Arbigraph: Verifiable Turing-Complete Execution Delegation

Michael Mirkin[1], Hongyin Chen[1,2],
Ohad Eitan[1], Gal Granot[1], and Ittay Eyal[1]

[1]Technion – Israel Institute of Technology
[2]Peking University

Abstract

Dependence on online infrastructure is rapidly growing as services like online payments and insurance replace traditional options, while others, like social networks, offer new capabilities. The centralized service operators wield unilateral authority over user conflicts, content moderation, and access to essential services. In the context of payments, blockchains provide a decentralized alternative. They also enable decentralized execution of stateful programs called *smart contracts*. But those lack the contextual understanding and interpretative capabilities that would enable reasoning about complex scenarios. Advancements in machine learning (ML) are raising interest in *actually-smart contracts*, but blockchain computation constraints prohibit direct ML inference execution. While many projects deploy computation delegation mechanisms, they lack Turing-completeness, prohibit parallel computation, or suffer from high overhead.

We present Arbigraph, a blockchain-based execution delegation protocol. Like previous optimistic solutions, the parties submit their computation results, allowing a smart contract to arbitrate in case of dispute. But Arbigraph employs a novel dual-graph data structure and takes advantage of the nature of the dispute process to achieve Turing completeness, constant-time memory access, and parallel execution. We formalize the problem and show that Arbigraph guarantees completeness, soundness, and progress. Experiments on LLM inference as well as matrix multiplication, which is at the core of ML inference, demonstrate that parallelization speedup grows linearly with matrix dimensions. We demonstrate Arbigraph's practical cost with a deployment on the Avalanche blockchain. Arbigraph thus enables decentralized, context-aware decision-making and unlocks unprecedented use cases for blockchains.

1 Introduction

The proliferation of digital services has fundamentally changed how we interact with everyday life. Traditional methods—from payment systems to insurance claims—are increasingly being replaced by online platforms. Meanwhile, social networks have created entirely new forms of digital interaction. However, this digital transformation comes with significant concerns: these services operate under centralized control, with providers often exercising power without adequate accountability [1, 2]. While users can seek recourse through external jurisdictions, this process is both expensive and time-consuming, often failing to prevent harm before it occurs [3, 4, 5, 6]. Notable

examples include the censorship of content by social media platforms [7, 8, 9, 10], inadequate handling of copyright claims [11], and the bias of insurance companies against low-income patients [12].

A promising path to an automatic alternative, not subject to human bias, is to use *Blockchains*, which are decentralized and censorship-resistant platforms. They were originally introduced for currency transactions and evolved to enable *smart contracts*—stateful programs that enforce simple conditions [13]. However, smart contracts lack the contextual understanding and interpretative capabilities that would enable reasoning about complex, ambiguous real-world scenarios. Machine learning (ML) algorithms, particularly Large Language Models (LLMs) [14, 15, 16], exhibit exactly these capabilities. However, implementing these computationally intensive tasks within a smart contract introduces prohibitive complexity, since blockchain systems require all their operators to replicate all computations.

Previous work (§2) aims to extend blockchain computational capabilities while maintaining decentralization. Some (e.g., [17, 18]) use cryptographic ZK proofs; however, they incur overhead for each operation, inappropriate for computationally intensive ML tasks, and require precomputed lookup tables to support non-linear operations, with compounding accuracy loss. Others follow an optimistic approach, incurring overhead only in case of foul play. But existing protocols (e.g. [19, 20, 21, 22]) either restrict parallelism, impose logarithmic overhead on memory access, or lack Turing-completeness, as they prohibit dynamic control flows. Note that parallelism and efficient memory access are critical for optimistic-protocol security, as resource-rich attackers can exclude legitimate users with limited capabilities from participating in the protocol.

Our approach belongs to the second category, where the problem is that of *Optimistic Execution Delegation* (*OED*) (§3). Two parties wish to agree on the result of a program using a smart contract as an arbitrator. Both parties know the program, while the smart contract maintains a *commitment* thereof. Each party submits the result and, in the absence of dishonest behavior, the protocol terminates with the smart contract outputting the result. In case of a dispute, the smart contract acts as an arbitrator and outputs the correct result and the dishonest party, which might be penalized. A protocol solving the problem must satisfy (1) *completeness*, ensuring that if both parties follow the protocol, the smart contract outputs the correct result; (2) *soundness*, guaranteeing that if exactly one party is dishonest, they are detected and the smart contract outputs the correct result; and (3) *progress*, ensuring that the dispute resolution time is logarithmic in the program size. This simple two-party model is trivially extendable to any number of participants.

We present Arbigraph (§4), a Turing-complete optimistic execution delegation protocol that enables constant-time dynamic memory access and parallel execution. Arbigraph utilizes a novel *dual-graph* data structure: A *symbolic graph* specifies the program with its operations and control flow as graph *nodes*, and their dependencies as edges. A matching *execution graph* records the runtime trace of the program with nodes mapped from the symbolic graph and instantiated with values.[1] This dual-graph architecture captures the program's execution structure while eliminating the need to know even the program size in advance.

The smart contract maintains a commitment to the symbolic graph, which is used to verify the correctness of the execution graph during disputes. Each party independently executes the program and commits its output value to the smart contract. In the common case when no dispute occurs, Arbigraph incurs no overhead. If the outputs differ, it triggers a dispute. Because node execution follows a predefined order imposed by the symbolic graph, there must exist a first point of disagreement between parties—the *dispute node*. Arbigraph first uses the *Bisection protocol* [20, 23, 24] to identify the dispute node. This protocol either (1) directly identifies the cheater—

[1] We show in Appendix C this computational model is Turing-complete.

revealing the honest party's output as correct—or (2) has both parties commit to a *common prefix* of the execution, isolating a single disputed node. In the latter case, the parties submit proofs of correctness for that node. The smart contract validates the proofs, determines the correct result, and identifies the cheater.

Previous optimistic approaches use either a single memory-state structure, incurring logarithmic overhead for each memory write operation and prohibiting parallel execution, or a static graphical structure, prohibiting dynamic control flows. Our dynamic execution presents a new challenge: The contract must verify the disputed node's variable reads observe the most recent writes, which is not enforced by a global state or by a static graph. To address this, we observe that at the point of dispute the parties have already committed to a common prefix of the execution. Our dual-graph data structure thus allows Arbigraph to employ a *recency rule*, which favors the party referencing the more recent writes.

Arbigraph satisfies the three essential properties of optimistic execution delegation (§5). Completeness holds because if both parties submit the same output, the smart contract accepts it without dispute. To show that soundness holds, we use the bisection protocol guarantees to identify the first dispute node. The malicious party can alter the dispute node in two main ways: (1) by changing its structure—the static properties dictated by the symbolic graph—or (2) by falsifying its value—representing the dynamic computation results. To detect structural tampering, the smart contract verifies the relevant Merkle proofs linking the dispute node to the Merkle root commitment, showing that its structure is correct and that it is the next node to be executed after the common prefix. To detect value tampering, the smart contract verifies the disputed node's dynamic value based on its dependencies, which reside in the common prefix, allowing for the straightforward recalculation of the correct values. The recency rule further guarantees that all computations reference the correct state. Finally, the protocol ensures progress, as the Bisection protocol operates with logarithmic complexity relative to the computation size, and the validation of the disputed node requires a single message per party.

Unlike previous Turing-complete approaches [20, 19, 21], Arbigraph enables parallel execution: As there is no unified state structure, different threads can read and write to different nodes without conflicts. To demonstrate this advantage (§6), we first implemented a translation layer that converts C++ code into a symbolic graph for commitment and executes it, building its execution graph. We implemented the arbitrator smart contract and deployed it on the Avalanche Fuji testnet, demonstrating that the dispute resolution cost is under $1. For performance evaluation, we first focus on the core of general ML workloads, namely matrix multiplication. We evaluate Arbigraph's parallelization speedup in dispute resolution compared to serial multiplication (along with execution graph generation). The speedup grows linearly with matrix dimensions, reaching $20x$. This demonstrates the significance of parallelization for blockchain-validated ML applications. We demonstrate Arbigraph's efficacy by executing full LLM inference based on the Llama2 framework [25] using three public models with millions of parameters, achieving a $7x$ speedup. Several practical challenges remain for broad deployment, including implementing existing solutions for deterministic execution across different architectures [26, 27] and leveraging GPUs for a higher degree of parallelism.

In summary, our main contributions are:

- a novel dual-graph data structure that enables constant-time memory access and dynamic control flows,
- a formal model of optimistic execution delegation (OED),

- a framework transforming C++ code into a dual graph,
- Arbigraph, a Turing-complete OED protocol,
- proof that Arbigraph is secure,
- a scalable matrix multiplication OED implementation, and
- a scalable LLM inference OED implementation.

Arbigraph opens the door to embedding ML inference directly into smart contracts, enabling decentralized alternatives to centralized digital services. These *actually smart* contracts can move funds based on nuanced real-world conditions, adjudicate insurance claims objectively, enforce intellectual property rights with contextual understanding, and provide fair, transparent governance for the digital platforms that increasingly mediate our daily interactions.

2 Related Work

Classical work on verifiable-computation [28, 29, 30, 31] optimize for cryptographic minimalism and theoretical verifier efficiency—but they often clash with on-chain gas budgets, prover scalability, high cost of interactive proofs, and trust assumptions that most blockchain systems require.

Several approaches aim to extend the computational capabilities of blockchain systems while maintaining security and verification guarantees. These approaches broadly fall into three categories: VM-based, zero-knowledge proof, and circuit-based.

Arbitrum [20] and Optimism [19] implement so-called *optimistic rollups* to move computation off-chain while providing mechanisms for fraud proofs on-chain. They employ virtual machines that emulate physical CPU architectures.

Both systems leverage a Merklized memory structure for state management. This structure enables verification through cryptographic proofs. However, each memory write requires an $\mathcal{O}(\log n)$ overhead to update the Merkle commitment. Arbigraph also follows the optimistic approach, but it avoids the memory access overhead inherent in VM-based designs.

StarkNet [32], zkSync [17], and Polygon zkEVM [33] utilize proof systems to verify computation integrity. Instead of disputes they generate cryptographic proofs that smart contracts can verify. Like optimistic approaches, proof system, based approaches such as those implemented in Cairo [34] and Miden [18], commit state in Merkle trees to facilitate proof generation. This design incurs an $\mathcal{O}(\log n)$ overhead for each memory access due to the required tree traversal. A major drawback of proof system based approaches is that they require precomputed look-up tables [35, 36] when implementing non-linear operations, which limits the computation accuracy. Additionally, generating proofs for every computation introduces substantial overhead, which is significant for high complexity computations. In contrast, Arbigraph supports non-linear operations and only imposes overheads during disputes, eliminating computational overhead during normal operation.

Recent years are seeing a rush for protocols to validate ML computations on blockchains. Agatha [22] relies on fixed-size circuits, which requires all execution paths to be pre-unrolled. This limits expressiveness and scalability, prohibiting dynamic, data-dependent computations. Verde [27], is another optimistic protocol that assumes fixed size circuits. Unlike Agatha, Verde divides ML inference or training into large computation units and uses a refereed delegation to resolve disputes. It assumes that referees hold the entire program and can compute large programs such as matrix multiplications, which makes it not trivially suitable for smart contracts.

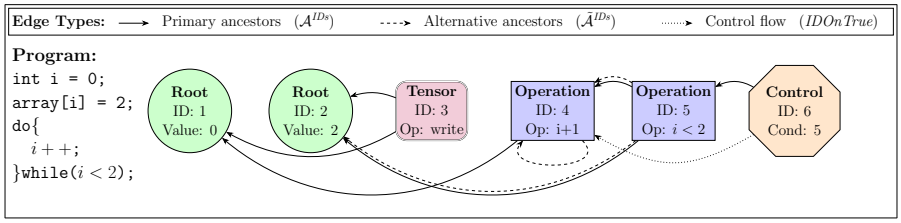

Figure 1: Symbolic graph representation of a simple program.

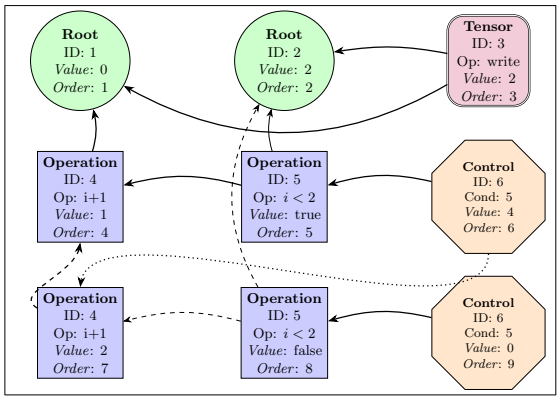

Figure 2: Execution graph of the program in Fig. 1

OPML [21] is ongoing open-source project that employs an optimistic MIPS-like virtual machine; to minimize computational overhead, it partitions programs into small units, pinpointing the disputed unit before compiling it into low-level commands for verification; this allows for parallel evaluation of decoupled computation units rather than local parallelization, such as matrix multiplications. However, each memory write still requires an $\mathcal{O}(\log n)$ overhead due to the Merkle tree-based state management, and the MIPS architecture does not support parallelization, which limits scalability, for certain workloads, specifically for large matrix operations. Unlike Agatha and Verde, Arbigraph supports dynamic control flows; moreover, whereas Verde requires the referee to hold the entire program, Arbigraph requires it to hold only a constant size commitment to the program; finally, in contrast to OPML, Arbigraph enables efficient local parallelization, particularly for computationally intensive operations such as matrix multiplications.

