zkMIPS: An Advanced Zero-Knowledge Proof Solution for MIPS Architecture

zkMIPS Team

Version 1.1
June 15, 2023

Abstract

This document introduces zkMIPS, an advanced Zero-Knowledge Protocol (ZKP) system designed for MIPS architecture. zkMIPS aims to provide a verifiable computing solution to trust the computation results generated by untrusted computers. The adoption of the MIPS architecture aligns perfectly with the vision of incorporating zkMIPS into diverse domains such as blockchains and IoT. In the blockchain realm, zkMIPS is seamlessly integrated with Optimism technology, offering a ZKP layer 2 rollup solution specifically tailored for Ethereum. By harnessing the power of ZKP and leveraging the robustness of the MIPS architecture, zkMIPS aims to fulfill its objectives effectively. In particular, Optimism has already established a MIPS platform, providing a strong foundation for the integration of zkMIPS and accelerating the development of Ethereum Layer 2 ZK rollup solutions. In non-blockchain systems including the Internet of Things (IoT), Virtual Reality (VR), wearable devices, and decentralized cloud computing, zkMIPS enables a secure communication channel by trusting devices' computation results. This document serves as a comprehensive introduction, shedding light on the integration of zkMIPS with Optimism's architecture, while offering an overview of the zkMIPS zero-knowledge approach.

Contents

1 Introduction

In recent years, the advancement of verifiable computing techniques, particularly in zero-knowledge proofs (ZKPs), has enabled developers to ensure the trustworthiness of computation results from untrusted parties. Among these achievements, zkMIPS has successfully developed a mechanism to demonstrate the integrity of any MIPS computation. Although initially focused on layer 2 zero-knowledge (ZK) rollup solutions, zkMIPS holds broad applicability, including Internet-of-Things (IoT), wearables, and more. zkMIPS facilitates quick and easy proof of the validity of the computation results performed by untrusted parties, offering a robust solution to ensure computational trust in a wide range of practical applications.

In an interactive proof system, one of the parties, called the Prover, wants to convince the other party, also known as the Verifier, that it possesses specific information [13, 14]. This can involve proving knowledge of a password, a specific solution to a problem, or the ability to execute a particular computation. Although existing methods in the area of ZKP are very promising, applying these techniques to develop a high-performance system requires further innovation and careful consideration, such as computation and storage overhead, to apply a ZKP system. zkMIPS aims to create a cryptographic proof that validates the execution of computations, with several concerns such as proof size, prover time, and verifier time. The size of the generated proof should be succinct to meet application requirements, minimizing storage and transmission overhead. Generating the proof by the Prover and verifying it by the Verifier introduce additional computational overhead, so the time it takes to generate or verify it should align with the application requirements. Balancing these factors requires ongoing innovation to ensure that zkMIPS can be applied effectively while addressing the challenges of proof size, prover time, and verifier time. More importantly, it should be easily integrated with existing applications with minimum effort.

zkMIPS is designed for the stable and well-established MIPS architecture to apply ZKP techniques resulting in offering numerous advantages. The adoption of MIPS architecture brings benefits, such as small instruction set and the simplified design of efficient ZKP circuits. Additionally, due to the stability of the MIPS architecture, integrating zkMIPS does not require significant alterations to its core design, making it compatible with systems that compile computations to MIPS. In the realm of blockchain, Optimism, a key player in Layer 2 Ethereum solutions, recognizes the importance of MIPS. The collaboration between Optimism and zkMIPS to accelerate the development of zkMIPS as a ZKP rollup solution for Ethereum, enhancing scalability and privacy. Furthermore, zkMIPS demonstrates to be a natural choice for IoT applications, as the popularity of MIPS in IoT devices allows for the seamless incorporation of verifiable computing capabilities. In general, zkMIPS takes advantage of the strengths of the MIPS architecture, enabling the widespread implementation of ZKP techniques in domains ranging from blockchain solutions like Optimism to IoT, Virtual Reality (VR), wearable devices, and more applications.

This document serves as an introduction to zkMIPS technology, covering various aspects of its implementation. Section 2 provides a comprehensive review of the applied MIPS architecture within zkMIPS. Section 3 delves into the software system architecture necessary for seamless integration, while also highlighting the integration of Optimism technology for Layer 2 (L2) rollup purposes. The document proceeds to explore ZK protocols in Section 4, explaining how they provide succinct proof through the conversion of computations into high-degree polynomials over a finite field, enabling efficient verifica-

tion by the Verifier. This section further elaborates on the ZKP approach employed by zkMIPS. Finally, in Section 5, the document examines various zkMIPS applications, such as zkRollup, decentralized cloud computing, and their suitability for IoT devices. Section 6 concludes the document.

2 MIPS

The Microprocessor without Interlocked Pipelined Stages (MIPS) is a well-known and widely adopted class of Reduced Instruction Set Computer (RISC) architectures that were developed by MIPS Computer Systems. Over the years, MIPS has gained significant prominence and found applications in various domains, including embedded systems and video game consoles.

While MIPS has a strong presence in the industry, other architectures such as RISC-V have emerged more recently and are rapidly gaining popularity. One of the advantages of MIPS is its fixed set of instructions, which simplifies the design of stable zero-knowledge products for the market. In contrast, RISC-V offers a modular instruction set that allows the incorporation of new custom instructions. This flexibility can be advantageous in certain scenarios, but it also means that RISC-based zero-knowledge products may require more modifications in the future to adapt to the evolving instruction set.

In the realm of Ethereum Layer 2 solutions, Optimism is a key player, and it employs a subset of the MIPS instruction set in its architecture. This makes MIPS an ideal fit for both Optimism and the zkMIPS solutions built upon it.

It is important to note that there are multiple versions of MIPS, and zkMIPS specifically utilizes MIPS32 [20], which refers to the 32-bit implementations of the MIPS R3000 architecture [17].

2.1 MIPS Processor Architecture

Any computation starts with a well-defined initial state and is then modified through a specific sequence of instructions. In the case of the MIPS processor, as emulated by Cannon [11], the associated state comprises various elements depicted in Figure 1.

By maintaining and modifying this state through a specific sequence of instructions, MIPS processors perform computations and execute programs. The emulator applied in zkMIPS emulates a MIPS processor and operates on states to simulate the execution of MIPS instructions and computations. The MIPS architecture is depicted in Fig. 1. Also, the CPU registers are as follows.

- R0, R1, ..., R31: The MIPS processor architecture includes a set of 32 general-purpose registers. These registers are used to store data during computations and operations such as arithmetic, data storage, and control flow. Each register is 32 bits wide.
- HI and LO: They are special-purpose registers to hold the results of operations, particularly useful when performing computations that involve larger data sizes or require extended precision. The HI and LO registers are used to store the results of integer multiply, divide, and multiply-accumulate operations.
- PC (program counter): It is a special-purpose register that keeps track of the memory address of the next instruction to be fetched and executed.
- A memory module consisting of a set of 4KB pages, each page consisting of 2^{12} words. The MIPS architecture incorporates a memory module that provides the necessary

storage for program instructions and provides storing and loading instructions to be used during the execution of MIPS-based computations and programs.

