BitVM: Quasi-Turing Complete Computation on Bitcoin

Abstract—A long-standing question in the blockchain community is which class of computations are efficiently expressible in cryptocurrencies with limited scripting languages, such as Bitcoin Script. Such languages expose a reduced trusted computing base, thereby being less prone to hacks and vulnerabilities, but have long been believed to support only limited classes of payments.

In this work, we confute this long-standing belief by showing for the first time that arbitrary computations can be encoded in today's Bitcoin Script, without introducing any language modification or additional security assumptions, such as trusted hardware, trusted parties, or committees with secure majority. In particular, we present BitVM, a two-party protocol realizing a generic virtual machine by a combination of cryptographic and incentive mechanisms. We conduct a formal analysis of BitVM, characterizing its functionality, system assumptions, and security properties. We further demonstrate the practicality of our approach: in the optimistic case (i.e., in the absence of disputes between parties), our protocol requires just three on-chain transactions, whereas in the pessimistic case, the number of transactions grows logarithmically with the size of the virtual machine. This work not only solves a long-standing theoretical problem, but it also promises a strong practical impact, enabling the development of complex applications in Bitcoin.

1. Introduction

Smart contracts play a fundamental role in blockchain ecosystems by enabling decentralized, automated execution of agreements without the need for trusted intermediaries. These self-executing contracts offer many advantages, such as transparency, security, and immutability. In particular, smart contracts enable programmable money and complex decentralized applications (dApps), fostering innovation in fields like decentralized finance (DeFi), governance, supply chain management, and more.

Smart contracts are typically stored and executed on the blockchain in a low-level language. Some blockchains support a very limited scripting language, such as Bitcoin Script, whereas others feature a quasi-Turing complete[1] language, such as EVM bytecode. The former choice is motivated by a reduced trusted code base, which in principle reduces the attack surface, whereas the latter is justified by the need to support advanced computations, such as those at the core of DeFi.

A long-standing question within the community is the extent of computational capabilities in Bitcoin Script. This is not only a compelling theoretical question but one with substantial practical implications: if limited scripting languages could support complex computations, they could pave the way for advanced dApps and DeFi applications on secure platforms like Bitcoin. Until now, the prevailing view has been that limited scripting languages, such as Bitcoin Script, are suited primarily for basic functionalities like conditional payments or hashed timelock contracts, with arbitrary computations considered out of reach. Realizing such functionalities would require (i) covenant opcodes [2], which are not yet available in Bitcoin or many other cryptocurrencies, and (ii) costly encoding methods, such as counter machines [3].

Contributions. In this work, we challenge this long-standing belief by showing arbitrary (bounded) computations can be executed securely on Bitcoin in a practical manner. We introduce BitVM, a two-party protocol where a prover P claims that a function evaluates to a specific output for a given input, with a verifier V able to prove fraud and penalize P if the claim is false.

With this mechanism, one can encode any computable function on Bitcoin and execute transactions depending on the function output. For instance, imagine a party V wants to challenge a party P to solve a chess puzzle[2], say a checkmate-in-1 puzzle, and bet 10 coins (5 coins each) on whether P will solve it in a given timeframe. Note that V may or may not know the solution itself. Both parties lock up the funds in a BitVM instance that, given the initial chessboard configuration and a sequence of chess moves, verifies whether P solves the puzzle in time and distributes the funds accordingly, e.g., if P solves the puzzle in time then 10 coins go to P, otherwise 10 coins go to V.

BitVM does not require any consensus change on Bitcoin and is practical. When parties agree, the on-chain footprint is minimal, needing only three transactions to complete the protocol. In case of a dispute, BitVM ensures a constant upper bound on the number of on-chain transactions.

The key to enabling expressiveness while retaining efficiency is the design of a virtual machine supporting a Turing-complete instruction set, which is then used to execute programs off-chain and produce a verifiable execution

---

1. This term is adopted in the blockchain community to indicate Turing-complete languages that enforce termination by bounding the execution (e.g., via gas consumption in Ethereum) [1].

2. According to https://en.wikipedia.org/wiki/Chess_puzzle, in a chess puzzle "[...] the goal is to find the single best, ideally aesthetic move or a series of single best moves in a chess position [...]". We challenge the reader to solve the following checkmate-in-1 puzzles [4], [5].

trace. In case of dispute, the parties can first identify the point of disagreement in the execution trace and then verify a single step of computation on-chain leveraging the Bitcoin Script and the UTXO model, thereby ensuring that disputes are resolved within Bitcoin's scripting limitations.

The contributions of this paper are summarized below.

- We present BitVM, the first protocol to encode arbitrary computations in Bitcoin Script;
- We conduct a formal analysis of BitVM, characterizing its functionality, system assumptions, and security properties termed balance security and rational correctness;
- We conduct a complexity analysis, detailing on-chain costs and settlement time in both optimistic (no disputes) and dispute scenarios. Specifically, optimistic execution requires only three on-chain transactions, costing approximately $10,707$ satoshis[3] (6.90\$ in Sept. 2024). In disputes, the on-chain footprint scales logarithmically with the virtual machine's computational power. For a virtual machine comparable to a high-end '90s workstation[4], contract settlement may require up to 81 transactions, costing around $327Ksat$, (210\$ in Sept. 2024).

Related work. Relatively complex smart contracts can be implemented in Bitcoin by combining UTXOs and scripts, effectively splitting functionality across multiple transactions. BitML [6] provides a high-level, domain-specific language and compiler that translates programs into Bitcoin transactions, illustrating Bitcoin's potential for intricate smart contract designs [7]. These methods, however, incur substantial on-chain costs, as compiled programs often result in numerous large transactions that must ultimately be recorded on-chain.

To mitigate these costs, some approaches leverage Trusted Execution Environments (TEEs), such as FastKitten [8] that facilitates off-chain computation within a secure hardware enclave. This approach, however, introduces dependencies on the TEE, a rational adversarial model, collateral, the involvement of a TEE operator, and a limited contract duration. POSE [9] improves upon FastKitten by eliminating the need for collateral and time limitations while enhancing privacy but it still relies on TEE hardware.

Solutions based on Hashed Timelock contracts (HTLCs) aim to shift computation off-chain in a way akin to *state channels* on Ethereum [10], [11] while remaining compatible with Bitcoin's inherent contraints [12]. For example, *Discreet Log Contracts (DLCs)* [13] and *Cryptographic Oracle-Based Conditional Payments* [14] are alternative approaches that utilize (semi-)trusted oracles to assert specific events and perform payments conditioned to them. These events, typically encoded in the preimage of the hash, have to be known upfront, which restricts the class of supported functions. For instance, the chess example from Section 1 would not be expressible through HTLCs since the outcome might not be known a-priori to any of the parties.

In contrast to prior work, BitVM enables quasi-Turing complete computation on Bitcoin, similar to Ethereum, without relying on additional assumptions such as TEEs or semi-trusted oracles. BitVM thus represents the first trustless protocol allowing arbitrary, yet bounded, computation on Bitcoin, unlocking a range of potential applications. Notable examples include bridges (e.g., [15], [16], [17]), some of which are already being deployed based on the initial informal BitVM concept [18].

Due to the industry's significant interest in this concept, a follow-up work proposed an alternative approach with the main goal of designing a bridge between Bitcoin and layer-2 systems [19]: The core idea in that work is to compile programs down to potentially huge Bitcoin Script programs, split them, and commit to intermediary results on-chain, which can then be disputed. This approach allows permissionless challengers to dispute false claims of correctness but incurs high on-chain costs. In case of dispute in [19], the protocol enforces at least one transaction on-chain that fills an entire 4 MB Bitcoin block, costing approximately $1.9K\$$. In contrast, the solution we present in this paper is better suited to permissioned environments, as it is significantly more cost-effective in terms of on-chain fees compared to [19]; the estimated cost for BitVM in case of dispute is approximately 210\$.

Approach	Expressiveness	Extra Assumptions	On-chain cost
BitML [6], [7]	QT	None	$O(n)$
TEEs [8], [9]	QT	TEE	$O(n)$
Gen. Channels [12]	Bitcoin	None	$O(1)$
Oracles [13], [14]	QT	trusted oracle	$O(1)$
BitVM	QT	None	$O(\log(n))$

TABLE 1. COMPARISON OF BITCOIN-BASED SMART CONTRACT APPROACHES. n DENOTES THE UPPER BOUND ON COMPUTATIONAL STEPS, AND QT REFERS TO QUASI-TURING COMPLETENESS.

2. Model

BitVM is a protocol between two mutually distrusting parties, the prover P and the verifier V, designed to enable P to prove on the Bitcoin blockchain that the outcome of a pre-agreed computation with V was performed correctly. Concretely, for an agreed-upon Turing-complete program Π, a BitVM instance secures collateral from both parties and it enables P to enforce a transaction on-chain based on the outcome $\Pi(x)$ for a specific input x. In other words, $\Pi(x)$ dictates the payout of the funds within the BitVM instance, typically allocating them to P and V[5]. If P or V stop collaborating during protocol execution, after a designated period all the funds are allocated to the other party.

2.1. System model

We assume time advances in discrete rounds $(1, 2, \dots)$. Protocol participants run in probabilistic polynomial time

---

3. A *satoshi* is a fraction of a bitcoin, i.e., $1sat = 10^{-8}$₿.
4. Configured with 2^{32} memory cells for 32-bit integers, supporting up to 2^{32} instructions and steps.

5. Note that P and V can agree to allocate the funds to a third party or, more generally, make the funds spendable under any condition that can be expressed in Bitcoin Script.

(PPT) in the security parameter κ. We assume synchronous communication, i.e., messages sent between parties arrive at the beginning of the next round, as well as authenticated communication channels. Our protocol employs a hash function modeled as a random oracle $H : \{0,1\}^* \to \{0,1\}^\kappa$ which maps an input of arbitrary length to a fixed κ-sized output. Moreover, our protocol builds upon a distributed ledger protocol (e.g., [20], [21], [22]).

Definition 1 (Distributed Ledger Protocol). A distributed ledger protocol is an interactive Turing machine exposing the following functionality on each party.

- execute(): executes one protocol round and enables the machine to communicate with the network, invoked by the environment in every round;
- write(tx): takes as input a transaction from the environment;
- read(): outputs a finite, ordered sequence of transactions, also known as *transaction ledger* L.

We denote L_r^P as the output of invoking read() on party P at the end of round r. We restrict honest parties to only include *valid* transactions in their ledgers[6]. As we are interested in building BitVM on Bitcoin, when we present the construction, transactions are deemed (in)valid based on Bitcoin's validation rules (see Section 3.1). However, BitVM can be built on top of any distributed ledger protocol with validation rules as expressive as those of Bitcoin. We assume that our protocol participants have access to the functionality exposed by the distributed ledger protocol, either by being an active participant or by running some (light) client protocol. We are interested in distributed ledger protocols that are safe and live, as defined below (cf. [20], [21], [22]). Given two sequences A and B, we use $A \preceq B$ to mean that A is a prefix of B.

Definition 2 (Stickiness). A distributed ledger protocol is sticky if for any honest party P and any rounds $r_1 \leq r_2$, it holds that $\mathsf{L}_{r_1}^P \preceq \mathsf{L}_{r_2}^P$.

Definition 3 (Safety). A distributed ledger protocol is safe, if it is sticky and for any pair of honest parties P_1, P_2 and any pair of rounds r_1, r_2, it holds that $\mathsf{L}_{r_1}^{P_1} \preceq \mathsf{L}_{r_2}^{P_2} \vee \mathsf{L}_{r_2}^{P_2} \preceq \mathsf{L}_{r_1}^{P_1}$.

Definition 4 (Liveness). A distributed ledger protocol execution is live(u), if any transaction that is written to an honest party's ledger at round r, appears in the ledger of all honest parties by round $r + u$, denoted as $\mathsf{L}_{r+u}^\cap$.

Throughout this paper, we say "publish a transaction tx (on L)" to denote calling the function write(tx). Furthermore, after publishing a valid transaction tx, we sometimes say "wait until tx appears (on L)", to denote calling the function read() every round until $tx \in \mathsf{L}$, which happens at most after u rounds due to liveness. When presenting the BitVM construction, we sometimes refer to the ledger as blockchain even though the distributed ledger protocol could be realized

---

6. This is not strictly necessary and is done mainly for convenience. Parties could also take an outputted ledger and remove invalid transactions from it.

differently. We say something happens *on-chain* if there are one or more corresponding transactions in the ledger, and something happens *off-chain* if there are no corresponding transactions on the ledger.

There is a ledger state that is induced by a ledger L, denoted as $\mathsf{st}(\mathsf{L})$, by executing each transaction in order, starting with a genesis state. The execution of transactions is captured by a state transition function, taking a state and a transaction and outputting a new state. We denote $\mathsf{bal}_\mathsf{L}(P) \in \mathbb{R}_{\geq 0}$ as the balance of party P in the state induced by L. A party can use parts of their balance $\mathsf{in}_P \in [0, \mathsf{bal}_\mathsf{L}(P)]$ as *monetary input* for a transaction. For a given ledger L, we define the on-chain (monetary) utility of a transaction $tx \in \mathsf{L}$ for a party P as $w_\mathsf{L}(P, tx) := \mathsf{bal}_{\mathsf{L}_1}(P) - \mathsf{bal}_{\mathsf{L}_2}(P)$, where $\mathsf{L}_1 \prec \mathsf{L}$ is the ledger up to (not including) tx and $\mathsf{L}_2 := \mathsf{L}_1 \| tx$. Usually, it is obvious which ledger we refer to, so we omit the subscript. In addition to balances of parties, a ledger state $\mathsf{st}(\mathsf{L})$ can include a string $s \in \{0,1\}^*$, denoted as $s \in \mathsf{st}(\mathsf{L})$, if there exists a transaction $tx \in \mathsf{L}$, such that tx contains the string s.

2.2. Threat model

We analyze BitVM in the presence of a PPT adversary that may corrupt any protocol party $\{P, V\}$ during the execution of the protocol. The adversary can corrupt parties, causing them to behave either as *Byzantine* or as *rational* actors. Byzantine parties can deviate arbitrarily from the honest protocol execution. Contrarily, rational parties deviate from the honest protocol execution only when such action increases their monetary utility.

The protocol gives different guarantees based on the type of corruption. On a high level, we want to show that (i) honest protocol participants are guaranteed their rightful balance even if the other party is Byzantine, (ii) rational parties follow the honest protocol execution, and (iii) if both parties behave rationally, the protocol follows an optimistic execution (which is efficient). We formally define these properties in Section 2.3.

2.3. Protocol goals

The core objectives of BitVM are termed *balance security* and *rational correctness*. Informally, balance security ensures an honest party will not lose their funds against Byzantine counterparties, whereas rational correctness guarantees that rational parties will follow the protocol. To formally define balance security we argue in terms of utility, i.e., the utility of the on-chain state of an honest party after the settlement of a BitVM instance will be at least equal to its utility of the correct final state, regardless of the actions of its counterparty. Rational correctness implies that if both parties are rational, they will commit on-chain the correct final state of the BitVM instance. These properties are standard in the literature: for instance, an honest user of a Lightning channel [23] can always dispute a malicious commitment and claim the channel funds, while rational players will always commit to the last agreed-upon state [24].

We formalize these objectives on a generic primitive, which we call *on-chain state verification* protocol and is defined as follows.

Definition 5 (On-chain State Verification Protocol). An on-chain state verification protocol, parameterized over a distributed ledger protocol that outputs a ledger L, is a two-party protocol that exposes the two following functionalities:

- setup($\mathsf{in}_P, \mathsf{in}_V, \Pi, f$): takes as input monetary inputs $\mathsf{in}_P \in [0, \mathsf{bal}_L(P)]$ and $\mathsf{in}_V \in [0, \mathsf{bal}_L(V)]$ of parties P and V, a computable function (or program) $\Pi : \mathcal{S} \to \mathcal{O}$ that maps a set of states $\mathcal{S}$ to a set of outcomes $\mathcal{O}$ and an outcome mapping function $f : \mathcal{O} \to \mathbb{R}^2_{\geq 0}$, that maps the set of outcomes $\mathcal{O}$ to pairs of utilities (v_P, v_V) where $v_P + v_V \leq \mathsf{in}_P + \mathsf{in}_V$ and returns an instance $\mathcal{I}$.
- execute($\mathcal{I}, x$): takes as input an instance $\mathcal{I}$ returned by the setup function and a function input $x \in \mathcal{S}$ (for function Π).

Consider an execution of this primitive for given inputs $\mathsf{in}_P, \mathsf{in}_V, \Pi, f$, where $\mathcal{I} \leftarrow$ setup($\mathsf{in}_P, \mathsf{in}_V, \Pi, f$), and then execute($\mathcal{I}, x$) are called, and finish in round r. Let $\mathcal{T}$ be the set of transactions that are included in $\mathsf{L}^{\cap}_{r+u}$ as a result of this execution. Moreover, we denote the utility of party $A \in \{P, V\}$ in $f(\Pi(x))$ by $f_A(\Pi(x))$.

Balance Security. An execution achieves balance security, if it holds that $\sum_{tx \in \mathcal{T}}(w(tx, A)) \geq v_A$ where $v_A = f_A(\Pi(x))$, for any honest $A \in \{P, V\}$.

Rational Correctness. An execution achieves rational correctness, if P and V are rational and $\sum_{tx \in \mathcal{T}}(w(tx, A)) = v_A$ where $v_A = f_A(\Pi(x))$, for any $A \in \{P, V\}$ and $\Pi(x) \in \mathsf{st}(\mathsf{L}^{\cap}_{r+u})$.

An on-chain state verification protocol achieves balance security and rational correctness, respectively, if for any $\mathsf{in}_P, \mathsf{in}_V, \Pi, f$ the probability that the corresponding execution does not achieve balance security and rational correctness, respectively, is negligible in κ.

3. Preliminaries

In this section, we present the necessary background concerning Bitcoin Script and some key primitives our construction builds upon.

Notation. Given a sequence $A := (a_1, \ldots, a_n)$, $A[i]$ represents its i-th element. We use $A[i : j]$ to denote the subsequence $(a_i, \ldots, a_j)$. We use $|A|$ to denote the length of a sequence, e.g., $|(a_1, \ldots, a_n)| = n$. For a string $s \in \{0, 1\}^*$, we use $|s|_{bit}$ to denote its bit length.

3.1. Transactions in the UTXO model

A user U on a ledger L is identified by the secret-public key pair $(\mathsf{pk}_\mathsf{U}, \mathsf{sk}_\mathsf{U})$; by $\sigma_\mathsf{U}(m)$ we denote the digital signature of U over the message $m \in \{0, 1\}^*$.

In the *unspent transaction output* (UTXO) model, a transaction Tx maps a (non-empty) list of existing, unspent, transaction outputs to a (non-empty) list of new transaction

outputs. A transaction output is defined as an attribute tuple out $:= (a\ddot{\mathsf{B}}, \mathsf{lockScript})$, where $\mathsf{out}.a \in \mathbb{R}_{\geq 0}$ is the amount of coins (expressed in $\ddot{\mathsf{B}}$) held by the output out and $\mathsf{out.lockScript}$ is the condition that needs to be fulfilled to spend it and transfer the coins to a new output, which we also call UTXO. We distinguish the already existing transaction outputs (input of a transaction Tx) from the newly created outputs calling them $\mathsf{Tx.inputs}$ and $\mathsf{Tx.outputs}$, respectively. A transaction input in is defined as $\mathsf{in} := (\mathsf{PrevTx}, \mathsf{outIndex}, \mathsf{lockScript})$, where the output being spent is uniquely identified by specifying the transaction PrevTx and an output index $\mathsf{outIndex}$. To improve readability, we also give the locking script $\mathsf{lockScript}$ that is being fulfilled.

We formally define a transaction as a tuple $\mathsf{Tx} := (\mathsf{inputs}, \mathsf{witnesses}, \mathsf{outputs})$ where $\mathsf{Tx.inputs} := [\mathsf{in}_1, \ldots, \mathsf{in}_n]$ are the transaction inputs, $\mathsf{Tx.outputs} := [\mathsf{out}_1, \ldots, \mathsf{out}_m]$ are the transaction outputs and $\mathsf{Tx.witnesses} := [\mathsf{w}_1, \ldots, \mathsf{w}_n]$ represents the witness data, i.e., the list of the tuples that fulfill the spending conditions of the inputs, one witness for each input. The locking script of an output is expressed in the scripting language of the ledger. To transfer the coins held in a UTXO, its locking script is executed with a witness as script input and must return True; if successful, the condition is considered fulfilled. If the script execution returns False, the condition is not fulfilled and the UTXO is not spendable[7].

A transaction is valid only if every UTXO in input is unspent, the witnesses fulfill the conditions of the corresponding locking scripts, and the sum of the coins held in the inputs is equal to or greater than the sum of the coins held in the outputs.

Transaction spending conditions. Bitcoin has a stack-based scripting language. Below, we describe the subset of Bitcoin spending conditions that we use in this paper.

- **Signature locks.** The spending condition $\mathsf{CheckSig}_{\mathsf{pk}_\mathsf{U}}(m)$ is fulfilled if the signature $\sigma_\mathsf{U}(m)$ is part of the witness.
- **Multisignature locks.** To fulfill this spending condition, k out of n signatures are required. In particular, for two users A and B, a spending condition that represents a 2-of-2 multi-signature of a message m between them is denoted as $\mathsf{CheckMSig}_{\mathsf{pk}_{A,B}}(m)$ and is fulfilled by giving the signature $\sigma_{A,B}(m)$ as part of the witness of the spending transaction.
- **Relative timelocks** make a transaction output spendable only after a specified time Δ has elapsed since the transaction was included on-chain. We denote the relative timelock spending condition as $\mathsf{TL}(\Delta)$.
- **Taproot trees** [26], also known as Taptrees, enable a UTXO to be spent by satisfying one of several possible spending conditions. These conditions, referred to as

---

7. In this work, we separate the locking script from the witness for readability. However, note that in practice, the protocol is implemented using SegWit [25] transactions, where the locking script is included in the witness.

Tapleaves, form the leaves of a Merkle tree. To spend a UTXO locked by a Taptree locking script, the user must provide a witness for one of the Tapleaves along with proof of inclusion of that leaf in the Taptree.

We denote the Tapleaves of a Taptree locking script as $\langle \mathsf{leaf}_1, \ldots, \mathsf{leaf}_r \rangle$. When a user fulfills the script leaf_i to unlock the j-th output of the transaction $\mathtt{Tx}$, the corresponding input is represented as $(\mathtt{Tx}, j, \langle \mathsf{leaf}_i \rangle)$. Whenever a user spends a UTXO via a Tapleaf of a Taptree, we assume that they have provided a valid Merkle proof of inclusion for that Tapleaf.

- **Other conditions.** We denote with True (False) a condition that is always fulfilled (can never be fulfilled).

We use $*$ to denote a generic transaction input, witness, or output that is not directly relevant to our protocol, provided it remains valid under Bitcoin consensus rules.

Combining spending conditions. When presenting spending conditions with complex logic, we explicitly provide their pseudocode. We use the conditions described in this section as building blocks, combining them with standard Bitcoin Script constructions using logical operators $\wedge$ (and) and $\vee$ (or). Furthermore, for convenience, inside long scripts we append the keyword Verify to sub-spending conditions that return either True or False with the following meaning: if the sub-spending condition returns True, pop True from the stack and continue to execute the rest of the script, if it returns False, mark the transaction as invalid (and thus fail to unlock the long script). This is meant to mimic how the Bitcoin OP_VERIFY opcode works.