3 Model

The system comprises two participants, $\mathcal{H}$ and $\mathcal{M}$, and a smart contract $\mathcal{C}$ (a stateful automaton with a clock). The system operates in two phases: a program commitment phase and an input processing phase.

Program Commitment Phase. In this initial phase, a trusted third party publishes a program Φ of size k to all participants. The execution trace of the program, denoted by $\tilde{\Phi}$, can have size n where n might be much larger than k. A trusted party initializes the smart contract $\mathcal{C}$ with

a commitment $commit_1(\Phi)$, which requires $\mathcal{O}(k)$ computation time.[2]

Processing Phase. Subsequently, the adversary chooses an input x of size m and the trusted third party sends it to both participants $\mathcal{H}$ and $\mathcal{M}$. It also submits a commitment $commit_2(x)$ to the smart contract $\mathcal{C}$, which requires $\mathcal{O}(m)$ computation time.

For simplicity of presentation, hereinafter we combine the program and input into a single program entity. This simplification maintains generality because we can incorporate the input into the program, forming a new program whose commitment is the concatenation of the original program and input commitment. We denote this combined entity by Φ, its output as ϕ and the combined commitment as $commit(\cdot)$ with runtime $\mathcal{O}(k+m)$, where it is possible that $k+m \ll n$.

Participants can send and receive messages to and from the smart contract $\mathcal{C}$. The network delivers any message to or from the smart contract within at most Δ time units.

Commit-Execution-Verification Framework. The processing phase follows a commit-then-verify protocol as follows:

1. *Initialization*: Both parties receive the program Φ, and the smart contract stores its commitment $commit(\Phi)$.
2. *Execution*: Each party independently executes the program according to protocol specifications, producing the trace $\tilde{\Phi}$.
3. *Result Commitment*: The parties commit to their execution outcomes by sending them to the smart contract.
4. *Dispute Resolution*: In case of a disagreement, the smart contract determines the correct result and identifies the dishonest party.

Regardless whether there is a dispute or not, the execution ends once the smart contract $\mathcal{C}$ *outputs* a tuple $(y, cheater)$. The enforcement of penalties against the identified dishonest party falls beyond the scope of our formal analysis.

Standard cryptographic assumptions hold[2]: Both participants possess a unique identity using a Public Key Infrastructure (PKI) and a public-private key pair (pk, sk) for signing messages. *Hash* is a cryptographic collision-resistant hash function.

We now define an optimistic execution delegation that ensures completeness, soundness, and progress.

Definition 1 (Optimistic Execution Delegation)**.** *A protocol* Π *is an* optimistic execution delegation *if it comprises functions defining the commitment, participant and smart contract actions, and satisfies the following three properties.*

1. ***Completeness:*** *For any program* Φ*, if both* $\mathcal{H}$ *and* $\mathcal{M}$ *follow the protocol* Π*, the smart contract* $\mathcal{C}$ *does not output anything but* $(\phi, \perp)$*.*

When considering adversarial behavior, we assume, without loss of generality, that $\mathcal{H}$ *acts according to the protocol specification, thus we call it the* honest party*, and that* $\mathcal{M}$ *acts arbitrarily, in a Byzantine manner, thus we call it the* malicious party*.*

[2]We omit for brevity the standard dependence on the system's security parameter.

2. ***Soundness:*** *For any program Φ, if $\mathcal{H}$ follows the protocol Π and $\mathcal{M}$ commits on an output different from ϕ, then the smart contract $\mathcal{C}$ outputs $(\phi, \mathcal{M})$, correctly identifying $\mathcal{M}$ as the malicious party.*

3. ***Progress:*** *For any program Φ with an execution trace of size n, the protocol terminates within $\mathcal{O}(\log(n))$ time units.*

4 Arbigraph

Arbigraph utilizes a novel dual-graph data structure (§4.1) that enables cryptographic proofs (§4.2) that an execution step is valid. The protocol (§4.3) uses this to implement OED.

4.1 Dual-Graph Architecture

Arbigraph employs two interdependent graphs: a *symbolic graph* defines the program's structure and an *execution graph* captures the dynamic runtime trace with computed values. We illustrate with an example of a simple program and its matching graphs (Fig. 1).

Symbolic Graph Arbigraph represents operations as interconnected data structures called *nodes*. Each node N has a unique *identifier* $ID(N)$. It also has ID references to nodes that N depends on, called *ancestors*. There are four types of nodes, where the *type* of node N is $Type(N)$.

Root Nodes ($ROOT$) initialize computation with predefined values, denoted $Val(N)$ (nodes with IDs 1 and 2 in Fig. 1).

Operation Nodes ($OPER$) perform computational operations using an *operation descriptor* $Op(N)$ applied to ancestor values. The values can be scalars, vectors, or matrices, and operations can be addition, multiplication, and division (IDs 4 and 5 in Fig. 1). They can also be more complex operations like convolution and activation functions (e.g., ReLU).

Control Nodes ($CTRL$) manage conditional branching by referencing a *condition node* $Condition$ and two target identifiers: $IDOnTrue$ and $IDOnFalse$. Their output value equals the selected branch identifier (ID 6 in Fig. 1).

Tensor Nodes (TSR) manipulate tensors, which are a dynamic memory, through three operations: read (accessing values at specific index of the tensor), write (storing values at index), and root (initializing entire array state). Each tensor node contains a tensor ID and references a node with the index to access. Write operations also reference a node that holds the value to store (ID 3 in Fig. 1).

Due to conditional branches, the execution might depend on different ancestors, for example, in the program in Fig. 1, the ancestor of ID 4 node in the first loop iteration is the ID 1 node, whereas in the second loop iteration, it is ID 4 node itself. Therefore, when the execution encounters a node with a given identifier for the first time, it uses the primary ancestor ID vector, denoted by $\mathcal{A}^{IDs}(N)$, to locate the most recent instances of that node's ancestors. If the node is revisited (e.g., after a loop), the execution uses an alternative ancestor vector, denoted by $\tilde{\mathcal{A}}^{IDs}(N)$, which identifies the most recent execution state rather than relying on the original $\mathcal{A}^{IDs}(N)$.

The symbolic graph thus defines a program Φ by encoding the set of operations and their execution order. This allows a party to execute the program in a deterministic way, going by the order of the nodes, except when encountering a control node, in which case the party jumps to the symbolic node with identifier matching the control node's output value.

A symbolic graph $\mathcal{S} = (V, E)$ is thus a directed acyclic graph, where V is a set of n nodes defining the static structure of a program, and $E \subseteq V \times V$ is a set of directed edges. Each node $N^{symb} \in V$ has a unique topological order index $\mathcal{O}_{\mathcal{S}}(N^{symb}) \in \{1, 2, \ldots, n\}$. For any edge $(N_1^{symb}, N_2^{symb}) \in E$, node N_1^{symb}, whose ID is in $\mathcal{A}^{IDs}(N_2^{symb}) \cup \tilde{\mathcal{A}}^{IDs}(N_2^{symb})$, is an ancestor of node N_2^{symb}, meaning that the computation at N_2^{symb} depends on the value of N_1^{symb}. The *field set* of a symbolic node N^{symb}, denoted by $Fields(N^{symb})$, includes $ID(N^{symb})$, $Type(N^{symb})$, $\mathcal{O}_{\mathcal{S}}(N^{symb})$, $Op(N^{symb})$, $\mathcal{A}^{IDs}(N^{symb})$, $\tilde{\mathcal{A}}^{IDs}(N^{symb})$, for root nodes, the field set also includes $Val(N^{symb})$.

We are interested in symbolic graph that represents a *valid program*.

Definition 2 (Valid Program)**.** *A program Φ represented by a symbolic graph $\mathcal{S} = (V, E)$ and a canonical ordering function $\mathcal{O}_{\mathcal{S}}(\cdot) : V \to \mathbb{N}$, is a* valid program *if it satisfies the following properties:*

***Well-formedness**: For all nodes $N \in \mathcal{S}$: If ID is the set of all valid identifiers, N has a unique identifier $ID(N) \in \mathsf{ID}$. N conforms to exactly one of the defined node types: $Type(N) \in \{ROOT, OPER, CTRL, TSR, END\}$ and contains all required fields $Fields(N)$ of $Type(N)$.*

***Topological ordering**: For any distinct nodes $N, N' \in \mathcal{S}$ it holds that $\mathcal{O}_{\mathcal{S}}(N) \neq \mathcal{O}_{\mathcal{S}}(N')$ and for all nodes $N \in \mathcal{S}$ and ancestor identifiers $ID(N_a) \in \mathcal{A}^{IDs}(N) \cup \tilde{\mathcal{A}}^{IDs}(N)$, it holds that $N_a \in \mathcal{S}$ and $\mathcal{O}_{\mathcal{S}}(N_a) < \mathcal{O}_{\mathcal{S}}(N)$.*

***Type consistency and determinism**: For all nodes $N \in \mathcal{S}$, the operation $Op(N)$ is compatible with the types of its ancestor nodes, that is, if $\mathcal{A}^{IDs}(N) = (ID(N_{a_1}), \ldots, ID(N_{a_k}))$, then $Op(N)$ is compatible for input types $Type(N_{a_1}), \ldots, Type(N_{a_k})$, and similarly for $\tilde{\mathcal{A}}^{IDs}(N)$. For any fixed set of input values, the operation $Op(N)$ produces a unique, deterministic output value.*

A consequence of the above definition is that the output ϕ of a valid program Φ (described by $\mathcal{S}$) is deterministic.

Execution Graph The execution graph captures the execution trace $\tilde{\Phi}$ of a program Φ with computed values at each step.

An execution graph $\mathcal{G} = (V_\epsilon, E_\epsilon)$ is a directed acyclic graph where each node $N^{exe} \in V_\epsilon$ holds an ID $ID(N^{exe})$, ancestors references vector $\mathcal{A}^{IDs}(N^{exe})$, and a unique execution order index $\mathcal{O}_{\mathcal{G}}(N^{exe}) \in \{1, 2, \ldots, |V_\epsilon|\}$, that is part of the field set $Fields(N^{exe})$. The node at index i in the execution graph $\mathcal{G}$ is denoted by $\mathcal{G}(i)$. Each node contains a computed value $Val(N^{exe})$ determined, for non-root nodes, by applying its operation $Op(N^{exe})$ to the values of its ancestors. For any edge $(N_1^{exe}, N_2^{exe}) \in E_\epsilon$, node N_1^{exe}, whose ID is in $\mathcal{A}^{IDs}(N_2^{exe})$, is an ancestor of node N_2^{exe}, indicating that the computation at N_2^{exe} depends on the value of N_1^{exe}.

Each execution node N^{exe} includes all the fields of a corresponding symbolic node N^{symb}, but also holds a computed value $Val(N^{exe})$. This value is determined during execution, and is present even in non-root nodes. The execution graph forms by iterating over symbolic graph nodes according to their order, executing each node's operation in combination with the most recent ancestor values. Each node in the execution graph is added to a vector of nodes, which forms the order of the execution graph. Upon encountering a control node, execution jumps to the symbolic node with identifier matching the control node's output value. Tensor operations maintain memory consistency through linking of reads to their most recent same-index writes or root tensor nodes. As a result

of the above process, each node in the execution graph $N^{exe} \in \mathcal{G}$ shares its ID with a node in the symbolic graph $N^{symb} \in \mathcal{S}$. The detailed algorithm for this deterministic construction is in Appendix A.

Node Equivalence We define two notions of equivalence between nodes. Two nodes are *equivalent* if they are identical in all attributes and *structurally equivalent* if they are identical in all attributes except maybe for the computed value.