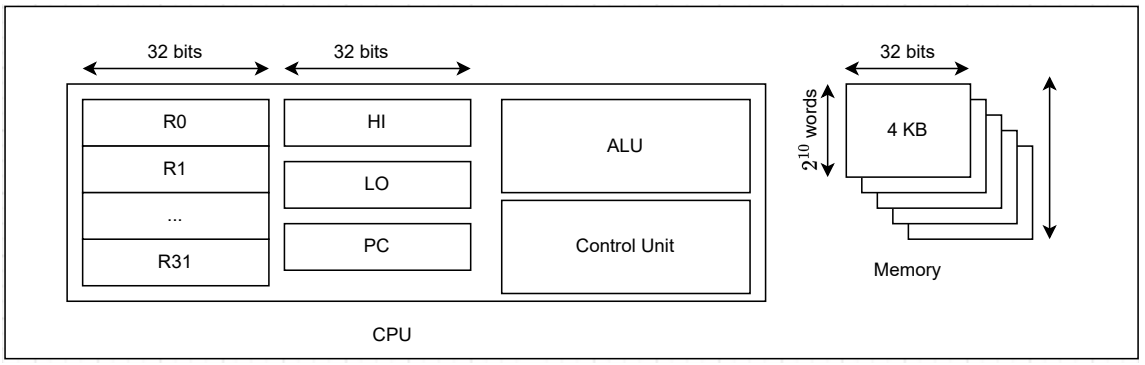

Figure 1: MIPS processor architecture

2.2 MIPS Instruction set

In the initial version of zkMIPS, we leverage the existing implementation of Cannon, which provides support for a subset of the official MIPS Instruction Set Architecture. This subset consists of 69 implemented instructions that are crucial for the functioning of zkMIPS. For a comprehensive list of these instructions, please refer to Appendix A.

3 Software System Architecture

ZkMIPS is applying the existing Optimism Bedrock architecture [2], originally developed by the Optimism team. The main feature of zkMIPS, compared to Bedrock, is the support for ZK rollups instead of Optimistic rollups, i.e. we chose to favor validity proofs instead of fault proofs.

In zkMIPS, Optimism architecture is extended with a single stateless executable - zkMIPS. It connects to L1 node on Ethereum and L2 node (op-geth) to get its inputs and submits ZK proofs to the ZK Verifier smart contract on L1. The Verifier is able to verify the generated proof and to make a state transition if it is valid.

In Bedrock, Cannon is a MIPS VM that is used to verify a Merkle state root implied by a new L2 block and generate the execution trace (explained in Section 4) for this validation process. This trace can later be used to initiate the challenge game, intrinsic to Optimistic rollups, if the generated Merkle state root does not match the one previously submitted. Similarly to this, in zkMIPS we develop a MIPS VM which has zero-knowledge proof generation mechanisms. The main components of the system as shown in Fig. 2.

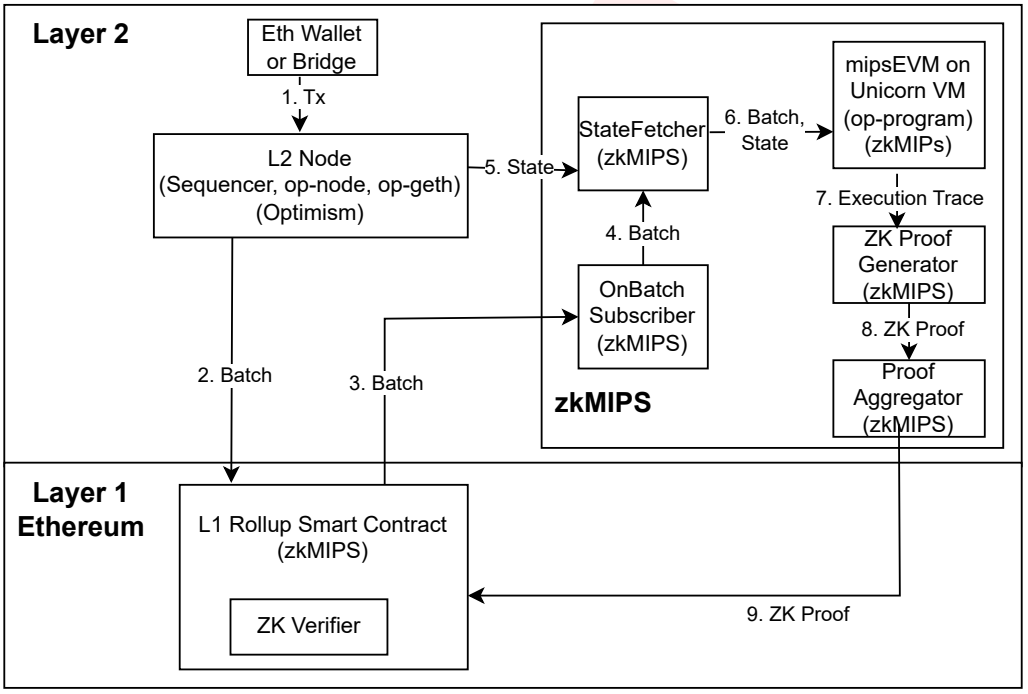

Figure 2: zkMIPS software architecture

3.1 Software System Components

In this section, the main components of the system will be explained.

- **L2 Sequencer Nodes** receive new L2 transactions (e.g., from an L2 wallet app or a bridge L1 smart contract) and use them to compose a new L2 block, which are submitted to L1 rollup smart contract as a transaction batch.

- **L1 Rollup Smart Contract** manages the unconfirmed batches produced by L2 Sequencers and stores the latest confirmed root of the L2 state.
- **L2 Validator Nodes** is a full L2 Ethereum node (op-node and op-geth from Optimism). They validate input L2 blocks, and keep track of the L2 state.
- **zkMIPS** is a stateless executable that validates the input batch and submits the ZK Proof to the L1 rollup smart contract.
- **OnBatch subscriber** is a software component of zkMIPS that is connected to L1 and subscribed to new batches.
- **State Fetcher** is a software component of zkMIPS that reads the L2 state necessary for input batch validation.
- **mipsEVM** is a MIPS VM that runs the zkMIPS program and generates an execution trace. The execution trace will be explained in the next section.
- **zkMIPS-Program** is a program that is compiled to MIPS and is executed by the zkMIPS VM (mipsEVM). It runs through the rollup state-transition to verify an L2 output from L1 inputs.
- **ZK Proof Generator** is a component of zkMIPS that generates ZK proof based on its inputs.
- **Proof Aggregator** generates a single ZK proof for the validity of the unconfirmed L2 block using ZK proofs for all pieces of this L2 block. Then, sends this aggregated ZK proof to zkMIPS Verifier smart contract on L1.

3.2 Architecture Workflow

Figure 2 illustrates zkMIPS workflow. The parts which are coming from Optimism technology stack are labeled with '(Optimisn)'. Also, the parts which are developed for zkMIPS are labeled with '(zkMIPS)'. The arrows show the general direction of the data flow.

