in case of Ronin). Committee breaches are far from being rare in practice. In a recent exploit against Ronin [27], the attacker obtained five of the nine validator keys, stealing 624 million USD. PolyNetwork and Wormhole were also recently attacked, losing \$611m [6] and \$326m [10] respectively. Key compromise was suspected in the PolyNetwork attack.

An alternative design is to leverage economic incentives. Nomad [7] (which recently lost more than \$190m to hackers due to an implementation bug [22]) and Near's Rainbow Bridge [4] are such examples. These systems require participants to deposit a collateral, and rely on a watchdog service to continuously monitor the blockchain and confiscate offenders' collateral upon detecting invalid updates. Optimistic protocols fundamentally require a long confirmation latency in order to ensure invalid updates can be detected with high probability (e.g., Near [4] requires 4 hours). Moreover, participants must deposit significantly collateral (e.g., 20 ETH in Near [4]). Both issues can be avoided by zkBridge.

In summary, compared to existing protocols, zkBridge achieve both efficiency and cryptographic assurance. zkBridge is "trustless" in that it does not require extra assumptions other than those of blockchains and underlying cryptographic protocols. It also avoids the long confirmation of optimistic protocols.

zk-rollups. Rollups are protocols that batch transaction execution using ZKPs to scale up the layer-1 blockchains. Starkware [28], ZkSync [31], and Polygon Zero [25] are a few examples.

These zk-rollup solutions have not been applied to the bridge setting, where our work is the first to use ZKP to enable a decentralized trustless bridge. In addition, the current zk-rollup work in general has not dealt with such large circuits as in zkBridge, whereas in our work, we need to design and develop a number of techniques including deVirgo and proof recursion to make building a ZKP-based bridge practical for the first time. In particular, we leverage the data parallelism of the circuits to obtain a ZKP protocol that is more than 100x faster than existing protocols for the workload in zkBridge and combine it with proof recursion for efficient on-chain verification. The idea behind deVirgo protocol may be applicable to zk-rollups too.

ACKNOWLEDGMENTS

This material is in part based upon work supported by the National Science Foundation (NSF) under Grant No. TWC-1518899 and Grant No. 2144625, DARPA under Grant No. N66001-15-C-4066, the Center for Responsible, Decentralized Intelligence at Berkeley (Berkeley RDI), the Center for Long-Term Cybersecurity (CLTC), the Simons Foundation, and NTT Research. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of these institutes.

REFERENCES

[1] 2014. Filecoin: A Decentralized Storage Network. (2014). https://filecoin.io/filecoin.pdf
[2] 2017. Hyperledger Sawtooth. (2017). https://sawtooth.hyperledger.org/
[3] 2020. Poly Network. https://poly.network/. (2020).
[4] 2020. Rainbow Bridge. https://near.org/bridge/. (2020).
[5] 2020. Wormhole Solana. https://solana.com/wormhole. (2020).
[6] 2021. At least $611 million stolen in massive cross-chain hack. (2021).
[7] 2021. Nomad Protocol. https://docs.nomad.xyz/the-nomad-protocol/overview. (2021).
[8] 2022. Average Price of Electricity. https://www.eia.gov/electricity/monthly/epm_table_grapher.php?t=epmt_5_6_a. (2022).
[9] 2022. Axelar. https://axelar.network/. (2022).
[10] 2022. Blockchain Bridge Wormhole Suffers Possible Exploit Worth Over $326M. (2022). https://www.coindesk.com/tech/2022/02/02/blockchain-bridge-wormhole-suffers-possible-exploit-worth-over-250m/
[11] 2022. Cosmos. https://cosmos.network/. (2022).
[12] 2022. Cryptocurrency prices, charts and market capitalizations. (2022). https://coinmarketcap.com/
[13] 2022. ed25519-circom. https://github.com/Electron-Labs/ed25519-circom. (2022).
[14] 2022. ed25519-circom. https://github.com/Electron-Labs/ed25519-circom. (2022).
[15] 2022. ETH-NEAR Rainbow Bridge – NEAR Protocol. (2022). https://near.org/blog/eth-near-rainbow-bridge/
[16] 2022. gnark. https://docs.gnark.consensys.net/en/latest/. (2022).
[17] 2022. Hetzner. https://www.hetzner.com/. (2022).
[18] 2022. LayerZero. https://layerzero.network/. (2022).
[19] 2022. libSNARK. https://github.com/scipr-lab/libsnark. (2022).
[20] 2022. Multi-chain future likely as Ethereum's DeFi dominance declines | Bloomberg Professional Services. (2022). https://www.bloomberg.com/professional/blog/multi-chain-future-likely-as-ethereums-defi-dominance-declines/
[21] 2022. A multichain approach is the future of the blockchain industry. (2022). https://cointelegraph.com/news/a-multichain-approach-is-the-future-of-the-blockchain-industry
[22] 2022. Nomad crypto bridge loses $200 million in "chaotic" hack. https://www.theverge.com/2022/8/2/23288785/nomad-bridge-200-million-chaotic-hack-smart-contract-cryptocurrency. (2022).
[23] 2022. Polygon Hermez. https://polygon.technology/solutions/polygon-hermez/. (2022).
[24] 2022. Polygon Miden. https://polygon.technology/solutions/polygon-miden/. (2022).
[25] 2022. Polygon Zero. https://polygon.technology/solutions/polygon-zero/. (2022).
[26] 2022. Risc Zero. https://www.risczero.com/. (2022).
[27] 2022. Ronin Attack Shows Cross-Chain Crypto Is a 'Bridge' Too Far. (2022). https://www.coindesk.com/layer2/2022/04/05/ronin-attack-shows-cross-chain-crypto-is-a-bridge-too-far/
[28] 2022. Starkware. https://starkware.co/. (2022).
[29] 2022. Vbuterin comments on [AMA] We are the EF's Research Team (Pt. 7: 07 January, 2022). (2022). https://old.reddit.com/r/ethereum/comments/rwojtk/ama_we_are_the_efs_research_team_pt_7_07_january/hrngyk8/
[30] 2022. YouTube includes NFTs in new creator tools. (2022). https://www.nbcnews.com/pop-culture/viral/youtube-includes-nfts-new-creator-tools-rcna15813
[31] 2022. ZkSync. https://zksync.io/. (2022).
[32] 2022-04-24. Beeple sold an NFT for $69 million - The Verge. (2022-04-24). https://www.theverge.com/2021/3/11/22325054/beeple-christies-nft-sale-cost-everydays-69-million
[33] Martin Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and Tyge Tiessen. 2016. MiMC: Efficient encryption and cryptographic hashing with minimal multiplicative complexity. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 191–219.
[34] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. 2017. Ligero: Lightweight sublinear arguments without a trusted setup. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*.
[35] Giuseppe Ateniese, Ilario Bonacina, Antonio Faonio, and Nicola Galesi. 2014. Proofs of space: When space is of the essence. In *International Conference on Security and Cryptography for Networks*. Springer, 538–557.
[36] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. 2018. Scalable, transparent, and post-quantum secure computational integrity. *Cryptology ePrint* (2018).
[37] Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P. Ward. 2019. Aurora: Transparent Succinct Arguments for R1CS. In *EUROCRYPT 2019*. 103–128.
[38] Iddo Bentov, Charles Lee, Alex Mizrahi, and Meni Rosenfeld. 2014. Proof of activity: Extending bitcoin's proof of work via proof of stake [extended abstract] y. *ACM SIGMETRICS Performance Evaluation Review* 42, 3 (2014), 34–37.
[39] Iddo Bentov, Rafael Pass, and Elaine Shi. 2016. Snow White: Provably Secure Proofs of Stake. *IACR Cryptol. ePrint Arch.* 2016, 919 (2016).
[40] Daniel J Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. 2012. High-speed high-security signatures. *Journal of cryptographic engineering* 2, 2 (2012), 77–89.
[41] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell. Bulletproofs: Short Proofs for Confidential Transactions and More. In *Proceedings of the Symposium on Security and Privacy (SP), 2018*, Vol. 00. 319–338.
[42] Panagiotis Chatzigiannis, Foteini Baldimtsi, and Konstantinos Chalkias. 2021. SoK: Blockchain Light Clients. *Cryptology ePrint Archive* (2021).
[43] Alessandro Chiesa, Michael A. Forbes, and Nicholas Spooner. 2017. A Zero Knowledge Sumcheck and its Applications. *CoRR* abs/1704.02086 (2017). arXiv:1704.02086 http://arxiv.org/abs/1704.02086
[44] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas Ward. 2020. Marlin: Preprocessing zkSNARKs with Universal and Updatable SRS. In *EUROCRYPT 2020*. 738–768.
[45] Alessandro Chiesa, Dev Ojha, and Nicholas Spooner. 2020. Fractal: Post-quantum and Transparent Recursive Proofs from Holography. In *EUROCRYPT 2020*. 769–793.
[46] Graham Cormode, Michael Mitzenmacher, and Justin Thaler. Practical Verified Computation with Streaming Interactive Proofs. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference (ITCS '12)*.
[47] Bernardo David, Peter Ga, Aggelos Kiayias, and Alexander Russell. 2017. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake protocol. *Cryptology ePrint Archive* (2017).
[48] Stefan Dziembowski, Sebastian Faust, Vladimir Kolmogorov, and Krzysztof Pietrzak. 2015. Proofs of space. In *Annual Cryptology Conference*. Springer, 585–605.
[49] Amos Fiat and Adi Shamir. How to Prove Yourself: Practical Solutions to Identification and Signature Problems. In *Crypto 1986*.
[50] Dario Fiore and Anca Nitulescu. 2016. On the (in) security of SNARKs in the presence of oracles. In *Theory of Cryptography Conference*. Springer, 108–138.
[51] Ariel Gabizon, Zachary J Williamson, and Oana Ciobotaru. 2019. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. *Cryptology ePrint Archive* (2019).
[52] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. 2017. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th symposium on operating systems principles*. 51–68.
[53] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. 2015. Delegating Computation: Interactive Proofs for Muggles. *J. ACM* 62, 4, Article 27 (Sept. 2015), 64 pages.
[54] Jens Groth. 2016. On the Size of Pairing-Based Non-interactive Arguments. In *EUROCRYPT 2016*. 305–326.