Definition 3 (Node Equivalence)**.** *Two execution nodes N_1^{exe} and N_2^{exe} are said to be* equivalent*, denoted by $N_1^{exe} \equiv_f N_2^{exe}$, if*

$$\forall f \in \mathit{Fields}(N_1^{exe}) \cup \mathit{Fields}(N_2^{exe}), \quad f(N_1^{exe}) = f(N_2^{exe}).$$

Two nodes, N_1 and N_2 (each either symbolic or execution), are structurally equivalent*, denoted by $N_1 \equiv_s N_2$, if*

$$\forall f \in \mathit{Fields}(N_1) \cup \mathit{Fields}(N_2) \setminus \{\mathit{value}\}, \quad f(N_1) = f(N_2).$$

Note that each execution node is structurally equivalent to a symbolic node with the same ID. Moreover, any two honestly generated execution nodes N_1^{exe} and N_2^{exe}, are structurally equivalent if and only if they share the same identifier: $\mathit{ID}(N_1^{exe}) = \mathit{ID}(N_2^{exe}) \iff N_1^{exe} \equiv_s N_2^{exe}$.

Dispute Identification When two executions graphs are not identical, the first point of disagreement is called the *earliest dispute node*. We denote by $N^{exe} \prec_{\mathcal{G}} \tilde{N^{exe}}$ that N^{exe} has lower index than $\tilde{N^{exe}}$ in $\mathcal{G}$.

Definition 4 (Earliest Dispute Node)**.** *Let $\mathcal{G}$ and $\tilde{\mathcal{G}}$ be two different execution graphs.*

The earliest dispute node *of $\mathcal{G}$ and $\tilde{\mathcal{G}}$, denoted by $N_D^{exe}(\mathcal{G}, \tilde{\mathcal{G}})$, is the first node only in $\mathcal{G}$, that is, the unique node in $\mathcal{G}$ satisfying both $N_D^{exe}(\mathcal{G}, \tilde{\mathcal{G}}) \notin \tilde{\mathcal{G}}$, and*

$$\forall N^{exe} \in (\mathcal{G} \setminus \tilde{\mathcal{G}}) \setminus \{N_D^{exe}(\mathcal{G}, \tilde{\mathcal{G}})\} : \quad N_D^{exe}(\mathcal{G}, \tilde{\mathcal{G}}) \prec_{\mathcal{G}} N^{exe}.$$

Denote the index of the dispute node by $\mathcal{I}$.

The *common prefix vector* is the vector of the execution graph $\mathcal{G}$ nodes, sorted by the canonical ordering, that are identical in both execution graphs, with the last node called the *last common node*.

Definition 5 (Common Prefix Vector and Last Common Node)**.** *The* common prefix vector *of $\mathcal{G}$ and $\tilde{\mathcal{G}}$, denoted by $\mathcal{A}(\mathcal{G}, \tilde{\mathcal{G}})$, consists of the first $\mathcal{I} - 1$ nodes of the honest execution graph:*

$$\mathcal{A}(\mathcal{G}, \tilde{\mathcal{G}}) = \big(\mathcal{G}(1), \ldots, \mathcal{G}(\mathcal{I} - 1)\big) = \big(\tilde{\mathcal{G}}(1), \ldots, \tilde{\mathcal{G}}(\mathcal{I} - 1)\big).$$

The last common node *N_L^{exe} is defined as:*

$$N_L^{exe}(\mathcal{G}, \tilde{\mathcal{G}}) = \mathcal{G}(\mathcal{I} - 1) = \tilde{\mathcal{G}}(\mathcal{I} - 1).$$

Note that it holds that $\mathcal{I}(\mathcal{G}, \tilde{\mathcal{G}}) = |\mathcal{A}(\mathcal{G}, \tilde{\mathcal{G}})| + 1$. For simplicity, we denote by $\mathcal{A}$, N_L^{exe}, and $\mathcal{I}$ the common prefix vector, the last common node, and the dispute index of the execution graphs of both parties and by $N_D^{exe}(\mathcal{H})$ and $N_D^{exe}(\mathcal{M})$ the early dispute nodes of the honest and malicious parties, respectively.

4.2 Proof Structure and Verification

The dual-graph of the Arbigraph protocol allows an arbitrator to verify the correctness of the disputed node given the common prefix and dispute node.

A central data structure the proof uses is a *Merkle tree* [37] that acts as a commitment to the symbolic and execution graphs. A Merkle tree $\mathcal{T}$ over a vector of nodes $(N_1, \ldots, N_n)$ defines a root $\mathcal{R}_\mathcal{T}$ that commits to the entire dataset. Each node has an efficient $\mathcal{O}(\log(n))$ inclusion proof $\mathsf{P}_\mathcal{T}(N)$ that the smart contract can verify in logarithmic time, called a *Merkle proof*. But a computationally-bounded adversary cannot forge a Merkle proofs for a node and its location in the vector. Arbigraph uses a novel concept of *recency rule* to ensure that when multiple instances of the same node ID exist, to prevent a malicious party from providing a less recent version of a node, a smart contract can compare the order of the nodes to identify the misbehaving party.

Definition 6 (Proof Structure)**.** *A proof for the dispute node N_D^{exe} by party $\mathcal{P}$, denoted by $Proof_\mathcal{P}$, compromises the following components:*

Node Representation: *The disputed execution node N_D^{exe} and its fields,*

Execution Merkle Proof ($\mathsf{P}_{\text{exe}}(N_D^{\boldsymbol{exe}}, \mathcal{R}_E)$): *A Merkle proof of the inclusion of N_D^{exe} in the execution Merkle tree with root $\mathcal{R}_E$,*

Symbolic Node ($N_D^{\boldsymbol{symb}}$): *The symbolic node corresponding to N_D^{exe},*

Symbolic Merkle Proof ($\mathsf{P}_{\text{symb}}(N_D^{\boldsymbol{symb}}), \mathcal{R}_{\boldsymbol{sym}}(\mathcal{S})$): *A Merkle proof demonstrating the inclusion of N_D^{symb} in the system constant symbolic Merkle root $\mathcal{R}_{sym}(\mathcal{S})$.*

Adjacency Proof ($\mathcal{J}_{\mathcal{R}_E}(N_L^{\boldsymbol{exe}}, N_D^{\boldsymbol{symb}})$): *A proof that confirms the disputed node is the correct successor of N_L^{exe}, decided by the execution of the program. The proof contains the previous node N_L^{exe}, its corresponding symbolic node N_D^{symb}, and their Merkle proofs. This proof proves only the structural correctness of node selection—whether the execution properly followed sequential ordering for operation nodes or branch selection for control nodes—while deferring value verification to a separate step. When N_L^{exe} is control node, it confirms that N_D^{symb} has the same ID as the value of the control node; For other cases, it uses Merkle paths to verify proper ordering in the symbolic graph.*

Tensor Read Proof (TensorProof($N_D^{\boldsymbol{exe}}$)): *For tensor read nodes, a proof includes the most recent tensor operation (write or root) with matching tensor ID and index from the common prefix, with its Merkle proof. To validate that the attached tensor operation is the most recent one, a smart contract compares the sequence number of both parties, and in case of dispute, it uses the recency rule to determine the correct operation. This enforces memory consistency by ensuring reads access only the latest valid state, preventing use of stale values while enabling constant-time memory writes.*

Ancestors Proofs: *A set of proofs for the input nodes required to compute the disputed node's value, accompanied by a useAlternativeAncestors flag. This flag indicates whether to use the primary ancestor vector $\mathcal{A}^{IDs}(N_D^{exe})$ or the alternative set $\tilde{\mathcal{A}}^{IDs}(N_D^{exe})$, enabling proper handling of loops and repeated node executions. For each ancestor ID, the proof includes the specific ancestor node instance and its Merkle proof demonstrating inclusion in the execution graph's common prefix. While ancestor nodes must exist in the common prefix, a malicious party might attempt to reference older*

instances of nodes with the same identifier that were executed earlier in the trace. The protocol's recency rule prevents this attack by requiring that when multiple instances of a node with the same ID exist, the verification always uses the most recent instance from the common prefix, ensuring that an honest party correctly referencing the freshest ancestors prevails in any dispute.

Collectively, these proof components ensure that N_D^{exe} has the correct structure, i.e., all fields except for the value are correct. To complete the verification, the smart contract computes the value Val(N_D^{exe}) using the operation Op(N_D^{exe}) and the values of its ancestors $\mathcal{A}_{N_D^{exe}}$ (or $\tilde{\mathcal{A}}_{N_D^{exe}}$, if useAlternativeAncestors is true).

4.3 Protocol Execution

Having defined its core data structures, we now describe how the Arbigraph smart contract orchestrates the protocol execution from initialization through dispute resolution.

4.3.1 Initialization

At initialization, parties $\mathcal{H}$ and $\mathcal{M}$ receive Φ as the symbolic graph $\mathcal{S}$. The smart contract $\mathcal{C}$ receives its Merkle root commitment $\mathcal{R}_{sym}(\mathcal{S})$.

4.3.2 Execution Phase

Once the symbolic graph is committed, each party independently executes the computation according to the protocol specification. The computation process of each party forms the execution graph, $\mathcal{G}_{\mathcal{S}}^{\mathcal{H}}$, where the number of nodes in the graph is $|\mathcal{G}_{\mathcal{S}}^{\mathcal{H}}|$. Each party forms a vector of the nodes in the execution graph according to the order of the nodes in the canonical ordering. The execution graph commitment is created using a Merkle tree that each party forms using the vector of nodes, where $\mathcal{R}_{\mathcal{H}}$ and $\mathcal{R}_{\mathcal{M}}$ are the Merkle roots of the execution graphs of the honest and malicious parties, respectively.

4.3.3 Commit and Verify Phase

After execution, each party commits on their output values and sends the commitment to $\mathcal{C}$. If the parties disagree on the output, the contract requests their execution graph Merkle roots, and the sizes of their execution graphs. Upon receiving execution graph commitments, the smart contract verifies that each Merkle root is consistent with the corresponding party's claimed output, that is the last node of the execution graph, using Merkle proofs (Algorithm 1, lines 11–13). If verification fails, that party is immediately identified as malicious. This prevents parties from submitting Merkle roots that don't correspond to their claimed outputs.

4.3.4 Dispute Phase - Common Prefix Identification

In the event of a disagreement on the program result, the protocol enters the *dispute phase*. The smart contract utilizes a *Bisection protocol* [20, 23, 24] to efficiently identify the index of the disputed node. This interactive protocol requires active participation from both parties and yields a tuple consisting of either a dispute index or identification of a misbehaving party. If one of the parties misbehaves during the Bisection protocol, the smart contract returns the identity of the misbehaving party $\mathcal{P}$ and outputs $(\phi, \mathcal{P})$ (Algorithm 1, line 17). To account for arbitrary behavior

by the attacker during the Bisection protocol, we define the notion of a *prefix commitment*, which ensures that the attacker is bound to a common prefix of the execution graph.

Definition 7 (Prefix Commitment)**.** *Let $\mathcal{G}$ be an execution graph and let $\mathcal{R}_{\mathcal{M}}$ be a Merkle root claimed to commit to it. We say that $\mathcal{R}_{\mathcal{M}}$ is a* $(\mathcal{I}, \mathcal{G})$ prefix commitment *if the following holds.*

For every index $j < \mathcal{I}$, a computationally bounded adversary, except with negligible probability, cannot produce a valid Merkle proof under $\mathcal{R}_{\mathcal{M}}$ that verifies a value different from $Val(N_j)$ at position j. At index $\mathcal{I}$, the adversary may produce at most one valid Merkle proof for any value.

The Arbigraph protocol ensures that if no cheating is detected during the Bisection protocol, the malicious party is bounded to all nodes with index less than the disputed index.

Definition 8 (Bisection Protocol Guarantee)**.** *Consider parties executing the Arbigraph protocol where an honest party $\mathcal{H}$ provides execution Merkle root $\mathcal{R}_{\mathcal{H}}$ derived from execution graph $\mathcal{G}_{\mathcal{S}}^{\mathcal{H}}$ and a malicious computationally-bounded party $\mathcal{M}$ provides a value $\mathcal{R}_{\mathcal{M}}$. If the Bisection protocol returns $(\mathcal{I}, \perp)$, then $\mathcal{R}_{\mathcal{M}}$ is an $(\mathcal{I}, \mathcal{G}_{\mathcal{S}}^{\mathcal{H}})$ prefix commitment. If the Bisection protocol completes with $(\perp, \mathcal{P})$, then $\mathcal{P}$ did not follow the protocol.*

The details of the mostly standard Bisection protocol are provided in Appendix B, for completeness.

4.3.5 Dispute Phase - Node Dispute Resolution

If the Bisection protocol completes with a dispute index, it means that the smart contract identifies the first disputed node; therefore, it initiates the node dispute resolution phase (Arbitrate function Algorithm 1, line 20), where it verifies the proofs for the disputed nodes, $N_D^{exe}(\mathcal{H})$ and $N_D^{exe}(\mathcal{M})$ respectively, and determines the identity of the misbehaving party, and by elimination the correct output, it proceeds as follows. The smart contract first verifies the correctness of the proofs (Algorithm 2, lines 6–7), this includes verifying Merkle proofs, the *adjacency proof* that proves that the structure of the dispute node is correct and computation of the value of the disputed node. If one of the proofs is invalid, the smart contract returns the output of the other party and the identity of the misbehaving party. If both proofs are valid, it means that one of the parties provided wrong ancestors, thus the smart contract proceeds to identify the misbehaving party using the *recency* test, that is, it determines the correct party using the more recent version of ancestors of the disputed node.