1. Wallet software or bridge smart contracts initiate new transactions and forward them to the L2 Sequencer.
2. The L2 Sequencer collects a batch of transactions and applies them to a local state, thereby creating a new L2 block. Once the block is validated and applied, a batch is formed, which includes the list of transactions and the latest state root of the L1 rollup.
3. The zkMIPS component, known as the OnBatch Subscriber, reads the new batch from the L1 rollup smart contract.
4. OnBatch Subscriber passes the batch to the State Fetcher component.
5. The State Fetcher also retrieves the required L2 state from the L2 Node to validate the input batch.
6. The retrieved state and the input batch are then passed to mipsEVM by the State-Fetcher component.

7. The mipsEVM carries out the execution of the zkMIPS-Program and meticulously verifies the input batch. As a result of this execution, an execution trace is generated, capturing the detailed sequence of operations. This execution trace serves as input for the subsequent step in the process, as it is handed over to the ZK Proof Generator for further processing and generation of a ZK proof.

8. The ZK Proof Generator generates a ZKP proof based on the program's execution trace.

9. The Proof Aggregator submits the ZK proof to the ZK Verifier smart contract on L1. The Proof is then verified, and if it is valid, the L1 rollup updates its current Ethereum state root to the value from the new batch.

4 zkMIPS Protocol - Proof Generation

Every ZK system comprises two main components: proof generation and proof verification. In this section, we explain the proof generation process. However, before delving into that, we provide an introduction to interactive and non-interactive ZKP approaches, followed by a review of our proof generation steps.

4.1 Introduction

We first review some fundamentals before introducing the proof generation process.

4.1.1 Interactive vs. non-interactive

Fiat and Shamir observed that the Prover can generate random bits by applying a hash function to the messages sent so far, simplifying the process [12]. For example, after sending the first message m_1, the Prover defines $c_1 = h(\text{public parameters}, m_1)$ as the Verifier's first challenge. For the second message, the Prover computes $m_2, c_2 = h(c_1, m_2)$; for the third, $m_3, c_3 = h(c_2, m_3)$, etc. This creates a dependency on earlier messages due to the hash of a hash. In a non-interactive protocol, the Prover independently produces the protocol transcript public parameters, $m_1, c_1, m_2, c_2, \ldots, m_n$ without requiring an actual Verifier. An external party can verify the correctness of the proof by replicating all of the Prover computations and confirming the accuracy of the final message m_n. Non-interactive methods offer several benefits over interactive methods. First, they eliminate the need for ongoing communication between the Verifier and Prover during the proof process. This simplifies the implementation and reduces communication overhead, making verification more efficient. Second, they enable the Prover to generate the entire proof transcript independently, without relying on the presence or availability of a Verifier. This self-contained nature allows for easier distribution and verification of proofs by third parties or independent verifiers [3, 22]. zkMIPS follows a non-interactive approach in its ZKP design.

4.1.2 Execution Trace

The overall state of an automaton is defined by the values of a finite list of variables. In this setup, a valid computation can be defined as a sequence of states from some well-defined initial state to a final state, in which each state transition represents a valid computation step. We can represent this valid sequence of states as a table whose columns represent the list of variables defining the automaton's overall state and whose rows represent each step of the computation. This table is known as the execution trace.

We call the minimal set of automaton variables the CPU variables. Although it suffices to evaluate the execution of a program, we may choose to include in the trace an additional set of variables intermediary to the execution of each opcode from the program. We call this additional set the program variables and, in practice, it helps to evaluate the correct execution of programs by modeling some opcodes as sequences of more ZKP-friendly operations.

Table 1 describes the list of columns/variables from each row/step of the execution trace. From this table, we can see that the execution trace contains the variables that are

the CPU registers. Thus, it contains all the information necessary to independently verify each computation step.

The set of CPU registers from the execution trace is inspired by TinyRAM [7] as this architecture describes a minimal working CPU that is simple enough to be easily converted to STARK instances. The set of variables shown in Table 1 help to evaluate the correct execution of complex opcodes during ZKP steps.

	Variable	Size	Type
1	PC	32 bits	CPU
2	HI	32 bits	
3	LO	32 bits	
4	Register[0]	32 bits	
5	Register[1]	32 bits	
6	Register[2]	32 bits	
⋮	⋮	⋮	
35	Register[31]	32 bits	

Table 1: List of Variables

4.2 Proof Generation Steps

We are now going to review our method. The first part applies the techniques described in [5, 6] by Eli Ben-Sasson et al. and [21] by StarkWare. We then continue presenting our method in Section 4.3 to demonstrate our approach to generating a small-size proof. To start let's review the basic terminologies.

4.2.1 Basic Terminologies

Once the execution trace for an instance of the computation is defined, it can be converted to an input witness and follow the ZK steps to generate a proof.

1. Let $\mathbb{F}$ be a finite field of large prime order p. $\mathbb{F}$ is the main field of the computation and is constructed by the p-order unity root, ω_p.
2. Let $\mathbb{F}^*$ be the corresponding multiplicative group of order $p-1$. Also, let k be the largest positive integer such that $2^k | p-1$.
3. Let G be the *trace evaluation domain* that is a proper cyclic subgroup of $\mathbb{F}^*$ with order N satisfying $N = 2^n$ where $n < k$. Note that G is constructed by the N-order unity root, $g = \omega_N$. Therefore, $G = \{g^i | i \in \{0, 1, .., N-1\}\}$.
4. Let $T \in \mathbb{F}^{N \times R}$ be the execution trace table with N rows and R columns such that T_{ij} is the jth column of the ith row.
5. Let $P_i(x) \in \mathbb{F}[x]$ be a polynomial constructed by column i of the trace table T such that $P_i(g^{j-1}) = T_{ij}$.
6. Let H be the *evaluation domain* that is a non-trivial coset of a cyclic subgroup of $\mathbb{F}^*$ of order M where $M = 2^m$ and $M = \beta N$. Also, β is the blowup factor. Consider generator h, $H = \{h^i | i \in \{0, 1, .., M-1\}\}$.

7. Let $\mathbb{K}$ be a field extension of $\mathbb{F}$, $|\mathbb{K}| = p^e$ where $e \geq 2$.

4.2.2 Algebraic Intermediate Representation (AIR): Transforming the Computation into a Polynomial

The first phase of the protocol models the instructions of the input MIPS program as a set of polynomial constraints. These constraints serve as input for subsequent phases, ensuring the correctness of the protocol by accurately capturing the program logic. This phase algebraically compiles the program into a preliminary representation that needs to be proven. In summary, the Algebraic Intermediate Representation (AIR) [5] is a polynomial representation of a computation specifically designed to facilitate efficient verification of large-scale computations. We briefly summarize the applied steps as follows.

1. The Prover executes the computation to generate an execution trace table that represents the input and output data of the computation being proven. Note that the execution trace columns are padded with $N - U$ zeros where U is the line of the codes of the computation.