3.2. Lamport digital signature scheme

Let $h : X \to Y$ be a one-way function, where $X := \{0,1\}^*$ and $Y := \{0,1\}^\lambda$, for a given security parameter λ. Let $m \in \{0,1\}^\ell$ be a ℓ-bit message, with $\ell \in \mathbb{N}_{>0}$. A *Lamport digital signature scheme* [27] Lamp consists of a triple of algorithms (KeyGen, Sig, Vrfy), where:

- $(pk_\mathcal{M}, sk_\mathcal{M}) \leftarrow \mathsf{Lamp.KeyGen}(\ell)$ (cf. Algorithm 1), is a Probabilistic Polynomial Time (PPT) algorithm that takes as input a positive integer ℓ and returns a key pair, consisting of a secret key $sk_\mathcal{M}$ and a public key $pk_\mathcal{M}$ which can be used for one-time signing an ℓ-bit message. We use $\mathcal{M} = \{0,1\}^\ell$ as an alias for the ℓ-bit message space.
- $c_m \leftarrow \mathsf{Lamp.Sig}_{sk_\mathcal{M}}(m)$ (cf. Algorithm 2), is a Deterministic Polynomial Time (DPT) algorithm parameterized by a secret key $sk_\mathcal{M}$, that takes as input a message $m \in \mathcal{M}$ and outputs the signature c_m, which we also call (Lamport) commitment.
- $\{\mathsf{True}, \mathsf{False}\} \leftarrow \mathsf{Lamp.Vrfy}_{pk_\mathcal{M}}(m, c_m)$ (cf. Algorithm 3), is a DPT algorithm parameterized by a public key $pk_\mathcal{M}$ that takes as input a message m, a signature c_m, and outputs True iff c_m is a valid signature for m generated by the secret key $sk_\mathcal{M}$, corresponding to $pk_\mathcal{M}$, i.e., $(pk_\mathcal{M}, sk_\mathcal{M})$ is a key pair generated by Lamp.KeyGen.

Lamport signatures are secure one-time signatures. Given a message space $\mathcal{M}$, it is possible to sign any message

---

Algorithm 1 The key generation algorithm Lamp.KeyGen for a ℓ-bit messages space $\mathcal{M}$. In the following algorithms, we use matrix notation, i.e., for a given two-dimensional matrix a, $a[i,j]$ refers to the element at row i and column j of it.

1: **function** Lamp.KeyGen(ℓ)
2:     Let $sk_\mathcal{M} \leftarrow \begin{pmatrix} x[0,0], \ldots, x[0,\ell-1] \\ x[1,0], \ldots, x[1,\ell-1] \end{pmatrix}$, where every element $x[i,j]$ is sampled uniformly at random from the set X;
3:     **for** $i = 0, 1$ and $j = 0, \ldots, \ell-1$ **do**
4:       $y[i,j] \leftarrow h(x[i,j])$;
5:     Let $pk_\mathcal{M} \leftarrow \begin{pmatrix} y[0,0], \ldots, y[0,\ell-1] \\ y[1,0], \ldots, y[1,\ell-1] \end{pmatrix}$;
6:     **return** $(sk_\mathcal{M}, pk_\mathcal{M})$.

Algorithm 2 The Lamport signature algorithm Lamp.Sig, parameterized over a secret key $sk_\mathcal{M}$ for a ℓ-bit sized message space $\mathcal{M}$.

1: **function** $\mathsf{LampSig}_{sk_\mathcal{M}}(m)$
2:     **for** $i = 0, \ldots, \ell-1$ **do**
3:       Let $c_m[i] \leftarrow sk_\mathcal{M}[m[i], i]$;
4:     **return** c_m.

$m \in \mathcal{M}$ by using the secret key $sk_\mathcal{M}$ of the key pair $(sk_\mathcal{M}, pk_\mathcal{M})$, i.e., the key pair associated to $\mathcal{M}$. When the message m is signed and c_m is created, the key pair becomes bound to m. No polynomially bounded adversary is able to forge a signature for a different message $m' \neq m$ with non-negligible probability. However, if the signer uses the same secret key $sk_\mathcal{M}$ to sign another different ℓ-bit messages $m'' \neq m$, they can be held accountable. We call this action *equivocation* and we show how to detect it in Algorithm 4.

Notice that signing the ℓ-bit message m with the secret key $sk_\mathcal{M}$ consists in revealing for every bit $i = 0, \ldots, \ell-1$ of m one of the two preimages that compose the $i - th$ column of secret key $sk_\mathcal{M}$, namely, revealing $x[0,i]$ to claim that $m[i] = 0$, or revealing $x[1,i]$ to claim that $m[i] = 1$. When the signer reveals both $x[0,i], x[1,i]$ for any bit i, they are equivocating.

For a formal discussion about one-time security and a proof that Lamport signatures are one-time secure (assuming the existence of one-way functions), see, e.g., [28]. One-time security is crucial for the correctness of BitVM as it enables the signer of a message to make a non-repudiable commitment to that message. Lamport signatures are implementable using Bitcoin Script, as demostrated in [29].

In the following, we are interested in Lamport signatures as a mechanism to enable a party to *commit* to (single or multiple bits) messages. Thus, we will refer to Algorithm 2

Algorithm 3 Lamport verification algorithm Lamp.Vrfy, parameterized over a public key $pk_\mathcal{M}$ for a ℓ-bit message space $\mathcal{M}$.

1: **function** $\mathsf{Lamp.Vrfy}_{pk_\mathcal{M}}(m, c_m)$
2:     **for** $i = 0, \ldots, \ell-1$ **do**
3:       **if** $h(c_m[i]) \neq pk_\mathcal{M}[m[i], i]$ **then**
4:         **return** False;
5:     **return** True.

Algorithm 4 The CheckEquivocation algorithm for a bit $b \in \mathcal{B} = \{0,1\}$. The input is the corresponding public key $pk_\mathcal{B}$ and two preimages $x', x'' \in X$.

1: **function** CheckEquivocation($pk_\mathcal{B}$, x', x'')
2:     **if** $\left(h(x') = pk_\mathcal{B}[0,0] \text{ and } h(x'') = pk_\mathcal{B}[1,0] \right)$ **then**
3:         **return** True;
4:         ▷ *The committer is trying to commit to both* 0 *and* 1 *for the bit* b. ◁
5:     **else**
6:         **return** False.

---

as Comm instead of Lamp.Sig and to Algorithm 3 as CheckComm instead of Lamp.Vrfy.

3.3. Stateful Bitcoin scripting

Although the Bitcoin scripting language is stateless, a clever use of one-time digital signature schemes, such as Lamport signatures, enables state preservation across different Bitcoin transactions.

Consider the following example: Let a user U hold a Lamport key pair $(sk_\mathcal{M}, pk_\mathcal{M})$ associated with $\mathcal{M}$, the set of all ℓ-bit messages. We can think of $\mathcal{M}$ as a variable that can hold any ℓ-bit string. U can assign a value m to $\mathcal{M}$ by creating the commitment $c_m \leftarrow \mathsf{Comm}_{sk_\mathcal{M}}(m)$.

By hard-coding CheckComm$_{pk_\mathcal{M}}$ for a public key $pk_\mathcal{M}$ in the locking script of multiple outputs, this variable assignment can not only be verified but also transferred from one output to another, effectively establishing a global state in Bitcoin. This is accomplished by reading m and c_m from the unlocking script of one output and passing them to another output through its witness. For example, consider two different transactions $\mathtt{Tx_1} := (*, *, [\mathsf{out}_1, *])$ and $\mathtt{Tx_2} := (*, *, [\mathsf{out}'_1, *])$, where the outputs are defined as $\mathsf{out}_1 := (a\,\dot{\mathsf{B}}, \mathsf{CheckComm}_{pk_\mathcal{M}})$ and $\mathsf{out}'_1 := (b\,\dot{\mathsf{B}}, \mathsf{CheckComm}_{pk_\mathcal{M}})$. To unlock both out_1 and out'_1, a Lamport commitment c_m must be provided. Since the same Lamport public key appears in both scripts, every party in the network knows that when U unlocks these scripts, U is assigning a value to the same variable $\mathcal{M}$. Following from *one-time security*, no user other than U can assign a different value to $\mathcal{M}$ without knowing $sk_\mathcal{M}$. Moreover, U cannot assign two different values $m_1 \neq m_2$ to $\mathcal{M}$ without equivocating, which is detectable and can be punished on-chain.

4. BitVM **Virtual Machine**

In the BitVM protocol, both parties employ a *Virtual Machine* (VM) to run off-chain any deterministic program Π. Although the underlying concept closely resembles an *abstract machine*, we choose to retain the term "VM" to stay consistent with the original naming of the construction. In this section, we describe the components of the VM and demonstrate how to initialize them for practical deployment of the protocol.

VM components. At a high level, the virtual machine (VM) executes programs composed of instructions written in a VM-compatible language. While the program is running, the VM continuously performs an instruction cycle, or *state transition function*. In each cycle, the VM fetches the instruction indicated by the program counter, loads the values stored at specific memory addresses referenced by the instruction, executes the operation defined by the instruction on those values, stores the result at the designated memory address, and updates the program counter accordingly (cf. Definition 7).

This process repeats until the program terminates or reaches a predefined execution limit. Throughout its execution, the VM produces an execution trace, recording (i) the current program counter value and (ii) a commitment to the state of memory at each step. The BitVM protocol leverages this execution trace for dispute resolution, as described in Section 6.3 and Section A.1.

Formally, let a *VM address* be an integer $addr \in \mathcal{A} := \{0, 1, \ldots, \mathsf{MemLen} - 1\}$ where $\mathsf{MemLen} \in \mathbb{N}_{>0}$ represents the memory length. We define the VM *memory* as the sequence $M \in \mathcal{M} := \{0, 1, \ldots, n\}^{\mathsf{MemLen}}$, where $n \in \mathbb{N}_{>0}$ specifies the range of values stored at any memory address. The *VM program counter*, denoted pc, is an element of the set $\mathcal{PC} := \{0, 1, \ldots, \ell-1\} \cup \{\bot\}$, where $\ell \in \{1, 2, \ldots, n\}$ is the maximum length of the program, and $\bot$ indicates termination. Let $\mathcal{OP} := \big\{ f_{\mathsf{OP}} : \mathcal{PC} \times \{0, \ldots, n\} \times \{0, \ldots, n\} \to \mathcal{PC} \times \{0, \ldots, n\} \cup \{\bot\} \big\}$ be a set of CPU instructions that the VM can execute[8]. The function f_{OP} takes as input a triple (pc, val_A, val_B) and outputs a pair (pc, val_C) or $\bot$. For any CPU instruction $f_{\mathsf{OP}} \in \mathcal{OP}$, we require that f_{OP} is executable in Bitcoin Script. A *VM program* is an ordered sequence of ℓ elements, denoted $\Pi \in \mathcal{I}^\ell$, where $\mathcal{I} := \big\{ (f_{\mathsf{OP}}, addr_A, addr_B, addr_C) \mid addr_A, addr_B, addr_C \in \mathcal{A}, f_{\mathsf{OP}} \in \mathcal{OP} \big\}$. We can now define the following.

Definition 6 (VM State). A *VM state*, or simply, *state*, is a triple $S := (M, pc, \Pi)$, where M is the VM memory, pc is the VM program counter, and Π is a VM program.

Definition 7 (State Transition Function). Let $\mathcal{S} := \mathcal{M} \times \mathcal{PC} \times \mathcal{I}^\ell$ be the set of all VM states. We define the *state transition function* $f_{ST} : \mathcal{S} \to \mathcal{S}$ with f_{ST} taking as argument the state (M_i, pc_i, Π) and giving as output the state (M_{i+1}, pc_{i+1}, Π) as specified in Algorithm 5.

Algorithm 5 State Transition Function f_{ST}.

1: **function** $f_{ST}(M, pc, \Pi)$
2:     $M' \leftarrow M$;
3:     $(f_{\mathsf{OP}}, addr_A, addr_B, addr_C) \leftarrow \Pi[pc]$;
4:     $val_A \leftarrow M[addr_A]$;
5:     $val_B \leftarrow M[addr_B]$;
6:     $(pc', val_C) \leftarrow f_{\mathsf{OP}}(val_A, val_B, pc)$;
7:     **if** $val_C \neq \bot$ **then**
8:         $M'[addr_C] \leftarrow val_C$;
9:     **return** (M', pc', Π).

---

8. Even though $\mathcal{OP}$ can be arbitrary, we are interested in a Turing-complete instruction set. In particular, we later use ADD, BEQ, and JMP, cf. Algorithm 7 – a well-known Turing-complete instruction set [30].
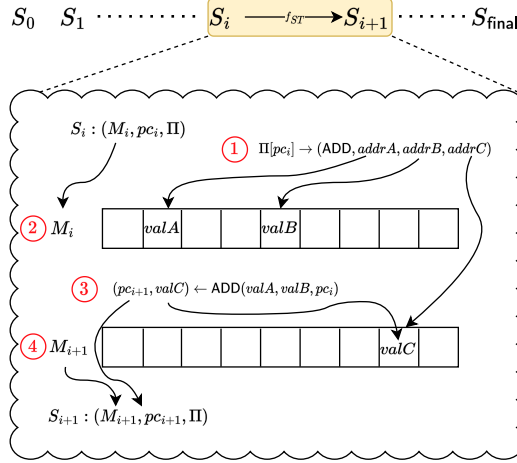
Figure 1. Overview of the state transition function execution f_{ST}. Given a state S_i: (1) instruction $\Pi[pc_i]$ is fetched, (2) values $valA, valB$ are taken from memory at their respective addresses, (3) the instruction is executed, and (4) part of the result (i.e., $valC$) is stored in the memory. The state transition function outputs the new state S_{i+1}.

Given a program Π and a memory configuration M, we assume that the entry point of the program, namely the first instruction that a program executes, is always $\Pi[0]$. Thus, we define as *initial* state the tuple $S_0 := (M, 0, \Pi)$. We use the shorthand notation $f_{ST}^i(S)$ when we apply the state transition function f_{ST} to a state S exactly i times, $f_{ST}^i(S) := f_{ST}(f_{ST}(\ldots(f_{ST}(S))))$. We say that a state $S_i := (M_i, pc_i, \Pi)$ at step i is *correct* with respect to an initial state S_0 iff $S_i = f_{ST}^i(S_0)$. We avoid the subscripts (and simply refer to the state S_i as (M, pc, Π)) when it is clear from the context which state we are referring to.

Finally, after a number of execution steps equal to final (final is decided when a VM instance is created), the program terminates. We denote the final state, or outcome, as $\Pi(S_0) := f_{ST}^{\mathsf{final}}(S_0)$. Fig. 1 provides a visual representation of the execution of the state transition function f_{ST}.

We define a VM instance as a tuple

$$\Gamma := \langle \Pi, \mathsf{MemLen}, n, \mathsf{final} \rangle.$$

We write Γ^A to refer to the VM instance executed by party A. We write S_i^A to denote a VM state S_i that A claims to have produced during the execution of A's VM instance Γ^A. We say that two parties A and B agree on the state S_i if $S_i^A = S_i^B$, and disagree on S_i otherwise.

Definition 8 (Execution Trace Element). Let $(M_i, pc_i, \Pi) := f_{ST}^i(S_0)$, and let MR_i be the root of the Merkle tree with the entries of M_i as its leaves. The i-th VM *execution trace element*, or simply, i-th *trace element* is the pair $E_i := (MR_i, pc_i)$, for $i \in \{0, \ldots, \mathsf{final}\}$.

We write E_i^A to denote a VM execution trace element E_i that A claims to have produced during the execution of A's VM instance Γ^A. The VM *execution trace* is defined as a sequence of consecutive trace elements $ExecTrace :=$ $(E_0, \ldots, E_{\mathsf{final}})$. We write $ExecTrace^A$ as a short-hand for $(E_0^A, \ldots, E_{\mathsf{final}}^A)$.

We describe how the VM behaves in Algorithm 6: starting from initial state S_0, it applies the state transition function f_{ST} to the state and records the related trace elements until the program Π ends, namely, once pc is set to be $\bot$. The VM algorithm is parameterized by final, a parameter that represents the maximum number of state transitions that the VM is allowed to perform. The VM algorithm returns as output the VM execution trace $ExecTrace$, along with the resulting memory M after the program execution.

Algorithm 6 The VM algorithm. S_0 is the initial VM state.

1: **function** $\mathsf{VM}_{\mathsf{final}}(S_0)$
2:     $stepCount \leftarrow 0$;
3:     **while** $stepCount < \mathsf{final}$ **do**
4:         $E_{stepCount} \leftarrow (MR, pc)$;
5:         $(M, pc, \Pi) \leftarrow f_{ST}(M, pc, \Pi)$;
6:         increment $stepCount$ by 1;
7:     $E_{stepCount} \leftarrow (MR, pc)$;
8:     $ExecTrace \leftarrow (E_0, \ldots, E_{\mathsf{final}})$;
9:     **return** $(ExecTrace, M)$.

A practical VM instance. For better readability and to provide a protocol instance that can be deployed in practice, in the rest of the paper, we will consider a VM instance $\Gamma := \langle \Pi, \mathsf{MemLen}, n, \mathsf{final} \rangle$ with the following initialization: We set the length of the memory as $\mathsf{MemLen} = 2^{32}$ and the greatest integer that can be stored in any entry of the memory as $n = 2^{32}$.

Furthermore, we assume that the input program Π has $\ell \leq 2^{32}$ number of instructions[9] and we set $\mathsf{final} = 2^{32}$.

As for the set $\mathcal{OP}$ of instructions that the VM can execute, our VM instance employs the following: $\mathcal{OP} := \{\mathsf{ADD}, \mathsf{BEQ}, \mathsf{JMP}\}$. This is a minimal set of computer instructions known to be Turing complete [30]. We underscore that the BitVM protocol can function with any Turing-complete instruction set, provided that each instruction within the set is implementable in Bitcoin script. In Algorithm 7, we give an implementation of ADD, BEQ and JMP that can be easily translated in Bitcoin script.

5. The BitVM protocol

The BitVM protocol enhanced the expressiveness of Bitcoin, allowing the encoding of spending conditions based on the outcome of quasi-Turing complete programs.

Protocol overview. The protocol proceeds in four phases.

1) In the *setup* phase, P and V agree on the program Π they wish to execute. For example, as described in Section 1, this program could verify the validity of a sequence of chess moves and check whether or not the chess puzzle (encoded in the program itself) has been solved. P and V also agree on the outcome mapping function f, create

---

9. In the BitVM protocol, we build a Taproot tree where every program instruction is a Tapleaf script. We chose such ℓ since $2^{32} \ll 2^{128}$, the maximum number of leaf scripts in the current specification of Bitcoin [26].

Algorithm 7 The Algorithms ADD, BEQ, and JMP, each taking as input the tuple (pc, val_A, val_B), and returning a pair (pc, val_C).

```
 1: function ADD(pc, val_A, val_B)
 2:     if pc = ⊥ then return (⊥, ⊥);
 3:     return (pc + 1, val_A + val_B).

 4: function BEQ(pc, val_A, val_B)
 5:     if pc = ⊥ then return (⊥, ⊥);
 6:     if val_A = val_B then
 7:         return (pc + 1, ⊥).
 8:     else
 9:         return (pc + 2, ⊥).

10: function JMP(pc, val_A, val_B)
11:     if pc = ⊥ then return (⊥, ⊥);
12:     return (val_A, ⊥).
```

and pre-sign all necessary transactions, and post the initial transaction that locks their coins on-chain, which we call `Setup`.

2) In the *VM execute* phase, performed off-chain, P and V generate the input S_0 for Π (e.g., the chess moves starting from a given chessboard state) and compute $\Pi(S_0)$.

3) In the *commit* phase, P posts the `CommitComputation` transaction on-chain, i.e., a transaction committing to the input S_0 and the result $\Pi(S_0)$ using Lamport signatures. Upon seeing this, V can either accept the claim as correct and wait for P to settle according to $f(\Pi(S_0))$ (via publishing a `Close` transaction), or, if the committed result does not match $\Pi(S_0)$, initiate a dispute.

4) In the event of a dispute, they enter the *resolve dispute* phase. This phase is the main technical challenge of the BitVM protocol, as it requires an on-chain mechanism to verify whether P's claimed result $\Pi(S_0)$ is correct, while remaining within Bitcoin's scripting limitations.

5.1. Resolving Disputes

The core challenge of BitVM is to enable the on-chain verification of the result of computation that is normally not expressible in Bitcoin Script. To address this, a novel approach is necessary to be able to verify the correct result, $\Pi(S_0)$, when executing Π on input S_0. Our solution leverages our VM (see Section 4) and the resulting execution trace $ExecTrace := (E_0, \ldots, E_{\text{final}})$ produced when computing $\Pi(S_0)$. Notably, each successive element in this execution trace results from applying a single VM instruction to the preceding element. We demonstrate that verifying each individual VM instruction can indeed be accomplished within Bitcoin Script, a process we detail later in this section. With this in place, the only remaining task is to identify two consecutive trace elements, where P and V agree on the former but disagree on the latter. Given that they both agree on S_0, any disagreement over the result $\Pi(S_0)$ ensures that such a pair of consecutive trace elements exists. We identify this pair via an on-chain bisection.

We present both of these components in this section, focusing on describing *what* the parties do by publishing

certain transactions, not *how* exactly these are implemented. The reader can assume that all necessary transaction logic is feasible within Bitcoin, utilizing Lamport signatures and other Bitcoin Script features. We defer the concrete implementation of these Bitcoin transactions and the full protocol specification to Section 6. Whenever we mention that one party must respond with a transaction, this is enforced by a timelock mechanism, allowing the other party to claim all funds in the event of prolonged inactivity by the first party.

In Fig. 2, we illustrate a protocol execution in case of disagreement.

Identify Disagreement. Recall that the total length of the execution trace is final. V initiates the bisection game by publishing a `Kickoff` transaction, forcing P to respond with a `TraceResponse`$_1$ transaction that reveals the middle trace element, $E_{n_{31}}$, where $n_{31} = 2^{31} = \text{final}/2$.

The loop then begins, where V forces P into progressively smaller sequences of the trace by publishing a `TraceChallenge`$_1$ transaction, Lamport committing to a value b_{31} in its witness.

- If V agrees with $E_{n_{31}}$, V commits to $b_{31} = 1$, indicating that P should reveal a trace element in the right half of the sequence.
- If V disagrees with $E_{n_{31}}$, V commits to $b_{31} = 0$, indicating that P should reveal a trace element in the left half of the sequence.
- Based on b_{31}, P responds by publishing a `TraceResponse`$_2$ transaction, revealing the middle element of the next sequence, $E_{n_{30}}$, where $n_{30} = b_{31} \cdot 2^{31} + 2^{30}$.

This process continues recursively, with each step revealing the middle trace element of the current sequence. V publishes a new `TraceChallenge`$_{32-k}$ at each step, setting a new bit b_k (for $k = 30, 29, \ldots, 1$), indicating whether to go left ($b_k = 0$) or right ($b_k = 1$). In each response, P commits to the next middle trace element $E_{n_{k-1}}$ in `TraceResponse`$_{32-k+1}$, for $n_{k-1} = \sum_{i=k}^{31}(b_i \cdot 2^i) + 2^{k-1}$.