5 Security Analysis

We show that the Arbigraph protocol meets the three security properties: (1) Completeness (Lemma 5.1), (2) Soundness (Lemma 5.5), and (3) Progress (Lemma 5.6).

5.1 Completeness

We show that the Arbigraph protocol is complete, meaning that if both parties follow the protocol correctly, they produce the same output.

Lemma 5.1 (Completeness)**.** *For any valid program Φ (Definition 2) represented by a symbolic graph $\mathcal{S}$ with Merkle root $\mathcal{R}_{sym}(\mathcal{S})$ committed to the smart contract, if both parties $\mathcal{H}$ and $\mathcal{M}$ follow*

the Arbigraph protocol, then: Both parties produce the same output: $\phi(\mathcal{H}) = \phi(\mathcal{M})$ and the smart contract outputs $(\phi(\mathcal{H}), \perp) = (\phi(\mathcal{M}), \perp)$.

Proof. Let $\mathcal{S}$ be a valid program Φ according to Definition 2. By the properties of valid programs: The symbolic graph $\mathcal{S}$ is well-formed, with each node having a unique identifier and conforming to a defined node type with all required fields,

there exists a canonical topological ordering of nodes in $\mathcal{S}$ such that all ancestors of a node precede it in this ordering, and for all node $N \in \mathcal{S}$, the operation $Op(N)$ is type-compatible with its ancestors and produces a unique, deterministic output for any fixed set of input values.

Since both parties $\mathcal{H}$ and $\mathcal{M}$ follow the Arbigraph protocol faithfully, each party: constructs an execution graph by sequentially evaluating nodes in the canonical topological order defined by $\mathcal{S}$, computes the value of each node deterministically using its operation descriptor $Op(N)$ applied to the values of its ancestor nodes, and handles control flow identically because control node evaluations, which determine execution paths, are deterministic functions of their conditions.

Given that both parties follow the exact same deterministic process with the same program Φ, they produce identical execution graphs $\mathcal{G}_{\mathcal{S}}^{\mathcal{H}} = \mathcal{G}^{\mathcal{M}}$ with identical Merkle roots $\mathcal{R}_{\mathcal{H}} = \mathcal{R}_{\mathcal{M}}$, and identical final outputs $\phi(\mathcal{H}) = \phi(\mathcal{M})$.

Since both parties produce the same output $\phi(\mathcal{H}) = \phi(\mathcal{M})$, no dispute is triggered at Algorithm 1, and the smart contract immediately outputs $(\phi(\mathcal{H}), \perp)$, where $\perp$ indicates that no party is identified as malicious. □

5.2 Soundness

We establish the soundness of the Arbigraph protocol by proving that any malicious behavior is detected.

5.2.1 Supporting Lemmas

Before proving the soundness, we establish a several guarantees of the Arbitrate function in the face of specific deviations.

Lemma 5.2 (Detection of Structural Discrepancies)**.** *For all valid symbolic graphs $\mathcal{S}$, if an adversary computes $\mathcal{R}_{\mathcal{M}}$ and $Proof_{\mathcal{M}}$ such that: $\mathcal{R}_{sym}(\mathcal{S})$ is the Merkle root of $\mathcal{S}$; $\mathcal{R}_{\mathcal{H}}$ is a Merkle root corresponding to a correct execution of $\mathcal{G}_{\mathcal{S}}^{\mathcal{H}}$, $\mathcal{R}_{\mathcal{M}}$ is an $(\mathcal{I}, \mathcal{G}_{\mathcal{S}}^{\mathcal{H}})$ prefix commitment provided by the computationally-bounded malicious party $\mathcal{M}$ that has access to $\mathcal{S}$; $Proof_{\mathcal{H}}$ is a proof (Definition 6) for a node $N_D^{exe}(\mathcal{H})$ at position $\mathcal{I}$ in the execution graph corresponding to $\mathcal{R}_{\mathcal{H}}$; $Proof_{\mathcal{M}}$ is a valid proof data structure that was created by $\mathcal{M}$ and does not contain a valid proof for any node $N_D^{exe}(\mathcal{M})$ at position $\mathcal{I}$ such that $N_D^{exe}(\mathcal{H}) \equiv_s N_D^{exe}(\mathcal{M})$ (it might not contain a proof or be corrupted in an arbitrary way). Then* Arbitrate *(Algorithm 2) outputs ϕ.*

Proof. Assume by contradiction that Arbitrate does not output $(\phi, \mathcal{M})$. We show this means that $Proof_{\mathcal{M}}$ must successfully pass all verification steps in the Arbitrate function, leading to a contradiction.

First, when execution graphs differ in length, the protocol handles this by having the party with the shorter graph provide a dummy node at the dispute point. Regardless of whether it is $\mathcal{H}$ with the dummy node and $\mathcal{M}$ with an invalid proof, or $\mathcal{M}$ with the dummy node and $\mathcal{H}$ with a valid node, the verification conditions (Algorithm 2, lines 8–11) correctly identify $\mathcal{M}$ as cheating, contradicting our assumption. Thus, neither party provides a dummy node at the dispute position.

If either of the proofs $N_D^{exe}(\mathcal{M})$ or $N_D^{symb}(\mathcal{M})$ are invalid, verification fails (Algorithm 3, lines 5 and 7) and Arbitrate outputs $(\phi, \mathcal{M})$, contradicting our assumption. Thus, both proofs are valid.

If $N_D^{exe}(\mathcal{M})$ is not structurally equivalent to $N_D^{symb}(\mathcal{M})$, verification fails (Algorithm 3, line 9) and Arbitrate outputs $(\phi, \mathcal{M})$, contradicting our assumption.

Thus, the execution node provided by $\mathcal{M}$ has the same structure (same ID, operation type, ancestor references, etc.) as its claimed symbolic counterpart.

If $N_D^{symb}(\mathcal{M})$ is not the correct successor to N_L^{exe} in the symbolic graph, verification fails and Arbitrate outputs $(\phi, \mathcal{M})$, contradicting our assumption.

Since $\mathcal{H}$ follows the protocol correctly, $N_D^{exe}(\mathcal{H})$ is structurally equivalent to the symbolic node that correctly follows N_L^{exe}. Due to the deterministic nature of the execution graph, there is exactly one symbolic node that can correctly follow N_L^{exe}. Therefore, $N_D^{symb}(\mathcal{M})$ and the symbolic node corresponding to $N_D^{exe}(\mathcal{H})$ must be the same node in the symbolic graph. This verification is performed in VerifyAdjacency (Algorithm 4), for control N_L^{exe} Verification checks that the dispute node's ID matches the expected branch value (Algorithm 4, lines 9–16). For other types of N_L^{exe} Verification ensures the dispute node is the correct next node in the canonical ordering (Algorithm 4, lines 6–8). Thus, $N_D^{symb}(\mathcal{M})$ is the symbolic node corresponding to the successor of N_L^{exe}.

This leads directly to a contradiction: Since an execution node that is structurally equivalent to $N_D^{symb}(\mathcal{M})$ is the proper successor of N_L^{exe} that $\mathcal{M}$ committed to (from steps 1 and 3), $N_D^{exe}(\mathcal{M}) \equiv_s N_D^{symb}(\mathcal{M})$ (from step 2), $N_D^{exe}(\mathcal{H}) \equiv_s N_D^{symb}(\mathcal{H}) \equiv_s N_D^{symb}(\mathcal{M})$ (by honest execution), it follows that $N_D^{exe}(\mathcal{M}) \equiv_s N_D^{exe}(\mathcal{H})$.

However, the lemma explicitly states that $Proof_{\mathcal{M}}$ does not contain a valid proof for any node structurally equivalent to $N_D^{exe}(\mathcal{H})$. This directly contradicts our derived conclusion.

Therefore, our initial assumption must be false, and Arbitrate must output $(\phi, \mathcal{M})$. □

Lemma 5.3 (Ancestor Integrity Verification)**.** *For all valid symbolic graphs $\mathcal{S}$, if an adversary computes $\mathcal{R}_{\mathcal{M}}$ and $Proof_{\mathcal{M}}$ such that: $\mathcal{R}_{sym}(\mathcal{S})$ is the Merkle root of a valid symbolic graph $\mathcal{S}$, $\mathcal{R}_{\mathcal{H}}$ is a Merkle root corresponding to a correct execution of $\mathcal{G}_{\mathcal{S}}^{\mathcal{H}}$, $\mathcal{R}_{\mathcal{M}}$ is an $(\mathcal{I}, \mathcal{G}_{\mathcal{S}}^{\mathcal{H}})$ prefix commitment provided by the computationally-bounded malicious party $\mathcal{M}$ that has access to $\mathcal{S}$; $Proof_{\mathcal{H}}$ contains a correct proof (as required by Definition 6) for a node $N_D^{exe}(\mathcal{H})$ at position $\mathcal{I}$ in the execution graph corresponding to $\mathcal{R}_{\mathcal{H}}$; $Proof_{\mathcal{M}}$ contains a proof for a node $N_D^{exe}(\mathcal{M})$ at position $\mathcal{I}$ with $N_D^{exe}(\mathcal{H}) \equiv_s N_D^{exe}(\mathcal{M})$ (i.e., the nodes are structurally equivalent), $Proof_{\mathcal{M}}$ does not provide a valid set of ancestors that matches those in $Proof_{\mathcal{H}}$. Then Arbitrate (Algorithm 2) outputs $(\phi, \mathcal{M})$.*

Proof. Assume by contradiction that Arbitrate does not output $(\phi, \mathcal{M})$. This means that $Proof_{\mathcal{M}}$ must successfully pass all verification steps in the Arbitrate function, despite containing invalid ancestors. We show this leads to a contradiction.

Given that ancestor validity is essential for verification, we must consider exactly what makes ancestors valid in Arbigraph. The ancestor ID uniquely determines structure, while the protocol requires using the most recent matching instance from the common prefix vector. Since $\mathcal{M}$ fails to provide valid ancestors matching those in $Proof_{\mathcal{H}}$, this deviation must occur in one of the following precisely defined ways, each of which our verification steps explicitly detect:

Case 1: Incorrect set of ancestor IDs. If $\mathcal{M}$ provides a set of ancestors whose IDs do not match the required set ($\mathcal{A}^{IDs}(N_D^{symb}(\mathcal{M}))$ or $\tilde{\mathcal{A}}^{IDs}(N_D^{symb}(\mathcal{M}))$), verification fails (Algorithm 3, line 13), since the provided ancestor IDs do not match the required set based on

useAlternativeAncestors. This leads Arbitrate to output $(\phi, \mathcal{M})$, contradicting our assumption. Thus, the provided ancestor IDs must match the required set.

Case 2: Inclusion of malformed ancestors. If $\mathcal{M}$ provides ancestors with malformed structure or garbage data, verification fails (Algorithm 3, line 16), where the ValidateAncestorStructure function explicitly checks that each ancestor has a valid structure. This leads Arbitrate to output $(\phi, \mathcal{M})$, contradicting our assumption. Thus, the provided ancestors must have valid structure.

Case 3: Ancestors with invalid Merkle proofs or not in the common prefix vector. If $\mathcal{M}$ provides ancestors that do not have valid Merkle proofs in the execution graph, verification fails (Algorithm 3, line 18). If the ancestor is not in the common prefix vector (Definition 7), verification fails (Algorithm 3, line 20). Either failure leads Arbitrate to output $(\phi, \mathcal{M})$, contradicting our assumption. Thus, the provided ancestors must be in the common prefix vector.

Case 4: Provision of older ancestor versions. Even if $\mathcal{M}$ provides the correct set of ancestor IDs with well-formed nodes and valid Merkle proofs, they might still provide older versions of correct ancestors. Let $N_{A^*}^{exe}$ be an ancestor in $Proof_{\mathcal{M}}$ and N_A^{exe} be the corresponding ancestor in $Proof_{\mathcal{H}}$ with the same ID. If $N_{A^*}^{exe} \prec_{\mathcal{G}_S^{\mathcal{H}}} N_A^{exe}$, then $N_{A^*}^{exe}$ is strictly older than N_A^{exe}. By the protocol's recency rule, the most recent version in the common prefix vector must be used. The CompareAncestors function (Algorithm 2, line 26) detects this discrepancy and returns A_MORE_RECENT, causing Arbitrate to output $(\phi(\mathcal{H}), \mathcal{M})$, contradicting our assumption. Thus, the provided ancestors must be the most recent versions.