[55] Jessica Hamlin. 2022. Big investors are finally serious about crypto. but experienced talent is still scarce. (Mar 2022). https://www.institutionalinvestor.com/article/b1x0gr2y3dzzp3/Big-Investors-Are-Finally-Serious-About-Crypto-But-Experienced-Talent-Is-Still-Scarce

[56] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. 2017. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Annual international cryptology conference*. Springer, 357–388.

[57] Jae Kwon. 2014. Tendermint: Consensus without mining. *Draft v. 0.6, fall* 1, 11 (2014).

[58] Carsten Lund, Lance Fortnow, Howard Karloff, and Noam Nisan. 1992. Algebraic Methods for Interactive Proof Systems. *J. ACM* 39, 4 (Oct. 1992), 859–868.

[59] Ralph C Merkle. 1987. A digital signature based on a conventional encryption function. In *Conference on the theory and application of cryptographic techniques*.

[60] Silvio Micali. 2000. Computationally Sound Proofs. *SIAM J. Comput.* (2000).

[61] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review* (2008), 21260.

[62] Kaihua Qin, Liyi Zhou, Benjamin Livshits, and Arthur Gervais. 2021. Attacking the defi ecosystem with flash loans for fun and profit. In *International Conference on Financial Cryptography and Data Security*. Springer, 3–32.

[63] Ling Ren and Srinivas Devadas. 2016. Proof of space from stacked expanders. In *Theory of Cryptography Conference*. Springer, 262–285.

[64] Srinath Setty. 2020. Spartan: Efficient and General-Purpose zkSNARKs Without Trusted Setup. In *CRYPTO 2020*. Springer International Publishing, 704–737.

[65] Shravan Srinivasan, Alexander Chepurnoy, Charalampos Papamanthou, Alin Tomescu, and Yupeng Zhang. 2021. Hyperproofs: Aggregating and Maintaining Proofs in Vector Commitments. *IACR Cryptol. ePrint Arch.* (2021), 599.

[66] Justin Thaler. 2013. Time-Optimal Interactive Proofs for Circuit Evaluation. In *Advances in Cryptology – CRYPTO 2013*, Ran Canetti and Juan A. Garay (Eds.).

[67] Justin Thaler. 2015. A Note on the GKR Protocol. (2015). Available at http://people.cs.georgetown.edu/jthaler/GKRNote.pdf.

[68] Riad S Wahby, Max Howald, Siddharth Garg, Abhi Shelat, and Michael Walfish. 2016. Verifiable asics. In *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 759–778.

[69] Riad S Wahby, Ioanna Tzialla, Abhi Shelat, Justin Thaler, and Michael Walfish. 2018. Doubly-efficient zkSNARKs without trusted setup. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 926–943.

[70] Lawrence Wintermeyer. 2021. Institutional money is pouring into the crypto market and its only going to grow. (Aug 2021). https://www.forbes.com/sites/lawrencewintermeyer/2021/08/12/institutional-money-is-pouring-into-the-crypto-market-and-its-only-going-to-grow/?sh=2660a69d1459

[71] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.