With the last transaction, `TraceChallenge`$_{32}$, V sets the last bit b_0, indicating (dis)agreement with E_{n_0}, so that we finally obtain the index $\mathcal{N} = \sum_{i=0}^{31} b_i \cdot 2^i$. Let $\mathcal{N}' := \mathcal{N} + 1$. For the pair $(E_\mathcal{N}, E_{\mathcal{N}'})$, both P and V agree on the former and disagree on the latter. This pair allows the protocol to resolve the dispute by examining the exact step in the computation where the disagreement occurred.

At this point, P is forced to publish a `CommitInstruction` transaction, committing to the necessary information to execute $S_{\mathcal{N}'} = f_{ST}(S_\mathcal{N})$:

- pc_θ, $pc_{\theta'}$: the program counter of the states $S_\mathcal{N}$ and $S_{\mathcal{N}'}$, respectively;
- $insType_\theta \in \mathcal{OP}$: the instruction type at $\Pi[pc_\theta]$;
- $addrA_\theta$, $addrB_\theta$, $addrC_\theta$: the memory addresses referenced in $\Pi[pc_\theta]$;
- $valA_\theta$, $valB_\theta$: the memory values at addresses $addrA_\theta$ and $addrB_\theta$ in $S_\mathcal{N}$;
- $valC_\theta$: the value at address $addrC_\theta$ in $S_{\mathcal{N}'}$, i.e., after executing $f_{ST}(S_\mathcal{N})$.
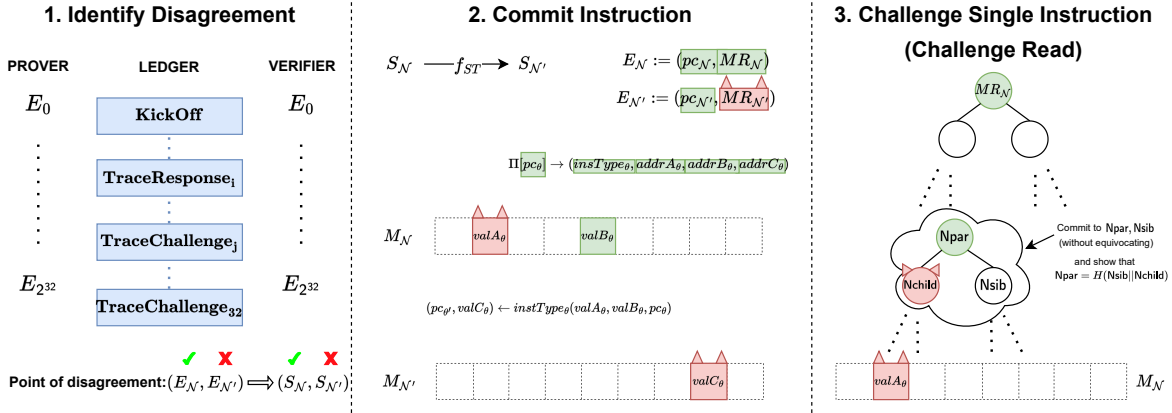
1. Identify Disagreement

PROVER LEDGER VERIFIER

E_0 | KickOff | E_0

$\vdots$ | TraceResponse$_i$ | $\vdots$

 | TraceChallenge$_j$ |

$E_{2^{32}}$ | TraceChallenge$_{32}$ | $E_{2^{32}}$

✓ ✗   ✓ ✗

Point of disagreement: $(E_{\mathcal{N}}, E_{\mathcal{N}'}) \Longrightarrow (S_{\mathcal{N}}, S_{\mathcal{N}'})$

2. Commit Instruction

$S_{\mathcal{N}} \xrightarrow{f_{ST}} S_{\mathcal{N}'}$

$E_{\mathcal{N}} := (pc_{\theta}, MR_{\mathcal{N}})$

$E_{\mathcal{N}'} := (pc_{\mathcal{N}'}, MR_{\mathcal{N}'})$

$\Pi[pc_{\theta}] \to (insType_{\theta}, addrA_{\theta}, addrB_{\theta}, addrC_{\theta})$

$M_{\mathcal{N}}$: $valA_{\theta}$ | $valB_{\theta}$

$(pc_{\theta'}, valC_{\theta}) \leftarrow insType_{\theta}(valA_{\theta}, valB_{\theta}, pc_{\theta})$

$M_{\mathcal{N}'}$: $valC_{\theta}$

3. Challenge Single Instruction (Challenge Read)

$MR_{\mathcal{N}}$

Commit to Npar, Nsib (without equivocating) and show that Npar = H(Nsib||Nchild)

Npar — Nchild, Nsib

$valA_{\theta}$ | $M_{\mathcal{N}}$

Figure 2. Example of dispute resolution in the BitVM protocol: To resolve a dispute, (1) P and V engage in a bisection game to identify the point of disagreement $(E_{\mathcal{N}}, E_{\mathcal{N}'})$ in their execution traces, indicating a disagreement in the transition from state $S_{\mathcal{N}}$ to $S_{\mathcal{N}'}$. Next, (2) P commits on-chain to all necessary information for executing $S_{\mathcal{N}'} = f_{ST}(S_{\mathcal{N}})$ (i.e., the values highlighted by colored boxes in the figure). In this example, we assume that P is committing to an incorrect value for $valA_{\theta}$, resulting in incorrect value for $valC_{\theta}$ and $MR_{\mathcal{N}'}$. (3) V challenges P through a bisection game over the path in the memory Merkle tree $M_{\mathcal{N}}$, from the root $MR_{\mathcal{N}}$ to the leaf containing $valA_{\theta}$. This bisection game reveals two intermediate nodes, Npar and Nchild, on which P and V disagree. To get away while using an incorrect value, P would need to prove that Nchild is indeed the left child of Npar, which is impossible without equivocating, leading to punishment.

In addition to committing to these values, the script of TraceChallenge$_{32}$ requires P to provide them so that $(pc_{\theta'}, valC_{\theta}) = insType_{\theta}(pc_{\theta}, valA_{\theta}, valB_{\theta})$ holds. In particular, there is a tapleaf that ensures this for each instruction type $\mathcal{OP} := \{\mathsf{ADD}, \mathsf{BEQ}, \mathsf{JMP}\}$.

Challenge Single Instruction. Armed with the prover's commitment to these values and the knowledge that $(pc_{\theta'}, valC_{\theta}) = insType_{\theta}(pc_{\theta}, valA_{\theta}, valB_{\theta})$ holds, it remains to verify that the committed values correspond to the committed states $S_{\mathcal{N}}$ and $S_{\mathcal{N}'}$. There are five potential errors in the committed values: (i) invalid program counter: pc_{θ} is not the program counter of $S_{\mathcal{N}}$, (ii) invalid program counter next: $pc_{\theta'}$ is not the program counter of $S_{\mathcal{N}'}$, (iii) invalid instruction: $insType_{\theta}, addrA_{\theta}, addrB_{\theta},$ or $addrC_{\theta}$ do not match $\Pi[pc_{\theta}]$, (iv) invalid read: $M_{\mathcal{N}}[addrA_{\theta}] \neq valA_{\theta}$ or $M_{\mathcal{N}}[addrB_{\theta}] \neq valB_{\theta}$, and (v) invalid write: $M_{\mathcal{N}'}[addrC_{\theta}] \neq valC_{\theta}$. Referring to Algorithm 5, if $S_{\mathcal{N}'} \neq f_{ST}(S_{\mathcal{N}})$ while $(pc_{\theta'}, valC_{\theta}) = insType_{\theta}(pc_{\theta}, valA_{\theta}, valB_{\theta})$ holds, one of these five errors must have occurred. In essence, either the values are not correctly derived from the agreed upon state $S_{\mathcal{N}}$, or the result is not stored correctly in $S_{\mathcal{N}'}$. By executing this step, V can pinpoint which of these errors occurred, and thus choose the appropriate challenge path.

Challenging the Current Program Counter. V can challenge the current program counter pc_{θ} by publishing a ChallengeCurrPC transaction. In this transaction, V commits to $\mathcal{N}$, revealing both P's commitment to $pc_{\mathcal{N}}$ and the currently disputed value pc_{θ}, proving that $pc_{\mathcal{N}} \neq pc_{\theta}$.

Since V already committed to $\mathcal{N}$ bit-by-bit during the bisection game with P to locate a disagreement point, V could attempt to equivocate on one of the bits when recommitting to $\mathcal{N}$ in the punishment phase. If equivocation occurs, P publishes a PunishCurrPC transaction, revealing

two different bitwise commitments to $\mathcal{N}$ linked to the verifier's public key. In this scenario, P claims all funds in the multisignature, thereby terminating the BitVM protocol.

If P fails to act within a specified time interval Δ, and V does not equivocate, V redeems the locked coins, terminating the protocol. This outcome happens either when P remains inactive or when V successfully proves that $pc_{\mathcal{N}} \neq pc_{\theta}$ without equivocation. This case is analogous to challenging the next program counter $pc_{\theta'}$.

Disproving a Wrong Program Instruction. In the CommitInstruction transaction's output locking script, a tapleaf is associated with each line of the program, hardcoding the program line number (pointed to by pc_{θ}) along with the corresponding values $insType_{\theta}, addrA_{\theta}, addrB_{\theta},$ and $addrC_{\theta}$. If the prover commits to an incorrect program instruction, the verifier can expose this by publishing a DisproveProgram transaction.

To spend one of these tapleaves, V must provide P's committed values: $pc_{\theta}, insType_{\theta}, addrA_{\theta}, addrB_{\theta},$ and $addrC_{\theta}$. The locking script ensures that the committed program counter pc_{θ} matches the program line number in the tapleaf. If the program counter matches but any other value does not, the DisproveProgram transaction succeeds, and BitVM terminates with V receiving the coins.

Challenge Read. The verifier challenges the correctness of $valA_{\theta}$ at memory address $addrA_{\theta}$ in $S_{\mathcal{N}}$, agreeing with the Merkle root $MR_{\mathcal{N}}$ but disputing the value $valA_{\theta}$. To start the challenge, V publishes a ChallengeRead transaction, spending the CommitInstruction output.

The parties engage in a bisection game on the memory Merkle path $\mathcal{P}_R := (MR_{\mathcal{N}}, \ldots, M_{\mathcal{N}}[addrA_{\theta}])$, i.e., the path from the root $MR_{\mathcal{N}}$ to the leaf $M_{\mathcal{N}}[addrA_{\theta}]$. The goal is to isolate a point of disagreement between V and P. Over 5 rounds of challenge-response (given $\mathsf{MemLen} = 2^{32}$), they progressively narrow down the path until they identify a pair

of nodes $(\mathsf{Npar}, \mathsf{Nchild})$ where they agree on Npar (i.e., the parent node) but disagree on Nchild (i.e., the child node). This bisection game proceeds in a similar fashion as the one for finding the disagreement in the execution trace.

Once this point of disagreement is found, V challenges P to provide a valid sibling node Nsib for Nchild, such that $H(\mathsf{Nsib} \parallel \mathsf{Nchild}) = \mathsf{Npar}$. If P cannot provide this due to an invalid commitment, V can claim the locked funds by publishing the `PunishRead` transaction. If P provides the correct sibling and no equivocation occurs, P claims the funds after the timelock expires. This process is analogous for $valB_\theta$ at memory address $addrB_\theta$.

Challenge Write. The process for challenging the result of a write operation is similar to *Challenge Read*, but with a key difference: V challenges the value $valC_\theta$ written to memory at $addrC_\theta$ in $S_{\mathcal{N}'}$. This affects both the Merkle path rooted at $MR_{\mathcal{N}}$ and the one rooted at $MR_{\mathcal{N}'}$. As a result, the bisection game involves two parallel Merkle paths: $\mathcal{P}_W$ (from $MR_{\mathcal{N}}$ to $M_{\mathcal{N}}[addrC_\theta]$) and $\mathcal{P}'_W$ (from $MR_{\mathcal{N}'}$ to $M_{\mathcal{N}'}[addrC_\theta]$). In each round, P reveals corresponding nodes from both paths. If P commits to incorrect values on $\mathcal{P}_W$, V focuses on that path (as in *Challenge Read*). Otherwise, if there is a disagreement on $\mathcal{P}'_W$, V focuses on that path. The game ends when a pair of parent-child nodes from both paths are isolated, and P must provide a valid sibling node Nsib to prove the correctness of the Merkle structure. If P equivocates, V can claim the coins by publishing `PunishWrite`, similarly to *Challenge Read*.

5.2. Honest Closure

In the happy path where the parties agree on the outcome of the computation, the on-chain footprint of the protocol is minimal, with only 3 transactions being published. After a set time period, P can spend the transaction in which they committed to the outcome by signing a pre-signed transaction that includes the result as a witness and distributes the coins according to the outcome mapping function f applied to this result. Should P equivocate, V claims all the funds.

Moreover, BitVM ensures a constant maximum number of transactions in case of dispute, once the VM instance is fixed. For example, consider a VM with 2^{32} memory cells, each capable of storing a 32-bit integer, executing a program up to 2^{32} instructions long, and requiring up to 2^{32} execution steps. Resolving a dispute in the execution of this VM would require publishing at most 81 transactions on-chain. This occurs when V first publishes a `Kickoff` transaction to initiate the Identify Disagreement phase. Once this phase is completed, V publishes on-chain a `ChallengeRead` transaction to initiate the Challenge Read path (or, similarly, V can initiate the Challenge Write path).

6. The BitVM full protocol

In this section, we present the full BitVM protocol specification. All scripts that we use comprise only (multi-) signature and Lamport signature verification, if/else statements, timelocks, and hashing, and are thus compatible with Bitcoin. Due to space constraints, we present the *setup*, *VM execute*, and *commit* phases and defer the *resolve dispute* phase to Appendix A.

6.1. Setup

In the *setup* phase, the prover P and the verifier V create and presign the necessary transactions for both honest protocol execution and potential dispute resolution; then both P and V lock an on-chain deposit, in_P and in_V, respectively.

At first, both P and V create all the transactions that are defined in this section and Appendix A, except `Setup`. Whenever such a transaction contains a new Lamport public key, the corresponding party creates one using Lamp.KeyGen and shares the public key with the other party.

Each transaction output either requires a 2-of-2 multisignature σ_{PV} to be spent and is presigned by both parties or requires a signature from one party along with a timelock. The timelock condition ensures that if a party ceases participation in the BitVM protocol, they forfeit the deposit, which the counterparty can then claim, along with their deposit.

After creating the transactions, the parties exchange them for presigning. For each transaction P (V) verifies it is well-formed according to the definitions below. If verified, the transaction is signed and sent to V (P). The

Finally, P and V sign and publish the `Setup` (cf. Eq. (1)) transaction on-chain. This transaction takes inputs from P (holding $\mathsf{in}_P\text{\textcent}$) and V (holding $(\mathsf{in}_V\text{\textcent})$, creating an output that can be unlocked by both parties under the following conditions: P can spend the output by presenting Lamport commitments to MR_0 and MR_{final} (i.e., the commitments to the input and output of program Π), while V can spend the output after a timeout period Δ. For brevity, we use σ_U when the signature is in a transaction's witness; the message signed in that case is the transaction body (inputs and outputs without witnesses).

$$
\begin{aligned}
\texttt{Setup} := \\
\Big(in &= [(*, *, \mathsf{CheckSig}_{\mathsf{pk}_P}), (*, *, \mathsf{CheckSig}_{\mathsf{pk}_V})], \\
wit &= [(\sigma_P), (\sigma_V)], \\
out &= [(d\text{\textcent}; \langle \mathsf{CommitComputationScript}, \mathsf{TL}(\Delta) \wedge \\
&\quad \mathsf{CheckSig}_{\mathsf{pk}_V})\rangle)] \Big),
\end{aligned} \tag{1}
$$

The script $\mathsf{CommitComputationScript}$ is defined below.

$$
\mathsf{CommitComputationScript} := \\
\mathsf{CheckMSig}_{\mathsf{pk}_{PV}} \wedge \mathsf{CheckComm}_{\mathsf{pk}_{E_0}} \wedge \mathsf{CheckComm}_{\mathsf{pk}_{E_{\mathsf{final}}}}.
$$

6.2. VM Execute

The prover P sends to the verifier V the input x of program Π via a communication channel. Both P and V execute off-chain the program Π with input x on their

VM instance. They copy x into the VM memory M and call Algorithm 6 with input $S_0 := (M, 0, \Pi)$. They get as output the VM execution trace $ExecTrace$ and the memory M, from which they fetch the output y of program Π with input x. We stress that this is the most resource-intensive phase of BitVM and it is entirely performed *off-chain*.

6.3. Commit

The prover P publishes the `CommitComputation` transaction (cf. Eq. (2)) on-chain, which spends the output of the `Setup` transaction by providing a Lamport commitment to $E_0 := (MR_0, pc_0)$ and $E_{\mathsf{final}} := (MR_{\mathsf{final}}, pc_{\mathsf{final}})$.

$$
\begin{aligned}
\texttt{CommitComputation} := \\
\Big([in = (\texttt{Setup}, 0, \texttt{CommitComputationScript})], \\
[wit = (\sigma_{PV}, E_0, c_{E_0}, E_{\mathsf{final}}, c_{E_{\mathsf{final}}})], \\
[out = (d\ddot{B}; \langle \mathsf{CheckMSig}_{\mathsf{pk}_{PV}}, \texttt{CloseScript}, \\
\mathsf{CheckSig}_{\mathsf{pk}_V} \wedge \mathsf{TL}(2\Delta) \rangle)] \Big).
\end{aligned}
\tag{2}
$$

The script CloseScript$_\mathsf{i}$ is defined in Algorithm 8.

Algorithm 8 The script CloseScript$_\mathsf{i}$. In the setup phase, the public key $\mathsf{pk}_{MR_{\mathsf{final}}}$ is hard-coded in the script.

1: **function** CloseScript$_\mathsf{i}(\sigma_{PV}, MR_{\mathsf{final}}, c_{MR_{\mathsf{final}}})$
2: $\quad \mathsf{TL}(\Delta)$;
3: $\quad \mathsf{CheckMSigVerify}_{\mathsf{pk}_{PV}}(\sigma_{PV})$;
4: $\quad \mathsf{CheckCommVerify}_{\mathsf{pk}_{MR_{\mathsf{final}}}}(MR_{\mathsf{final}}, c_{MR_{\mathsf{final}}})$;
5: $\quad$ **if** $MR_{\mathsf{final}} = \mathcal{MR}_i$ **then**
6: $\quad\quad$ **return** True;
7: $\quad$ **return** False.

The verifier V can either challenge P if they disagree with the MR_{final} published on-chain by P or simply take no action if they agree. Since the VM execution is deterministic, honest parties running the same program on the same input naturally agree on MR_{final}. A disagreement, therefore, implies that one party is behaving dishonestly.

Close. V agrees with P's commitment to MR_{final} and does not dispute it. The BitVM protocol follows the happy path: after a timeout period Δ, P publishes one of the close transactions, $\texttt{Close}_1, \ldots, \texttt{Close}_m$. Each of these transactions distributes the funds according to the outcome mapping function f, applied to one of the possible results of the computation[10]. If P does not publish any $\texttt{Close}_i$ transaction after that $\mathsf{TL}(2\Delta)$ expires after the publication of $\texttt{CommitComputation}$ transaction, V can unlock $\texttt{CommitComputation}$ output with their signature and claim all the funds.

Transaction $\texttt{Close}_i$ (cf. Eq. (3)) spends the output of $\texttt{CommitComputation}$ by unlocking CloseScript$_\mathsf{i}$ and creates two outputs. The first output carries $o_P\ddot{B}$ and can be

---

10. During the setup phase, P and V agree on f and jointly create and sign a finite set of closing transactions, one for each possible outcome. The funds are distributed to P and V according to the result of f.

---

unlocked by P after a timeout period Δ or by V if P equivocates on MR_{final} (as shown in Algorithm 9). The second output carries $o_V\ddot{B}$ and can be unlocked by V.

$$
\begin{aligned}
\texttt{Close}_\mathsf{i} := \\
\Big([in = (\texttt{CommitComputation}, 0, \texttt{CloseScript})], \\
[wit = (\sigma_{PV}, MR_{\mathsf{final}}, c_{MR_{\mathsf{final}}})], \\
[out = (v_P; \langle \mathsf{CheckSig}_{\mathsf{pk}_P} \wedge \mathsf{TL}(\Delta), \\
\texttt{PunishCloseScript} \rangle), (v_V; \mathsf{CheckSig}_{\mathsf{pk}_V})] \Big).
\end{aligned}
\tag{3}
$$

Algorithm 9 The script PunishCloseScript. In the setup phase, the public key $\mathsf{pk}_{MR_{\mathsf{final}}}$ is hard-coded in the script.

1: **function** PunishCloseScript(σ_{PV}, c_0, c_1)
2: $\quad \mathsf{CheckMSigVerify}_{\mathsf{pk}_{PV}}(\sigma_{PV})$;
3: $\quad$ **for** $i = 1, \ldots, |MR_{\mathsf{final}}|_{bit}$ **do**
4: $\quad\quad$ **if** $\mathsf{Equivocation}(\mathsf{pk}_{MR_{\mathsf{final}}[i]}, c_0, c_1) = \mathsf{True}$ **then**
5: $\quad\quad\quad$ **return** True;
6: $\quad$ **return** False.

Identify Disagreement. V disagrees with P's commitment to MR_{final}. To dispute P's result, V publishes the `KickOff` transaction (cf. Eq. (4)) by spending `CommitComputation`'s output, unlocking it through the multisignature.

$$
\begin{aligned}
\texttt{KickOff} := \\
\Big([in = (\texttt{CommitComputation}, 0, \mathsf{CheckMSig}_{\mathsf{pk}_{PV}})], \\
wit = [(\sigma_{PV})], \\
out = [(d\ddot{B}; \langle \texttt{ChallScript}_1, \mathsf{TL}(\Delta) \wedge \mathsf{CheckSig}_{\mathsf{pk}_V} \rangle)] \Big).
\end{aligned}
\tag{4}
$$

The script ChallScript$_\mathsf{j}$, with $\mathsf{j} \in \{1, \ldots, 31\}$, is defined as follows:

$$
\texttt{ChallScript}_\mathsf{j} := \mathsf{CheckMSig}_{\mathsf{pk}_{PV}} \wedge \mathsf{CheckComm}_{\mathsf{pk}_{E_{n_{32-j}}}}.
$$

The parties engage in an on-chain interactive protocol known as dispute bisection game (cf. Section B.1): the game is played over the VM execution trace $ExecTrace := (E_0, \ldots, E_{\mathsf{final}})$ and has the goal to determine a pair of consecutive VM trace elements $(E_{\mathcal{N}}, E_{\mathcal{N}'})$, where $\mathcal{N}' := \mathcal{N} + 1$, such that they agree on $E_{\mathcal{N}}$ and disagree on $E_{\mathcal{N}'}$.