Since the provided ancestors are the correct and most recent versions, the set in $Proof_{\mathcal{M}}$ must match the set in $Proof_{\mathcal{H}}$, which contradicts our assumption. Therefore, Arbitrate must output $(\phi, \mathcal{M})$ when the conditions of the lemma are met. □

Building on ancestor integrity, the next lemma addresses the most direct form of computational fraud: falsifying the final computed result. It proves that even when structure and ancestors appear valid, any deviation in the output value is detected. This completes our security framework by ensuring computational accuracy at every step.

Algorithm 1: Commit and Verify

```
1.  Function CommitAndVerify (outputA, outputB):
2.      R_sym ← getGlobalSymbolicMerkleRoot()
        // Check for initial output agreement
3.      if outputA = outputB then
            // No dispute, return agreed result
4.          return (outputA, ⊥)
        // Outputs differ, request execution graph commitments
5.      dataA ← RequestData(A)
6.      dataB ← RequestData(B)
        // Extract execution roots and sizes from submitted data
7.      R_A ← dataA.merkleRoot
8.      sizeA ← dataA.size
9.      R_B ← dataB.merkleRoot
10.     sizeB ← dataB.size
        // Verify that merkle roots correspond to the outputs
        // Verify A's output proof is consistent with claimed result
11.     if ¬VerifyOutputProof(R_A, sizeA, outputA, dataA.outputProofA) then
            // A's merkle root doesn't match claimed output
12.         return (outputB, A)
        // Verify B's output proof is consistent with claimed result
13.     if ¬VerifyOutputProof(R_B, sizeB, outputB, dataB.outputProofB) then
            // B's merkle root doesn't match claimed output
14.         return (outputA, B)
        // Initiate dispute resolution
15.     I, P ← BisectionProtocol(R_A, R_B, sizeA, sizeB)
16.     if P ≠ ⊥ then
            // Bisection identified a cheating party directly
17.         return (φ(OtherParty(P)), P)
        // Extract proofs for disputed node
18.     Proof_A ← dataA.proofs[I]
19.     Proof_B ← dataB.proofs[I]
        // Arbitrate between the proofs
20.     return Arbitrate(Proof_A, Proof_B, R_sym, R_A, R_B, I)
```

Algorithm 2: Single Step Dispute Resolution

Function Arbitrate *($Proof_A$, $Proof_B$, $\mathcal{R}_{sym}$, $\mathcal{R}_A$, $\mathcal{R}_B$, $\mathcal{I}$)*:
  if ¬ ValidateProofStructure *($Proof_A$)* **then**
    return $(\phi(B), A)$
  if ¬ ValidateProofStructure *($Proof_B$)* **then**
    return $(\phi(A), B)$
  validA ← VerifyProof ($Proof_A$, $\mathcal{R}_{sym}$, $\mathcal{R}_A$, $\mathcal{I}$)
  validB ← VerifyProof ($Proof_B$, $\mathcal{R}_{sym}$, $\mathcal{R}_B$, $\mathcal{I}$)
  // party with dummy node loses if other party is valid
  if *validB* $\neq$ *false* ***and*** *validA* $\neq$ *true* **then**
    return $(\phi(B), A)$
  else if *validA* $\neq$ *false* ***and*** *validB* $\neq$ *true* **then**
    return $(\phi(A), B)$
  else if *validA* ***and*** *validB* **then**
    // Both proofs valid, check recency
    nodeA ← $Proof_A$.node
    nodeB ← $Proof_B$.node
    if *Type*(*nodeA*) $=$ *TSR* ***and*** *nodeA.op* $=$ *READ* **then**
      // Verify tensor read recency
      writeNodeA ← $Proof_A$.recentWrite
      writeNodeB ← $Proof_B$.recentWrite
      if ¬ VerifyPathAndIdx *(writeNodeA, writeNodeA.idx, $Proof_A$.writePath, $\mathcal{R}_A$)* **then**
        return $(\phi(B), A)$
      if ¬ VerifyPathAndIdx *(writeNodeB, writeNodeB.idx, $Proof_B$.writePath, $\mathcal{R}_B$)* **then**
        return $(\phi(A), B)$
      if *writeNodeA.idx* $>$ *writeNodeB.idx* **then**
        return $(\phi(A), B)$
      else if *writeNodeB.idx* $>$ *writeNodeA.idx* **then**
        return $(\phi(B), A)$
    // Compare recency of common ancestors
    result ← CompareAncestors ($Proof_A$, $Proof_B$)
    if *result* $\neq$ *EQUAL* **then**
      return $\begin{cases} \textit{PartyA\_Correct}, & \textit{if result} = \textit{A\_MORE\_RECENT} \\ \textit{PartyB\_Correct}, & \textit{otherwise} \end{cases}$
    // This point should never be reached in practice
    return $(\phi(A), \perp)$
Function CompareAncestors *($Proof_A$, $Proof_B$)*:
  commonIDs ← GetCommonAncestorIDs($Proof_A$, $Proof_B$)
  foreach *id* $\in$ *commonIDs* **do**
    ancA ← FindAncestorByID (id, $Proof_A.ancestors$)
    ancB ← FindAncestorByID (id, $Proof_B.ancestors$)
    if *ancA.idx* $>$ *ancB.idx* **then**
      return A_MORE_RECENT
    if *ancB.idx* $>$ *ancA.idx* **then**
      return B_MORE_RECENT
  return EQUAL
Function *FindAncestorByID(id, ancestors)*:
  foreach $anc \in$ *ancestors* **do**
    if $ID(anc.node) =$ *id* **then**
      return $anc.node$
  return null

Algorithm 3: Proof Verification

Function VerifyProof *(proof, $\mathcal{R}_{sym}$, exeRoot, $\mathcal{I}$)*:
  $node \leftarrow$ proof.$node$
  $sym \leftarrow$ proof.$symNode$
  $useAlt \leftarrow$ proof.$useAlternativeAncestors$
  // 1. Check Merkle proofs
  if ¬ VerifyPathAndIdx *(node, $\mathcal{I}$, proof.exePath, exeRoot)* **then**
    return false
  if ¬ VerifyPathAndIdx *(sym, node.idx, proof.symPath, $\mathcal{R}_{sym}$)* **then**
    return false
  // 2. Check structural match
  if ¬ AreEquivalent *(node, sym)* **then**
    return false
  // 3. Check ancestors
  requiredIDs $\leftarrow \begin{cases} \tilde{\mathcal{A}}^{IDs}(node), & \text{if } useAlt \\ \mathcal{A}^{IDs}(node), & \text{otherwise} \end{cases}$
  // Check that the ancestor IDs match exactly the required set
  providedIDs $\leftarrow \{ID(anc.node) \mid anc \in$ proof.$ancestors\}$
  if *providedIDs* $\neq$ *requiredIDs* **then**
    return false // Ancestors don't match required set exactly
  // Check each ancestor's validity
  foreach $anc \in$ *proof.ancestors* **do**
    if ¬ ValidateAncestorStructure *(anc)* **then**
      return false
    if ¬ VerifyPathAndIdx *(anc.node, anc.node.idx, anc.path, exeRoot)* **then**
      return false
    if *anc.node.idx* $\geq$ *node.idx* **then**
      return false // Ancestor must precede current node
  // 4. For tensor read, check recent write
  if *Type*$(node) = TSR$ ***and node.op*** $=$ *READ* **then**
    $writeNode \leftarrow$ proof.$recentWrite$
    if ¬ VerifyPathAndIdx *(**writeNode**, **writeNode**.idx, proof.**writePath**, exeRoot)* **then**
      return false
    if *writeNode.idx* $\geq$ *node.idx* **then**
      return false // Write must precede read
    if *writeNode.**tid*** $\neq$ *node.**tid** **or** writeNode.**idx*** $\neq$ *proof.**ancestors.idxID*** **then**
      return false // Write must match tensor and index
    if *writeNode.op* $\neq$ *WRITE* ***and*** *writeNode.op* $\neq$ *ROOT* **then**
      return false // Node must be a write operation
  // 5. Check adjacency
  $adjacencyResult \leftarrow$ VerifyAdjacency (node, sym, proof.prevAgreedNode, proof.prevAgreedSymNode, proof, $\mathcal{I}$)
  if $adjacencyResult = \perp$ **then**
    return DUMMY_NODE // Special return value for dummy nodes
  else if $\neg adjacencyResult$ **then**
    return false
  // 6. Check computation
  if ¬ CheckComp *(node, sym, proof.**ancestors**, proof.**recentWrite**)* **then**
    return false
  return true

Algorithm 4: Verify Adjacency

```
1.  Function VerifyAdjacency (node, sym, prevNode, prevSym, proof, ℐ):
2.    if ¬ VerifyPathAndIdx (prevNode, ℐ − 1, proof.exePath, exeRoot) then
3.      return false
      // Check if node is a dummy node
4.    if IsDummyNode (node) then
5.      return ⊥ // Indicate this is a dummy node
6.    if Type(prevNode) ≠ CTRL then
        // Check that the symbolic index numbers are sequential
7.      if sym.idx ≠ prevSym.idx + 1 then
8.        return false
9.    else
        // For control nodes, check branch outcome matches
10.     cond ← proof.ancestors.cond
11.     if cond then
12.       expectedBranch ← prevNode.IDOnTrue
13.     else
14.       expectedBranch ← prevNode.IDOnFalse
15.     if ID(node) ≠ expectedBranch then
16.       return false
17.   return true
```

Algorithm 5: Node Computation Verification

```
1.  Function CheckComp (node, sym, ancs, recentWrite):
2.    switch Type(node) do
3.      case ROOT do
4.        return Val(node) = Val(sym)
5.      case OPER do
6.        ins ← {Val(anc) | anc ∈ ancs}
7.        return Val(node) = Op(node)(ins)
8.      case CTRL do
9.        cond ← ancs.cond if cond then
10.         exp ← node.IDOnTrue
11.       else
12.         exp ← node.IDOnFalse
13.       return Val(node) = exp
14.     case TSR do
15.       switch node.op do
16.         case ROOT do
17.           return Val(node) = Val(sym)
18.         case WRITE do
19.           val ← GetValue(node.valID, ancs)
20.           return Val(node) = val
21.         case READ do
22.           return Val(node) = Val(recentWrite)
23.   return false
```

Lemma 5.4 (Detection of Value Tampering)**.** *For all valid symbolic graphs $\mathcal{S}$, if an adversary computes $\mathcal{R}_{\mathcal{M}}$ and $Proof_{\mathcal{M}}$ such that: $\mathcal{R}_{sym}(\mathcal{S})$ is the Merkle root of a valid symbolic graph $\mathcal{S}$, $\mathcal{R}_{\mathcal{H}}$ is a Merkle root corresponding to a correct execution of $\mathcal{G}_{\mathcal{S}}^{\mathcal{H}}$, $\mathcal{R}_{\mathcal{M}}$ is an $(\mathcal{I}, \mathcal{G}_{\mathcal{S}}^{\mathcal{H}})$ prefix commitment provided by the computationally-bounded malicious party $\mathcal{M}$ that has access to $\mathcal{S}$; $Proof_{\mathcal{H}}$ contains a correct proof (as required by Definition 6) for a node $N_D^{exe}(\mathcal{H})$ at position $\mathcal{I}$ in the execution graph*

corresponding to $\mathcal{R}_{\mathcal{H}}$; $Proof_{\mathcal{M}}$ contains a valid proof for a node $N_D^{exe}(\mathcal{M})$ at position $\mathcal{I}$ where: $N_D^{exe}(\mathcal{H}) \equiv_s N_D^{exe}(\mathcal{M})$ (the nodes are structurally equivalent), and $Val(N_D^{exe}(\mathcal{H})) \neq Val(N_D^{exe}(\mathcal{M}))$ (the computed values differ). Then Arbitrate *(Algorithm 2) outputs $(\phi, \mathcal{M})$.*

Proof. Assume by contradiction that Arbitrate does not output $(\phi, \mathcal{M})$. This means that $Proof_{\mathcal{M}}$ must successfully pass all verification steps in the Arbitrate function, despite having an incorrect computed value. We show this leads to a contradiction.

Since $N_D^{exe}(\mathcal{H}) \equiv_s N_D^{exe}(\mathcal{M})$ (as stated in the lemma), the nodes are structurally equivalent—they have the same ID, operation descriptor, and ancestor references, differing only in their computed values. By Lemma 5.3, the ancestors provided by both parties are valid and consistent, as otherwise the smart contract detects the discrepancy and outputs $(\phi(\mathcal{H}), \mathcal{M})$, contradicting our assumption. As a result, for the remaining proof, we assume that the operation descriptor $Op(N_D^{exe})$ and the values of its ancestor nodes, denoted by $\{Val(N_{a_1}), \ldots, Val(N_{a_k})\}$, are uncontested. Given these fixed inputs, the correct output value is deterministically computable.