2. The Prover calculates separate trace polynomials $P_i : G \to \mathbb{F}$, $i \in \{1, .., R\}$ for each column in the execution trace table using the Inverse Fast Fourier Transform (IFFT). Graphically, this means that we model the execution trace table as Table 2.

step	Variable[1]	Variable[2]	...	Variable[R]
1	$P_1(g^0)$	$P_2(g^0)$	...	$P_R(g^0)$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
U	$P_1(g^{U-1})$	$P_2(g^{U-1})$	...	$P_R(g^{U-1})$

Table 2: Execution trace table as AIR polynomials

Note: This encoding proceeds using elements from G, which means the j-th trace column $\vec{x} = [x_1, \ldots, x_N]$ is encoded by P_j such that $P_j(g^{i-1}) = x_i$. In addition, $(P_1(g^{i-1}), \ldots, P_R(g^{i-1}))$ and $(P_1(g^i), \ldots, P_R(g^i))$ represent a valid state transition from state i to the next one if and only if Eq. 1 holds.

$$C_i(P_1(g^{i-1}), \ldots, P_R(g^{i-1}), P_1(g^i), \ldots, P_R(g^i)) = 0, \ \forall i, \ 0 < i < U \tag{1}$$

3. The Prover generates $P_i : H \to \mathbb{F}$ using the Fast Fourier Transform (FFT) [10]. This improves the soundness of the protocol. The new polynomial is a Low Degree Extension (LDE) of the first one. The coset H is the evaluation domain, while the ratio $\beta = \frac{M}{N} = 2^{m-n}$ is the blow up-factor that is the ratio of the size of the evaluation domain H over the size of the trace evaluation domain G.

4. The Prover generates a commitment for the polynomials P_i, $i \in \{1, .., R\}$ using the Merkle tree root of $P_1, P_2, ..., P_R$ values on H, and sends it to the Verifier.

The summerize the described AIR steps as shown in Protocol 1.

Protocol 1 Algebraic Intermediate Representation (AIR)

1: The Prover creates the execution trace.
2: The Prover generates polynomials P_i on G.
3: The Prover evaluates P_i on the evaluation domain H using FFT.
4: The Prover generates Merkle tree of P_i values on H and considers its root as $commit_{AIR}$.
5: The Prover sends $commit_{AIR}$ to the Verifier.

The explained steps allow the verification of subsequent computational states, we will follow an extended AIR approach that improves the described method by verifying dependency between non-subsequent states. Therefore, it reduces the number of necessary constraints.

4.2.3 DEEP-Algebraic Linking Interactive Oracle Proof (DEEP-ALI): Generating constraints for the polynomial

Once AIR constraints have been generated, the protocol proceeds to a phase where a few additional constraints are considered, namely **boundary**, **memory** and **general constraints** that restrict the behavior of the polynomial at the edges of the computation domain, check the integrity of memory operations, and ensure generic constraints on all types of MIPS opcodes, respectively (see Appendix B for a formal definition).

In Algebraic Linking IOP (ALI), new constraints are defined algebraically and model the last properties that a correct execution of the target MIPS program must fulfill. Subsequently, all constraints are compiled into a polynomial comprising all of them by applying the random coefficients sent by the Verifier. Therefore, the Prover and the Verifier can finally engage in an Interactive Oracle Proof (IOP) to evaluate its validity.

We also apply the Domain Extending for Eliminating Pretenders (DEEP) method to prevent cheating in the opening phase of the ALI method [8]. This method can be used to evaluate w.h.p. whether a committed polynomial function $H \to \mathbb{F}$ indeed corresponds to a polynomial. The method increases soundness of the system as it allows to verify in a single round that the rational function is not only close to a polynomial of degree at most d, but is indeed a polynomial of degree at most d. We can see this technique as an LDE being applied to the opening phase after the commitment phase. This domain extension only aims to catch cheating provers in the opening phase, since extending the commitment phase as well would require committing to the whole $\mathbb{F}^* \setminus G$.

The idea behind this method is to sample points *outside the box*, where the box refers to $G \cup H$ and the space we sample is $\mathbb{F}^* \setminus (G \cup H)$ (remember that the polynomials input to ALI is LDEs from G to H, so we consider a box containing both). More specifically, we let the Prover commit to some polynomial function $f : H \to \mathbb{F}$ and check this commitment through a polynomial difference in $f : \mathbb{F}^* \setminus G \to \mathbb{F}$, i.e. a difference between values from H and $\mathbb{F}^* \setminus (G \cup H)$.

We apply DEEP-ALI method by Ben-Sasson et al. [5,6,8] which is summarized below.

Protocol 2 Domain Extending for Eliminating Pretenders in Algebraic Linking IOP (DEEP-ALI)

1: The Verifier chooses $\alpha \in_R \mathbb{F}^*$ and sends it to the Prover.
2: The Prover generates $f_{validity} : G \to \mathbb{F}$ as follows

$$f_{validity}(x) = \sum_{i=1}^{|C_{set}|} \alpha^i \frac{C_i(P_1(x), ..., P_R(x), P_1(gx), ..., P_R(gx))}{x - g^{i-1}} \tag{2}$$

where C_{set} is a set of constraints.
3: The Prover evaluates $f_{validity}$ on the evaluation domain H using FFT.
4: The Prover generates Merkle tree of $f_{validity}$ values on H and considers its root as $commit_{ALI}$.
5: The Prover sends the commitment $commit_{ALI}$ to the Verifier.
6: The Verifier chooses $z \in_R \mathbb{F}^* \setminus (G \cup H)$ and queries $f_{validity}(z), P_1(z), \ldots, P_R(z), P_1(gz), \ldots, P_R(gz)$
7: The Verifier chooses $t \in_R H$ and queries $f_{validity}(t), P_1(t), \ldots, P_R(t), P_1(gt), \ldots, P_R(gt)$
8: **if** Eq. 2 does not hold for $f_{validity}(z)$ and $f_{validity}(t)$ **then**
9: The Verifier rejects
10: **else**
11: The Verifier accepts

4.2.4 Fast Reed-Solomon IoP of proximity (FRI)

We continue the system by applying the Fast Reed-Solomon IOP of Proximity (FRI) technique. In FRI, the Prover aims to prove the closeness of $f_{validity}$ to low-degree polynomials. In fact, it aims to prove that it is close to a polynomial of low degree instead of verifying that $f_{validity}$ *is* of low degree [4]. For this section, we follow the method described in [19] by Masip-Ardevol et al. and [16] by Haböck. We first review the general approach and then explain it in Protocol 3.

The idea is that the Prover splits a polynomial g_i into even and odd polynomials, according to the degree of each of its factors. Namely, g_i is split into $g_{i,1}, g_{i,2} : H_{i+1} \to \mathbb{K}$ where $H_{i+1} = \{x^2 | x \in H_i\}$ and $\deg(g_{i,1}), \deg(g_{i,2}) \leq \frac{1}{2} \cdot \deg(g_i)$. Algebraically, g_i can be represented as a combination of $g_{i,1}$ and $g_{i,2}$ as in Eq. 3. Then, the Prover substitutes this factor x by some value $v_i \in \mathbb{K}^*$ randomly chosen by the Verifier, denotes the resulting polynomial by g_{i+1}, as in Eq. 4, and commits to it.

$$g_i(x) = g_{i,1}(x^2) + x g_{i,2}(x^2) \tag{3}$$

$$g_{i+1}(x) = g_{i,1}(x) + v_i g_{i,2}(x) \tag{4}$$

Note that the above procedure will be performed repeatedly for a number of $l = log(d)$ times where $d = deg(f_{validity}(x))$. Furthermore, the PCS we use to commit to each g_i is a Merkle tree with M leaves. When the Verifier queries $g_i(s)$, $s \in H_i$, the Prover can simply respond with the Merkle path corresponding to the leaf of $g_i(s)$. In summary, FRI is shown in Protocol 3.