After that, V initiates the bisection game by publishing the `Kickoff` transaction, P responds by publishing the $\texttt{TraceResponse}_1$ transaction (cf. Eq. (5)), committing to $E_{n_{31}}$ in the witness, where $n_{31} = 1 \cdot 2^{31}$.

$$
\begin{aligned}
\texttt{TraceResponse}_1 := \\
\Big(in = [(\texttt{KickOff}, 0, \mathsf{CheckMSig}_{\mathsf{pk}_{PV}} \wedge \\
\mathsf{CheckComm}_{\mathsf{pk}_{E_{n_{31}}}})], \\
wit = [(\sigma_{PV}, E_{n_{31}}, c_{E_{n_{31}}})], \\
out = [(d\ddot{B}; \langle \texttt{RespScript}_1, \mathsf{TL}(\Delta) \wedge \mathsf{CheckSig}_{\mathsf{pk}_P} \rangle)] \Big).
\end{aligned}
\tag{5}
$$

The script RespScript$_i$, with $i \in \{1, \ldots, 32\}$, is defined as follows:

$$\mathsf{RespScript_i} := \mathsf{CheckMSig}_{\mathsf{pk}_{PV}} \wedge \mathsf{CheckComm}_{\mathsf{pk}_{b_{32-i}}}.$$

Next, V publishes the $\mathtt{TraceChallenge_1}$ transaction (cf. Eq. (6)), committing to bit b_{31} in the witness.

$$
\begin{aligned}
\mathtt{TraceChallenge_1} := \\
\Big(in &= [(\mathtt{TraceResponse_1}, 0, \mathsf{RespScript_1})], \\
wit &= [(\sigma_{PV}, b_{31}, c_{b_{31}})], \\
out &= [(d\mathsf{\ddot{B}}; \langle \mathsf{ChallScript_2}, \mathsf{TL}(\Delta) \wedge \mathsf{CheckSig}_{\mathsf{pk}_V} \rangle)] \Big).
\end{aligned}
\tag{6}
$$

During the dispute bisection game, P publishes transactions $\mathtt{TraceResponse_i}$ (cf. Eq. (7)), with $i = 1, \ldots, 32$, and V publishes transactions $\mathtt{TraceChallenge_j}$ (cf. Eq. (8)), with $j = 1, \ldots, 31$.

$$
\begin{aligned}
\mathtt{TraceResponse_i} := \\
\Big(in &= [(\mathtt{TraceChallenge_{i-1}}, 0, \mathsf{ChallScript_{i-1}})], \\
wit &= [(\sigma_{PV}, E_{n_{32-i}}, c_{E_{n_{32-i}}})], \\
out &= [(d\mathsf{\ddot{B}}; \langle \mathsf{RespScript_i}, \mathsf{TL}(\Delta) \wedge \mathsf{CheckSig}_{\mathsf{pk}_P} \rangle)] \Big),
\end{aligned}
\tag{7}
$$

where $n_{32-i} = 1 \cdot 2^{32-i} + \sum_{k=32-i+1}^{31} b_k \cdot 2^k$.

$$
\begin{aligned}
\mathtt{TraceChallenge_j} := \\
\Big(in &= [(\mathtt{TraceResponse_j}, 0, \mathsf{RespScript_j})], \\
wit &= [(\sigma_{PV}, b_{32-j}, c_{b_{32-j}})], \\
out &= [(d\mathsf{\ddot{B}}; \langle \mathsf{ChallScript_{j+1}}, \mathsf{TL}(\Delta) \wedge \mathsf{CheckSig}_{\mathsf{pk}_V} \rangle)] \Big),
\end{aligned}
\tag{8}
$$

Finally, V publishes $\mathtt{TraceChallenge_{32}}$ (cf. Eq. (9)).

$$
\begin{aligned}
\mathtt{TraceChallenge_{32}} := \\
\Big(in &= [(\mathtt{TraceResponse_{32}}, 0, \mathsf{RespScript_{32}})], \\
wit &= [(\sigma_{PV}, b_0, c_{b_0})], \\
out &= [(d\mathsf{\ddot{B}}; \langle \mathsf{ADDScript}, \mathsf{BEQScript}, \mathsf{JMPScript}, \\
& \quad \mathsf{TL}(\Delta) \wedge \mathsf{CheckSig}_{\mathsf{pk}_V} \rangle)] \Big),
\end{aligned}
\tag{9}
$$

To unlock the $\mathtt{TraceChallenge_{32}}$ output, P is forced to provide a commitment for pc_θ, $pc_{\theta'}$, $insType_\theta$, $addrA_\theta$, $addrB_\theta$, $addrC_\theta$, $valA_\theta$, $valB_\theta$, $valC_\theta$. The instruction $insType_\theta$ must match the leaf being spent and pc_θ, $pc_{\theta'}$, $valA_\theta$, $valB_\theta$, $valC_\theta$ must align with the instruction's semantics. For instance, if P unlocks the ADDScript tapleaf (cf. Algorithm 10), the condition $\mathrm{ADD}(pc_\theta, valA_\theta, valB_\theta) = (pc_{\theta'}, valC_\theta)$ must hold, where ADD is the VM instruction defined in Algorithm 7, lines 1 to 3. The leaves BEQScript and JMPScript are analogous to ADDScript but they encode the semantics of the BEQ and JMP instructions, respectively. The *resolve dispute* phase is deferred to Appendix A.

Algorithm 10 The script ADDScript. In the setup phase, the public keys pk_{pc_θ}, $\mathsf{pk}_{pc_{\theta'}}$, $\mathsf{pk}_{insType_\theta}$, $\mathsf{pk}_{addrA_\theta}$, $\mathsf{pk}_{addrB_\theta}$, $\mathsf{pk}_{addrC_\theta}$, $\mathsf{pk}_{valA_\theta}$, $\mathsf{pk}_{valB_\theta}$, $\mathsf{pk}_{valC_\theta}$ and the sematics of the ADD instruction are hard-coded in the script.

1: **function** ADDScript(σ_{PV}, pc_θ, c_{pc_θ}, $pc_{\theta'}$, $c_{pc_{\theta'}}$, $insType_\theta$, $c_{insType_\theta}$, $addrA_\theta$, c_{addrA_θ}, $addrB_\theta$, c_{addrB_θ}, $addrC_\theta$, c_{addrC_θ}, $valA_\theta$, c_{valA_θ}, $valB_\theta$, c_{valB_θ}, $valC_\theta$, c_{valC_θ})
2:   $\mathsf{CheckMSigVerify}_{\mathsf{pk}_{PV}}(\sigma_{PV})$;
3:   $\mathsf{CheckCommVerify}_{\mathsf{pk}_{pc_\theta}}(pc_\theta, c_{pc_\theta})$;
4:   $\mathsf{CheckCommVerify}_{\mathsf{pk}_{pc_{\theta'}}}(pc_{\theta'}, c_{pc_{\theta'}})$;
5:   $\mathsf{CheckCommVerify}_{\mathsf{pk}_{insType_\theta}}(insType_\theta, c_{insType_\theta})$;
6:   $\mathsf{CheckCommVerify}_{\mathsf{pk}_{addrA_\theta}}(addrA_\theta, c_{addrA_\theta})$;
7:   $\mathsf{CheckCommVerify}_{\mathsf{pk}_{addrB_\theta}}(addrB_\theta, c_{addrB_\theta})$;
8:   $\mathsf{CheckCommVerify}_{\mathsf{pk}_{addrC_\theta}}(addrC_\theta, c_{addrC_\theta})$;
9:   $\mathsf{CheckCommVerify}_{\mathsf{pk}_{valA_\theta}}(valA_\theta, c_{valA_\theta})$;
10:   $\mathsf{CheckCommVerify}_{\mathsf{pk}_{valB_\theta}}(valB_\theta, c_{valB_\theta})$;
11:   $\mathsf{CheckCommVerify}_{\mathsf{pk}_{valC_\theta}}(valC_\theta, c_{valC_\theta})$;
12:   **if** $insType_\theta = \mathsf{ADD} \wedge \mathsf{ADD}(pc_\theta, valA_\theta, valB_\theta) = (pc_{\theta'}, valC_\theta)$ **then**
13:    **return** True;
14:   **else**
15:    **return** False.

7. Security Analysis

In this section, we show that BitVM is On-chain State Verification Protocol. BitVM satisfies *Balance Security* and *Rational Correctness*. To achieve that, we first argue about the correctness of each phase of the protocol, i.e., setup, execution, dispute, and punishment phase, in Lemmas D.1–D.13. Then, we model BitVM as an Extensive Form Game (EFG) for which we provide the related background in Appendix C. Last, we combine the aforementioned Lemmas and the EFG representation to prove that BitVM satisfies *Rational Correctness* and *Balance Security*.

7.1. Balance Security

We consider two scenarios: (i) both parties act honestly, and (ii) one party, $A \in \{P, V\}$, deviates at any step of the protocol. In both cases, we prove that an honest party retains their funds.

Setup. If either party deviates during Setup, the honest party will refuse to sign the $\mathtt{Setup}$ transaction, ensuring no coins are locked unless both parties have received all necessary pre-signed transactions (cf. Lemma D.1).

Both parties honest. When both parties are honest, BitVM follows an optimistic path: P posts the correct computation result on-chain in $\mathtt{CommitComputation}$ and, after a timelock Δ, publishes $\mathtt{Close}$ to distribute funds according to the outcome function f (cf. Lemma D.3).

V **is honest and** P **is Byzantine.** If P fails to publish $\mathtt{CommitComputation}$, because of either being inactive or executing invalid computations, or subsequently fails to post $\mathtt{Close}$, V can claim the coins after the relevant timelocks

expire (Δ or 2Δ) (cf. Lemmas D.2, D.3). This mechanism prevents hostage scenarios by enabling V to reclaim funds in case of non-responsiveness.

If P has committed an incorrect computation result in `CommitComputation`, V publishes `KickOff` (Identify Disagreement phase). As shown in (cf. Lemma D.4), if P is inactive, V can claim the coins after the relevant timelock expires. If the Identify Disagreement phase completes, V knows a step for which P has committed on-chain to an execution of the VM state transition function (Algorithm 5) incorrectly (cf. Lemma D.7).

P can deviate in the following ways: by using an incorrect program counter (current or next), making an incorrect memory read or write, or using invalid instructions. For each of these deviations, V can post the corresponding transaction on-chain (e.g., `ChallengeCurrPC`, `ChallengeNextPC`, `ChallengeRead`, `ChallengeWrite`, `PunishInstruction`). Following the respective dispute path, V is able to disprove P's computation and claim the coins, as we conclude in Lemma D.14.

P **is honest and** V **is Byzantine.** V can misbehave by publishing `KickOff` on-chain to initiate the Dispute Phase, although P has committed to the correct output of the execution in `CommitComputation`. We show that if V is inactive during the Identify Disagreement phase, P can claim the coins after the timelock expires (cf. Lemma D.5). Otherwise, if the Identify Disagreement phase completes, V has to publish one of the transactions `ChallengeCurrPC`, `ChallengeNextPC`, `ChallengeRead`, `ChallengeWrite`, `PunishInstruction`. Since P has committed to the correct values, V cannot disprove P, as we conclude in Lemma D.15.

7.2. Rational Correctness

We have shown that if a party $A \in \{P, V\}$ misbehaves at any point, the other party A' can claim all the coins. Therefore, when both parties are rational, no party will misbehave but instead follow the optimistic path of BitVM.

8. Discussion

In the following, we outline BitVM's costs to demonstrate its feasibility and efficiency, discuss potential applications, its limitations – particularly its permissioned nature – and conclude with possible improvements.

Feasibility & Cost Evaluation. To assess the feasibility of our approach, we estimate the transaction fees for both an optimistic run and the most expensive dispute run of BitVM, using the VM instance defined in Section 4.

We assume constant transaction fees of $3sat/vB$[11], a Bitcoin price of $64,300\$$ (as of 24 September 2024). For the Lamport signatures, we set $\lambda = 160$, thus public key

and commitment are of length $20B$; for an a-bit message they occupy $a \times 20\text{B} = a \times 5\text{vB}$. We also assume that $\Delta = 12$ hours, meaning each timelock expires after half a day. Different concrete values can be chosen, but any such selection would require scaling the evaluation accordingly. For details on how we computed the actual transaction sizes, we refer the reader to Appendix E.

Optimistic case. In the optimistic case, three on-chain transactions are published: `Setup` ($187vB$), `CommitComputation` ($2,093vB$), and `Close` ($1,289vB$), totaling $3,569vB$. The protocol's execution cost is $10,707$ sat ($6.90\$$). In terms of execution time, once `Setup` is published, BitVM completes in at most $2u$ rounds.

Dispute case. We focus on the most expensive path in terms of fees, the *Challenge Write* path. Besides the `Setup` and the `CommitComputation` transactions, the following transactions are also published on-chain in case of dispute:

- 1 `KickOff`, 32 `TraceChallenge`, 32 `TraceResponse`, 1 `ChallengeWrite`, 5 `WriteChallenge` transactions, with up to $1,112vB$.
- 5 `WriteResponse`, transactions, with up to $2,093vB$.
- The transactions `CommitWrite1` and `PunishWrite1` with up to $7,286vB$.
- The transaction `CommitInstruction` with up to $2,751vB$.

Overall, the path weighs $109,020vB$. The total protocol execution cost is roughly $327Ksat$, or about $210\$$, and, once the `Setup` transaction is published on-chain, it takes at most $80 \times \Delta = 40$ days to complete its execution.

Applications, Limitations & Improvements. This paper aims to formalize BitVM and prove its security, establishing for the first time that quasi-Turing complete on-chain state verification is feasible on blockchains like Bitcoin. By facilitating the verification of such programs on Bitcoin, BitVM unlocks a diverse array of applications, including cross-chain bridges, light clients, zk-proof verification, state channels, games, and escrow contracts. Off-chain computation by users eliminates additional costs for miners, making BitVM a practical and scalable solution for complex applications.

Nevertheless, the current construction is not optimized for efficiency. Possible improvements include undergoing engineering efforts [15], [16], replacing Lamport signatures with Winternitz signatures [28], or incorporating future opcodes, such as covenants [2].

Furthermore, the current construction is a two-party permissioned protocol, limiting its direct applicability compared to [19], for example, which permits anyone to act as the verifier. However, this added flexibility incurs a significantly higher cost of $1.9K\$$ (due to filling an entire 4MB Bitcoin block), compared to BitVM's practical cost range of $7-200\$$.

---

11. In Bitcoin, the size of a SegWit [25] transaction is expressed in *virtual Bytes*, or *vBytes* (vB). The number of vBytes of a transaction witness is equal to its number of Bytes divided by four.

References

[1] "Bitcoin script," https://en.bitcoin.it/wiki/Script, accessed: 2024-11.

[2] M. Bartoletti, S. Lande, and R. Zunino, "Bitcoin covenants unchained," in *Leveraging Applications of Formal Methods, Verification and Validation: Applications*, 2020, pp. 25–42.

[3] M. Bartoletti, R. Marchesin, and R. Zunino, "Secure compilation of rich smart contracts on poor UTXO blockchains," in *9th IEEE European Symposium on Security and Privacy, EuroS&P*, 2024. [Online]. Available: https://doi.org/10.1109/EuroSP60621.2024.00021

[4] "Mate in 1: white to move," https://lichess.org/analysis/1Bb3BN/R2Pk2r/1Q5B/4q2R/2bN4/4Q1BK/1p6/1bq1R1rb_w_-.

[5] "Mate in 1: white to move," https://lichess.org/analysis/r1qr1b2/1R3pkp/P2p2pN/ppnPp1Q1/bn2P3/4P1NP/PBBPp1P1/5RK1_w_Q_e6_0_1.

[6] M. Bartoletti and R. Zunino, "Bitml: A calculus for bitcoin smart contracts," in *ACM CCS*, 2018, p. 83–100. [Online]. Available: https://doi.org/10.1145/3243734.3243795

[7] M. Bartoletti, T. Cimoli, and R. Zunino, "Fun with bitcoin smart contracts," in *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*, Cham, 2018, pp. 432–449.

[8] P. Das, L. Eckey, T. Frassetto, D. Gens, K. Hostáková, P. Jauernig, S. Faust, and A.-R. Sadeghi, "FastKitten: Practical smart contracts on bitcoin," in *USENIX Security 19*, 2019.

[9] T. Frassetto, P. Jauernig, D. Koisser, D. Kretzler, B. Schlosser, S. Faust, and A.-R. Sadeghi, "Pose: Practical off-chain smart contract execution," in *NDSS Symposium*. Internet Society, 2023. [Online]. Available: http://dx.doi.org/10.14722/ndss.2023.23118

[10] Ethereum Foundation, "State channels," 2024, accessed: 2024-11. [Online]. Available: https://ethereum.org/en/developers/docs/scaling/state-channels/

[11] S. Dziembowski, S. Faust, and K. Hostáková, "General state channel networks," in *ACM CCS*, 2018, p. 949–966. [Online]. Available: https://doi.org/10.1145/3243734.3243856

[12] L. Aumayr, O. Ersoy, A. Erwig, S. Faust, K. Hostáková, M. Maffei, P. Moreno-Sanchez, and S. Riahi, "Generalized channels from limited blockchain scripts and adaptor signatures," in *ASIACRYPT*, 2021.

[13] C. DLC, "Discreet log contracts: An overview," 2018. [Online]. Available: https://adiabat.github.io/dlc.pdf

[14] V. Madathil, S. A. Thyagarajan, D. Vasilopoulos, L. Fournier, G. Malavolta, and P. Moreno-Sanchez, "Cryptographic oracle-based conditional payments," in *NDSS Symposium*, 2023. [Online]. Available: https://dx.doi.org/10.14722/ndss.2023.24024

[15] FairGate Labs, "Bitvmx & bitvm," 2024, accessed: 2024-11. [Online]. Available: https://fairgate.io

[16] Sovryn, "Bitcoinos," 2024, accessed: 2024-11. [Online]. Available: https://sovryn.com/bitcoinos

[17] Citrea, "Bitcoin settlement trust-minimized btc bridge: Bitvm," 2024, accessed: 2024-11. [Online]. Available: https://docs.citrea.xyz/technical-specs/characteristics/bitcoin-settlement-trust-minimized-btc-bridge/bitvm

[18] Robin Linus, "Bitvm: Compute anything on bitcoin," 2023, accessed: 2024-11. [Online]. Available: https://bitvm.org/bitvm.pdf

[19] R. Linus, L. Aumayr, A. Zamyatin, A. Pelosi, Z. Avarikioti, and M. Maffei, "BitVM2: Bridging bitcoin to second layers," 2024, accessed: 2024-11. [Online]. Available: https://bitvm.org/bitvm_bridge.pdf

[20] J. Garay, A. Kiayias, and N. Leonardos, "The bitcoin backbone protocol: Analysis and applications," *J. ACM*, vol. 71, no. 4, Aug. 2024. [Online]. Available: https://doi.org/10.1145/3653445

[21] A. Tzinas, S. Sridhar, and D. Zindros, "On-chain timestamps are accurate," Cryptology ePrint Archive, Paper 2023/1648, 2023. [Online]. Available: https://eprint.iacr.org/2023/1648

[22] L. Aumayr, Z. Avarikioti, M. Maffei, G. Scaffino, and D. Zindros, "Blink: An optimal proof of proof-of-work," Cryptology ePrint Archive, 2024. [Online]. Available: https://eprint.iacr.org/2024/692

[23] J. Poon and T. Dryja, "The bitcoin lightning network: Scalable off-chain instant payments," 2016.

[24] S. Rain, G. Avarikioti, L. Kovács, and M. Maffei, "Towards a game-theoretic security analysis of off-chain protocols," in *IEEE CSF*, 2023, pp. 107–122. [Online]. Available: https://doi.org/10.1109/CSF57540.2023.00003

[25] E. Lombrozo and P. Wuille, "Bip 141, segregated witness," 2015, accessed: 2024-11. [Online]. Available: https://github.com/bitcoin/bips/blob/master/bip-0141.mediawiki

[26] P. Wuille, J. Nick, and A. Towns, "Bip 0341, taproot: Segwit version 1 spending rules," jan 2020, https://en.bitcoin.it/wiki/BIP_0341.

[27] L. Lamport, "Constructing digital signatures from a one way function," Tech. Rep. CSL-98, October 1979, https://www.microsoft.com/en-us/research/publication/constructing-digital-signatures-one-way-function/.

[28] D. Boneh and V. Shoup, "A graduate course in applied cryptography," *Draft 0.6*, 2023, https://toc.cryptobook.us/.

[29] ZeroSync, "BitVM Github repository," dec 2023, https://github.com/BitVM/BitVM.

[30] Esolangs, "Addleq," https://esolangs.org/wiki/Addleq.

[31] M. J. Osborne and A. Rubinstein, *A Course in Game Theory*. Cambridge, MA: MIT Press, 1994.

[32] "Bitcoin transactions," https://en.bitcoin.it/wiki/Transaction, accessed: 2024-11.

Appendix A.
The BitVM Full Protocol: Resolve Dispute

A.1. Resolve Dispute

P spends the TraceChallenge$_{32}$ output by publishing the CommitInstruction transaction (cf. Eq. (10)).

$$
\begin{aligned}
\texttt{CommitInstruction} := \\
\Big(in = [(\texttt{TraceChallenge}_{32}, 0, \texttt{OPScript})], \\
wit = [(\sigma_{PV}, pc_\theta, c_{pc_\theta}, pc_{\theta'}, c_{pc_{\theta'}}, insType_\theta, \\
c_{insType_\theta}, addrA_\theta, c_{addrA_\theta}, addrB_\theta, c_{addrB_\theta}, addrC_\theta, \\
c_{addrC_\theta}, valA_\theta, c_{valA_\theta}, valB_\theta, c_{valB_\theta}, valC_\theta, c_{valC_\theta})], \\
out = [(d_P^B; \langle \texttt{CheckMSig}_{\mathsf{pk}_{PV}}, \{\texttt{CIScriptPCCurr}_i\}_{i\in\{1,...,32\}}, \\
\{\texttt{CIScriptPCNext}_i\}_{i\in\{1,...,32\}}, \{\texttt{CIScriptInstr}_j\}_{j\in\{1,...,\ell\}}, \\
\texttt{TL}(\Delta) \wedge \texttt{CheckSig}_{\mathsf{pk}_P} \rangle)] \Big).
\end{aligned}
$$

$$(10)$$

The tapleaf that P unlocks when publishing CommitInstruction is OPScript $\in$ {ADDScript, BEQScript, JMPScript}.

By publishing the CommitInstruction transaction, P reveals all the information necessary for the state transition from $S_\mathcal{N}$ to $S_{\mathcal{N}'}$. Depending on the specific error that V claims P made, V spends the output of CommitInstruction in one of the following ways.

A.1.1. Challenging the Current Program Counter. V is claiming that, by publishing `CommitInstruction`, P is committing to a program counter pc_θ at step $\mathcal{N}$ that differs from the program counter $pc_\mathcal{N}$ (previously committed by P during the dispute bisection game). V challenges the current program counter pc_θ by unlocking one of the leaves $\mathsf{CIScriptPCCurr_i}$ (cf. Algorithm 11) via the publication of transaction `ChallengeCurrPC` (cf. Eq. (11)). We use Algorithm 12 to map the challenge-response rounds to the leaves $\mathsf{CIScriptPCCurr_0}, \ldots, \mathsf{CIScriptPCCurr_{31}}$. When V unlocks leaf $\mathsf{CIScriptPCCurr_i}$, they challenge the program counter of the $(32-i)$-th challenge-response round of the dispute bisection game.