As shown in Algorithm 3, the smart contract performs computation verification using the CheckComp function in Algorithm 3. We analyze each possible node type:

Case 1: Operation Node. For an operation node, the expected computed result is:

$$Val(N_D^{exe})_{\text{expected}} \;=\; Op(N_D^{exe})\big(Val(N_{a_1}), \ldots, Val(N_{a_k})\big).$$

Since both $Op(N_D^{exe})$ and the input values are agreed upon, $Val(N_D^{exe})_{\text{expected}}$ is uniquely determined. The verification (Algorithm 5, line 5) detects if $Val(N_D^{exe}(\mathcal{M})) \neq Val(N_D^{exe})_{\text{expected}}$, causing Arbitrate to output $(\phi, \mathcal{M})$, contradicting our assumption.

Case 2: Control Node. For a control node, the expected value is:

$$Val(N_D^{exe})_{\text{expected}} \;=\; \begin{cases} IDOnTrue, & \text{if } Val(Condition) = 1, \\ IDOnFalse, & \text{otherwise.} \end{cases}$$

The verification (Algorithm 5, line 8) detects if $Val(N_D^{exe}(\mathcal{M})) \neq Val(N_D^{exe})_{\text{expected}}$, causing Arbitrate to output $(\phi, \mathcal{M})$, contradicting our assumption.

Case 3: Tensor Node. For tensor nodes, we have three subcases:

Root Node: The value is directly verified against the symbolic node (Algorithm 5, line 16). Any deviation causes Arbitrate to output $(\phi, \mathcal{M})$, contradicting our assumption.

Write Node: The expected value is verified (Algorithm 5, line 18) to be the same as the relevant ancestor's value. Any deviation causes Arbitrate to output $(\phi, \mathcal{M})$, contradicting our assumption.

Read Node: The expected value is verified (Algorithm 5, line 21) to match the value from the most recent write. If $\mathcal{M}$ provides an older write node or a write node not in the common prefix vector, the verification (Algorithm 2, lines 15–25) detects this, causing Arbitrate to output $(\phi, \mathcal{M})$, contradicting our assumption.

Conclusion. Since we have exhaustively covered all node types and shown that any deviation in the computed value is detected, leading to a contradiction of our assumption, the original assumption must be false. Therefore, Arbitrate must output $(\phi, \mathcal{M})$. □

5.2.2 Soundness Theorem

Lemma 5.5 (Soundness)**.** *For any valid program Φ represented by a symbolic graph $\mathcal{S}$ with Merkle root $\mathcal{R}_{sym}(\mathcal{S})$ committed to the smart contract, the following holds: Let $\mathcal{H}$ be a party that follows the Arbigraph protocol, producing a correct execution graph $\mathcal{G}_{\mathcal{S}}^{\mathcal{H}}$ with Merkle root $\mathcal{R}_{\mathcal{H}}$ and submitting the correct output ϕ, and let $\mathcal{M}$ be a computationally-bounded party that deviates from the protocol and submits an output $\phi_{\mathcal{M}}$ where $\phi_{\mathcal{M}} \neq \phi$. Then, upon completion of the Arbigraph protocol, the smart contract outputs $(\phi, \mathcal{M})$, correctly identifying the malicious party and returning the valid result.*

Proof. Assume by contradiction that upon completion of the Arbigraph protocol, the smart contract does not output $(\phi, \mathcal{M})$. We show that this leads to a contradiction for all possible cases.

Commitment and Bisection Phases: Given that $\mathcal{M}$ submits $\phi_{\mathcal{M}} \neq \phi$ while $\mathcal{H}$ submits the correct output ϕ, a dispute is triggered (Algorithm 1, line 3). The contract then requests execution graph commitments from both parties to initiate the dispute resolution process. The contract verifies each party's Merkle root consistency with their claimed output using Merkle proofs (Algorithm 1, lines 11–13). Failed verification immediately identifies that party as malicious, resulting in $(\phi, \mathcal{M})$, contradicting our assumption. Thus, parties must submit Merkle roots that are consistent with their claimed outputs. The protocol proceeds with the Bisection protocol (§4.3.4, Algorithm 1, line 15).

If $\mathcal{M}$ fails to send messages in time during bisection or during the initial commitment phase, the protocol identifies $\mathcal{M}$ as the cheater due to the timeout mechanism §3. The contract then returns $(\phi, \mathcal{M})$ (Algorithm 1, lines 16–17), contradicting our assumption. Therefore, the Bisection protocol completes successfully and returns $(\mathcal{I}, \perp)$ (guaranteed by the Bisection protocol §4.3.4), identifying the index where the parties first diverge. Importantly, the Bisection protocol guarantees that all nodes with indices less than $\mathcal{I}$ form an identical common prefix vector for both parties, but makes no guarantees about the validity or well-formedness of any nodes at or after index $\mathcal{I}$, including the disputed node itself.

Arbitration Phase: Following the successful bisection, both parties should submit proofs $Proof_{\mathcal{H}}$ and $Proof_{\mathcal{M}}$ to the Arbitrate function (Algorithm 2, line 1).

Case 1: Refusal to Provide Proof or Timeout. If $\mathcal{M}$ fails to submit $Proof_{\mathcal{M}}$ within the required timeframe Δ, the timeout mechanism identifies $\mathcal{M}$ as the cheater (§3). This results in $(\phi, \mathcal{M})$, contradicting our assumption.

Therefore, $\mathcal{M}$ submitted a proof within the required timeframe.

Case 2: Submission of Invalid or Garbage Data. A proof is *well-formed* if it has all the required fields, even if they are incorrect. If $\mathcal{M}$ submits $Proof_{\mathcal{M}}$, a proof that is not well-formed, i.e., it does not conform to the proof structure (Definition 6), the Arbitrate function detects this during validation (Algorithm 2, lines 2 and 4), contradicting our assumption. Thus $Proof_{\mathcal{M}}$ is a valid proof data structure.

Case 3: Structural Tampering with Valid Proof Format. Let us examine the case where $\mathcal{M}$ submits a well-formed proof but for a node $N_D^{exe}(\mathcal{M})$ that is structurally different from the correct node $N_D^{exe}(\mathcal{H})$ at position $\mathcal{I}$. The conditions of Lemma 5.2 are satisfied:

$\mathcal{R}_{sym}(\mathcal{S})$ is the Merkle root of the symbolic graph $\mathcal{S}$ (given in the theorem statement), $\mathcal{R}_{\mathcal{H}}$ corresponds to a correct execution of $\mathcal{S}$ (since $\mathcal{H}$ follows the protocol), $\mathcal{I}$ identifies the earliest disputed node (returned by the Bisection protocol), guaranteeing identical common prefix vectors, $\mathcal{R}_{\mathcal{M}}$ is an $(\mathcal{I}, \mathcal{G}_{\mathcal{S}}^{\mathcal{H}})$ prefix commitment provided by the computationally-bounded $\mathcal{M}$ that has access to $\mathcal{S}$;and $\mathcal{R}_{\mathcal{H}}$ (by assumption), $Proof_{\mathcal{H}}$ contains a correct proof for node $N_D^{exe}(\mathcal{H})$ (since $\mathcal{H}$ follows the protocol), and $Proof_{\mathcal{M}}$ does not contain a valid proof for any node $N_D^{exe}(\mathcal{M})$ such that $N_D^{exe}(\mathcal{H}) \equiv_s N_D^{exe}(\mathcal{M})$ (this is the case we are examining).

All conditions are satisfied. Therefore, by Lemma 5.2, the Arbitrate function outputs $(\phi, \mathcal{M})$, contradicting our assumption.

Therefore, $N_D^{exe}(\mathcal{M})$ must be structurally equivalent to $N_D^{exe}(\mathcal{H})$.

Case 4: Ancestor Tampering with Structurally Correct Node. If $\mathcal{M}$ submits a proof for a structurally correct node but with invalid ancestors, all conditions of Lemma 5.3 are satisfied:

$\mathcal{R}_{sym}(\mathcal{S})$ is the Merkle root of a valid symbolic graph $\mathcal{S}$ (given in the theorem statement), $\mathcal{R}_{\mathcal{H}}$ corresponds to a correct execution of $\mathcal{S}$ (since $\mathcal{H}$ follows the protocol), $\mathcal{I}$ identifies the earliest disputed node (returned by the Bisection protocol), guaranteeing identical common prefix vectors, $\mathcal{R}_{\mathcal{M}}$ is an $(\mathcal{I}, \mathcal{G}_{\mathcal{S}}^{\mathcal{H}})$ prefix commitment provided by the computationally-bounded $\mathcal{M}$ that has access to $\mathcal{S}$;(by assumption), $Proof_{\mathcal{H}}$ contains a correct proof for node $N_D^{exe}(\mathcal{H})$ (since $\mathcal{H}$ follows the protocol), and $Proof_{\mathcal{M}}$ contains a proof for a node $N_D^{exe}(\mathcal{M})$ where $N_D^{exe}(\mathcal{H}) \equiv_s N_D^{exe}(\mathcal{M})$ (established in Case 3), and $Proof_{\mathcal{M}}$ does not provide a valid set of ancestors that matches those in $Proof_{\mathcal{H}}$ (this is the case we are examining).

Therefore, by Lemma 5.3, the Arbitrate function outputs $(\phi(\mathcal{H}), \mathcal{M})$, contradicting our assumption. Thus, $\mathcal{M}$ must provide valid and consistent ancestors in $Proof_{\mathcal{M}}$.

Case 5: Value Tampering. If $\mathcal{M}$ submits a proof for a structurally correct node with valid ancestors but an incorrect computed value (i.e., $N_D^{exe}(\mathcal{H}) \equiv_s N_D^{exe}(\mathcal{M})$ but $Val(N_D^{exe}(\mathcal{M})) \neq Val(N_D^{exe}(\mathcal{H}))$), all conditions of Lemma 5.4 are satisfied:

$\mathcal{R}_{sym}(\mathcal{S})$ is the Merkle root of a valid symbolic graph $\mathcal{S}$ (given in the theorem statement), $\mathcal{R}_{\mathcal{H}}$ corresponds to a correct execution of $\mathcal{S}$ (since $\mathcal{H}$ follows the protocol), $\mathcal{I}$ identifies the earliest disputed node (returned by the Bisection protocol), guaranteeing identical common prefix vectors, $\mathcal{R}_{\mathcal{M}}$ is an $(\mathcal{I}, \mathcal{G}_{\mathcal{S}}^{\mathcal{H}})$ prefix commitment provided by the computationally-bounded $\mathcal{M}$ that has access to $\mathcal{S}$;(by assumption), $Proof_{\mathcal{H}}$ contains a correct proof for node $N_D^{exe}(\mathcal{H})$ (since $\mathcal{H}$ follows the protocol), $Proof_{\mathcal{M}}$ contains a proof for a node $N_D^{exe}(\mathcal{M})$ where $N_D^{exe}(\mathcal{H}) \equiv_s N_D^{exe}(\mathcal{M})$ (established in Case 3), and $Val(N_D^{exe}(\mathcal{H})) \neq Val(N_D^{exe}(\mathcal{M}))$ (this is the case we are examining). Thus, by Lemma 5.4, the Arbitrate function outputs $(\phi(\mathcal{H}), \mathcal{M})$, contradicting our assumption.

Therefore, after considering all possible cases, the contradiction assumption is wrong, and upon completion of the Arbigraph protocol, the smart contract outputs $(\phi, \mathcal{M})$, correctly identifying the malicious party and returning the valid result. □

5.3 Progress

Lemma 5.6 (Progress)**.** *For any valid program Φ represented by a symbolic graph $\mathcal{S}$ with Merkle root $\mathcal{R}_{sym}(\mathcal{S})$ committed to the smart contract, the following holds: Let $\mathcal{H}$ be a party that follows the Arbigraph protocol, producing a correct execution graph $\mathcal{G}_{\mathcal{S}}^{\mathcal{H}}$ of size n with Merkle root $\mathcal{R}_{\mathcal{H}}$, let $\mathcal{M}$ be a byzantine computationally-bounded party. Then, the Arbigraph protocol terminates within at most time $\mathcal{O}(\log(n))$.*

Proof. Let n be the total number of nodes in the final execution graph. If outputs differ, the protocol requests execution data from both parties, after which the Bisection protocol halves the search space at each iteration. It takes $\log(n)$ rounds to isolate a single node of disagreement. Each round must complete within 2 time units otherwise the smart contract identifies the stalling party as the cheater and outputs $(\phi(\mathcal{H}), \mathcal{M})$. Thus, in the worst case of a single dispute, the protocol finalizes in $\mathcal{O}(\log(n))$ time. If a malicious party stalls or fails to respond, the smart contract eventually outputs $(\phi(\mathcal{H}), \mathcal{M})$. □

5.4 Security of Arbigraph

The security of Arbigraph follows directly from the established properties of completeness (Lemma 5.1), soundness (Lemma 5.5), and progress (Lemma 5.6).