[72] Howard Wu, Wenting Zheng, Alessandro Chiesa, Raluca Ada Popa, and Ion Stoica. 2018. DIZK: A Distributed Zero-Knowledge Proof System. (2018).

[73] Tiacheng Xie, Jiaheng Zhang, Yupeng Zhang, Charalampos Papamanthou, and Dawn Song. 2019. Libra: Succinct Zero-Knowledge Proofs with Optimal Prover Computation. In *Advances in Cryptology (CRYPTO)*.

[74] Jiaheng Zhang, Tianyi Liu, Weijie Wang, Yinuo Zhang, Dawn Song, Xiang Xie, and Yupeng Zhang. 2021. Doubly efficient interactive proofs for general arithmetic circuits with linear prover time. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 159–177.

[75] Jiaheng Zhang, Tiancheng Xie, Thang Hoang, Elaine Shi, and Yupeng Zhang. 2022. Polynomial Commitment with a {One-to-Many} Prover and Applications. In *31st USENIX Security Symposium (USENIX Security 22)*. 2965–2982.

[76] Jiaheng Zhang, Tiancheng Xie, Y. Zhang, and D. Song. 2020. Transparent Polynomial Delegation and Its Applications to Zero Knowledge Proof. *2020 IEEE Symposium on Security and Privacy (SP)* (2020), 859–876.

[77] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. 2017. vSQL: Verifying Arbitrary SQL Queries over Dynamic Outsourced Databases. In *IEEE Symposium on Security and Privacy (S&P) 2017*.

[78] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. 2018. vRAM: Faster verifiable RAM with program-independent preprocessing. In *Proceeding of IEEE Symposium on Security and Privacy (S&P)*.

PROTOCOL 3 (**SUMCHECK**). *The protocol proceeds in ℓ rounds.*

- *In the first round, $\mathcal{P}$ sends a univariate polynomial*

$$f_1(x_1) \stackrel{def}{=} \sum_{b_2,\ldots,b_\ell \in \{0,1\}} f(x_1,b_2,\ldots,b_\ell),$$

  $\mathcal{V}$ checks $H = f_1(0)+f_1(1)$. Then $\mathcal{V}$ sends a random challenge $r_1 \in \mathbb{F}$ to $\mathcal{P}$.
- *In the i-th round, where $2 \le i \le \ell-1$, $\mathcal{P}$ sends a univariate polynomial*

$$f_i(x_i) \stackrel{def}{=} \sum_{b_{i+1},\ldots,b_\ell \in \{0,1\}} f(r_1,\ldots,r_{i-1},x_i,b_{i+1},\ldots,b_\ell),$$

  $\mathcal{V}$ checks $f_{i-1}(r_{i-1}) = f_i(0)+f_i(1)$, and sends a random challenge $r_i \in \mathbb{F}$ to $\mathcal{P}$.
- *In the ℓ-th round, $\mathcal{P}$ sends a univariate polynomial*

$$f_\ell(x_\ell) \stackrel{def}{=} f(r_1,r_2,\ldots,r_{l-1},x_\ell),$$

  $\mathcal{V}$ checks $f_{\ell-1}(r_{\ell-1}) = f_\ell(0) + f_\ell(1)$. The verifier generates a random challenge $r_\ell \in \mathbb{F}$. Given oracle access to an evaluation $f(r_1, r_2, \ldots, r_\ell)$ of f, $\mathcal{V}$ will accept if and only if $f_\ell(r_\ell) = f(r_1,r_2,\ldots,r_\ell)$. The oracle access can be instantiated by PC.

A BACKGROUND: THE SUMCHECK PROTOCOL

The sumcheck protocol is given in Protocol 3.

B THE DISTRIBUTED SUMCHECK PROTOCOL

The distributed sumcheck protocol is given in Protocol 4.

C BACKGROUND: THE GKR PROTOCOL

Notations in GKR protocol. We follow the convention in prior works of GKR protocols [46, 66, 73, 76, 77]. We denote the number of gates in the i-th layer as S_i and let $s_i = \lceil \log S_i \rceil$. (For simplicity, we assume S_i is a power of 2, and we can pad the layer with dummy gates otherwise.) We then define a function $V_i : \{0,1\}^{s_i} \to \mathbb{F}$ that takes a binary string $\mathbf{b} \in \{0,1\}^{s_i}$ and returns the output of gate $\mathbf{b}$ in layer i, where $\mathbf{b}$ is called the gate label. With this definition, V_0 corresponds to the output of the circuit, and V_d corresponds to the input layer. Finally, we define two additional functions $add_i, mult_i : \{0,1\}^{s_{i-1}+2s_i} \to \{0,1\}$, referred to as *wiring predicates* in the literature. add_i ($mult_i$) takes one gate label $\mathbf{z} \in \{0,1\}^{s_{i-1}}$ in layer $i-1$ and two gate labels $\mathbf{x},\mathbf{y} \in \{0,1\}^{s_i}$ in layer i, and outputs 1 if and only if gate $\mathbf{z}$ is an addition (multiplication) gate that takes the output of gate $\mathbf{x},\mathbf{y}$ as input. With these definitions, for any $\mathbf{z} \in \{0,1\}^{s_i}$, V_i can be written as:

$$V_i(\mathbf{z}) = \sum_{\mathbf{x},\mathbf{y} \in \{0,1\}^{s_{i+1}}} (add_{i+1}(\mathbf{z},\mathbf{x},\mathbf{y})(V_{i+1}(\mathbf{x})+V_{i+1}(\mathbf{y})) + mult_{i+1}(\mathbf{z},\mathbf{x},\mathbf{y})V_{i+1}(\mathbf{x})V_{i+1}(\mathbf{y})) \quad (3)$$

In the equation above, V_i is expressed as a summation, so $\mathcal{V}$ can use the sumcheck protocol to check that it is computed correctly. As the sumcheck protocol operates on polynomials defined on $\mathbb{F}$,

PROTOCOL 4 (**DISTRIBUTED SUMCHECK**). *Suppose the prover has N machines $\mathcal{P}_0,\cdots,\mathcal{P}_{N-1}$ and suppose $\mathcal{P}_0$ is the master node. Each P_i holds a polynomial $f^{(i)}:\mathbf{F}^{\ell-n}\to\mathbf{F}$ such that $f^{(i)}(\mathbf{x})=f(\mathbf{x}[1:\ell-n],\mathbf{i})$. Suppose $\mathcal{V}$ is the verifier. The protocol proceeds in 3 phases consisting of ℓ rounds.*