Algorithm 11 The script $\mathsf{CIScriptPCCurr_i}$, for $i \in \{0, \ldots, 31\}$. For each $\mathsf{CIScriptPCCurr_i}$, in the setup phase, we hard-code the public keys pk_{pc_θ}, $\mathsf{pk}_\mathcal{N}$. For each $\mathsf{CIScriptPCCurr_i}$, for $i \in \{1, \ldots, 31\}$, we hard-code the same public key pk_{pc_i} hard-coded in $\mathsf{ChallScript_i}$. For $\mathsf{CIScriptPCCurr_0}$, we hard-code the same public key pk_{pc_0} hard-coded in $\mathsf{CommitComputationScript}$.

1: **function** $\mathsf{CIScriptPCCurr_i}(\sigma_{PV}, \mathcal{N}, c_\mathcal{N}, pc_i, c_{pc_i}, pc_\theta, c_{pc_\theta})$
2:      $\mathsf{CheckMSigVerify}_{\mathsf{pk}_{PV}}(\sigma_{PV})$;
3:      $\mathsf{CheckCommVerify}_{\mathsf{pk}_\mathcal{N}}(\mathcal{N}, c_\mathcal{N})$;
4:      **if** $\mathsf{CountZeroes}(\mathcal{N}) \neq i$ **then**
5:          $\triangleright$ *Maps $\mathcal{N}$ to one of the 32 program counters pc_{n_0},*
             $\ldots, pc_{n_{31}}$. $\triangleleft$
6:          **return** False;
7:      $\mathsf{CheckCommVerify}_{\mathsf{pk}_{pc_i}}(pc_i, c_{pc_i})$;
8:      $\mathsf{CheckCommVerify}_{\mathsf{pk}_{pc_\theta}}(pc_\theta, c_{pc_\theta})$;
9:      **if** $pc_i \neq pc_\theta$ **then**
10:        **return** True;
11:      **else**
12:        **return** False.

Algorithm 12 The algorithm $\mathsf{CountZeroes}$. It counts the number of consecutive bits set to 0 in the binary representation of a number N, starting from the least significant bit (LSB), until the first occurrence of a bit set to 1.

1: **function** $\mathsf{CountZeroes}(N)$
2:      $counter \leftarrow 0$;
3:      $flag \leftarrow$ False;
4:      **for** $i = 0, \ldots, |N|_{bit} - 1$ **do**
5:        **if** $N[|N|_{bit} - i] = 1$ **then**
6:          $flag \leftarrow$ True;
7:          $\triangleright$ *Set the flag, stop incrementing the counter.* $\triangleleft$
8:        **else**
9:          **if** $flag =$ False **then**
10:            $counter \leftarrow counter + 1$;
11:      **return** $counter$.

$\mathsf{ChallengeCurrPC} :=$
$$\Big(in = [(\mathtt{CommitInstruction}, 0, \mathsf{CIScriptPCCurr}_\mathcal{N})],$$
$$wit = [(\sigma_{PV}, \mathcal{N}, c_\mathcal{N}, pc_\mathcal{N}, c_{pc_\mathcal{N}}, pc_\theta, c_{pc_\theta})],$$
$$out = [(d\overset{..}{\mathsf{B}}; \langle \mathsf{ChallPCScript}, \mathsf{TL}(\Delta) \wedge \mathsf{CheckSig}_{\mathsf{pk}_V} \rangle)] \Big).$$
(11)

In the `ChallengeCurrPC` transaction, V commits again

---

to $\mathcal{N}$, potentially equivocating. P can punish equivocation by unlocking $\mathsf{ChallPCScript}$ script (cf. Algorithm 13).

Algorithm 13 The script $\mathsf{ChallPCScript}$. In the setup phase, the public key $\mathsf{pk}_\mathcal{N}$ is hard-coded in the script.

1: **function** $\mathsf{ChallPCScript}(\sigma_{PV}, c_0, c_1)$
2:      $\mathsf{CheckMSigVerify}_{\mathsf{pk}_{PV}}(\sigma_{PV})$;
3:      **for** $i = 1, \ldots, |\mathcal{N}|_{bit}$ **do**
4:        **if** $\mathsf{Equivocation}(\mathsf{pk}_{\mathcal{N}[i]}, c_0, c_1) =$ True **then**
5:          **return** True;
6:      **return** False.

If V equivocates, P publishes `PunishCurrPC` (cf. Eq. (12)), redeeming all the funds in the multisignature.

$\mathsf{PunishCurrPC} :=$
$$\Big(in = [(\mathtt{ChallengeCurrPC}, 0, \mathsf{ChallPCScript})],$$
$$wit = [(\sigma_{PV}, c_0, c_1)],$$
$$out = [(d\overset{..}{\mathsf{B}}; \mathsf{CheckSig}_{\mathsf{pk}_P})] \Big).$$
(12)

A.1.2. Challenging the Next Program Counter. V is claiming that, by publishing `CommitInstruction`, P is committing to a program counter $pc_{\theta'}$ at step $\mathcal{N}'$ (output of the VM operation executed on-chain) that differs from the previously committed program counter $pc_{\mathcal{N}'}$. V challenges the next program counter $pc_{\theta'}$ by unlocking one of the leaves $\mathsf{CIScriptPCNext_i}$ (cf. Algorithm 14) via the publication of transaction `ChallengeNextPC` (cf. Eq. (13)).

Algorithm 14 The script $\mathsf{CIScriptPCNext_i}$, for $i \in \{0, \ldots, 31\}$. In the script $\mathsf{CIScriptPCNext_i}$, during the setup phase we hard-code the same public keys that we hard-code in the script $\mathsf{CIScriptPCCurr_i}$, except for public key pk_{pc_θ}. We hard-code $\mathsf{pk}_{pc_{\theta'}}$ instead.

1: **function** $\mathsf{CIScriptPCNext_i}(\sigma_{PV}, \mathcal{N}', c_{\mathcal{N}'}, pc_i, c_{pc_i}, pc_{\theta'}, c_{pc_{\theta'}})$
2:      $\mathsf{CheckMSigVerify}_{\mathsf{pk}_{PV}}(\sigma_{PV})$;
3:      $\mathsf{CheckCommVerify}_{\mathsf{pk}_\mathcal{N}}(\mathcal{N}', c_{\mathcal{N}'})$;
4:      **if** $\mathsf{CountZeroes}(\mathcal{N}') \neq i$ **then**
5:          $\triangleright$ *Maps $\mathcal{N}'$ to one of the 32 program counters pc_{n_0},*
             $\ldots, pc_{n_{31}}$. $\triangleleft$
6:        **return** False;
7:      $\mathsf{CheckCommVerify}_{\mathsf{pk}_{pc_i}}(pc_i, c_{pc_i})$;
8:      $\mathsf{CheckCommVerify}_{\mathsf{pk}_{pc_{\theta'}}}(pc_{\theta'}, c_{pc_{\theta'}})$;
9:      **if** $pc_i \neq pc_{\theta'}$ **then**
10:        **return** True;
11:      **else**
12:        **return** False.

$\mathsf{ChallengeNextPC} :=$
$$\Big(in = [(\mathtt{CommitInstruction}, 0, \mathsf{CIScriptPCNext}_{\mathcal{N}'})],$$
$$wit = [(\sigma_{PV}, \mathcal{N}', c_{\mathcal{N}'}, pc_{\mathcal{N}'}, c_{pc_{\mathcal{N}'}}, pc_{\theta'}, c_{pc_{\theta'}})],$$
$$out = [(d\overset{..}{\mathsf{B}}; \langle \mathsf{ChallPCScript}, \mathsf{TL}(\Delta) \wedge \mathsf{CheckSig}_{\mathsf{pk}_V} \rangle)] \Big).$$
(13)

In this challenge path, V can equivocate on $\mathcal{N}'$[12]. P can punish equivocation by publishing the PunishNextPC transaction (cf. Eq. (14)), which unlocks ChallPCScript by proving the equivocation. Upon doing so, P redeems all the funds locked in the multisignature.

$$
\begin{aligned}
\texttt{PunishNextPC} := \\
\Big(in = [(\texttt{ChallengeNextPC}, 0, \texttt{ChallPCScript})], \\
wit = [(\sigma_{PV}, c_0, c_1)], out = [(d\texttt{B}; \texttt{CheckSig}_{\mathsf{pk}_P})] \Big).
\end{aligned}
\tag{14}
$$

A.1.3. Punish Wrong Instruction. P has committed to a current program counter pc_θ that does not correspond to the correct program instruction, specifically: $\Pi[pc_\theta] \neq (insType_\theta, addrA_\theta, addrB_\theta, addrC_\theta)$.

V spends the CommitInstruction output by unlocking the script CIScriptInstr$_j$ (cf. Algorithm 15) and publishing the DisproveProgram transaction (cf. Eq. (15)). A script CIScriptInstr exists for each of the ℓ instructions in the program Π.

Algorithm 15 The script CIScriptInstr$_j$, for $j \in \{1, ..., \ell\}$. In the script CIScriptInstr$_j$, during the setup phase we hard-code the public keys $\mathsf{pk}_{pc_\theta}, \mathsf{pk}_{insType_\theta}, \mathsf{pk}_{addrA_\theta}, \mathsf{pk}_{addrB_\theta}, \mathsf{pk}_{addrC_\theta}$, for $j \in \{1, \ldots, \ell\}$. In addition to the public keys, the j-th instruction of Π is also hard-coded into the script CIScriptInstr$_j$.

1: **function** CIScriptInstr$_j(\sigma_{PV}, pc_\theta, c_{pc_\theta}, insType_\theta, c_{insType_\theta},$
   $addrA_\theta, c_{addrA_\theta}, addrB_\theta, c_{addrB_\theta}, addrC_\theta, c_{addrC_\theta})$
2:   CheckMSigVerify$_{\mathsf{pk}_{PV}}(\sigma_{PV})$;
3:   CheckCommVerify$_{\mathsf{pk}_{pc_\theta}}(pc_\theta, c_{pc_\theta})$;
4:   CheckCommVerify$_{\mathsf{pk}_{insType_\theta}}(insType_\theta, c_{insType_\theta})$;
5:   CheckCommVerify$_{\mathsf{pk}_{addrA_\theta}}(addrA_\theta, c_{addrA_\theta})$;
6:   CheckCommVerify$_{\mathsf{pk}_{addrB_\theta}}(addrB_\theta, c_{addrB_\theta})$;
7:   CheckCommVerify$_{\mathsf{pk}_{addrC_\theta}}(addrC_\theta, c_{addrC_\theta})$;
8:   **if** $\Big((pc_\theta = j) \wedge (insType_j \neq insType_\theta \vee addrA_j \neq$
     $addrA_\theta \vee addrB_j \neq addrB_\theta \vee addrC_j \neq addrC_\theta)\Big)$
     then
9:      **return** True;
10: **else**
11:     **return** False.

$$
\begin{aligned}
\texttt{DisproveProgram} := \\
\Big(in = [(\texttt{CommitInstruction}, 0, \texttt{CIScriptInstr}_{pc_\theta})], \\
wit = [(\sigma_{PV}, pc_\theta, c_{pc_\theta}, insType_\theta, c_{insType_\theta}, \\
addrA_\theta, c_{addrA_\theta}, addrB_\theta, c_{addrB_\theta}, addrC_\theta, c_{addrC_\theta})], \\
out = [(d\texttt{B}; \texttt{CheckSig}_{\mathsf{pk}_V})] \Big).
\end{aligned}
\tag{15}
$$

A.1.4. Challenge Read. V starts the challenge by publishing the ChallengeRead transaction (cf. Eq. (16)), spending the CommitInstruction output[13].

---

12. We use $\mathcal{N}'$ to emphasize that challenging the next program counter is a distinct path from challenging the current program counter. However, in practice, V commits to the same bits $b_0, \ldots, b_{31}$, i.e., the same public key $\mathsf{pk}_\mathcal{N}$ is used in both current and next program counter challenge paths.

13. We explain how *Challenge Read* works by presenting a challenge to $valA_\theta$; the process for challenging $valB_\theta$ is analogous.

$$
\begin{aligned}
\texttt{ChallengeRead} := \\
\Big(in = [(\texttt{CommitInstruction}, 0, \texttt{CheckMSig}_{\mathsf{pk}_{PV}})], \\
wit = [(\sigma_{PV})], \\
out = [(d\texttt{B}; \langle \texttt{ReadChallScript}_1, \texttt{TL}(\Delta) \wedge \texttt{CheckSig}_{\mathsf{pk}_V} \rangle)] \Big).
\end{aligned}
\tag{16}
$$

The script ReadChallScript$_j$, with $j \in \{1, \ldots, 5\}$ is defined as follows:

$$
\texttt{ReadChallScript}_j := \texttt{CheckMSig}_{\mathsf{pk}_{PV}} \wedge \texttt{CheckComm}_{\mathsf{pk}_{\mathsf{Node}_{d_{5-j}}}}.
$$

The parties engage in the read bisection game (cf. Section B.2). The game is played over the sequence $\mathcal{P}_R := (MR_\mathcal{N}, \ldots, M_\mathcal{N}[addrA_\theta])$, namely, a path from the root to one of the leaves in $MerkleTree_{M_\mathcal{N}}$, i.e., the Merkle tree of the memory at step $\mathcal{N}$. P responds by publishing the ReadResponse$_1$ transaction (cf. Eq. (17)), committing to $\mathsf{Node}_{d_4} := \mathcal{P}_R[d_4]$ in the witness, where $d_4 = 1 \cdot 2^4$.

$$
\begin{aligned}
\texttt{ReadResponse}_1 := \\
\Big(in = [(\texttt{ChallengeRead}, 0, \texttt{ReadChallScript}_1)], \\
wit = [(\sigma_{PV}, \mathsf{Node}_{d_4}, c_{\mathsf{Node}_{d_4}})], \\
out = [(d\texttt{B}; \langle \texttt{ReadRespScript}_1, \texttt{TL}(\Delta) \wedge \texttt{CheckSig}_{\mathsf{pk}_P} \rangle)] \Big).
\end{aligned}
\tag{17}
$$

ReadRespScript$_i$ with $i \in \{1, \ldots, 5\}$ is defined as:

$$
\texttt{ReadRespScript}_i := \texttt{CheckMSig}_{\mathsf{pk}_{PV}} \wedge \texttt{CheckComm}_{\mathsf{pk}_{b'_{5-i}}}.
$$

Then, V publishes ReadChallenge$_1$ transaction (cf. Eq. (18)), committing to bit b'_4 in the witness, where $b'_4 = 1$ if V agrees with Node_{d_4}, and $b'_4 = 0$ otherwise.

$$
\begin{aligned}
\texttt{ReadChallenge}_1 := \\
\Big(in = [(\texttt{ReadResponse}_1, 0, \texttt{ReadRespScript}_1)], \\
wit = [(\sigma_{PV}, b'_4, c_{b'_4})], \\
out = [(d\texttt{B}; \langle \texttt{ReadChallScript}_2, \texttt{TL}(\Delta) \wedge \texttt{CheckSig}_{\mathsf{pk}_V} \rangle)] \Big);
\end{aligned}
\tag{18}
$$

P and V continue playing the read bisection game by publishing transactions ReadResponse$_i$ (cf. Eq. (19) and ReadChallenge$_j$ (cf. Eq. (20)), respectively, with $i = 2, \ldots, 5$ and $j = 1, \ldots, 4$.

$$
\begin{aligned}
\texttt{ReadResponse}_i := \\
\Big(in = [(\texttt{ReadChallenge}_{i-1}, 0, \texttt{ReadChallScript}_i)], \\
wit = [(\sigma_{PV}, \mathsf{Node}_{d_{5-i}}, c_{\mathsf{Node}_{d_{5-i}}})], \\
out = [(d\texttt{B}; \langle \texttt{ReadRespScript}_i, \texttt{TL}(\Delta) \wedge \texttt{CheckSig}_{\mathsf{pk}_P} \rangle)] \Big);
\end{aligned}
\tag{19}
$$

$$
\begin{aligned}
\texttt{ReadChallenge}_j := \\
\Big(in = [(\texttt{ReadResponse}_j, 0, \texttt{ReadRespScript}_j)], \\
wit = [(\sigma_{PV}, b'_{5-j}, c_{b'_{5-j}})], \\
out = [(d\texttt{B}; \langle \texttt{ReadChallScript}_{j+1}, \texttt{TL}(\Delta) \wedge \texttt{CheckSig}_{\mathsf{pk}_V} \rangle)] \Big);
\end{aligned}
\tag{20}
$$

where $d_{5-i} = 1 \cdot 2^{5-i} + \sum_{k=5-i+1}^{4} b'_k \cdot 2^k$.

Then, V publishes the `ReadChallenge`$_5$ transaction (cf. Eq. (21)). In total, V has committed to the bits $b'_4, \ldots, b'_0$. These bits determine the last element on the path $\mathcal{P}_R$ upon which P and V agree. Let $\mathcal{N}_{Mer}$ be the corresponding integer, computed as $\mathcal{N}_{Mer} = \sum_{k=0}^{4} b'_k \cdot 2^k$.

$$
\begin{aligned}
&\texttt{ReadChallenge}_5 := \\
&\Big(in = [(\texttt{ReadResponse}_5, 0, \texttt{ReadRespScript}_5)], \\
&\quad wit = [(\sigma_{PV}, b'_0, c_{b'_0})], \\
&\quad out = [(d\ddot{\beta}; \langle \texttt{HashReadScript}_1, \ldots, \texttt{HashReadScript}_{20}, \\
&\quad \texttt{RootReadScript}_1, \ldots, \texttt{RootReadScript}_{32}, \texttt{ValueAScript}, \\
&\quad \texttt{TL}(\Delta) \wedge \texttt{CheckSig}_{\texttt{pk}_V} \rangle)] \Big).
\end{aligned}
$$

(21)

The integer $\mathcal{N}_{Mer}$, chosen by V, conditions which Tapleaves P can unlock to spend the `ReadChallenge`$_5$ output. We can distinguish three cases.

(A) Commit Read. The point of disagreement is between two consecutive elements of the path $\mathcal{P}_R$, excluding the first and the last. P publishes the `CommitRead1` transaction (cf. Eq. (22)) to spend the `ReadChallenge`$_5$ output. To do so, P provides a witness that unlocks one of the scripts `HashReadScript`$_1$, ..., `HashReadScript`$_{20}$. Each script hard-codes the public key of a pair of nodes belonging to $\{\texttt{Node}_{d_0}, \ldots, \texttt{Node}_{d_4}\}$, the first being the parent node in $MerkleTree_{MR_\mathcal{N}}$ and the second being the child node[14]. Additionally, P provide a sibling node Nsib, claiming whether it is the left or right sibling by committing to the bit v_{pos}, the $\mathcal{N}_{Mer}$-th bit of $addrA_\theta$. To unlock the script, it must hold that the child node, when concatenated with the sibling node, hashes to the parent node.

We present the pseudocode of the script in `HashReadScript`$_1$ in Algorithm 16. The scripts `HashReadScript`$_2$, ..., `HashReadScript`$_{20}$ are identical except for the public keys hard-coded to set the parent and the child nodes, and the mapping from $\mathcal{N}_{Mer}$ to the appropriate Tapleaf.

$$
\begin{aligned}
&\texttt{CommitRead1} := \\
&\Big(in = [(\texttt{ReadChallenge}_5, 0, \texttt{HashReadScript}_i)], \\
&\quad wit = [(\sigma_{PV}, \mathcal{N}_{Mer}, c_{\mathcal{N}_{Mer}}, v_{pos}, c_{v_{pos}}, c_{addrA_\theta}, \texttt{Nsib}, \\
&\quad \texttt{Npar}, c_{\texttt{Npar}}, \texttt{Nchild}, c_{\texttt{Nchild}})], \\
&\quad out = [(d\ddot{\beta}; \langle \texttt{CommitRead1Script}, \texttt{TL}(\Delta) \wedge \texttt{CheckSig}_{\texttt{pk}_P} \rangle)] \Big).
\end{aligned}
$$

(22)

V can punish P if they equivocate either on Npar, Nchild, v_{pos} by publishing `PunishRead1` (cf. Eq. (23)), which requires to unlock the `CommitRead1Script` (cf. Algorithm 17) script.

$$
\begin{aligned}
&\texttt{PunishRead1} := \\
&\Big(in = [(\texttt{CommitRead1}, 0, \texttt{CommitRead1Script})], \\
&\quad wit = [(\sigma_{PV}, c_0, c_1)], out = [(d\ddot{\beta}; \texttt{CheckSig}_{\texttt{pk}_V}] \Big).
\end{aligned}
$$

(23)

---

14. Since any node can be the parent of any other, we need 20 scripts to capture all the possibilities.

Algorithm 16 The script HashReadScript$_1$. The bit $v_{pos} \in \{0, 1\}$ represents the position of the child node ($v_{pos} = 0$ means that $\texttt{Node}_{d_0}$ is the left child of $\texttt{Node}_{d_4}$, $v_{pos} = 1$ means the opposite. Nsib is the sibling node of $\texttt{Node}_{d_0}$ that P presents. In the setup phase, the public keys $\texttt{pk}_{\mathcal{N}_{Mer}}$, $\texttt{pk}_{addrA_\theta}$, $\texttt{pk}_{\texttt{Node}_{d_4}}$, $\texttt{pk}_{\texttt{Node}_{d_0}}$, are hard-coded in the script.

1: **function** HashReadScript$_1$(σ_{PV}, $\mathcal{N}_{Mer}$, $c_{\mathcal{N}_{Mer}}$, v_{pos}, $c_{v_{pos}}$, c_{addrA_θ}, Nsib, $\texttt{Node}_{d_4}$, $c_{\texttt{Node}_{d_4}}$, $\texttt{Node}_{d_0}$, $c_{\texttt{Node}_{d_0}}$)
2:      CheckMSigVerify$_{\texttt{pk}_{PV}}$(σ_{PV});
3:      CheckCommVerify$_{\texttt{pk}_{\mathcal{N}_{Mer}}}$($\mathcal{N}_{Mer}, c_{\mathcal{N}_{Mer}}$);
4:      ▷ *Since V committed to $\mathcal{N}_{Mer}$, P does not know* $\texttt{sk}_{\mathcal{N}_{Mer}}$. *Therefore, to satisfy this guard, has to provide the commitment that V made* ◁
5:      **if** CountZeroes($\mathcal{N}_{Mer}$) $\neq 5 - 1$ **then**
6:          ▷ *Since $\texttt{Node}_{d_4}$ is the parent node here,* CountZeroes($\mathcal{N}_{Mer}$) *should be 4.* ◁
7:          **return** False;
8:      CheckCommVerify$_{\texttt{pk}_{addrA_\theta[\mathcal{N}_{Mer}]}}$($v_{pos}, c_{addrA_\theta[\mathcal{N}_{Mer}]}$);
9:      ▷ *The whole public key* $\texttt{pk}_{addrA_\theta}$ *is hard-coded in the script, but only the the $\mathcal{N}_{Mer}$-th entry is used* ◁
10:     CheckCommVerify$_{\texttt{pk}_{\texttt{Node}_{d_4}}}$($\texttt{Node}_{d_4}, c_{\texttt{Node}_{d_4}}$); ▷ *Parent node*
11:     CheckCommVerify$_{\texttt{pk}_{\texttt{Node}_{d_0}}}$($\texttt{Node}_{d_0}, c_{\texttt{Node}_{d_0}}$); ▷ *Child node*
12:     **if** $v_{pos} = 0$ **then**
13:        **if** $H(\texttt{Node}_{d_0} || \texttt{Nsib}) = \texttt{Node}_{d_4}$ **then**
14:          **return** True;
15:        **else**
16:          **return** False;
17:     **else**
18:        **if** $H(\texttt{Nsib} || \texttt{Node}_{d_0}) = \texttt{Node}_{d_4}$ **then**
19:          **return** True;
20:        **else**
21:          **return** False.