Theorem 5.7 (OED Security)**.** *The Arbigraph protocol satisfies the optimistic execution delegation security properties.*

6 Evaluation

To evaluate Arbigraph, we implement the graph generation and dispute process (§6.1). We use it to demonstrate the scalability of matrix multiplication, which is the core building block of general ML workloads (§6.2), and of a full LLM implementation (§6.3). A practical deployment would call for some additional considerations (§6.4).

6.1 Implementation

To evaluate Arbigraph, we first implemented a C++ library that allows developers to write C++ code, simply replacing primitive types with template counterparts. Operator overloading allows the library to transparently track the program without requiring developers to learn specialized languages or frameworks. To form the symbolic graph, the library traces the possible execution paths of the C++ program. Then, the program can be executed as usual, with nominal overhead, to compute its result. In case of a dispute, running the program again, the library transparently constructs the execution graph. This approach permits the use of flexible programming constructs including dynamic loops, conditional branches, and complex data structures. We further implemented the dispute process user-side code in C++ and the arbitration in a Solidity smart contract. We deployed the contract to the Avalanche [38] Fuji testnet.[3]. The cost of the bisection protocol would translate to a cost of under $0.03 for both parties per round in the Avalanche C-chain, where the number of rounds is $\log_2(n) + 1$. The cost of the dispute resolution would be under $0.30 for both parties.

6.2 Marix Parallelization

To evaluate the advantage of parallelization on ML tasks, we focus on scaling matrix operations. Beside the algebraic operation, Arbigraph requires the disputing parties to generate nodes for each operation, serialize them, and form a Merkle commitment.

[3]The contract can be found by the contract address `0x1ab3afaF2265118791B22e499f9B52E2C50d9742`

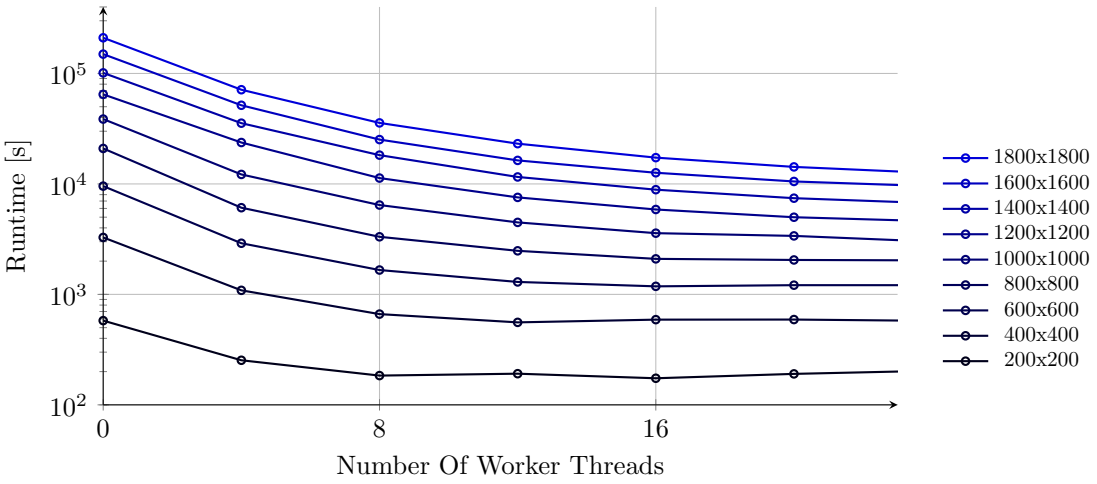

Figure 3: Runtime of MATMUL.

We use a pool of worker threads fed by a main program and set CPU affinity to ensure effective parallelization. To achieve a deterministic order of operations independent of race conditions without preallocating all Arbigraph nodes, we employ a trie-like lock-free structure [39, 40]: The main program forms an array of nodes. When enqueuing a task for worker threads, the main thread pushes a pointer to the workers' nodes into the array; done recursively, we obtain a trie. The canonical node order is an ordered traversal of the leaves.

All experiments run on a Linux machine (Ubuntu 24.04.2) with 256 logical cores @ 2.25 GHz in two AMD EPYC 7742 CPUs and with 2TB of RAM.

We demonstrate the scalability of Arbigraph. Figure 3 illustrates how execution time drops with the number of worker threads. A setup with zero worker threads is a serial execution of the code, which we compare with parallel executions to compute speedup. With more worker threads runtime drops until saturation. As expected, larger matrices take longer to multiply, but can utilize higher degrees of concurrency as they require more independent computations—they reach saturation with more worker threads.

As the matrix size grows from 100×100 to 1800×1800, the shortest execution time grows with matrix dimensions (Figure 4b); fortunately, we observe a linear improvement ($R^2 = 0.98$) in speedup (Figure 4a), ranging from 2.2× to 20.1×. The trend indicates parallelization will become even more effective with larger matrices.

6.3 LLM Implementation

We demonstrate the efficacy of Arbigraph with a complete Llama2 architecture implementation [25] and using our parallel OED matrix multiplication implementation. We use three public models [41], a 260K parameter model (matrix size 64×64), 15M (288×288), and 42M (512×512).

Like many ML workloads, LLM performance relies heavily on matrix multiplication, and therefore benefits from concurrency. Figure 5 shows that the saturation points and speedups of the LLM models match those of the matrix experiments. Here we measure performance with a single thread in the thread pool and observe the parallelization introduces a runtime overhead, which is quickly

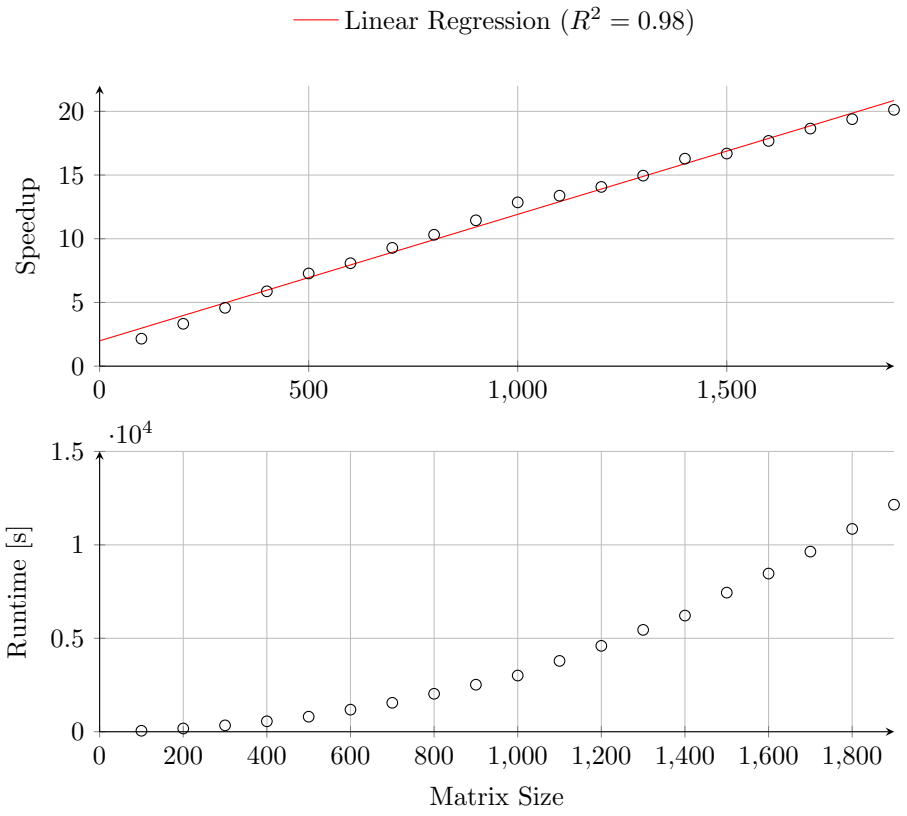

Figure 4: Optimal (a) Speedup and (b) Runtime of MATMUL.

overcome by the substantial performance gains with more threads.

6.4 Practical Considerations

Beyond this proof-of-concept implementation that demonstrates the qualitative advancement of Arbigraph, practitioners can further improve performance by taking advantage of GPU acceleration to further parallelize matrix operations. Additionally, a deployment of Arbigraph should ensure deterministically bitwise-identical execution across different hardware. The determinism can be enforced by our overloaded operations; the technique depends on the architecture. For example, CPU floating-point operation ordering can be controlled with the help of the RepOps library [27] and GPU execution can use hardware-level techniques like GPUDet [26].

7 Conclusion

Arbigraph is a novel protocol for verifiable Turing-complete execution delegation on blockchain systems. Its dual-graph architecture separates the program structure from the execution trace, enabling dynamic branching and loops. Its recency rule eliminates the logarithmic overhead of

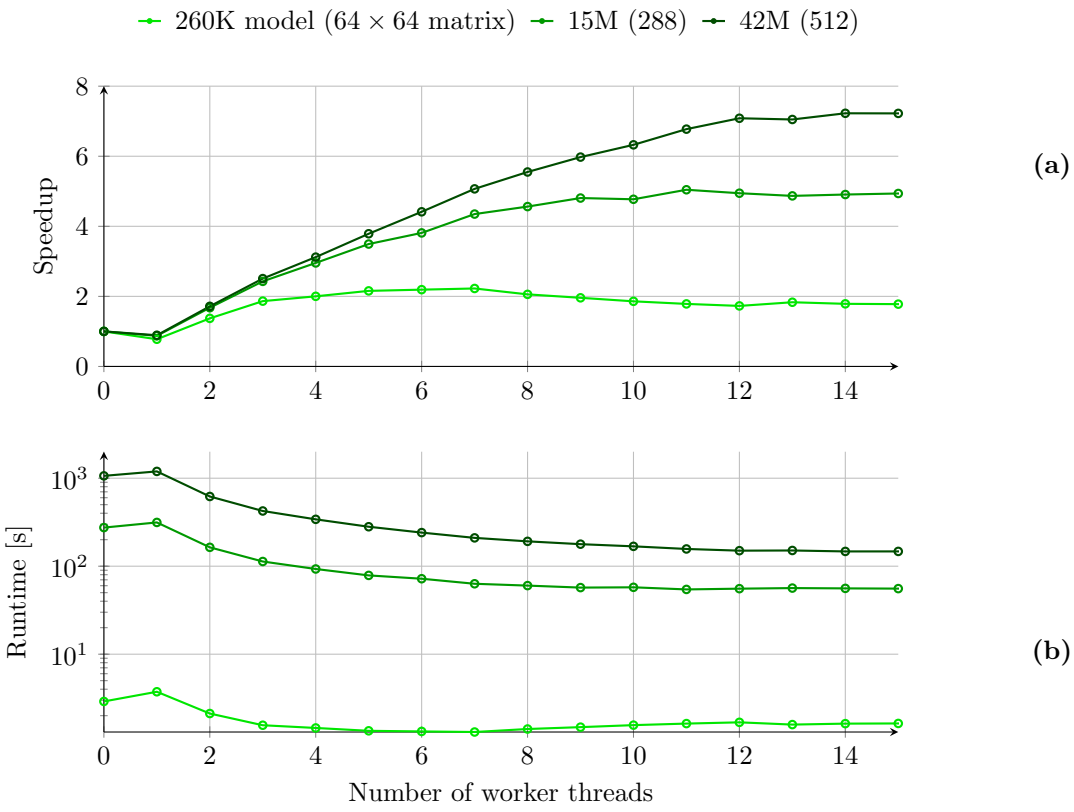

Figure 5: (a) Speedup and (b) Runtime of TRANSFORMER.

Merkle-based approaches and enables constant-time access to memory. Arbigraph enables parallelization; our implementation demonstrates $7x$ speedup for transformer inference with small public models. Moreover, focusing on matrix multiplication, which is at the core of many ML workloads, we demonstrate speedups scaling linearly with matrix dimensions—our experiments reach a $20x$ speedup, demonstrating significance of parallelization for ML computation delegation. We demonstrate with a deployment on Avalanche that the cost of the dispute resolution is under $1 for realistic workloads.

As natural language models and machine learning systems are becoming more complex and increasingly central to societal functions, Arbigraph offers a path toward trustworthy AI computation in decentralized environments. It can lead blockchain technology into domains where objective agreed-upon models can replace human oversight by providing transparent, auditable reasoning that establishes trust in automated systems.

Acknowledgments

This work was supported in part by IC3 and by Avalanche Foundation.

References

[1] J. Rose, "The meaning of corruption: Testing the coherence and adequacy of corruption definitions," *Public Integrity*, vol. 20, no. 3, pp. 220–233, 2018.

[2] J. Stiglitz *et al.*, "Transparency in government," *The right to tell: The role of mass media in economic development*, vol. 25070, pp. 27–44, 2002.