1. *In the j-th round, where $1\le j\le \ell-n$, each $\mathcal{P}_i$ sends P_0 a univariate polynomial*

$$f_j^{(i)}(x_j)\stackrel{def}{=}\sum_{\mathbf{b}\in\{0,1\}^{\ell-n-j}} f^{(i)}(\mathbf{r}[1:j-1],x_j,\mathbf{b}),$$

   After receiving all univariate polynomials from $P_1,\cdots,P_{N-1}$, P_0 computes

$$f_j(x_j)=\sum_{i=0}^{N-1} f_j^{(i)}(x_j)$$

   then sends $f_j(x_j)$ to $\mathcal{V}$. $\mathcal{V}$ checks $f_{j-1}(r_{j-1})=f_j(0)+f_j(1)$, and sends a random challenge $r_j\in\mathbb{F}$ to $\mathcal{P}_0$. $\mathcal{P}_0$ relays r_j to $P_1,\cdots,P_{N-1}$.
2. *In the j-th round, where $j=\ell-n$, after receiving r_j from $\mathcal{P}_0$, each $\mathcal{P}_i$ computes $f^{(i)}(\mathbf{r}[1:j])$ and sends $f^{(i)}(\mathbf{r}[1:j])$ to $\mathcal{P}_0$. Then $\mathcal{P}_0$ constructs a multi-linear polynomial $f':\mathbf{F}^n\to\mathbf{F}$ such that $f'(\mathbf{i})=f^{(i)}(\mathbf{r}[1:j])$ for $0\le i<N$.*
3. *In the j-th round, where $\ell-n<j\le\ell$, $\mathcal{P}_0$ and $\mathcal{V}$ run Protocol 3 on the statement:*

$$H'=\sum_{\mathbf{b}\in\{0,1\}^n} f'(\mathbf{b}),$$

   where $H'=\sum_{i=0}^{N-1} f^{(i)}(\mathbf{r}[1:\ell-n])$.

we rewrite the equation with their multi-linear extensions:

$$\begin{aligned}\tilde{V}_i(\mathbf{g})=&\sum_{\mathbf{x},\mathbf{y}\in\{0,1\}^{s_{i+1}}} h_i(\mathbf{g},\mathbf{x},\mathbf{y})\\ =&\sum_{\mathbf{x},\mathbf{y}\in\{0,1\}^{s_{i+1}}} (\tilde{add}_{i+1}(\mathbf{g},\mathbf{x},\mathbf{y})(\tilde{V}_{i+1}(\mathbf{x})+\tilde{V}_{i+1}(\mathbf{y}))\\ &+\tilde{mult}_{i+1}(\mathbf{g},\mathbf{x},\mathbf{y})\tilde{V}_{i+1}(\mathbf{x})\tilde{V}_{i+1}(\mathbf{y})),\end{aligned}\tag{4}$$

where $\mathbf{g}\in\mathbb{F}^{s_i}$ is a random vector.

The GKR Protocol. With Equation 4, the GKR protocol proceeds as following. The prover $\mathcal{P}$ first sends the claimed output of the circuit to $\mathcal{V}$. From the claimed output, $\mathcal{V}$ defines polynomial $\tilde{V}_0$ and computes $\tilde{V}_0(\mathbf{g})$ for a random $\mathbf{g}\in\mathbb{F}^{s_0}$. $\mathcal{V}$ and $\mathcal{P}$ then invoke a sumcheck protocol on Equation 4 with $i=0$. As described in Protocol 3, at the end of the sumcheck, $\mathcal{V}$ needs an oracle access to $h_i(\mathbf{g},\mathbf{u},\mathbf{v})$, where $\mathbf{u},\mathbf{v}$ are randomly selected in $\mathbb{F}^{s_{i+1}}$. To compute $h_i(\mathbf{g},\mathbf{u},\mathbf{v})$, $\mathcal{V}$ computes $\tilde{add}_{i+1}(\mathbf{g},\mathbf{u},\mathbf{v})$ and $\tilde{mult}_{i+1}(\mathbf{g},\mathbf{u},\mathbf{v})$ locally (they only depend on the wiring pattern of the circuit, not on the values), asks $\mathcal{P}$ to send $\tilde{V}_1(\mathbf{u})$ and $\tilde{V}_1(\mathbf{v})$ and computes $h_i(\mathbf{g},\mathbf{u},\mathbf{v})$ to complete the sumcheck protocol. In this way, $\mathcal{V}$ and $\mathcal{P}$ reduce a claim about the output to two claims about values in layer 1. $\mathcal{V}$ and $\mathcal{P}$ could invoke two sumcheck protocols on $\tilde{V}_1(\mathbf{u})$ and $\tilde{V}_1(\mathbf{v})$ recursively to layers above, but the number of the sumcheck protocols would increase exponentially.

Combining two claims using a random linear combination. One way to combine two claims $\tilde{V}_i(\mathbf{u})$ and $\tilde{V}_i(\mathbf{v})$ is using random linear combinations, as proposed in [43, 69]. Upon receiving the two claims $\tilde{V}_i(\mathbf{u})$ and $\tilde{V}_i(\mathbf{v})$, $\mathcal{V}$ selects $\alpha_{i,1},\alpha_{i,2}\in\mathbb{F}$ randomly and computes $\alpha_{i,1}\tilde{V}_i(\mathbf{u})+\alpha_{i,2}\tilde{V}_i(\mathbf{v})$. Based on Equation 4, this random linear combination can be written as

$$\begin{aligned}&\alpha_{i,1}\tilde{V}_i(\mathbf{u})+\alpha_{i,2}\tilde{V}_i(\mathbf{v})\\ =&\alpha_{i,1}\sum_{\mathbf{x},\mathbf{y}\in\{0,1\}^{s_{i+1}}}\tilde{add}_{i+1}(\mathbf{u},\mathbf{x},\mathbf{y})(\tilde{V}_{i+1}(\mathbf{x})+\tilde{V}_{i+1}(\mathbf{y}))\\ &+\tilde{mult}_{i+1}(\mathbf{u},\mathbf{x},\mathbf{y})\tilde{V}_{i+1}(\mathbf{x})\tilde{V}_{i+1}(\mathbf{y})\\ &+\alpha_{i,2}\sum_{\mathbf{x},\mathbf{y}\in\{0,1\}^{s_{i+1}}}\tilde{add}_{i+1}(\mathbf{v},\mathbf{x},\mathbf{y})(\tilde{V}_{i+1}(\mathbf{x})+\tilde{V}_{i+1}(\mathbf{y}))\\ &+\tilde{mult}_{i+1}(\mathbf{v},\mathbf{x},\mathbf{y})\tilde{V}_{i+1}(\mathbf{x})\tilde{V}_{i+1}(\mathbf{y})\\ =&\sum_{\mathbf{x},\mathbf{y}\in\{0,1\}^{s_{i+1}}}(\alpha_{i,1}\tilde{add}_{i+1}(\mathbf{u},\mathbf{x},\mathbf{y})+\alpha_{i,2}\tilde{add}_{i+1}(\mathbf{v},\mathbf{x},\mathbf{y}))(\tilde{V}_{i+1}(\mathbf{x})+\tilde{V}_{i+1}(\mathbf{y}))\\ &+(\alpha_{i,1}\tilde{mult}_{i+1}(\mathbf{u},\mathbf{x},\mathbf{y})+\alpha_{i,2}\tilde{mult}_{i+1}(\mathbf{v},\mathbf{x},\mathbf{y}))\tilde{V}_{i+1}(\mathbf{x})\tilde{V}_{i+1}(\mathbf{y})\end{aligned}\tag{5}$$