Algorithm 17 The script CommitRead1Script. The public keys $\texttt{pk}_{\texttt{Npar}}$, $\texttt{pk}_{\texttt{Nchild}}$, and $\texttt{pk}_{addrA_\theta}$ are hard-coded during the setup.

1: **function** CommitRead1Script(σ_{PV}, c_0, c_1)
2:      CheckMSigVerify$_{\texttt{pk}_{PV}}$(σ_{PV});
3:      **for** $i = 1, \ldots, |\texttt{Npar}|_{bit}$ **do**
4:        **if** Equivocation($\texttt{pk}_{\texttt{Npar}[i]}, c_0, c_1$) $=$ True $\vee$ Equivocation($\texttt{pk}_{\texttt{Nchild}[i]}, c_0, c_1$) $=$ True $\vee$ Equivocation($\texttt{pk}_{addrA_\theta[i]}, c_0, c_1$) $=$ True **then**
5:          **return** True;
6:      **return** False.

(B) Commit Value A. If V agrees with every element that P committed (i.e., $b'_4 = \cdots = b'_0 = 1$), $\mathcal{N}_{Mer}$ is set to 31. The point of disagreement is between the last intermediate node published by P, $\texttt{Node}_{d_0}$, and $valA_\theta$; To spend the `ReadChallenge`$_5$ output, P unlocks `ValueAScript`. `ValueAScript` is analogous to `HashReadScript`$_i$ with the following differences: (i) CountZeroes($\mathcal{N}_{Mer}$) $= 0$; (ii) the parent node is $\texttt{Node}_{d_0}$; (iii) the child node is not one of the nodes $\texttt{Node}_{d_4}$, ..., $\texttt{Node}_{d_0}$, but $valA_\theta$ instead.

P publishes `CommitRead2` transaction (analogous to `CommitRead1`, but unlocking `ValueAScript` instead). V can publish transaction `PunishRead2` (analogous to `PunishRead1`) if P equivocates on the values committed

in the `CommitRead2` transaction.

(C) Commit Read Root. If V disagrees with every element that P committed (i.e., $b'_4 = \cdots = b'_0 = 0$), $\mathcal{N}_{Mer}$ is set to 0. The point of disagreement is between the last intermediate node published by P, Node_{d_0}, and $MR_{\mathcal{N}}$. P unlocks one of the leaves $\mathsf{RootReadScript}_1, \ldots, \mathsf{RootReadScript}_{32}$, according to which number $\mathcal{N}$ V committed at the end of the dispute bisection game. We provide $\mathsf{RootReadScript}_i$ in Algorithm 18.

Algorithm 18 The script $\mathsf{RootReadScript}_i$. In the setup phase, the public keys $\mathsf{pk}_{\mathcal{N}_{Mer}}$, $\mathsf{pk}_{\mathcal{N}}$, $\mathsf{pk}_{\mathsf{Node}_{d_0}}$, pk_{MR_i} are hard-coded in the script.

1: **function** $\mathsf{RootReadScript}_i(\sigma_{PV}, \mathcal{N}_{Mer}, c_{\mathcal{N}_{Mer}}, v_{pos}, c_{v_{pos}},$
   $c_{addrA_\theta}, \mathsf{Nsib}, \mathcal{N}, c_{\mathcal{N}}, \mathsf{Node}_{d_0}, c_{\mathsf{Node}_{d_0}}, MR_i, c_{MR_i})$
2:     $\mathsf{CheckMSigVerify}_{\mathsf{pk}_{PV}}(\sigma_{PV})$;
3:     $\mathsf{CheckCommVerify}_{\mathsf{pk}_{\mathcal{N}_{Mer}}}(\mathcal{N}_{Mer}, c_{\mathcal{N}_{Mer}})$;
4:     ▷ *Since V committed to $\mathcal{N}_{Mer}$, P does not know*
   $\mathsf{sk}_{\mathcal{N}_{Mer}}$. Therefore, to satisfy this guard, P has to
   provide the commitment that V made ◁
5:     $\mathsf{CheckCommVerify}_{\mathsf{pk}_{\mathcal{N}}}(\mathcal{N}, c_{\mathcal{N}})$;
6:     ▷ *P has to provide the commitment that V made in the*
   dispute phase ◁
7:     **if** $\mathsf{CountZeroes}(\mathcal{N}) \neq i$ **then**
8:         **return** False;
9:     **if** $\mathsf{CountZeroes}(\mathcal{N}_{Mer}) \neq 5$ **then** ▷ *$\mathcal{N}_{Mer}$ must be equal*
   to 0
10:         **return** False;
11:     $\mathsf{CheckCommVerify}_{\mathsf{pk}_{addrA_\theta[\mathcal{N}_{Mer}]}}(v_{pos}, c_{addrA_\theta[\mathcal{N}_{Mer}]})$;
12:     $\mathsf{CheckCommVerify}_{\mathsf{pk}_{\mathsf{Node}_{d_0}}}(\mathsf{Node}_{d_0}, c_{\mathsf{Node}_{d_0}})$;
13:     ▷ *for any $\mathsf{RootReadScript}_i$, Node_{d_0} is always the child*
   node ◁
14:     $\mathsf{CheckCommVerify}_{\mathsf{pk}_{MR_i}}(\mathsf{Node}_{d_0}, c_{\mathsf{Node}_{d_0}})$;
15:     **if** $v_{pos} = 0$ **then**
16:         **if** $H(\mathsf{Node}_{d_0}||\mathsf{Nsib}) = MR_i$ **then**
17:             **return** True;
18:         **else**
19:             **return** False;
20:     **else**
21:         **if** $H(\mathsf{Nsib}||\mathsf{Node}_{d_0}) = MR_i$ **then**
22:             **return** True;
23:         **else**
24:             **return** False.

P unlocks $\mathsf{RootReadScript}_i$ by publishing the `CommitRead3` transaction (cf Eq. (24)).

$$\mathsf{CommitRead3} :=$$
$$\Big(in = [(\mathsf{ReadChallenge}_5, 0, \mathsf{ReadRootScript}_i)],$$
$$wit = [(\sigma_{PV}, \mathcal{N}_{Mer}, c_{\mathcal{N}_{Mer}}, \mathcal{N}, c_{\mathcal{N}}, \mathsf{Node}_{d_0}, c_{\mathsf{Node}_{d_0}})],$$
$$out = [(d\overset{\text{B}}{P}; \langle \mathsf{CommitRead3Script}, \mathsf{TL}(\Delta) \wedge \mathsf{CheckSig}_{\mathsf{pk}_P} \rangle)] \Big).$$
$$(24)$$

V can punish P if they equivocate on Node_{d_0}, MR_i or $addrA_\theta$ by publishing `PunishRead3` (cf. Eq. (25)), which unlocks $\mathsf{CommitRead3Script}$, analogous to $\mathsf{CommitRead1Script}$ but with pk_{MR_i}, $\mathsf{pk}_{\mathsf{Node}_{d_0}}$ instead of $\mathsf{pk}_{\mathsf{Npar}}$, $\mathsf{pk}_{\mathsf{Nchild}}$.

$$\mathsf{PunishRead3} :=$$
$$\Big(in = [(\mathsf{CommitRead3}, 0, \mathsf{CommitRead3Script})],$$
$$wit = [(\sigma_{PV}, c_0, c_1)], out = [(d\overset{\text{B}}{P}; \mathsf{CheckSig}_{\mathsf{pk}_V}]\Big). \quad (25)$$

A.1.5. Challenge Write. V challenges the result of the writing operation. Specifically, V claims that P is writing $valC'_\theta \neq valC_\theta$ in $M_{\mathcal{N}'}[addrC_\theta]$ in their local VM execution[15]. As a result, the memory root $MR_{\mathcal{N}'}$ is incorrect.

The parties engage in the write bisection game (cf. Section B.3) over the sequences $\mathcal{P}_W := (MR_{\mathcal{N}}, \ldots, M_{\mathcal{N}}[addrC_\theta])$ and $\mathcal{P}'_W := (MR_{\mathcal{N}'}, \ldots, M'_{\mathcal{N}}[addrC_\theta])$, that are paths in the merkle trees $MerkleTree_{M_{\mathcal{N}}}$ and $MerkleTree_{M_{\mathcal{N}'}}$, respectively. The transactions and locking scripts in the challenge write branch of the protocol closely follow the structure of those in the challenge read branch, with the following differences:

- The structure of the `WriteResponse`$_i$ transaction is analogous to `ReadResponse`$_i$ transaction but, in the witness, P provides two values (and their commitments) instead of one. These values are the d_{5-i}-th elements of $\mathcal{P}_W$ and $\mathcal{P}'_W$, respectively.
- As long as V agrees on the elements of the path $\mathcal{P}_W$, they focus on finding the disagreement in the path $\mathcal{P}'_W$. In the `WriteChallenge`$_j$ transaction (analogously to `ReadChallenge`$_j$), V sets (and commits to) the bit $b'_{5-j} = 0$ if V agrees with the element of $\mathcal{P}'_W$ provided by P. Otherwise, V sets (and commits to) the bit $b'_{5-j} = 1$. However, once V finds a disagreement in an element of $\mathcal{P}_W$, from that point on, V focuses on $\mathcal{P}_W$ and set the bit b'_{5-j} as in the *Challenge Read* branch.

During the write bisection game, P commits to the pairs nodes $\{(\mathsf{Node}_{d_4}, \mathsf{Node}'_{d_4}), \ldots, (\mathsf{Node}_{d_0}, \mathsf{Node}'_{d_0})\}$, where $\mathsf{Node}_{d_4}, \ldots, \mathsf{Node}_{d_0} \in \mathcal{P}_W$ and $\mathsf{Node}'_{d_4}, \ldots, \mathsf{Node}'_{d_0} \in \mathcal{P}'_W$. Analogous to the *Challenge Read* branch, V commits bit by bit to an integer $\mathcal{N}_{Mer} = \sum_{k=0}^4 b'_k \cdot 2^k$, which conditions how P can unlock `WriteChallenge`$_5$. There are three cases.

Note that P does not explicitly know which pair of elements in $\mathcal{P}_W$ or $\mathcal{P}'_W$ V disagrees with. However, as long as P is able to provide a pair of nodes $(\mathsf{Npar}, \mathsf{Nchild})$ for $\mathcal{P}_W$, a pair of nodes $(\mathsf{Npar}', \mathsf{Nchild}')$ for $\mathcal{P}'_W$, and a node Nsib such that $H(\mathsf{Nsib}||\mathsf{Nchild}) = \mathsf{Npar}$ and $H(\mathsf{Nsib}||\mathsf{Nchild}') = \mathsf{Npar}'$, they will be able to unlock `WriteChallenge`$_5$.

(A) CommitWrite. The point of disagreement is between two consecutive elements of $\mathcal{P}_W$ or between two consecutive elements of $\mathcal{P}'_W$, excluding for both paths the first and the last elements. P can unlock one of the scripts $\mathsf{HashWriteScript}_1$ (cf. Algorithm 19) , ..., $\mathsf{HashWriteScript}_{20}$ via publishing the `CommitWrite1` transaction (cf. Eq. (26)).

---

15. We assume P commits correctly to $valC_\theta$ in the witness of the `CommitInstruction` transaction, regardless of local execution. For example, if $insType_\theta := \mathsf{ADD}$, then $valA_\theta + valB_\theta = valC_\theta$. If $valC_\theta$ is incorrect, V can challenge $valA_\theta$ or $valB_\theta$.

Algorithm 19 The script HashWriteScript$_1$. The bit $v_{pos} \in \{0,1\}$ represents the position of the child nodes ($v_{pos} = 0$ means that Node$_{d_0}$ and Node$'_{d_0}$ are the left childs of Node$_{d_4}$ and Node$'_{d_4}$, respectively). Nsib is the sibling node of Node$_{d_0}$ in $\mathcal{P}_W$ and of Node$'_{d_0}$ in $\mathcal{P}'_W$. In the setup phase, the public keys pk$_{\mathcal{N}_{Mer}}$, pk$_{addrC_\theta}$, pk$_{Node_{d_4}}$, pk$_{Node_{d_0}}$, pk$_{Node'_{d_4}}$, pk$_{Node'_{d_0}}$ are hard-coded in the script.

```
 1: function HashWriteScript₁(σ_PV, 𝒩_Mer, c_𝒩_Mer, v_pos, c_v_pos,
       c_addrC_θ, Nsib, Node_d₄, c_Node_d₄, Node_d₀, c_Node_d₄, Node'_d₄,
       c_Node'_d₄, Node'_d₀, c_Node'_d₄)
```
2:     CheckMSigVerify$_{pk_{PV}}$(σ_{PV});
3:     CheckCommVerify$_{pk_{\mathcal{N}_{Mer}}}$($\mathcal{N}_{Mer}, c_{\mathcal{N}_{Mer}}$);
4:     ▷ *Since V committed to $\mathcal{N}_{Mer}$, P does not know* sk$_{\mathcal{N}_{Mer}}$. *Therefore, to satisfy this guard, has to provide the commitment that V made* ◁
5:     **if** CountZeroes($\mathcal{N}_{Mer}$) $\neq 5 - 1$ **then**
6:         ▷ *Since* Node$_{d_4}$, Node$'_{d_4}$ *are the parent nodes here,* CountZeroes($\mathcal{N}_{Mer}$) *should be 4.* ◁
7:         **return** False;
8:     CheckCommVerify$_{pk_{addrC_\theta}[\mathcal{N}_{Mer}]}$($v_{pos}, c_{addrC_\theta}[\mathcal{N}_{Mer}]$);
9:     ▷ *The whole public key* pk$_{addrC_\theta}$ *is hardcoded in the script, but only the the $\mathcal{N}_{Mer}$-th entry is used* ◁
10:     CheckCommVerify$_{pk_{Node_{d_4}}}$(Node$_{d_4}, c_{Node_{d_4}}$); ▷ *Parent node in $\mathcal{P}_W$*
11:     CheckCommVerify$_{pk_{Node_{d_0}}}$(Node$_{d_0}, c_{Node_{d_0}}$); ▷ *Child node in $\mathcal{P}_W$*
12:     CheckCommVerify$_{pk_{Node'_{d_4}}}$(Node$'_{d_4}, c_{Node'_{d_4}}$); ▷ *Parent node in $\mathcal{P}'_W$*
13:     CheckCommVerify$_{pk_{Node'_{d_0}}}$(Node$'_{d_0}, c_{Node'_{d_0}}$); ▷ *Child node in $\mathcal{P}'_W$*
14:     **if** $v_{pos} = 0$ **then**
15:         **if** $H($Node$_{d_0}||$Nsib$) = $Node$_{d_4} \wedge H($Node$'_{d_0}||$Nsib$) = $Node$'_{d_4}$ **then**
16:             **return** True
17:         **else**
18:             **return** False
19:     **else**
20:         **if** $H($Nsib$||$Node$_{d_0}) = $Node$_{d_4} \wedge H($Nsib$||$Node$'_{d_0}) = $Node$'_{d_4}$ **then**
21:             **return** True
22:         **else**
23:             **return** False

CommitWrite1 :=

$\Big(in = [(\texttt{WriteChallenge}_5, 0, \mathsf{HashWriteScript_i})],$

$wit = [(\sigma_{PV}, \mathcal{N}_{Mer}, c_{\mathcal{N}_{Mer}}, v_{pos}, c_{v_{pos}}, c_{addrC_\theta}, \mathsf{Nsib},$

$\mathsf{Npar}, c_{\mathsf{Npar}}, \mathsf{Nchild}, c_{\mathsf{Nchild}}, \mathsf{Npar'}, c_{\mathsf{Npar'}}, \mathsf{Nchild'},$

$c_{\mathsf{Nchild'}})],$

$out = [(d\overset{!!}{\mathbb{B}}; \langle \mathsf{CommitWrite1Script}, \mathsf{TL}(\Delta) \wedge \mathsf{CheckSig}_{pk_P} \rangle)] \Big);$

$$\tag{26}$$

The script CommitWrite1Script is identical to CommitRead1Script (cf. Algorithm 17) except that it also checks for potential equivocation on Npar', Nchild', and $addrC_\theta$ rather than $addrA_\theta$. As a consequence, the PunishWrite1 transaction is analogous to PunishRead1. Thus, if P equivocates while committing to Npar, Nchild, Npar', Nchild', V can claim all the coins locked in the multisignature.

(B) Commit Value C. $\mathcal{N}_{Mer} = 31$, the point of disagreement is between Node$_{d_0}$ and $valC_\theta$ or between Node$'_{d_0}$ and $valC_\theta$. This case is analogous to the "commit value A" case of the *Challenge Read* branch. For ValueCScript, the difference with HashWriteScript$_i$, is that: (i) CountZero $= 0$; (ii) the parent nodes are Node$_{d_0}$ and Node$'_{d_0}$, and (iii) the child node is $valC_\theta$.

P publishes CommitWrite2 transaction (analogous to CommitWrite1, but unlocking ValueCScript instead). V can publish transaction PunishWrite2 (analogous to PunishWrite1) if P equivocates on the values committed in the CommitWrite2 transaction.

(C) Commit Write Root. $\mathcal{N}_{Mer} = 0$, the point of disagreement is between $MR_{\mathcal{N}}$ and Node$_{d_0}$ or between $MR_{\mathcal{N}'}$ and Node$'_{d_0}$. This case is analogous to the "commit read root" case of the *Challenge Read* branch. The script RootWriteScript$_i$, with i $\in \{0, \ldots, 31\}$ is the same as RootReadScript$_i$ but takes as additional inputs Node$'_{d_0}$, MR_{i+1} (their public key are hard-coded in the script accordingly), and takes as input c_{addrC_θ} instead of c_{addrA_θ}.

In RootWriteScript$_i$, instead of lines 15 to 24 of Algorithm 18 the code is the one in Algorithm 20. V can punish P if they equivocate on Node$_{d_0}$, Node$'_{d_0}$, MR_i, MR_{i+1}, $addrC_\theta$.

Algorithm 20 The script RootWriteScript$_i$. In the setup phase, the public keys pk$_{\mathcal{N}_{Mer}}$, pk$_{\mathcal{N}}$, pk$_{Node_{d_0}}$, pk$_{MR_i}$ are hard-coded in the script.

1: **function** RootWriteScript$_i$
2:     ▷ ... ◁
3:     **if** $v_{pos} = 0$ **then**
4:         **if** $H($Node$_{d_0}||$Nsib$) = MR_i \wedge H($Node$'_{d_0}||$Nsib$) = MR_{i+1}$ **then**
5:             **return** True;
6:         **else**
7:             **return** False;
8:     **else**
9:         **if** $H($Nsib$||$Node$_{d_0}) = MR_i \wedge H($Nsib$||$Node$'_{d_0} = MR_{i+1})$ **then**
10:             **return** True;
11:         **else**
12:             **return** False.

Appendix B.
Bisection game

In this section, we formally describe the bisection games that the prover and verifier play interactively during the *dispute*, *challenge read*, and *challenge write* subphases of the BitVM protocol. These are referred to as the *dispute bisection game*, *read bisection game*, and *write bisection game*, respectively.

In general, the bisection game is played as follows. P and V each hold a sequence of values, which are assumed to

be identical. The prover makes public the first and the last elements of their sequence. If the verifier disagrees with one of these two values, V initiates a bisection game to find a *point of disagreement*, i.e. , a pair of consecutive sequence elements such that they agree on one of them and disagree on the other. Given sequences A^P and A^V, a point of disagreement is defined as a tuple $(A^P[i], A^P[i+1], A^V[i], A^V[i+1])$ such that either $A^P[i] = A^V[i] \land A^P[i+1] \neq A^V[i+1]$ or $A^P[i] \neq A^V[i] \land A^P[i+1] = A^V[i+1]$ (for brevity, we refer to such a point of disagreement as $(A[i], A[i+1])$).

The first stage of the game is called *disagreement phase*: the game progresses as the prover responds to the verifier's challenges by revealing specific elements of their sequence. A response consists of publishing an on-chain transaction with a commitment to a sequence element in the witness, while a challenge consists of publishing a transaction with a commitment to a bit, indicating which element should be revealed next.

After a point of disagreement has been found, the dispute bisection game ends, while the read and write bisection games proceed to the *solve phase*. At the end of the solve phase, either P or V is declared the winner, and the other one is declared the loser of the bisection game.

For brevity and readability, we use a shorthand notation for transactions that abstracts away all but the fundamental components needed to present the bisection game. Specifically, if party $A \in \{P, V\}$ wants to publish a transaction with m variables $v_1, \ldots, v_m$ as part of the witness, and for n of them they want to publish their Lamport commitment as well, we express this by writing $\mathsf{Tx}^A[\{v_1, \ldots, v_n\}, v_{n+1}, \ldots, v_m]$. Furthermore, we assume that every transaction that we describe in this section has a timelock mechanism that punishes inactivity (as we explain in Section 5.1).

B.1. Dispute bisection game

Disagreement phase. P and V play the dispute bisection game to find a point of disagreement in their VM execution traces. P runs $\mathsf{DisagreeP}(ExecTrace^P, |ExecTrace^P|)$ (cf. Algorithm 21, lines 1 to 14), where $ExecTrace^P$ is the VM execution trace of the VM instance Γ^P run by the prover during the BitVM protocol. V runs $\mathsf{DisagreeV}(ExecTrace^V, |ExecTrace^V|)$ (cf. Algorithm 21, lines 15 to 32).

B.2. Read bisection game

Disagreement phase. P and V play this phase of the read bisection game to find a point of disagreement in the path from the root to $M_{\mathcal{N}}[addr A_\theta]$ in the merkle tree of the memory $M_{\mathcal{N}}$. P runs $\mathsf{DisagreeP}(\mathcal{P}_R^P, |\mathcal{P}_R^P|)$ (cf. Algorithm 21, lines 1 to 14), where $\mathcal{P}_R^P := (MR_{\mathcal{N}}^P, \ldots, M_{\mathcal{N}}^P[addr A_\theta])$. The algorithm outputs the index of a point of disagreement. V runs $\mathsf{DisagreeV}(\mathcal{P}_R^V, |\mathcal{P}_R^P|)$ (cf. Algorithm 21, lines 15 to 32). The algorithm outputs the index of a point of disagreement.

Algorithm 21 DisagreeP and DisagreeV are the algorithms run by P and V as they interact with each other through the ledger L through the dispute/read bisection game. The variable I_P denotes the prover's sequence and n denotes its length. Likewise, the variable I_V denotes the verifier's sequence and n denotes its length.

1: **function** DisagreeP(I_P, n)
2:     $l \leftarrow 1$;   ▷ *Left search boundary*
3:     $r \leftarrow n$;   ▷ *Right search boundary*
4:     $i \leftarrow 1$;   ▷ *Counter*
5:     **while** $l + 1 < r$ **do**
6:         $m \leftarrow \lfloor \frac{l+r}{2} \rfloor$;
7:         Publish $\mathsf{Tx}_i^P[\{I_P[m]\}]$ on L;
8:         Wait until $\mathsf{Tx}_i^V[\{b_i\}]$ appears in L, where b_i is part of the witness of transaction Tx_i^V published by V. Then, fetch b_i from $\mathsf{Tx}_i^V[\{b_i\}]$;
9:         **if** $b_i = 0$ **then**
10:             $r \leftarrow m$;
11:         **else**
12:             $l \leftarrow m$;
13:         $i \leftarrow i + 1$;
14:     **return** $r - 1$.