Protocol 3 DEEP version - Fast Reed-Solomon IOP of Proximity (FRI) for $f_{validity}$

1: The Prover sets $g_0(x) = \frac{f_{validity}(x) - f_{validity}(z)}{x-z}$.
2: The prover creates commit $commit^f_{g_0}$ using $g_0(x)$ values on H and send it to the Verifier.
3: Let $d = deg(g_0(x))$
4: **for all** $0 \leq i < log(d)$ **do**
5: The Prover splits g_i into $g_{i,1}, g_{i,2} : H_{i+1} \to \mathbb{K}$ based on Eq. 3.
6: The Verifier chooses $v_i \in_R \mathbb{K}^*$ and sends it to the Prover.
7: The Prover generates g_{i+1} based on Eq. 4
8: The Prover generates a Merkle tree of g_{i+1} values on H_{i+1} and considers its root as $commit^f_{g_{i+1}}$.
9: The Prover sends the commitment $commit^f_{g_{i+1}}$ to the Verifier.
10: The Verifier chooses $s \in_R H_i$ and sends it to the Prover.
11: The Prover generates $\pi_1 = \big(g_i(s), g_i(-s), g_{i+1}(s^2)\big)$
12: The Prover generates π_2 as the Merkle tree path corresponding to the leaf that contains $g_{i+1}(s^2)$.
13: The Verifier generates $g^* = \frac{g_i(s)+g_i(-s)}{2} + v_i \frac{g_i(s)-g_i(-s)}{2s}$.
14: **if** $(g^* \neq g_{i+1}(s^2))$ or (π_2 does not match $commit^f_{g_{i+1}}$) **then**
15: The Verifier rejects and terminates the protocol.
16: The Verifier accepts

The system continues running FRI with a different setup as follows.

Protocol 4 DEEP version - Fast Reed-Solomon IOP of Proximity (FRI) for trace poly nomials

1: The Prover sets $g_0(x) = \sum_{i=1}^{R} \alpha^{i-1} \cdot \frac{P_i(x) - P_i(z)}{(x-z)(x-gz)}$.
2: The prover creates commit $commit^P_{g_0}$ using $g_0(x)$ values on H and send it to the Verifier.
3: Let $d = deg(g_0(x))$
4: **for all** $0 \leq i < log(d)$ **do**
5: The Prover splits g_i into $g_{i,1}, g_{i,2} : H_{i+1} \to \mathbb{K}$ based on Eq. 3.
6: The Verifier chooses $v_i \in_R \mathbb{K}^*$ and sends it to the Prover.
7: The Prover generates g_{i+1} based on Eq. 4
8: The Prover generates a Merkle tree of g_{i+1} values on H_{i+1} and considers its root as $commit^P_{g_{i+1}}$.
9: The Prover sends the commitment $commit^P_{g_{i+1}}$ to the Verifier.
10: The Verifier chooses $s \in_R H_i$ and sends it to the Prover.
11: The Prover generates $\pi_1 = \big(g_i(s), g_i(-s), g_{i+1}(s^2)\big)$
12: The Prover generates π_2 as the Merkle tree path corresponding to the leaf that contains $g_{i+1}(s^2)$.
13: The Verifier generates $g^* = \frac{g_i(s)+g_i(-s)}{2} + v_i \frac{g_i(s)-g_i(-s)}{2s}$.
14: **if** $(g^* \neq g_{i+1}(s^2))$ or (π_2 does not match $commit^P_{g_{i+1}}$) **then**
15: The Verifier rejects and terminates the protocol.
16: The Verifier accepts

Finally, to prove and verify the validity of the whole MIPS program execution with soundness ϵ, the Prover and the Verifier engage in Protocol 5.

Protocol 5 Scalable and Transparent Argument of Knowledge (STARK)

1: **for all** $0 \leq i < \log_\beta(\epsilon)$ **do**
2: The Prover and the Verifier engage in Protocol 3.
3: The Prover and the Verifier engage in Protocol 4.
4: The Verifier accepts

4.3 Composition, Recursion, and Batching

4.3.1 Composition

Different types of ZK proof require different amounts of resources, such as memory, prover time, verifier time, proof size, and gas cost (for L1), among others. Proof composition allows us to combine two different types of proof, creating the best of both worlds.

As an example, consider π_b, which is proof of the correctness of the execution of a batch of transactions. When STARK is applied, a large proof is generated, resulting in longer verification times. The Prover can execute the verification process using its own generated proof, π_b, and provides a new proof demonstrating the correctness of running the verification process. This second proof is called π_v. Since the second proof is meant to show the correctness of a fixed verification program, it can be optimized to provide a shorter proof size when combined with π_b. Finally, the last proof generated by the Prover, π_v, can be verified by an actual verifier, as shown in Figure 3.

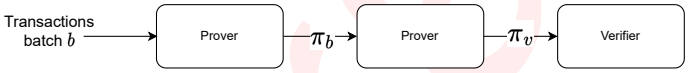

Figure 3: Composition

4.3.2 Recursion

In this stage, the first proof is generated by the first Prover using program execution. Then, the output proof is passed as input to the proof generation process, and the process is recursively repeated. Figure 4 shows the process of recursive provers.

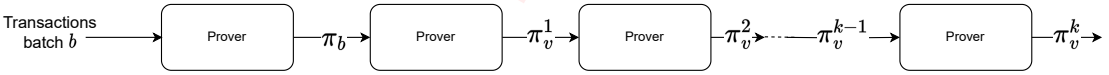

Figure 4: Recursion

4.3.3 Batching

In this stage, we use the batched verification method for the KZG polynomial commitment scheme where we use randomized techniques for batch pairing equations [15, 18]. For this section, we follow the method as described in [9] by Boneh et al.

4.3.4 Composition-Recursion-Batching

We want to use composition, recursion, and batching to prove the correct execution of a batch of transactions (or several batches). Figure 5 shows the overall process.