$\mathcal{V}$ and $\mathcal{P}$ then execute the sumcheck protocol on Equation 5 instead of Equation 4. At the end of the sumcheck protocol, $\mathcal{V}$ still receives two claims about $\tilde{V}_{i+1}$, computes their random linear combination and proceeds to the layer above recursively until the input layer.

The formal GKR protocol is presented in Protocol 5.

D THE DISTRIBUTED PC PROTOCOL

The formal protocol of distributed polynomial commitment is given in Protocol 6.

E BACKGROUND: THE VIRGO PROTOCOL

By combining the GKR protocol and the polynomial commitment in Section 4.3 We present the formal protocol of Virgo in Protocol 7 and the the complexity of Protocol 7 in the following[7].

Complexity of Virgo [76]. Given a layered arithmetic circuit C with d layers and m inputs, Protocol 7 is a zero-knowledge proof protocol as defined in Definition 2.2 for the function computed by

[7]Protocol 7 is a knowledge argument system rather than a zero-knowledge proof protocol as we actually use the knowledge argument system in our construction.

PROTOCOL 5 (**GKR**). *Let $\mathbb{F}$ be a finite field. Let $C: \mathbb{F}^m \to \mathbb{F}^k$ be a d-depth layered arithmetic circuit. $\mathcal{P}$ wants to convince that $C(\mathbf{x}) = \mathbf{1}$ where $\mathbf{x}$ is the input from $\mathcal{V}$, and $\mathbf{1}$ is the output. Without loss of generality, assume m and k are both powers of 2 and we can pad them if not.*

1. *$\mathcal{V}$ chooses a random $\mathbf{g} \in \mathbb{F}^{s_0}$ and sends it to $\mathcal{P}$.*
2. *$\mathcal{P}$ and $\mathcal{V}$ run a sumcheck protocol on*

$$1 = \sum_{\mathbf{x},\mathbf{y} \in \{0,1\}^{s_1}} (\tilde{add}_1(\mathbf{g}^{(0)},\mathbf{x},\mathbf{y})(\tilde{V}_1(\mathbf{x}) + \tilde{V}_1(\mathbf{y})) + \tilde{mult}_1(\mathbf{g}^{(0)},\mathbf{x},\mathbf{y})\tilde{V}_1(\mathbf{x})\tilde{V}_1(\mathbf{y}))$$

   At the end of the protocol, $\mathcal{V}$ receives $\tilde{V}_1(\mathbf{u}^{(1)})$ and $\tilde{V}_1(\mathbf{v}^{(1)})$. $\mathcal{V}$ computes $\tilde{mult}_1(\mathbf{g}^{(0)},\mathbf{u}^{(1)},\mathbf{v}^{(1)})$, $\tilde{add}_1(\mathbf{g}^{(0)},\mathbf{u}^{(1)},\mathbf{v}^{(1)})$ and checks that $\tilde{add}_1(\mathbf{g}^{(0)},\mathbf{u}^{(1)},\mathbf{v}^{(1)})$ $(\tilde{V}_1(\mathbf{u}^{(1)}) + \tilde{V}_1(\mathbf{v}^{(1)})) + \tilde{mult}_1(\mathbf{g}^{(0)},\mathbf{u}^{(1)},\mathbf{v}^{(1)})$ $\tilde{V}_1(\mathbf{u}^{(1)})\tilde{V}_1(\mathbf{v}^{(1)})$ equals to the last message of the sumcheck.
3. *For $i = 1,\ldots,d-1$:*
   - *$\mathcal{V}$ randomly selects $\alpha_{i,1}, \alpha_{i,2} \in \mathbf{F}$ and sends them to $\mathcal{P}$.*
   - *$\mathcal{P}$ and $\mathcal{V}$ run the sumcheck on the equation*

$$\alpha_{i,1}\tilde{V}_i(\mathbf{u}^{(i)}) + \alpha_{i,2}\tilde{V}_i(\mathbf{v}^{(i)}) = \sum_{\mathbf{x},\mathbf{y} \in \{0,1\}^{s_{i+1}}} ((\alpha_{i,1}\tilde{add}_{i+1}\mathbf{u}^{(i)},\mathbf{x},\mathbf{y}) + \alpha_{i,2}\tilde{add}_{i+1}(\mathbf{v}^{(i)},\mathbf{x},\mathbf{y}))(\tilde{V}_{i+1}(\mathbf{x}) + \tilde{V}_{i+1}(\mathbf{y}))$$
$$+ (\alpha_{i,1}\tilde{mult}_{i+1}(\mathbf{u}^{(i)},\mathbf{x},\mathbf{y}) + \alpha_{i,2}\tilde{mult}_{i+1}(\mathbf{v}^{(i)},\mathbf{x},\mathbf{y}))\tilde{V}_{i+1}(\mathbf{x})\tilde{V}_{i+1}(\mathbf{y}))$$

   - *At the end of the sumcheck protocol, $\mathcal{P}$ sends $\mathcal{V}$ $\tilde{V}_{i+1}(\mathbf{u}^{(i+1)})$ and $\tilde{V}_{i+1}(\mathbf{v}^{(i+1)})$.*
   - *$\mathcal{V}$ computes the following and checks if it equals to the last message of the sumcheck. For simplicity, let $Mult_{i+1}(\mathbf{x}) = \tilde{mult}_{i+1}(\mathbf{x},\mathbf{u}^{(i+1)},\mathbf{v}^{(i+1)})$ and $Add_{i+1}(\mathbf{x}) = \tilde{add}_{i+1}(\mathbf{x},\mathbf{u}^{(i+1)},\mathbf{v}^{(i+1)})$.*

$$(\alpha_{i,1}Mult_{i+1}(\mathbf{u}^{(i)}) + \alpha_{i,2}Mult_{i+1}(\mathbf{v}^{(i)})(\tilde{V}_{i+1}(\mathbf{u}^{(i+1)})\tilde{V}_{i+1}(\mathbf{v}^{(i+1)})) +$$
$$(\alpha_{i,1}Add_{i+1}(\mathbf{u}^{(i)}) + \alpha_{i,2}Add_{i+1}(\mathbf{v}^{(i)})(\tilde{V}_{i+1}(\mathbf{u}^{(i+1)}) + \tilde{V}_{i+1}(\mathbf{v}^{(i+1)}))$$