15: **function** DisagreeV(I_V, n)
16:     $l \leftarrow 1$;   ▷ *Left search boundary*
17:     $r \leftarrow n$;   ▷ *Right search boundary*
18:     $i \leftarrow 1$;   ▷ *Counter*
19:     **while** $l + 1 < r$ **do**
20:         Wait until $\mathsf{Tx}_i^P[\{I_P[m]\}]$ appears in L, where $I_P[m]$ is part of the witness of transaction Tx_i^P published by P. Then, fetch $I_P[m]$ from $\mathsf{Tx}_i^P[\{I_P[m]\}]$;
21:         $m \leftarrow \lfloor \frac{l+r}{2} \rfloor$;
22:         **if** $I_P[m] \neq I_V[m]$ **then**   ▷ *Disagreement*
23:             $b_i \leftarrow 0$;
24:         **else**
25:             $b_i \leftarrow 1$;
26:         Publish $\mathsf{Tx}_i^V[\{b_i\}]$ on L;
27:         **if** $b_i = 0$ **then**
28:             $r \leftarrow m$;▷ *Challenge the left half of at the next step*
29:         **else**
30:             $l \leftarrow m$;▷ *Challenge the right half of at the next step*
31:         $i \leftarrow i + 1$;
32:     **return** $r - 1$.

Solve phase. Let $(\mathsf{Npar}, \mathsf{Nchild})$ be the point of disagreement identified by P and V during the disagreement phase. In the read bisection game, a point of disagreement is a pair of intermediate Merkle tree nodes, where one is the parent of the other. P runs $\mathsf{SolveReadP}(\mathsf{Npar}^P, \mathsf{Nchild}^P, \mathsf{Nsib}, v_{pos})$ (cf. Algorithm 22, lines 1 to 7). In the algorithm, P asserts that Nchild^P is the left or right child of Npar^P by setting the bit v_{pos} to 0 or 1, respectively. To do so, P provides a sibling node Nsib. P publishes the transaction $\mathsf{CommitRead}^P$, where they provide a commitment for $\mathsf{Npar}^P, \mathsf{Nchild}^P, v_{pos}^P$. V runs $\mathsf{SolveReadV}(.)$ (cf. Algorithm 22, lines 8 to 14), where V publishes the transaction $\mathsf{PunishRead}^V$ if P equivocated on any of the values published as part of the witness of $\mathsf{CommitRead}^P$.

Notice that P does not risk equivocation only if Nchild^P is a real child node of Npar^P, meaning that the leaf that they provided in $M_{\mathcal{N}}^P[addr A_\theta]$ is really the $addr A_\theta$-th leaf of the

Merkle tree with root $MR_{\mathcal{N}}^P$.

Algorithm 22 SolveReadP and SolveReadV are the algorithms executed by P and V, respectively, as they interact through the ledger L to resolve the disagreement in the read bisection game in favor of either P or V. The variables Npar, Nchild, and Nsib represent a triple, where Npar is the parent node in a Merkle tree, and Nchild and Nsib are the child nodes, with Nchild being the left or right child based on the bit v_{pos}.

1: **function** SolveReadP(Npar, Nchild, Nsib, v_{pos})
2:     **if** $v_{pos} = 0$ **then**
3:         **if** $H($Nchild$||$Nsib$) =$ Npar **then**
4:             Publish CommitRead^P[{Npar, Nchild, v_{pos}}, Nsib] on L.
5:     **else**
6:         **if** $H($Nsib$||$Nchild$) =$ Npar **then**
7:             Publish CommitRead^P[{Npar, Nchild, v_{pos}}, Nsib] on L.

8: **function** SolveReadV(.)
9:     Wait until CommitRead^P[{Npar, Nchild, v_{pos}}, Nsib] appears in L, where Npar, Nchild, v_{pos}, Nsib is part of the witness of transaction CommitRead^P published by P.
10:     **if** there is a bit b of Npar, Nchild, v_{pos} for which there are two different commitments **then**
11:         ▷ *Recall that V cannot forge such commitments if P has not equivocated* ◁
12:         Let c_0 be the commitment for $b = 0$;
13:         Let c_1 be the commitment for $b = 1$;
14:         Publish PunishRead^V[c_0, c_1] on L.

The winning conditions of the prover and the verifier for the read bisection game are shown in Fig. 3.

B.3. Write bisection game

Disagreement phase. Let $\mathcal{P}_W := (MR_{\mathcal{N}}^P, \ldots, M_{\mathcal{N}}^P[addrC_\theta])$ be the path from the root to $M_{\mathcal{N}}^P[addrC_\theta]$ in the merkle tree of the memory $M_{\mathcal{N}}^P$. Let $\mathcal{P}'_W := (MR_{\mathcal{N}'}^P, \ldots, M_{\mathcal{N}'}^P[addrC_\theta])$ be the path in the merkle tree of the memory $M_{\mathcal{N}'}^P$ from the root to $M_{\mathcal{N}'}^P[addrC_\theta]$.

P runs DisagreeWriteP($\mathcal{P}_W^P, \mathcal{P}'_W{}^P, |\mathcal{P}_W^P|$), which returns the index of a point of disagreement. V runs DisagreeWriteV($\mathcal{P}_W^V, \mathcal{P}'_W{}^V, |\mathcal{P}_W^V|$), which returns the index of a point of disagreement.

In the disagreement phase, the verifier seeks a point of disagreement in the path $\mathcal{P}'_W$, given that P and V agree on the sequence $\mathcal{P}_W$. However, as soon as the verifier identifies a disagreement in $\mathcal{P}_W$ (cf. Algorithm 23, l. 24), V shifts focus to finding a point of disagreement within $\mathcal{P}_W$. From this point forward, V disregards the elements of $\mathcal{P}'_W$ published by P and considers only the elements of $\mathcal{P}_W$ to determine how to set the bit b_i.

Solve phase. The point of disagreement is either the pair (Npar, Nchild) or the pair (Npar$'$, Nchild$'$). P runs SolveWriteP(Npar^P, Nchild^P, Npar$'^P$, Nchild$'^P$, Nsib, v_{pos}) (cf. Algorithm 24, Lines 1 to 7). In the algorithm, P asserts

---

Algorithm 23 DisagreeWriteP and DisagreeWriteV are the algorithms run by P and V as they interact with each other through the ledger L through the write bisection game. The variable I_P denotes the prover's sequence and n denotes its length. Likewise, the variable I_V denotes the verifier's sequence and n denotes its length.

1: **function** DisagreeWriteP(I_P, I'_P, n)
2:     $l \leftarrow 1$;   ▷ *Left search boundary*
3:     $r \leftarrow n$;   ▷ *Right search boundary*
4:     $i \leftarrow 1$;   ▷ *Counter*
5:     **while** $l + 1 < r$ **do**
6:         $m \leftarrow \lfloor \frac{l+r}{2} \rfloor$;
7:         Publish $\mathsf{Tx}_i^P[\{I_P[m], I'_P[m]\}]$ on L;
8:         Wait until $\mathsf{Tx}_i^V[\{b_i\}]$ appears in L, where b_i is part of the witness of transaction Tx_i^V published by V. Then, fetch b_i from $\mathsf{Tx}_i^V[\{b_i\}]$;
9:         **if** $b_i = 0$ **then**
10:             $r \leftarrow m$;
11:         **else**
12:             $l \leftarrow m$;
13:         $i \leftarrow i + 1$;
14:     **return** $r - 1$.

15: **function** DisagreeWriteV(I_V, I'_V, n)
16:     $l \leftarrow 1$;   ▷ *Left search boundary*
17:     $r \leftarrow n$;   ▷ *Right search boundary*
18:     $i \leftarrow 1$;   ▷ *Counter*
19:     $flag \leftarrow$ False;
20:     **while** $l + 1 < r$ **do**
21:         Wait until $\mathsf{Tx}_i^P[\{I_P[m], I'_P[m]\}]$ appears in L, where $I_P[m], I'_P[m]$ is part of the witness of transaction Tx_i^P published by P. Then, fetch $I_P[m], I'_P[m]$ from $\mathsf{Tx}_i^P[\{I_P[m], I'_P[m]\}]$;
22:         $m \leftarrow \lfloor \frac{l+r}{2} \rfloor$;
23:         **if** $flag =$ False **then**
24:             **if** $I_P[m] \neq I_V[m]$ **then**
25:                 $flag =$ True;▷ *From now on, look only for disagreement on I_V*
26:                 $b_i \leftarrow 0$;
27:             **else**
28:                 **if** $I'_P[m] = I'_V[m]$ **then**
29:                     $b_i \leftarrow 0$;
30:                 **else**
31:                     $b_i \leftarrow 1$;
32:         **else**
33:             **if** $I_P[m] \neq I_V[m]$ **then**
34:                 $b_i \leftarrow 0$;
35:             **else**
36:                 $b_i \leftarrow 1$;
37:         Publish $\mathsf{Tx}_i^V[\{b_i\}]$ on L;
38:         **if** $b_i = 0$ **then**
39:             $r \leftarrow m$;▷ *Challenge the left half of at the next step*
40:         **else**
41:             $l \leftarrow m$;▷ *Challenge the right half of at the next step*
42:         $i \leftarrow i + 1$;
43:     **return** $r - 1$.

Verifier wins. The verifier wins once one of these events happens:

1) During the execution of DisagreeP (DisagreeWriteP) algorithm, P fails to publish Response$_i$ transaction within Δ rounds after Challenge$_i$ transaction has been published.
2) During the execution of SolveReadP (SolveWriteP) algorithm, P fails to publish CommitRead (CommitWrite) transaction within Δ rounds after the last tx Challenge$_i$ has been published.
3) V publishes PunishRead (PunishWrite) transaction.

Prover wins. The prover wins once one of these events happens:

1) During the execution of DisagreeV (DisagreeWriteV) algorithm, V fails to publish Challenge$_i$ transaction within Δ rounds after Response$_i$ transaction has been published.
2) During the execution of SolveReadV (SolveWriteV) algorithm, V fails to publish PunishRead (PunishWrite) transaction within Δ rounds after CommitRead (CommitWrite) transaction has been published.

Figure 3. The conditions that determine the winner of the read bisection game (write bisection game).

that Nchild^P and Nchild$'^P$ are the left or right child of the nodes Npar^P and Npar$'^P$, respectively, based on the bit v_{pos} (similar to the read bisection game). P demonstrates this by providing a node Nsib that serves as the sibling of both Nchild^P and Nchild$'^P$. P publishes the transaction CommitWrite^P, where they provide a commitment for Npar^P, Nchild^P, Npar$'^P$, Nchild$'^P$, v_{pos}^P. Intuitively, the prover can commit to all the aforementioned elements without equivocating only if they previously committed to $MR_{\mathcal{N}'}^P$ and $M_{\mathcal{N}'}^P[addrC_\theta]$ such that $M_{\mathcal{N}'}^P[addrC_\theta]$ is really the $addrC_\theta$-th leaf of the Merkle tree with root $MR_{\mathcal{N}'}^P$.

V runs SolveWriteV(.)(cf. Algorithm 24, lines 8 to 14), where V publishes the transaction PunishWrite^V if P equivocated on any of the values published as part of the witness of CommitWrite^P.

The winning conditions of the prover and the verifier for the write bisection game are the same as the read bisection game, thus in Fig. 3.

Appendix C.
Extensive Form Games with Perfect Information

We introduce the concept of Extensive Form Games (EFG) as follows. In an EFG, a game tree encapsulates all possible protocol executions, with nodes representing players' decision points, branches indicating possible actions, and leaves denoting the utility outcomes associated with chosen strategies.

Definition 9 (Extensive Form Game-EFG). An Extensive Form Game (EFG) is a tuple $\mathcal{G} = (N, H, P, u)$, where set N represents the game player, the set H captures EFG game history, $T \subseteq H$ is the set of terminal

---

Algorithm 24 SolveWriteP and SolveWriteV are the algorithms executed by P and V, respectively, as they interact through the ledger L to resolve the disagreement in the write bisection game in favor of either P or V. The variables Npar, Nchild, Npar', Nchild', and Nsib represent two triples, where Npar and Npar' are the parent nodes in a Merkle tree, with Nchild, Nsib and Nchild', Nsib as the child nodes, respectively. The nodes Nchild, Nchild' are the left or right children based on the bit v_{pos}.

1: **function** SolveWriteP(Npar, Nchild, Npar', Nchild', Nsib, v_{pos})
2:     **if** $v_{pos} = 0$ **then**
3:         **if** $(H(\text{Nchild}||\text{Nsib}) = \text{Npar}) \wedge (H(\text{Nchild}'||\text{Nsib}) = \text{Npar}')$ **then**
4:             Publish CommitWrite^P[{Npar, Nchild, Npar', Nchild', v_{pos}}, Nsib] on L.
5:     **else**
6:         **if** $(H(\text{Nsib}||\text{Nchild}) = \text{Npar}) \wedge (H(\text{Nsib}||\text{Nchild}') = \text{Npar}')$ **then**
7:             Publish CommitWrite^P[{Npar, Nchild, Npar', Nchild', v_{pos}}, Nsib] on L.

8: **function** SolveWriteV(.)
9:     Wait until CommitWrite^P[{Npar, Nchild, Npar', Nchild', v_{pos}}, Nsib] appears in L, where Npar, Nchild, Npar', Nchild', v_{pos}, Nsib is part of the witness of transaction CommitWrite^P published by P.
10:     **if** there is a bit b of Npar, Nchild, Npar', Nchild', v_{pos} for which there are two different commitments **then**
11:         ▷ *Recall that V cannot forge such commitments if P has not equivocated* ◁
12:         Let c_0 be the commitment for $b = 0$;
13:         Let c_1 be the commitment for $b = 1$;
14:         Publish PunishWrite^V[c_0, c_1] on L.

---

histories, P denotes the next player function, and u is the utility function. The following properties are satisfied.

(A) The set H of histories is a set of sequence actions with

1) $\emptyset \in H$;
2) if the action sequence $(a_k)_{k=1}^K \in H$ and $L < K$, then also $(a_k)_{k=1}^L \in H$;
3) an action sequence is terminal $(a_k)_{k=1}^K \in T$, if there is no further action a_{K+1} that $(a_k)_{k=1}^{K+1} \in H$.

(B) The next player function P

1) assigns the next player $p \in N$ to every non-terminal history $(a_k)_{k=1}^K \in H \setminus T$;
2) after a non-terminal history h, it is player $P(h)$'s turn to choose an action from the set $A(h) = \{a : (h, a) \in H\}$.

A player p's strategy is a function σ_p mapping every history $h \in H$ with $P(h) = p$ to an action from $A(h)$. Formally,

$$\sigma_p : \{h \in H : P(h) = p\} \to \{a : (h, a) \in H, \forall h \in H\},$$

such that $\sigma_p(h) \in A(h)$.

A subgame of an EFG is defined as a subtree rooted at a specific history node, representing the last decision point in that sequence of actions.

Definition 10 (EFG subgame). The subgame of an EFG $\varphi = (N, H, P, u)$ associated to history $h \in H$ is the EFG $\varphi(h) = (N, H_{|h}, P_{|h}, u_{|h})$ defined as follows: $H_{|h} := h'|(h, h') \in H$, $P_{|h}(h') := P(h, h')$, and $u_{|h}(h') := u(h, h')$.

The core concept of our proof methodology is to demonstrate that utility-maximizing players will choose to adhere to the protocol specification at each step of the protocol. We further show that this implies rational parties will follow the optimistic path of BitVM. This is accomplished by leveraging the notion of a Subgame Perfect Nash Equilibrium (Definition 11) [31]. Specifically, we show that the strategy profile encompassing the "correct protocol execution" of BitVM constitutes an SPNE of our game, using a technique known as backward induction.

In backward induction, we evaluate each decision point by traversing backwards the EFG, i.e., starting from the final outcomes and moving backward to the initial decision. At each step, the player selects the action that maximizes their utility, assuming that subsequent players will also choose optimal actions in response. This process continues up the game tree until the root is reached, yielding a sequence of optimal strategies that together form a Subgame Perfect Nash Equilibrium.

Definition 11 (Subgame Perfect Nash Equilibrium (SPNE)). A subgame perfect equilibrium strategy is a joint strategy $\sigma = (\sigma_1, ..., \sigma_n) \in S$, s.t. $\sigma_{|h} = (\sigma_{1|h}, ..., \sigma_{n|h})$ is a Nash Equilibrium of the subgame $\varphi(h)$, for every $h \in H$. The strategies $\sigma_{i|h}$ are functions that map every $h' \in H_{|h}$ with $P_{|h}(h') = i$ to an action from $A_{|h}(h')$.

Appendix D.
Security Analysis

Notation and Assumptions. We denote by N_1 the number of execution steps of the VM and by N_2 the size of the memory. For convenience, we denote the logarithm of a quantity x as $\tilde{x} = log(x)$ and the nested logarithmic value as $\tilde{\tilde{x}} = log(log(x))$.

Moreover, we denote the balance account of a user $A \in \{P, V\}$ by $\langle u \rangle_A$, meaning that there are u coins to the account associated with A. We consider only funds related to the execution of BitVM and assume a constant fee f for each transaction.

In the Setup phase, P locks $\mathsf{in}_P = \alpha + (2\tilde{N}_1 + 2\tilde{\tilde{N}}_2 + 7)f$ coins in the multisignature, and V $\mathsf{in}_V = \beta + (2\tilde{N}_1 + 2\tilde{\tilde{N}}_2 + 7)f$ coins. This amount ensures that if, w.l.o.g., P deviates from the protocol and the execution follows the longest path V will not lose money even if $\beta = 0$. That is because, as we will show in Lemma D.11, in the worst case $2\tilde{N}_1 + 2\tilde{\tilde{N}}_2 + 7$ transactions are posted on-chain.

Last, for the pair of utilities corresponding to any final state by the outcome mapping function we assume that $v_P + v_V \leq \mathsf{in}_P + \mathsf{in}_V - (2\tilde{N}_1 + 2\tilde{\tilde{N}}_2 + 7)f$.

D.1. Agreement phase

Lemma D.1 (Correctness of the setup). We denote by $Presigned_P$ the set of presigned transactions V handovers to P and by $Presigned_V$ the set of presigned transactions P handovers to V during the Setup phase. If $\mathtt{Setup}$ is published on chain, then:

1) *Presigned transactions availability:* P possesses all the transactions $\in Presigned_P$ along with V's signature. V possesses all the transactions $\in Presigned_V$ along with P's signature.
2) *Locking the deposit:* $\alpha + \beta + (4\tilde{N}_1 + 4\tilde{\tilde{N}}_2 + 14)f$ coins are locked in the multisig σ_{PV} of P and V.

Proof: First, since $\mathtt{Setup}$ is accepted by the miners, both P and V must have signed $\mathtt{Setup}$. Given that, the following holds:

1) P signed $\mathtt{Setup}$ only after receiving the set of transactions in $Presigned_P$ signed by V. Similarly, V signed $\mathtt{Setup}$ only after receiving the transactions in $Presigned_V$ signed by P.
2) $\mathtt{Setup}$ has an output of $\alpha + \beta + 2C$ to the multisig σ_{PV} of P and V.
$\square$

D.2. Execution phase

Lemma D.2 (P does not post $\mathtt{CommitComputation}$). If $\mathtt{Setup}$ is published on chain and $\mathtt{CommitComputation}$ is not published on chain within the timelock Δ, then it is a dominant strategy for V to claim the output of $\mathtt{Setup}$. As a result, P's balance account is $\langle 0 \rangle_P$ and V's $\langle \alpha + \beta + (4\tilde{N}_1 + 4\tilde{\tilde{N}}_2 + 12)f \rangle_v$.

Proof: P can only spend $\mathtt{Setup}$ by publishing $\mathtt{CommitComputation}$ on chain. If P does not publish $\mathtt{CommitComputation}$ after Δ when the timelock in $\mathtt{Setup}$ expires, V can claim the output of $\mathtt{Setup}$. If V claims the collateral, she spends f coins in transaction fees. Moreover, since P has already posted $\mathtt{Setup}$ on-chain, $2f$ has been spent in transaction fees in total. Therefore V's account is $u_1 = \langle \alpha + \beta + (4\tilde{N}_1 + 4\tilde{\tilde{N}}_2 + 12)f \rangle_V$. Otherwise, if V does not claim the collateral, her balance account is $u_2 = \langle 0 \rangle_V$. Since $u_1 > u_2$, it is a dominant strategy for V to claim the output of $\mathtt{Setup}$ when the timelock expires. $\square$

D.3. Identify Disagreement phase

D.3.1. Normal closing.

Lemma D.3 (V does not publish $\mathtt{KickOff}$ to initiate a dispute and P is supposed publish a $\mathtt{Close}$ transaction). Assume that $\mathtt{CommitComputation}$ is published on-chain and $\mathtt{KickOff}$ is not published on-chain within the timelock Δ. Moreover, consider $f(R_{final}^P, \alpha, \beta) = (f_\alpha, f_\beta)$ where R_{final}^P is the result of the computation that uniquely corresponds to MR_{final} that P has

committed in `CommitComputation` and f is the outcome mapping function, and consider `Close`$_i$ for some $i \in \{1, ..., m\}$ is the transaction that distributes the funds accordingly. Then, the following statements hold:

- If P publishes on-chain `Close`$_i$, P's balance account will be $\langle f_\alpha + (2\tilde{N}_1 + 2\tilde{N}_2 + 5.5)f \rangle_P$, and V's balance account will be $\langle f_\beta + (2\tilde{N}_1 + 2\tilde{N}_2 + 5.5)f \rangle_V$.
- If P publishes on-chain `Close`$_j$, for some $j \in \{1, ..., m\}, j \neq i$, it is dominant strategy for V to claim the coins in the multi-signature. Then, P's balance account is $\langle \alpha + \beta + (4\tilde{N}_1 + 4\tilde{N}_2 + 11)f \rangle_P$ and V's $\langle 0 \rangle_v$.
- If none of transactions in the set $\mathcal{S} = \cup_{\{1,...,m\}} \text{Close}_i$ is published on-chain within 2Δ since `CommitInstruction` was published, it is dominant strategy for V to claim the coins in the multi-signature. Then, V's balance account is $\langle \alpha + \beta + (4\tilde{N}_1 + 4\tilde{N}_2 + 11)f \rangle_V$ and P's $\langle 0 \rangle_S$.

Proof: Since `CommitComputation` is posted on chain, the transaction `Setup` must have been previously published. That is because `CommitComputation` spends `Setup`. Therefore $2f$ of the coins in the multisignature have already been spent in transaction fees.