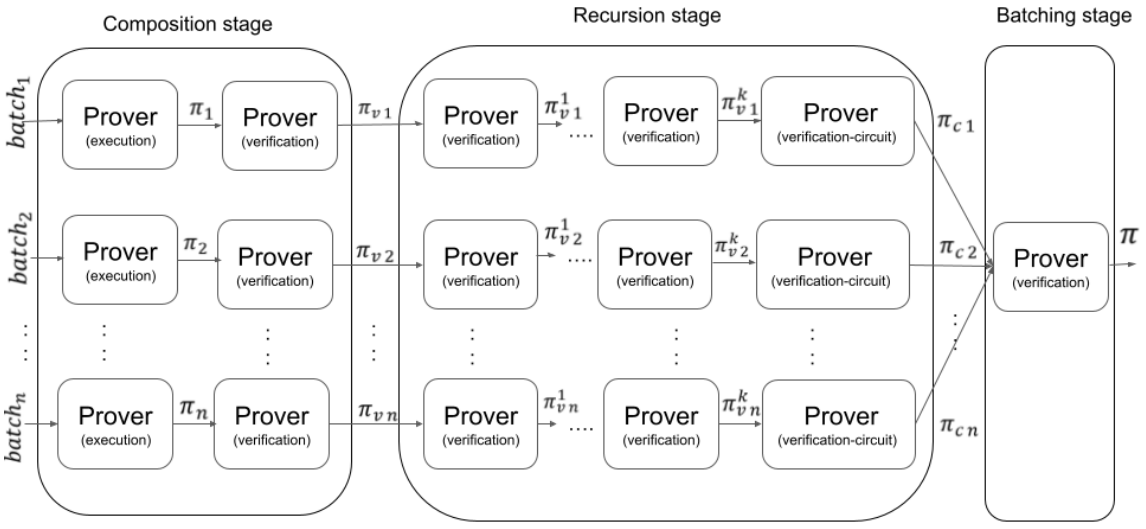

Figure 5: Proving architecture with composition, recursive, and batching

In this architecture, the first proofs for each batch, $\pi_1, \pi_2, ..., \pi_n$, are of the STARK type. In the composition stage, the system generates $\pi_{v1}, \pi_{v2}, ..., \pi_{vn}$ with a shorter size. Each proof π^i_{vj} where $i \in \{1, .., k\}$ results in proof π_{cj} after applying the recursion stage. Now, the resulted proofs $\pi_{c1}, \pi_{c2}, ..., \pi_{cn}$ can be used in batching step to make them into one proof and then, validate only the final proof π.

5 Applications of zkMIPS

5.1 Ethereum Layer 2 ZK Rollup Solutions

zkMIPS has valuable applications in Layer 2 (L2) blockchain solutions. L2 solutions aim to alleviate the scalability limitations of the underlying Layer 1 blockchain by processing transactions off-chain. Using zkMIPS, participants can cryptographically prove the validity of off-chain transactions in the Ethereum blockchain without revealing the specific details of each transaction on the public blockchain. This ensures the integrity of transactions. In addition, multiple transactions can be aggregated into a single proof, reducing the computational overhead required for verification and enhancing scalability. The provided proof guarantees the correctness of the transaction execution and hence reduces the withdrawal time of the processed fund. Fig. 6 provides an overview.

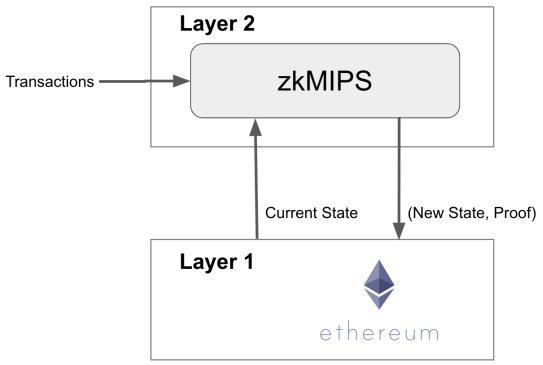

Figure 6: zkMIPS in Ethereum L2 solutions

5.2 Internet-of-Things (IoT)

zkMIPS, utilizing the MIPS instruction set, holds promises to add verifiable commuting functionalities to IoT, VR, and wearable devices that incorporate MIPS processors. In the realm of IoT, zkMIPS can offer secure and privacy-preserving communication protocols for connected devices. By leveraging ZKP, zkMIPS can enable secure data exchange, authentication, and integrity verification within IoT networks. This ensures that sensitive information remains confidential and protected against unauthorized access.

In the context of Virtual Reality (VR), zkMIPS can contribute to improved privacy and security in VR environments. zkMIPS can enable users to interact and engage in virtual experiences while maintaining the confidentiality of their personal data and activities. This includes safeguarding sensitive user information, such as biometric data or behavioral patterns, from potential threats. One such application can be explained through VR concerts that are rising in popularity. In one such example, a concert organizer may want to issue tickets in the form of non-fungible tokens (NFTs) to concert givers, but to avoid keeping data due to General Data Protection Regulation (GDPR) and other information laws, they can utilize this feature in order to register concert attendees through the public addresses of these NFTs - due to power of zero-knowledge, the organizer only needs the proof that a ticket has been paid for by a certain address, while the attendee can enjoy the concert live, without worrying about whether their data are being tracked.

Furthermore, in wearable devices that utilize MIPS architecture, zkMIPS can offer privacy-enhancing features and provide secure and private interactions with connected devices and networks. This can be particularly crucial for wearables that handle sensitive user data, such as health-related information or personal fitness metrics. Fig. 7 provides an overview.

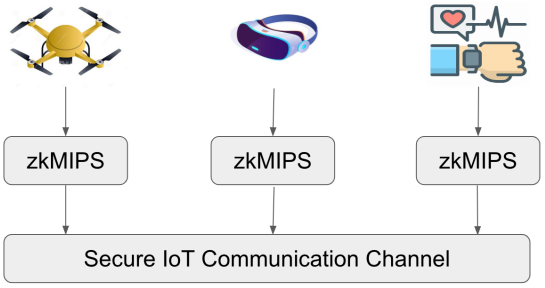

Figure 7: zkMIPS in IoT

5.3 Decentralized Cloud Computing

zkMIPS introduces an exciting opportunity for decentralized cloud computing systems. With zkMIPS, the computations can be offload to the cloud, leveraging the power of thousands of untrusted processors to execute them swiftly. Despite the untrusted nature of these processors, the computation result can still be trusted when using zkMIPS. This breakthrough paves the way for innovative applications in secure and scalable cloud computing.

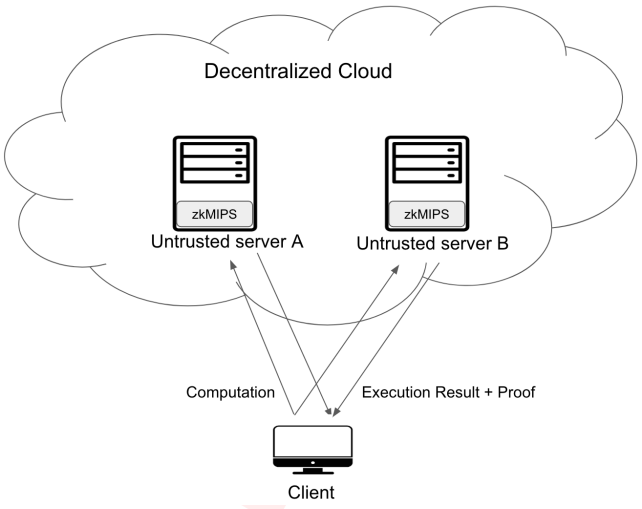

Figure 8: zkMIPS in Decentralized Cloud Computing

6 Conclusion

This document provides an overview of the steps involved in the zkMIPS zero-knowledge architecture. As the project continues to evolve, we anticipate updating this document on a monthly basis or more frequently, if necessary.

We are also aware that other teams are simultaneously working on implementing new techniques and solutions that could significantly improve the performance of zkMIPS. We closely monitor the progress of these projects to evaluate their potential integration into zkMIPS as soon as the corresponding software libraries or final techniques become accessible. If such an integration occurs, we will report on it in this document.