   If all checks in the sumcheck pass, V uses $\tilde{V}_{i+1}(\mathbf{u}^{(i+1)})$ and $\tilde{V}_{i+1}(\mathbf{v}^{(i+1)})$ to proceed to the $(i+1)$-th layer. Otherwise, $\mathcal{V}$ outputs $\mathtt{0}$ and aborts.
4. *At the input layer d, $\mathcal{V}$ has two claims $\tilde{V}_d(\mathbf{u}^{(d)})$ and $\tilde{V}_d(\mathbf{v}^{(d)})$. $\mathcal{V}$ evaluates $\tilde{V}_d$ at $\mathbf{u}^{(d)}$ and $\mathbf{v}^{(d)}$ using the input and checks that they are the same as the two claims. If yes, output $\mathtt{1}$; otherwise, output $\mathtt{0}$.*

Hardware type	Hardware name	Power consumption	Price	Quantity
CPU	AMD Ryzen Threadripper 3970X	435W	$2325.99	1
Memory	CMK256GX4M8D3600C18	96W	$1129.99	1
Motherboard	MSI TRX40 PRO WIFI	80W	$565.57	1
Power Supply	EVGA 220-T2-1000-X1	94% efficiency	$332.88	1
SSD	MZ-V8P1T0B/AM	6.2W	$129.99	1
Total		657W	$4484.42	

Table 3: Prover hardware configuration.

C. The prover time is $O(|C| + m\log m)$. The proof size is $O(d\log|C| + \lambda\log^2 m)$ and The verification time is also $O(d\log|C| + \lambda\log^2 m)$.

F THE DISTRIBUTED VIRGO PROTOCOL

By combining Protocol 6 and Protocol 4, we present the formal protocol of deVirgo in Protocol 8.

G ON-CHAIN GAS COST OPTIMIZATION

To further optimize the on-chain gas cost of block header verification and storage for a universal zkBridge, we propose the following approach, in which the prover won't bother to pay for on-chain proof verification or block header storage, and users are encouraged to submit the proof they need by our incentive design.

In our optimization, the same as the aforementioned batched proof, the prover generates one single proof for every 2^d blocks where d is a system configuration, and each proof checks and shows the validity of all signatures in the corresponding 2^d blocks. However, instead of submitting the Merkle root of the batch along with the proof on-chain immediately, provers simply post the proof to the users (e.g., through a website), and it's up to the users to retrieve and post the proof on-chain. Thus there's no more on-chain gas cost for provers through the approach.

For users who want to verify a transaction tx in a block blk, the workflow is as follows.

1. If blk has already been submitted on-chain, go to the next step. Otherwise, retrieve the proofs for the sequence of blocks

Protocol 6 (**Distributed PC**). *Suppose the prover has N machines of $\mathcal{P}_0,\cdots,\mathcal{P}_{N-1}$ and suppose $\mathcal{P}_0$ is the master node. Each P_i holds a polynomial $f^{(i)}:\mathbf{F}^{\ell-n}\to\mathbf{F}$ such that $f(\mathbf{x})=\tilde{\beta}(\mathbf{x}[\ell-n+1:\ell],\mathbf{i})f^{(i)}(x[1:\ell-n])$. Suppose $\mathcal{V}$ is the verifier. Let $\mathbb{H}$ and $\mathbb{L}$ be two disjoint multiplicative subgroups of $\mathbf{F}$ such that $|\mathbb{H}|=\frac{2^\ell}{N}$ and $|\mathbb{L}|=\rho|\mathbb{H}|$. For simplicity, We assume $\rho=1$. Let $\mathsf{pp}=\mathsf{PC.KeyGen}(1^\lambda)$. The protocol proceeds in following steps.*

1. *Each $\mathcal{P}_i$ invokes $\mathsf{PC.Commit}(f^{(i)},\mathsf{pp})$ to compute $\mathbf{f}_{\mathbb{L}}^{(i)}$ by IFFT and FFT.*
2. *Each $\mathcal{P}_i$ sends $\mathbf{f}_{\mathbb{L}}^{(i)}[1],\cdots,\mathbf{f}_{\mathbb{L}}^{(i)}[N]$ to $\mathcal{P}_0,\cdots,\mathcal{P}_{N-1}$ separately.*
3. *Each $\mathcal{P}_i$ receives $\mathbf{f}_{\mathbb{L}}^{(0)}[i{+}1],\cdots,\mathbf{f}_{\mathbb{L}}^{(N-1)}[i{+}1]$ from other machines. Assuming $\mathbf{h}_{\mathbb{L}}^{(i)}=(\mathbf{f}_{\mathbb{L}}^{(0)}[i{+}1],\cdots,\mathbf{f}_{\mathbb{L}}^{(N-1)}[i{+}1])$, $\mathcal{P}_i$ computes $\mathsf{com}_{h^{(i)}}=\mathsf{MT.Commit}(\mathbf{h}_{\mathbb{L}}^{(i)})$ and sends $\mathsf{com}_{h^{(i)}}$ to $\mathcal{P}_0$.*
4. *Suppose $\mathbf{h}=(\mathsf{com}_{h^{(0)}},\cdots,\mathsf{com}_{h^{(N-1)}})$, $\mathcal{P}_0$ computes $\mathsf{com}=\mathsf{MT.Commit}(\mathbf{h})$ and sends com to $\mathcal{V}$.*
5. *After receiving the random vector $\mathbf{r}$ from $\mathcal{V}$, $\mathcal{P}_0$ relays $\mathbf{r}$ to each $\mathcal{P}_i$. Each $\mathcal{P}_i$ computes $f^{(i)}(\mathbf{r}[1:\ell-n])$ and sends it to $\mathcal{V}$ via $\mathcal{P}_0$.*
6. *To prove the correctness of $f^{(i)}(\mathbf{r}[1:\ell-n])$, given random index of $k_1,\cdots,k_c$ from $\mathcal{V}$, $\mathcal{P}_{k_1-1},\cdots,\mathcal{P}_{k_c-1}$ send $\mathbf{h}_{\mathbb{L}}^{(k_1-1)},\cdots,\mathbf{h}_{\mathbb{L}}^{(k_c-1)}$ to $\mathcal{V}$ via $\mathcal{P}_0$.$\mathcal{P}_0$ also generates $(\mathbf{h}[k_1],\pi_{k_1})=\mathsf{MT.Open}(\mathbf{h},k_1),\cdots,(\mathbf{h}[k_c],\pi_{k_c})=\mathsf{MT.Open}(\mathbf{h},k_c)$ and send them to $\mathcal{V}$.*
7. *$\mathcal{V}$ checks $f(\mathbf{r})=\sum_{i=0}^{N-1}\tilde{\beta}(\mathbf{r}[\ell-n+1:\ell],\mathbf{i})f^{(i)}(\mathbf{r}[1:\ell-n])$. $\mathcal{V}$ checks $\mathbf{h}[k_1]{=}\mathsf{MT.Commit}(\mathbf{h}_{\mathbb{L}}^{(k_1-1)}),\cdots,\mathbf{h}[k_c]{=}\mathsf{MT.Commit}(\mathbf{h}_{\mathbb{L}}^{(k_c-1)})$. Then $\mathcal{V}$ checks $\pi_{k_1},\cdots,\pi_{k_c}$ by $\mathsf{MT.Verify}(\pi_{k_1},\mathbf{h}[k_1],\mathsf{com}),\cdots,\mathsf{MT.Verify}(\pi_{k_c},\mathbf{h}[k_c],,\mathsf{com})$. $\mathcal{V}$ also checks $q(\mathbf{f}_{\mathbb{L}}^{(i)}[k_1],\cdots,\mathbf{f}_{\mathbb{L}}^{(i)}[k_c],\mathbf{f}^{(i)}(\mathbf{r}[1:\ell-n]))=0$ for each i as shown in $\mathsf{PC.Verify}$. If all checks pass, $\mathcal{V}$ outputs 1, otherwise $\mathcal{V}$ outputs 0.*