[3] Federal Trade Commission, "Federal trade commission launches inquiry into tech censorship," 2025. Accessed: 2025-04-05.

[4] FindLaw Editorial Team, "Social media censorship and the law," 2023. Accessed: 2025-04-05.

[5] G. A. Manne, "Net neutrality and the paradox of private censorship." `https://truthonthemarket.com/2024/04/25/net-neutrality-and-the-paradox-of-private-censorship/`, 2024. Accessed: 2025-04-05.

[6] Anonymous, "Algorithmic arbitrariness in content moderation." `https://arxiv.org/abs/2402.16979`, 2024. Accessed: 2025-04-05.

[7] D. Wong and L. Floridi, "Meta's oversight board: A review and critical assessment," *Minds and Machines*, vol. 33, no. 2, pp. 261–284, 2023.

[8] P. H. Ang and S. Haristya, "The governance, legitimacy and efficacy of facebook's oversight board: A model for global tech platforms?," *Emerging Media*, vol. 2, no. 2, pp. 169–180, 2024.

[9] E. L. Ungless, N. Markl, and B. Ross, "Experiences of censorship on tiktok across marginalised identities," *arXiv preprint arXiv:2407.14164*, 2024.

[10] K. Jaidka, S. Mukerjee, and Y. Lelkes, "Silenced on social media: the gatekeeping functions of shadowbans in the american twitterverse," *Journal of Communication*, vol. 73, no. 2, pp. 163–178, 2023.

[11] Z. Rahman, "Enforcing copyright on online streaming platforms: Challenges faced by rights holders in the digital era," *International Journal for Multidisciplinary Research (IJFMR)*, vol. 5, pp. 1–14, 2023.

[12] A. Hoagland, O. Yu, and M. Hornỳ, "Social determinants of health and insurance claim denials for preventive care," *JAMA Network Open*, vol. 7, no. 9, pp. e2433316–e2433316, 2024.

[13] M. Bartoletti and L. Pompianu, "An empirical analysis of smart contracts: platforms, applications, and design patterns," in *Financial Cryptography and Data Security: FC 2017 International Workshops, WAHC, BITCOIN, VOTING, WTSC, and TA, Sliema, Malta, April 7, 2017, Revised Selected Papers 21*, pp. 494–509, Springer, 2017.

[14] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[15] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat, *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[16] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, *et al.*, "Llama: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

[17] M. Labs, "zksync: Scaling ethereum with zero-knowledge." `https://zksync.io/`, 2022.

[18] Polygon, "Miden vm: A stark-based virtual machine." `https://polygon.technology/blog/introducing-miden-vm`, 2023.

[19] O. Team, "Optimistic rollups: Scalable execution with economic security." `https://optimism.io/`, 2019.

[20] H. Kalodner, S. Goldfeder, X. Chen, S. M. Weinberg, and E. W. Felten, "Arbitrum: Scalable smart contracts," in *USENIX Security Symposium*, 2018.

[21] K. Conway, C. So, X. Yu, and K. Wong, "opml: Optimistic machine learning on blockchain," *arXiv preprint arXiv:2401.17555*, 2024.

[22] Z. Zheng, P. Xie, X. Zhang, S. Chen, Y. Chen, X. Guo, G. Sun, G. Sun, and L. Zhou, "Agatha: Smart contract for dnn computation," *arXiv preprint arXiv:2105.04919*, 2021.

[23] L. Bousfield, R. Bousfield, C. Buckland, B. Burgess, J. Colvin, E. W. Felten, S. Goldfeder, D. Goldman, B. Huddleston, H. Kalonder, *et al.*, "Arbitrum nitro: A second-generation optimistic rollup," 2022.

[24] M. M. Alvarez, H. Arneson, B. Berger, L. Bousfield, C. Buckland, Y. Edelman, E. W. Felten, D. Goldman, R. Jordan, M. Kelkar, *et al.*, "Bold: Fast and cheap dispute resolution," *arXiv preprint arXiv:2404.10491*, 2024.

[25] A. Karpathy, "llama2.c: Inference llama 2 in one file of pure c." `https://github.com/karpathy/llama2.c`, 2023.

[26] H. Jooybar, W. W. Fung, M. O'Connor, J. Devietti, and T. M. Aamodt, "Gpudet: a deterministic gpu architecture," in *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, pp. 1–12, 2013.

[27] A. Arun, A. S. Arnaud, A. Titov, B. Wilcox, V. Kolobaric, M. Brinkmann, O. Ersoy, B. Fielding, and J. Bonneau, "Verde: Verification via refereed delegation for machine learning programs," *arXiv preprint arXiv:2502.19405*, 2025.

[28] L. Babai and S. Moran, "Arthur-merlin games: a randomized proof system, and a hierarchy of complexity classes," *Journal of Computer and System Sciences*, vol. 36, no. 2, pp. 254–276, 1988.

[29] A. Shamir, "Ip= pspace," *Journal of the ACM (JACM)*, vol. 39, no. 4, pp. 869–877, 1992.

[30] C. Lund, L. Fortnow, H. Karloff, and N. Nisan, "Algebraic methods for interactive proof systems," *Journal of the ACM (JACM)*, vol. 39, no. 4, pp. 859–868, 1992.

[31] S. Arora and S. Safra, "Probabilistic checking of proofs: A new characterization of np," *Journal of the ACM (JACM)*, vol. 45, no. 1, pp. 70–122, 1998.

[32] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev, "Scalable, transparent, and post-quantum secure computational integrity," *Cryptology ePrint Archive*, 2018.

[33] Polygon Team, "Polygon zkevm: A zero-knowledge ethereum virtual machine." `https://polygon.technology/polygon-zkevm`. Accessed: 2025-04-04.

[34] L. Goldberg, E. Ben-Sasson, M. Riabzev, and S. Setty, "Cairo: A turing-complete stark-friendly cpu architecture." StarkWare Technical Report, 2021.

[35] D. Kang, T. Hashimoto, I. Stoica, and Y. Sun, "Scaling up trustless dnn inference with zero-knowledge proofs," *arXiv preprint arXiv:2210.08674*, 2022.

[36] H. Sun, J. Li, and H. Zhang, "zkllm: Zero knowledge proofs for large language models," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pp. 4405–4419, 2024.

[37] M. Szydlo, "Merkle tree traversal in log space and time," in *International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 541–554, Springer, 2004.

[38] T. Rocket, M. Yin, K. Sekniqi, R. van Renesse, and E. G. Sirer, "Scalable and probabilistic leaderless bft consensus through metastability," *arXiv preprint arXiv:1906.08936*, 2019.

[39] A. Prokopec, P. Bagwell, and M. Odersky, "Cache-aware lock-free concurrent hash tries," *arXiv preprint arXiv:1709.06056*, 2017.

[40] D. Drachsler-Cohen, M. Vechev, and E. Yahav, "Practical concurrent traversals in search trees," *ACM SIGPLAN Notices*, vol. 53, no. 1, pp. 207–218, 2018.

[41] A. Karpathy, "Tinyllamas," 2024. Accessed: 27 March 2025.

[42] J. C. Shepherdson and H. E. Sturgis, "Computability of recursive functions," *Journal of the ACM (JACM)*, vol. 10, no. 2, pp. 217–255, 1963.

Appendices

A Execution Graph Construction

Definition 9 (Execution Graph Construction)**.** *The execution graph $\mathcal{G}$ is constructed through the following deterministic process:*

1. *Initialize an empty execution graph $\mathcal{G} = \emptyset$.*
2. *Initialize an execution pointer execPtr to the symbolic node with the smallest $\mathcal{O}_{\mathcal{S}}(\cdot)$ value.*
3. *While execPtr does not point to an index greater than n:*
   - *Let N^{symb} be the symbolic node pointed to by execPtr.*
   - *Create a new execution node N^{exe} with structural properties matching N^{symb}.*

- *Determine the ancestor vector to use:*
  - *Let* $S = \{N_j \in \mathcal{G} \mid ID(N_j) = ID(N^{symb})\}$ *be the set of previously executed nodes with the same ID.*
  - *If* $S = \emptyset$ *(first execution of this node), use* $\mathcal{A}^{IDs}(N^{symb})$.
  - *If* $S \neq \emptyset$ *(repeated execution), use* $\tilde{\mathcal{A}}^{IDs}(N^{symb})$.
- *Compute* $Val(N^{exe})$ *based on the node type:*
  - *If* N^{symb} *is a root node, set* $Val(N^{exe}) = Val(N^{symb})$.
  - *If* N^{symb} *is an operation node or tensor node:*
    - *Let* $A = \{N_a \in \mathcal{G} \mid ID(N_a) \in$ *selected ancestor IDs*$\}$.
    - *If any required ancestor* $N_a \in A$ *has* $Val(N_a) = \perp$ *or if any required ancestor does not exist yet (forward jump), set* $Val(N^{exe}) = \perp$.
    - *Otherwise, compute* $Val(N^{exe}) = Op(N^{symb})(\{\, Val(N_a) \mid N_a \in A\})$.
  - *If* N^{symb} *is a control node:*
    - *Let* N_c *be the ancestor node representing the condition, where* $ID(N_c) = Condition$.
    - *If* $Val(N_c) = 1$ *(true), set* $Val(N^{exe}) = IDOnTrue$.
    - *If* $Val(N_c) = 0$ *(false), set* $Val(N^{exe}) = IDOnFalse$.
    - *If* $Val(N_c) = \perp$, *set* $Val(N^{exe}) = \perp$.
- *Add* N^{exe} *to* $\mathcal{G}$ *with execution order* $\mathcal{O}_{\mathcal{S}}(N^{exe}) = |\mathcal{G}|$.
- *Update the execution pointer execPtr:*
  - *If* N^{exe} *is a control node and* $Val(N^{exe}) \neq \perp$*:*
    - *Set execPtr to the symbolic node* $N_j \in \mathcal{S}$ *where* $ID(N_j) = Val(N^{exe})$.
  - *Otherwise:*
    - *Increase execPtr by 1.*

This construction guarantees that for a given symbolic graph and input, the resulting execution graph is deterministic. The propagation of $\perp$ *values ensures that executions with invalid control flow or missing dependencies are handled consistently.*

B Bisection Protocol Specification

For efficient bisection, the Merkle trees representing execution graphs are structured to accommodate binary search, even when the number of nodes is not a power of two. This is achieved without materializing unnecessary placeholder nodes. Specifically, for a graph with n nodes, the Merkle tree is logically structured as if it contained $2^{\lceil \log_2 n \rceil}$ nodes by using virtual placeholders only at tree levels where an odd number of nodes exists. For example, with $2^{20} + 1$ nodes, virtual placeholders are added only at specific points in the tree where needed for pairing, rather than materializing all $2^{21} - 1 - (2^{20} + 1)$ extra nodes. If parties have different-sized execution graphs, the smaller graph is logically extended to match the larger one's power-of-two ceiling using this same virtual placeholder technique.

The protocol operates as follows:

1. Both parties initially submit only their program output values
2. If the outputs differ, both parties submit their Merkle root commitments $\mathcal{R}_{\mathcal{H}}$ and $\mathcal{R}_{\mathcal{M}}$ respectively, and the sizes of their execution graphs.
3. The bisection process proceeds on the virtual power-of-two-sized trees:
    - Both parties submit Merkle hashes for the corresponding left and right child nodes at each step
    - The protocol identifies which subtree contains the discrepancy
    - This binary search continues until reaching an index of a leaf node, identified as $\mathcal{I}$
4. At each step, parties must respond within timeout period Δ
5. If a party fails to respond within Δ, they are automatically designated as the cheating $\mathcal{P}$ and forfeit their collateral

C Turing-Completeness of Arbigraph

The Unlimited Register Machine (URM) model [42] consists of an infinite sequence of registers $R_0, R_1, R_2, \ldots$, each capable of storing an arbitrarily large natural number, and a small set of primitive instructions: zeroing a register ($Z(n)$), incrementing a register ($S(n)$), copying the contents of one register to another ($T(m, n)$), and conditionally jumping to a specified instruction based on register equality ($J(m, n, q)$).

Arbigraph simulates this model using its symbolic and execution graphs as follows. Registers are mapped to fixed indices in a tensor; register R_i is represented as the i-th index of a tensor node. The *root tensor node* can initializes all registers to zero. Arbigraph's tensor operations enable arbitrary-index *read* and *write* capabilities, which map exactly to the URM's memory model.

The operation nodes encode the arithmetic instructions Z, S, and T, while control nodes implement the conditional jumps J by dynamically selecting the next instruction node. The current instruction number is implicitly maintained by Arbigraph's execution algorithm, i.e., conversion from symbolic graph to execution graph.

Therefore, any URM program can be translated into an Arbigraph symbolic graph with equivalent behavior, demonstrating that Arbigraph is at least as expressive as the URM and hence is Turing-complete.