We would like to acknowledge the MetisDAO Foundation for its support in the zkMIP project. Their participation has been instrumental in advancing the project's goals. We appreciate the guidance and assistance provided by them throughout this journey.

A Implemented MIPS Instructions

Tables 3 to 10 describe instructions from our MIPS Instruction Set, classified according to [17]. A detailed description for each instruction can be found in the MIPS32 [20] or MIPS R3000 [17] documentation, as specified in the last column of each instruction row.

	Mnemonic	Operands	Instruction	Reference
1	ADD	rt, rs, rt	Add Word	MIPS32
2	ADDI	rt, rs, imm	Add Immediate Word	MIPS32
3	ADDIU	rt, rs, imm	Add Immediate Unsigned Word	MIPS32
4	ADDU	rd, rs, rt	Add Unsigned Word	MIPS32
5	CLZ	rd, rs	Count Leading Zeros in Word	MIPS32
6	DIV	rs, rt	Divide Word	MIPS32
7	DIVU	rs, rt	Divide Unsigned Word	MIPS32
8	MUL	rd, rs, rt	Multiply Word to GPR	MIPS32
9	MULT	rs, rt	Multiply Word	MIPS32
10	MULTU	rs, rt	Multiply Unsigned Word	MIPS32
11	NEGU	rd, rs	Negate Unsigned Integer	R3000
12	SLT	rd, rs, rt	Set on Less Than	MIPS32
13	SLTI	rt, rs, imm	Set on Less Than Immediate	MIPS32
14	SLTIU	rt, rs, imm	Set on Less Than Immediate Unsigned	MIPS32
15	SLTU	rd, rs, rt	Set on Less Than Unsigned	MIPS32
16	SUBU	rd, rs, rt	Subtract Unsigned Word	MIPS32

Table 3: List of CPU Arithmetic Instructions

	Mnemonic	Operands	Instruction	Reference
17	B	offset	Unconditional Branch	MIPS32
18	BEQ	rs, rt, offset	Branch on Equal	MIPS32
19	BEQZ	rs, offset	Branch on Equal to Zero	R3000
20	BGEZ	rs, offset	Branch on Greater Than or Equal to Zero	MIPS32
21	BGTZ	rs, offset	Branch on Greater Than Zero	MIPS32
22	BLEZ	rs, offset	Branch on Less Than or Equal to Zero	MIPS32
23	BLTZ	rs, offset	Branch on Less Than Zero	MIPS32
24	BNE	rs, rt, offset	Branch on Not Equal	MIPS32
25	BNEZ	rs, offset	Branch on Not Equal to Zero	R3000
26	J	target	Jump	MIPS32
27	JAL	target	Jump and Link	MIPS32
28	JALR	rd, rs	Jump and Link Register	MIPS32
29	JR	rs	Jump Register	MIPS32

Table 4: List of CPU Branch and Jump Instructions

	Mnemonic	Operands	Instruction	Reference
30	NOP		No Operation	MIPS32

Table 5: List of CPU Instruction Control Instructions

	Mnemonic	Operands	Instruction	Reference
31	LB	rt, offset	Load Byte	MIPS32
32	LBU	rt, offset	Load Byte Unsigned	MIPS32
33	LH	rt, offset	Load Halfword	MIPS32
34	LHU	rt, offset	Load Halfword Unsigned	MIPS32
35	LI	rt, imm	Load Immediate	MIPS32
36	LL	rt, offset	Load Linked Word	MIPS32
37	LW	rt, offset	Load Word	MIPS32
38	LWR	rt, offset	Load Word Right	MIPS32
39	LWL	rt, offset	Load Word Left	MIPS32
40	SB	rt, offset	Store Byte	MIPS32
41	SC	rt, offset	Store Conditional Word	MIPS32
42	SH	rt, offset	Store Haldword	MIPS32
43	SW	rt, offset	Store Word	MIPS32
44	SWL	rt, offset	Store Word Left	MIPS32
45	SWR	rt, offset	Store Word Right	MIPS32
46	SYNC	(stype = 0)	Synchronize Shared Memory	MIPS32

Table 6: List of CPU Memory Instructions

	Mnemonic	Operands	Instruction	Reference
47	AND	rd, rs, rt	And	MIPS32
48	ANDI	rt, rs, imm	And Immediate	MIPS32
49	LUI	rt, imm	Load Upper Immediate	MIPS32
50	NOT	rd, rs	Not	R3000
51	OR	rd, rs, rt	Or	MIPS32
52	ORI	rt, rs, imm	Or Immediate	MIPS32
53	XOR	rd, rs, rt	Exclusive Or	MIPS32
54	XORI	rt, rs, imm	Exclusive Or Immediate	MIPS32

Table 7: List of CPU Logical Instructions

	Mnemonic	Operands	Instruction	Reference
55	MFHI	rd	Move From HI Register	MIPS32
56	MFLO	rd	Move From LO Register	MIPS32
57	MOVE	rd, rs	Move	R3000
58	MOVN	rd, rs, rt	Move Conditional on Not Zero	MIPS32
59	MOVZ	rd, rs, rt	Move Conditional on Zero	MIPS32
60	MTLO	rs	Move To LO Register	MIPS32
61	MTHI	rs	Move To HI Register	MIPS32

Table 8: List of CPU Move Instructions

	Mnemonic	Operands	Instruction	Reference
62	SLL	rd, rt, sa	Shift Word Left Logical	MIPS32
63	SLLV	rd, rt, rs	Shift Word Left Logical Variable	MIPS32
64	SRA	rd, rt, sa	Shift Word Right Arithmetic	MIPS32
65	SRAV	rd, rt, rs	Shift Word Right Arithmetic Variable	MIPS32
66	SRL	rd, rt, sa	Shift Word Right Logical	MIPS32
67	SRLV	rd, rt, rs	Shift Word Right Logical Variable	MIPS32

Table 9: List of CPU Shift Instructions

	Mnemonic	Operands	Instruction	Reference
68	TEQ	rs, rt	Trap If Equal	MIPS32
69	SYSCALL		System Call	MIPS32

Table 10: List of CPU Trap Instructions

B Constraints for MIPS Instructions

To ensure the correct execution of MIPS program and make the execution trace to ALI constraints transition smoother, zkMIPS supports the following additional types of constraints:

Boundary constraints ensure registers take certain values at certain steps. Typically, these constraints verify that the correct values were input and output to the first and last steps of the execution trace, respectively, as in Table 11. The Boundary constraints can be annotated as tuples (i, j, α), meaning that at the i-th step the j-th register should take the value α (Equations 5 and 6). Then, following the indexing from Table 1, they can be expressed as constraint polynomials (Equations 7 and 8) and verified as polynomial evaluations (Equations 9 and 10).

step	instruction	...	input_register[0]	input_register[1]	input_register[2]	...
1	-	...	0	1	2	...
⋮	⋮	⋱	⋮	⋮	⋮	⋱
U	-	...	1	2	4	...