Protocol 7 (**Virgo**). *Let $\mathbb{F}$ be a finite field. Let $C:\mathbb{F}^m\to\mathbb{F}^k$ be a d-depth layered arithmetic circuit. $\mathcal{P}$ wants to convince that $\mathbf{1}=C(\mathbf{x},\mathbf{w})$ where $\mathbf{x}$ and $\mathbf{w}$ are input and $\mathbf{1}$ is the output. Without loss of generality, assume m and k are both powers of 2 and we can pad them if not.*

1. *Set $\mathsf{pp}\leftarrow\mathsf{PC.KeyGen}(1^\lambda)$. $\mathcal{P}$ invokes $\mathsf{PC.Commit}(\tilde{V}_d,\mathsf{pp})$ to generate $\mathsf{com}_{\tilde{V}_d}$ and sends $\mathsf{com}_{\tilde{V}_d}$ to $\mathcal{V}$.*
2. *$\mathcal{P}$ and $\mathcal{V}$ run step 1.-3. in Protocol 5.*
3. *At the input layer d, $\mathcal{V}$ has two claims $\tilde{V}_d(\mathbf{u}^{(d)})$ and $\tilde{V}_d(\mathbf{v}^{(d)})$. $\mathcal{P}$ and $\mathcal{V}$ invoke $\mathsf{PC.Open}$ and $\mathsf{PC.Verify}$ on $\tilde{V}_d(\mathbf{u}^{(d)})$ and $\tilde{V}_d(\mathbf{v}^{(d)})$ with $\mathsf{com}_{\tilde{V}_d}$ and pp. If they are equal to $\tilde{V}_d(\mathbf{u}^{(d)})$ and $\tilde{V}_d(\mathbf{v}^{(d)})$ sent by $\mathcal{P}$, $\mathcal{V}$ outputs 1, otherwise $\mathcal{V}$ outputs 0.*

from the first unsubmitted one to *blk*, and then invoke the updater contract to verify all the proofs on-chain and store the information of the corresponding sequence of blocks. The process can be expensive. However, once the proofs are verified and the blocks are confirmed by the updater contract, the user becomes the owner of all these proofs on-chain, and can benefit from the proofs by charging later users who rely on these proofs to verify their transactions on-chain.

2. Thanks to the previously submitted proofs, the validity of the corresponding block is already proved at this step. And the work can never be accomplished without the efforts of proving all the blocks prior to *blk* (including *blk*). Suppose *blk* is the i^{th} block, then for each block with index in the range $[i-t+1,i]$, the user should pay a certain amount of fee to the block proof owner in compensation, where t is a system configuration and the definition of block proof owner is defined in the previous step.

In this case, provers don't bother to pay for on-chain verification any more, and the proofs are only submitted and verified on demand, which is more cost-efficient and can reduce possible waste. Moreover, through carefully-designed incentive, we can actually encourage users to submit the proofs as a possible investment, and it can also help with the popularity of our bridge.

Through the optimization, the cost performance of our bridge can be summarized as follows. If there is high demand, then each proof will be submitted immediately upon generation, and in this case each user needs to pay for at most one time of on-chain proof verification. It then degenerates into our original batched proof, but users are responsible of paying for the on-chain verification instead. If the sender chain is so unpopular that there is little bridging demand from the chain, then we successfully avoid unnecessarily submitting the proofs on-chain for meaningless but costly verification. And even if a user suddenly exists and requires bridging in this case, the request can also be fulfilled by retrieving the proofs from provers and sending them for on-chain verification one by one.

And thus we can see that, the new design can actually benefit both the provers and the users.

H PROVER MACHINE CONFIGURATION

To estimate the power consumption, we simulated a computer build. The total power consumption are based on the spec provided by the manufacturer. We present our hardware configuration in Table 3. Our prover machine doesn't need to be highly reliable since proof generation can be interrupted and restart at any time so we choose consumer grade hardware to be cost effective.

Protocol 8 (**Distributed Virgo**). *Let $\mathbb{F}$ be a finite field. Let $C: \mathbb{F}^{mN} \to \mathbb{F}^k$ be a d-depth layered arithmetic circuit. Suppose C is also a data-parallel circuit with N identical copies. $\mathcal{P}$ is a prover with N distributed machines and wants to convince $\mathcal{V}$ that $\mathbf{1} = C(\mathbf{x}, \mathbf{w})$ where $\mathbf{x}$ and $\mathbf{w}$ are input, and $\mathbf{1}$ is the output. Without loss of generality, assume m, N, and k are powers of 2 and we can pad them if not.*

1. *Set* $\mathsf{pp} \leftarrow \mathsf{PC.KeyGen}(1^\lambda)$. *Define the multi-linear extension of array* $(\mathbf{x}, \mathbf{w})$ *as* $\tilde{V}_d$. $\mathcal{P}$ *invokes step 1.-4. in Protocol 6 on* $\tilde{V}_d$ *to get* $\mathsf{com}_{\tilde{V}_d}$ *and sends* $\mathsf{com}_{\tilde{V}_d}$ *to* $\mathcal{V}$.
2. *Define the multi-linear extension of array* $\mathbf{1}$ *as* $\tilde{V}_0$. $\mathcal{V}$ *chooses a random* $g \in \mathbb{F}^{s_0}$ *and sends it to* $\mathcal{P}$.
3. $\mathcal{P}$ *and* $\mathcal{V}$ *run Protocol 4, the distributed sumcheck protocol, on*

$$1 = \sum_{\mathbf{x},\mathbf{y} \in \{0,1\}^{s_1}} (\tilde{add}_1(\mathbf{g}^{(0)}, \mathbf{x}, \mathbf{y})(\tilde{V}_1(\mathbf{x}) + \tilde{V}_1(\mathbf{y})) + \tilde{mult}_1(\mathbf{g}^{(0)}, \mathbf{x}, \mathbf{y})\tilde{V}_1(\mathbf{x})\tilde{V}_1(\mathbf{y}))$$

   At the end of the protocol, $\mathcal{V}$ *receives* $\tilde{V}_1(\mathbf{u}^{(1)})$ *and* $\tilde{V}_1(\mathbf{v}^{(1)})$. $\mathcal{V}$ *computes* $\tilde{mult}_1(\mathbf{g}^{(0)}, \mathbf{u}^{(1)}, \mathbf{v}^{(1)})$, $\tilde{add}_1(\mathbf{g}^{(0)}, \mathbf{u}^{(1)}, \mathbf{v}^{(1)})$ *and checks that* $\tilde{add}_1(\mathbf{g}^{(0)}, \mathbf{u}^{(1)}, \mathbf{v}^{(1)})\ (\tilde{V}_1(\mathbf{u}^{(1)}) + \tilde{V}_1(\mathbf{v}^{(1)})) + \tilde{mult}_1(\mathbf{g}^{(0)}, \mathbf{u}^{(1)}, \mathbf{v}^{(1)})\ \tilde{V}_1(\mathbf{u}^{(1)})\tilde{V}_1(\mathbf{v}^{(1)})$ *equals to the last message of the sumcheck.*
4. *For* $i = 1, \ldots, d-1$*:*
   - $\mathcal{V}$ *randomly selects* $\alpha_{i,1}, \alpha_{i,2} \in \mathbf{F}$ *and sends them to* $\mathcal{P}$.
   - $\mathcal{P}$ *and* $\mathcal{V}$ *run Protocol 4, the distributed sumcheck protocol, on*

$$\begin{aligned}\alpha_{i,1}\tilde{V}_i(\mathbf{u}^{(i)}) + \alpha_{i,2}\tilde{V}_i(\mathbf{v}^{(i)}) = \sum_{\mathbf{x},\mathbf{y} \in \{0,1\}^{s_{i+1}}} &((\alpha_{i,1}\tilde{add}_{i+1}(\mathbf{u}^{(i)}, \mathbf{x}, \mathbf{y}) + \alpha_{i,2}\tilde{add}_{i+1}(\mathbf{v}^{(i)}, \mathbf{x}, \mathbf{y}))(\tilde{V}_{i+1}(\mathbf{x}) + \tilde{V}_{i+1}(\mathbf{y})) \\ &+ (\alpha_{i,1}\tilde{mult}_{i+1}(\mathbf{u}^{(i)}, \mathbf{x}, \mathbf{y}) + \alpha_{i,2}\tilde{mult}_{i+1}(\mathbf{v}^{(i)}, \mathbf{x}, \mathbf{y}))\tilde{V}_{i+1}(\mathbf{x})\tilde{V}_{i+1}(\mathbf{y}))\end{aligned}$$

   - *At the end of the distributed sumcheck protocol,* $\mathcal{P}$ *sends* $\mathcal{V}$ $\tilde{V}_{i+1}(\mathbf{u}^{(i+1)})$ *and* $\tilde{V}_{i+1}(\mathbf{v}^{(i+1)})$.
   - $\mathcal{V}$ *computes the following and checks if it equals to the last message of the sumcheck. For simplicity, let* $Mult_{i+1}(\mathbf{x}) = \tilde{mult}_{i+1}(\mathbf{x}, \mathbf{u}^{(i+1)}, \mathbf{v}^{(i+1)})$ *and* $Add_{i+1}(\mathbf{x}) = \tilde{add}_{i+1}(\mathbf{x}, \mathbf{u}^{(i+1)}, \mathbf{v}^{(i+1)})$.

$$\begin{aligned}&(\alpha_{i,1}Mult_{i+1}(\mathbf{u}^{(i)}) + \alpha_{i,2}Mult_{i+1}(\mathbf{v}^{(i)}))(\tilde{V}_{i+1}(\mathbf{u}^{(i+1)})\tilde{V}_{i+1}(\mathbf{v}^{(i+1)})) + \\ &(\alpha_{i,1}Add_{i+1}(\mathbf{u}^{(i)}) + \alpha_{i,2}Add_{i+1}(\mathbf{v}^{(i)}))(\tilde{V}_{i+1}(\mathbf{u}^{(i+1)}) + \tilde{V}_{i+1}(\mathbf{v}^{(i+1)}))\end{aligned}$$

     If all checks in the sumcheck pass, V *uses* $\tilde{V}_{i+1}(\mathbf{u}^{(i+1)})$ *and* $\tilde{V}_{i+1}(\mathbf{v}^{(i+1)})$ *to proceed to the* $(i+1)$*-th layer. Otherwise,* $\mathcal{V}$ *outputs* $\mathtt{0}$ *and aborts.*
5. *At the input layer* d, $\mathcal{V}$ *has two claims* $\tilde{V}_d(\mathbf{u}^{(d)})$ *and* $\tilde{V}_d(\mathbf{v}^{(d)})$. $\mathcal{P}$ *invokes step 5.-6. in Protocol 6 to open* $\tilde{V}_d(\mathbf{u}^{(d)})$ *and* $\tilde{V}_d(\mathbf{v}^{(d)})$ *while* $\mathcal{V}$ *invokes step 7. in Protocol 6 to validate* $\tilde{V}_d(\mathbf{u}^{(d)})$ *and* $\tilde{V}_d(\mathbf{v}^{(d)})$. *If they are equal to* $\tilde{V}_d(\mathbf{u}^{(d)})$ *and* $\tilde{V}_d(\mathbf{v}^{(d)})$ *sent by* $\mathcal{P}$, $\mathcal{V}$ *outputs* $\mathtt{1}$, *otherwise* $\mathcal{V}$ *outputs* $\mathtt{0}$.