Table 11: Example of boundary constraints

$$(0,0,0) \qquad (0,1,1) \qquad (0,2,2) \tag{5}$$
$$(U,0,1) \qquad (U,1,2) \qquad (U,2,4) \tag{6}$$

$$C_{i'}(X) \coloneqq X - 0 \qquad C_{i'+1}(X) \coloneqq X - 1 \qquad C_{i'+2}(X) \coloneqq X - 2 \tag{7}$$
$$C_{i''}(X) \coloneqq X - 1 \qquad C_{i''+1}(X) \coloneqq X - 2 \qquad C_{i''+2}(X) \coloneqq X - 4 \tag{8}$$

$$C_{i'}(P_{151}(g^0)) \stackrel{?}{=} C_{i'+1}(P_{152}(g^0)) \stackrel{?}{=} C_{i'+2}(P_{153}(g^0)) \stackrel{?}{=} 0 \tag{9}$$
$$C_{i''}(P_{151}(g^U)) \stackrel{?}{=} C_{i''+1}(P_{152}(g^U)) \stackrel{?}{=} C_{i''+2}(P_{153}(g^U)) \stackrel{?}{=} 0 \tag{10}$$

Memory constraints ensure values loaded from memory positions match the latest values stored there. Just like boundary constraints, memory constraints can be annotated as tuples (Table 12 and Equation 11) and implemented as polynomial constraints (Equations 12 and 13) additionally sorted by ascending memory locations and time. By analyzing memory transcripts, it is possible to verify the consistency of memory locations during execution.

step	instruction	...	input_register[0]	input_register[1]	input_register[2]	...
1	`SW r0 M`	...	0	1	2	...
2	`LW r1 M`	...	0	0	2	...
3	`SW r2 M`	...	0	0	2	...
4	`LW r1 M`	...	0	2	2	...
⋮	⋮	⋱	⋮	⋮	⋮	⋱

Table 12: Example of memory constraints

$$(1, M, 0) \qquad (3, M, 2) \tag{11}$$

$$C_{i'}(X) \coloneqq X - 0 \qquad C_{i'+1}(X) \coloneqq X - 2 \tag{12}$$

$$C_{i'}(P_{151}(g^0)) \stackrel{?}{=} C_{i'}(P_{152}(g^1)) \stackrel{?}{=} C_{i'+1}(P_{153}(g^2)) \stackrel{?}{=} C_{i'+1}(P_{152}(g^3)) \stackrel{?}{=} 0 \tag{13}$$

General constraints ensure generic constraints on all type of MIPS opcodes such as flag constraint, if-else conditions with respect to range of values that opcode accept, etc. Just like boundary and memory constraints, general constraints can be annotated as tuples (Table 13 and Equation 14) and implemented as polynomial constraints (Equations 15 and 16) .

step	instruction	...	pInverse	...	input_flag	...	output_flag	...
1	-	...	`false`	...	`true`	...	`false`	...
2	-	...	`true`	...	`false`	...	-	...

Table 13: Example of memory constraints

$$(1, \text{pInverse}, 0) \qquad (1, \text{input\_flag}, 1) \qquad (2, \text{output\_flag}, 0) \tag{14}$$

$$C_{i'}(X) \coloneqq 1 - X \qquad C_{i'+1}(X) \coloneqq X - 0 \qquad C_{i'+2}(X) \coloneqq 1 - X \tag{15}$$

$$C_{i'}(P_1(g^0)) \stackrel{?}{=} C_{i'}(P_{147}(g^1)) \stackrel{?}{=} C_{i'+1}(P_{192}(g^1)) \stackrel{?}{=} C_{i'+1}(P_{147}(g^2)) \stackrel{?}{=} 0 \tag{16}$$

References

[1] Introduction to 0x. https://0x.org/docs/introduction/introduction-to-0x.

[2] Bedrock Explainer. https://community.optimism.io/docs/developers/bedrock/explainer/.

[3] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In Dorothy E. Denning, Raymond Pyle, Ravi Ganesan, Ravi S. Sandhu, and Victoria Ashby, editors, CCS '93, Proceedings of the 1st ACM Conference on Computer and Communications Security, Fairfax, Virginia, USA, November 3-5, 1993, pages 62–73. ACM, 1993.

[4] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Fast reed-solomon interactive oracle proofs of proximity. In Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella, editors, 45th International Colloquium on Automata, Languages, and Programming, ICALP 2018, July 9-13, 2018, Prague, Czech Republic, volume 107 of LIPIcs, pages 14:1–14:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[5] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive, Paper 2018/046, 2018. https://eprint.iacr.org/2018/046.

[6] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable Zero Knowledge with No Trusted Setup. 11694:701–732, 2019.

[7] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. TinyRAM Architecture Specification. https://www.scipr-lab.org/doc/TinyRAM-spec-2.000.pdf, 2020.

[8] Eli Ben-Sasson, Lior Goldberg, Swastik Kopparty, and Shubhangi Saraf. DEEP-FRI: sampling outside the box improves soundness. IACR Cryptol. ePrint Arch., page 336, 2019.

[9] Dan Boneh, Justin Drake, Ben Fisch, and Ariel Gabizon. Efficient polynomial commitment schemes for multiple points and polynomials. Cryptology ePrint Archive, 2020.

[10] Gilles Brassard and Paul Bratley. Algorithmics - theory and practice. Prentice Hall, 1988.

[11] Cannon Repository. https://github.com/ethereum-optimism/cannon.

[12] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings, volume 263 of Lecture Notes in Computer Science, pages 186–194. Springer, 1986.

[13] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof-systems (extended abstract). In Robert Sedgewick, editor, Proceedings of the 17th Annual ACM Symposium on Theory of Computing, May 6-8, 1985, Providence, Rhode Island, USA, pages 291–304. ACM, 1985.

[14] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. SIAM J. Comput., 18(1):186–208, 1989.

[15] Jens Groth. On the size of pairing-based non-interactive arguments. IACR Cryptol. ePrint Arch., page 260, 2016.

[16] Ulrich Haböck. A summary on the fri low degree test. Cryptology ePrint Archive, 2022.

[17] IDT R30xx Family Software Reference Manual. https://student.cs.uwaterloo.ca/~cs350/common/r3000-manual.pdf, 1994.

[18] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In Masayuki Abe, editor, Advances in Cryptology - ASIACRYPT 2010 - 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings, volume 6477 of Lecture Notes in Computer Science, pages 177–194. Springer, 2010.

[19] Hector Masip-Ardevol, Marc Guzman-Albiol, Jordi Baylina-Mele, and Jose Luis Munoz-Tapia. eSTARK: Extending STARKs with Arguments. Cryptology ePrint Archive, Paper 2023/474, 2023. https://eprint.iacr.org/2023/474.

[20] MIPS Architecture for Programmers Volume II-A: The MIPS32 Instruction Set Manual. https://s3-eu-west-1.amazonaws.com/downloads-mips/documents/MD00086-2B-MIPS32BIS-AFP-6.06.pdf, 2016.

[21] StarkwareTeam. ethSTARK Documentation – Version 1.1. Cryptology ePrint Archive, Paper 2021/582, 2021. https://eprint.iacr.org/2021/582.

[22] Justin Thaler. Proofs, arguments, and zero-knowledge, 2023.