- `Close`$_i$ redistributes the rest of the coins to the parties according to the outcomes mapping function f, namely $(f_\alpha + (2\tilde{N}_1 + 2\tilde{N}_2 + 5.5)f)$ coins to P and $(f_\beta + (2\tilde{N}_1 + 2\tilde{N}_2 + 5.5)f)$ coins to V, where $f(S_{final}, \alpha, \beta) = (f_\alpha, f_\beta)$. Since the result R^P_{final} corresponds to MR_{final} V cannot spend `Close`$_i$.
- Since $i \neq j$ and each transaction in $\mathcal{S}$ uniquely corresponds to an outcome of the computation, in `Close`$_j$ P commits to an $MR'_{\text{final}} \neq MR_{\text{final}}$. V can show the equivocation of P by providing the conflicting commitments since for each $k \in \{1, ..., m\}$ the Script `CloseScript`$_i$ (Algorithm 8) has the same hard-code keys with `CommitComputationScript`. As a result P will have a balance $u_1 = \langle \alpha + \beta + (4\tilde{N}_1 + 4\tilde{N}_2 + 10)f \rangle_V$, since $4f$ are spent in the transaction fees for `Setup`, `CommitComputation`, `Close`$_j$ and the transaction sending the collateral to their account. In this scenario, P's balance account is $\langle 0 \rangle_P$. Otherwise, if V does not utilize the timelock, their balance account is $u_2 = \langle 0 \rangle_P$. Since $u_1 > u_2$ it is a dominant strategy for V to use the timelock.
- P can only spend `CommitComputation` by posting exactly one transaction in $\mathcal{S}$, otherwise V can utilize the timelock after 2Δ. Since P any transaction in $\mathcal{S}$, P can claim the coins of the multisig after the timelock 2Δ, leading to a balance account $u_1 = \langle \alpha + \beta + (4\tilde{N}_1 + 4\tilde{N}_2 + 11)f \rangle_V$, since $3f$ are spent in the transaction fees for `Setup`, `CommitComputation`, and the transaction sending the collateral to their account. In this scenario, P's balance account is $\langle 0 \rangle_P$. Otherwise, if V does not utilize the timelock, their balance account is $u_2 = \langle 0 \rangle_P$.

Since $u_1 > u_2$ it is a dominant strategy for V to use the timelock. $\square$

D.3.2. Dispute bisection game.

Lemma D.4 (P is inactive during the challenge path). Consider a set of transactions $\{tx\} \subseteq \{\text{KickOff}\} \cup \{\text{TraceChallenge}_i\}_{i \in \{1,...,\tilde{N}_1\}}$, where $\{tx\} \neq \emptyset$, is published on-chain and one of the following scenarios is true:
1) `KickOff` $\in \{tx\}$ (let $j = 0$) and P has not posted `TraceResponse`$_1$ within Δ,
2) `TraceChallenge`$_i \in \{tx\}$ for $i < \tilde{N}_1$ (let $j = i$) and P has not posted `TraceResponse`$_{i+1}$ within Δ,
3) `TraceChallenge`$_{\tilde{N}_1} \in \{tx\}$ (let $j = \tilde{N}_1$) and P has not posted `CommitInstruction` within Δ.

Then, it is a dominant strategy for V to utilize the timelock. P's balance account is then $\langle 0 \rangle_P$, and V's balance account is $\langle \alpha + \beta + (4\tilde{N}_1 + 4\tilde{N}_2 - 2j + 10)f \rangle_V$.

Proof: Since $\{tx\} \neq \emptyset$, `KickOff` has been published on-chain, P has previously published on-chain `Setup`, and `CommitComputation`, which cost $3f$ in transaction fees. If scenario 1 is true and V utilizes the timelock to claim the output of `KickOff`, which costs another f in fees (thus $4f$ in total), her balance account is $u_1 = \langle \alpha + \beta + (4\tilde{N}_1 + 4\tilde{N}_2 + 10) \rangle_V$, and P's balance account is $\langle 0 \rangle_P$. If V does not utilize the timelock, her balance account is $u_2 = \langle 0 \rangle_v < u_1$, which shows that it is a dominant strategy for her to utilize the timelock.

Otherwise, if scenario 2 or 3 is true, P has posted on-chain before $\{\text{TraceResponse}_k\}_{k \in \{1,...,j\}}$ paying extra jf in transaction fees, and V, in turn, has posted $\{\text{TraceChallenge}_k\}_{k \in \{1,...,j\}}$ paying jf in fees too. Now, if V utilizes the timelock, her balance account is $\langle u_1 = \alpha + \beta + (4\tilde{N}_1 + 4\tilde{N}_2 - 2j + 10)f \rangle_V$ and P's balance account is $\langle 0 \rangle_P$. Otherwise, if V does not utilize the timelock, V's balance account is $u_2 = \langle 0 \rangle_V < u_1$, which again shows that it is a dominant strategy for her to use the timelock. $\square$

Lemma D.5 (V is inactive during the challenge path). Consider a set of transactions $\{tx\} \subseteq \{\text{TraceChallenge}_i\}_{i \in \{1,...,\tilde{N}_1\}} \cup \{\text{CommitInstruction}\}$, where $\{tx\} \neq \emptyset$, is published on-chain and one of the following scenarios is true:
1) `TraceResponse`$_i \in \{tx\}$ for some $i \leq \tilde{N}_1$ (let $j = i$), and V has not posted `TraceChallenge`$_i$ within Δ,
2) `CommitInstruction` $\in \{tx\}$ (let $j = \tilde{N}_1 + 1$) and V has not posted any transaction $tx' \in \{\text{ChallengeCurrPC}, \text{PunishInstruction}, \text{ChallengeRead}, \text{ChallengeWrite}\}$ within Δ.

Then, it is a dominant strategy for P to utilize the timelock. P's balance account is then $\langle \alpha + \beta + (4\tilde{N}_1 + 4\tilde{N}_2 - 2j + 11)f \rangle_P$, and V's balance account is $\langle 0 \rangle_V$.

Proof: Since `TraceResponse`$_i$ is posted on-chain, `Setup`, `Close`, `KickOff` are posted on-chain as well. Moreover, if $i > 1$ $\{\text{TraceResponse}_k\}_{k \in \{1,...,i-1\}}$ and

$\{\texttt{TraceChallenge}_{\texttt{k}}\}_{k \in \{1,...,i-1\}}$ are also published on-chain. More specifically, first, P committed the $\texttt{Setup}$ and the $\texttt{Close}$. Then, V posted $\texttt{KickOff}$. Furthermore, if $i > 1$, P has also published on-chain $\{\texttt{TraceResponse}_{\texttt{k}}\}_{k \in \{1,...,i-1\}}$ and V the respective $\{\texttt{TraceChallenge}_{\texttt{k}}\}_{k \in \{1,...,i-1\}}$. This results in $(2j+2)f$ in transaction fees. Now, if Scenario 2 is true, then V has published on-chain $\texttt{TraceChallenge}_{\tilde{N_1}}$ which P spent by publishing on-chain $\texttt{CommitInstruction}$. In both scenarios, 1 and 2, P spends extra f in transaction fees to claim the collateral. Therefore, P's balance account will now be $u_1 = \langle \alpha + \beta + (4\tilde{N_1} + 4\tilde{N_2} - 2j + 11) \rangle_P$. In that case, V's balance account is $\langle 0 \rangle_V$. Otherwise, P's balance account is $u_2 = \langle 0 \rangle_P < u_1$, which means that it is a dominant strategy for P to use the timelock. $\qquad \square$

Lemma D.6. Consider parties P and V play the bisection game on-chain described in Algorithm 21. P runs the function $\mathsf{DisagreeP}(\mathcal{A}, \mathsf{n})$ where $\mathcal{A}$ is a sequence of n values, and V runs the function $\mathsf{DisagreeV}(\mathcal{B}, \mathsf{n})$ where $\mathcal{B}$ is a sequence of n values. The following statements hold:

- if $\mathcal{A}[1] = \mathcal{B}[1]$ and $\mathcal{A}[n] \neq \mathcal{B}[n]$, the protocol pinpoints an index j such that $\mathcal{A}[j] = \mathcal{B}[j]$ and $\mathcal{A}[j+1] \neq \mathcal{B}[j+1]$ where $j \in \{1, ..., n-1\}$,
- The bisection game finishes after $O(log n)$ steps.

Proof: We denote the local variable i of P and V by i^P and i^V respectively. We say we are in the $i-$th step of the bisection game (or the loop) if $i^P = i$. Moreover, we denote the local variables l, r of P and V in the i-th step of the loop by l_i^P, r_i^P, and l_i^V, r_i^V respectively.

Precondition. The following condition holds: $\mathcal{A}[1] = \mathcal{B}[1]$, $\mathcal{A}[n] \neq \mathcal{B}[n]$, P starts with the local variables $l_0^P = 1, r_0^P = n, l_0^P < r_0^P, i^P = 1$ and V with the local variables $l_0^V = 1, r_0^V = n, l_0^V < r_0^V, i^V = 1$.

Loop invariant. We will prove that, in every step of the loop the protocol maintains the following loop invariant by induction on the number of steps.

After the i-th step of the loop, P and V have the same local variables $i^P = i^L = i$, and $l_i^P = l_i^V = l_i$, $r_i^P = r_i^l = r_i$ with $l_i < r_i$, and therefore, they continue with the subsequences $\mathcal{A}[l_i : r_i], \mathcal{B}[l_i : r_i]$ respectively. Moreover, it holds that $\mathcal{A}[l_i] = \mathcal{B}[l_i]$ and $\mathcal{A}[r_i] \neq \mathcal{B}[r_i]$.

Base Case. In the base case where $n = 2$ and $\mathcal{A}[1] = \mathcal{B}[1]$, $\mathcal{A}[2] \neq \mathcal{B}[2]$, then $l^P = l^V = l = 2$, $r^P = r^V = 2$, and since $r - l = 1$ the condition in line 19 is not satisfied so the bisection game pinpoints as the point of disagreement $j = 1$.

Induction Step. Assume that in the $i-$th step of the loop the invariant holds. So, P and V have the same local variables $i^P = i^L = i, l_i^P = l_i^V = l_i, r_i^P = r_i^V = r_i, l_i < r_i$, and for their subsequences $\mathcal{A}[l_i] = \mathcal{B}[l_i]$ and $\mathcal{A}[s_i] \neq \mathcal{B}[s_i]$ respectively. We will show that the invariant holds for the step $i + 1$.

First, P publishes on-chain the value $\mathcal{A}[m]$, where $m = \lfloor \frac{l_i + r_i}{2} \rfloor$ (line 7). When V witnesses $\mathcal{A}[m]$ on-chain, we have the following cases:

- Case 1, $\mathcal{A}[m] \neq \mathcal{B}[m]$: V sets $b_i = 0$ (line 23), publishes b_i on-chain (line 26) and updates its local variables $i^V \leftarrow i + 1$ (line 31) and $r_{i+1}^V \leftarrow m$ (line 28). As soon as P witnesses $b_i = 0$ on-chain it updates its local variables $i^P \leftarrow i + 1$ (line 13) and $r_{i+1}^P \leftarrow m$ (line 10). Moreover, since P and V entered the loop at step i, $r_i \neq l_i + 1$, and $r_i > l_i$ (by assumption), it must be $r_i \geq l_i + 2$. Therefore, $r_{i+1}^P = m = \lfloor \frac{l_i + r_i}{2} \rfloor \geq \lfloor \frac{2l_i + 2}{2} \rfloor = l_i + 1 = l_{i+1}^P + 1$. Therefore, since $l_{i+1} = l_{i+1}^P = l_{i+1}^V = l_i$, $r_{i+1} = r_{i+1}^P = r_{i+1}^l = m$, $r_{i+1} > l_{i+1}$, $i^V = i^P = i + 1$, P and V continue with the subsequences $\mathcal{A}[l_{i+1} : r_{i+1}], \mathcal{B}[l_{i+1} : r_{i+1}]$ respectively s.t. $\mathcal{A}[l_{i+1}] = \mathcal{B}[l_{i+1}]$, $\mathcal{A}[r_{i+1}] \neq \mathcal{B}[r_{i+1}]$, the invariant still holds.
- Case 2, $\mathcal{A}[m] = \mathcal{B}[m]$: V sets $b_i = 1$ (line 25), publishes b_i on-chain (line 26) and updates its local variables $l^V \leftarrow m$ (line 30) and $i^V \leftarrow i + 1$ (line 13). As soon as party P witnesses $b_i = 1$ on-chain it updates its local variables $l^P \leftarrow m$ (line 12) and $i^P \leftarrow i + 1$ (line 13). Moreover, since P and V entered the loop at step i, $l_i \leq r_i - 2$ and by assumption $r_i > l_i$, which means that $l_{i+1} = m = \lfloor \frac{l_i + r_i}{2} \rfloor \leq \lfloor \frac{2r_i - 2}{2} \rfloor = r_i - 1$. Since $r_{i+1} = r_i$, it holds that $r_{i+1} > l_{i+1}$.
  Again, since, $l^P = l^V = l_{i+1}, r^P = r^l = r_{i+1}, r_{i+1} > l_{i+1}, i^V = i^P = i+1$ and both parties continue with the subsequences $\mathcal{A}[l_{i+1} : r_{i+1}], \mathcal{B}[l_{i+1} : r_{i+1}]$, such that $\mathcal{A}[l_{i+1}] = \mathcal{B}[l_{i+1}], \mathcal{A}[r_{i+1}] \neq \mathcal{B}[r_{i+1}]$ the invariant still holds.

Termination: In every step of the bisection game the interval of the sequences of P and V remains the half. Moreover, after the step i of the loop for the local variables of P and V it holds that $l_{i+1}^P = l_{i+1}^V = l_{i+1}$, $r_{i+1}^P = r_{i+1}^l = r_{i+1}$, $r_{i+1} > l_{i+1}$. Since, the subsequences are decreasing to the half after every step i and $r_{i+1} > l_{i+1}$, after $O(log n)$ steps the algorithm will pinpoint the point of disagreement. $\qquad \square$

Lemma D.7 (P has committed to the wrong state and V initiates a dispute). Assume that P has committed to an execution trace E_{final}^P in $\texttt{CommitComputation}$ different than the VM execution trace element at step $final$, i.e., $E_{final}^P \neq E_{final}$. Assume that V follows the protocol specifications, publishes on-chain $\texttt{Kickoff}$ and P and V run Algorithm 21. Algorithm 21 outputs a step $\mathcal{N}$ such that P has committed to the execution traces $E_{\mathcal{N}}^P = E_{\mathcal{N}}$ at step $\mathcal{N}$ and $E_{\mathcal{N}+1}^P \neq E_{\mathcal{N}+1}$ at step $\mathcal{N} + 1$.

Proof: Let $\mathcal{I}$ be the set which consists of i) all the VM steps to which P has committed to an execution trace on-chain (line 7), ii) step $i = 1$, for which P has committed to $E_0^P]$, and $i = final$ for which P has committed to E_{final}^P in $\texttt{CommitInstruction}$.

By assumption V follows the protocol specification and, therefore, runs the function $\mathsf{DisagreeV}(\mathcal{B}, final)$ of Algorithm 21, where the sequence $\mathcal{B}$ consists of the VM execution trace element at each step, i.e., $\forall i \in \{1, ..., final\} : \mathcal{B}[i] = E_i$.

P runs the function $\mathsf{DisagreeP}(\mathcal{A}, final)$ of Algorithm 21, where the sequence of values $\mathcal{A}$ is constructed

as follows. For every $i \in \mathcal{I}$, $\mathcal{A}[i] = E_{final}^P$, i.e., $\mathcal{A}[i]$ is the execution trace to which P has committed on-chain for step i. For the rest indices, $i \in \{1, ..., final\} \setminus \mathcal{I}$, without loss of generality, we assume that $\mathcal{A}[i] = E_i$, i.e, $\mathcal{A}[i]$ is the correct VM execution trace element at step i.

By assumption $\mathcal{A}[1] = \mathcal{B}[1]$ and $\mathcal{A}[final] \neq \mathcal{B}[final]$, and therefore according to Lemma D.6 the bisection game outputs a step $\mathcal{N}$ such that $\mathcal{A}[\mathcal{N}] = \mathcal{B}[\mathcal{N}]$ and $\mathcal{A}[\mathcal{N}+1] \neq \mathcal{B}[\mathcal{N}+1]$. $\qquad\square$

D.4. Punishment phase

In this section, we consider the case where P has posted on-chain `CommitInstruction` along with the witness, which consists of the values pc_θ, $pc_{\theta'}$, $insType_\theta$, $addrA_\theta$, $addrB_\theta$, $addrC_\theta$, $valA_\theta$, $valB_\theta$, $valC_\theta$ and the respective commitments that correspond to the VM state at step θ. This means that the following arguments are true:

- Since `CommitInstruction` is accepted by the miners, `Setup` must have been first published on the chain. That is because CommitInstruction spends `TraceChallenge`$_{\tilde{N}_1}$, which can only exist on-chain if `Setup` has previously been published on-chain too. Therefore, according to Lemma D.1, P has presigned and sent to V the transactions `ChallengeCurrPC`, (or `ChallengeNextPC`), `PunishInstruction`, `ChallengeRead`, `ChallengeWrite`.
- Moreover, during the dispute bisection game V has committed to the bits $b_j \in \{0, 1\}$, $j \in \{0, ..., \tilde{N}_1 - 1\}$ which form the $\tilde{N}_1$-bit integer $\mathcal{N} = \sum_{j=0}^{\tilde{N}_1-1} 2^j b_j$.
- All the presigned transaction assiociated with the resolve dispute phase are available to the parties according to Theorem D.1.

Lemma D.8 (Inconsistent program counter). Consider that V publishes on-chain the transaction `ChallengeCurrPC`, committing to a number $N' \in \{0, ..., N_1\}$ and providing P's commitment for the program counter at a step $N^* \in \{0, ..., N_1\}$, $pc_{N^*} \neq pc_\theta$. Then the following scenarios hold true.

- *V equivocates*: If $pc_\theta = pc_\mathcal{N}$, namely, the program counter that P included as part of the witness to `CommitInstruction` is the one he committed during the dispute bisection game at step $\mathcal{N}$, then it is a dominant strategy for P to claim the output of `CommitInstruction`. As a result, P's balance account will be $\langle \alpha + \beta + (2\tilde{N}_1 + 4\tilde{N}_2 + 8)f \rangle_P$, and V's balance account $\langle 0 \rangle_V$.
- *P misbehaved:* If $pc_\theta \neq pc_\mathcal{N}$ and $N' = \mathcal{N}$, namely the program counter that P included as part of witness to `CommitInstruction` is different than the one he committed during the dispute bisection game at step $\mathcal{N}$, then it is a dominant strategy for V to claim the output of `CommitInstruction`. As a result, P's balance account will be $\langle 0 \rangle_P$, and V's balance account $\langle \alpha + \beta + (2\tilde{N}_1 + 4\tilde{N}_2 + 9)f \rangle_V$.

Proof: Since `ChallengeCurrPC` is published on-chain, P has published on chain the set of transactions `Setup` $\cup$

`CommitComputation`, $\cup \{\texttt{TraceResponse}_i\}_{i \in \{1, ..., \tilde{N}_1\}}$ $\cup$ `CommitInstruction` and V has published `KickOff` $\cup$ $\{\texttt{TraceChallenge}_i\}_{i \in \{1, ..., \tilde{N}_1\}}$ $\cup$ `ChallengeCurrPC` so $(2\tilde{N}_1 + 5)f$ coins have been already spent in transaction fees.

Moreover, `ChallengeCurrPC` is accepted by the miners which means that V unlocked the $i-th$ spending condition of `CommitInstruction` for some $i \in \{0, ..., \tilde{N}_1 - 1\}$, by committing to the bits b_j, $j \in \{0, ..., \tilde{N}_1 - 1\}$ which form the $\tilde{N}_1$-bit integer $N' = \sum_{j=0}^{\tilde{N}_1-1} 2^j b_j$ such that $\texttt{CountZeroes}(N') = i$.

Furthermore, for P's commitment pc_{N^*} it must be that $pc_{N^*} = pc_{N'}$. Namely, V can only provide as a witness P's commitment at step N', which is the execution step V claimed they disagree. That is, because `ChallengeCurrPCi` and the i-th spending condition have the same hard-coded public key. Therefore, from all of P's commitments to program counters shared during the dispute bisection game, only the one for $pc_{N'}$ satisfies the condition in Algorithm 11, line 8 .

- *P is honest ($pc_\theta = pc_\mathcal{N}$)*: Since $pc_\theta = pc_\mathcal{N}$ and $pc_\theta \neq pc_{N^*}$ by assumption, and $pc_{N^*} = pc_{N'}$ as explained before, it must hold that $pc_\mathcal{N} \neq pc_{N'}$. Therefore $N' \neq N$, which means that V committed now to N' which is different than N, to which V committed during the dispute bisection game. Since $N' \neq N$, the two numbers must differ in at least one bit. W.l.o.g., assume the two numbers differ in the $k-th$ bit. P can spend the transaction `ChallengeCurrPC` to claim the coins in the multisignature, spending extra f in transaction fees too, showing that V equivocates by providing the secret key to the commitment of $b'_k \neq b_k$.
- *P is malicious ($pc_\theta \neq pc_\mathcal{N}$ and $N' = \mathcal{N}$)*: P cannot spend the transaction `ChallengeCurrPC`, since V included to `ChallengeCurrPC` indeed the number $\mathcal{N}$ to which she has committed before. V will do Therefore, after Δ where the timelock expires, V can claim the coins in the multisignature. $\qquad\square$

Lemma D.9 (P claims a wrong instruction). If and only if the instruction that P claims for the program counter pc_θ is the wrong instruction, i.e., $\Pi(pc_\theta) \neq (insType_\theta, addrA_\theta, addrB_\theta, addrC_\theta)$, then V can claim the output of `CommitInstruction` by publishing on-chain `PunishInstruction`. In that scenario, P's balance account is $\langle 0 \rangle_P$, and V's balance account $\langle \alpha + \beta + (2\tilde{N}_1 + 4\tilde{N}_2 + 9)f \rangle_V$.

Proof: First, since `PunishInstruction` is published on-chain, P has published on-chain the set of transactions `Setup` $\cup$ `CommitComputation`, $\cup \{\texttt{TraceResponse}_i\}_{i \in \{1, ..., \tilde{N}_1\}}$ $\cup$ `CommitInstruction` and V has published `KickOff` $\cup$ $\{\texttt{TraceChallenge}_i\}_{i \in \{1, ..., \tilde{N}_1\}}$ $\cup$ `PunishInstruction` so $(2\tilde{N}_1 + 5)f$ coins have been spent in transaction fees.

We remind that the pc_θ-th spending condition of `CIScriptPCCurr`$_i$ in `CommitInstruction` is true if and only if it receives as witness P's

commitment to the program counter pc_θ and to the tuple $(insType_\theta, addrA_\theta, addrB_\theta, addrC_\theta)$ such that $(insType_\theta, addrA_\theta, addrB_\theta, addrC_\theta) \neq \Pi(pc_\theta)$.

$\Rightarrow V$ provides P's commitments to pc_θ and $(insType_\theta, addrA_\theta, addrB_\theta, addrC_\theta)$ as witnesses to unlock the transaction PunishInstruction by spending the pc_θ-th condition of CIScriptPCCurr$_i$. Since $\Pi(pc_\theta) \neq (insType_\theta, addrA_\theta, addrB_\theta, addrC_\theta)$ by assumption, V will successfully unlock the pc_θ-th locking script and spend PunishInstruction to claim the output of CommitInstruction. Therefore, V's balance account is $\langle \alpha + \beta + (2\tilde{N}_1 + 4\tilde{N}_2 + 9)f \rangle_V$ and P's balance account $\langle 0 \rangle_P$.

$\Leftarrow$ Assume that V has managed to spend the transaction PunishInstruction. That means that V has unlocked the $pc_j - th$ spending condition for $pc_j \in \{1, ..., len(\Pi)\}$, which in turn means V provided as witness P's commitment to pc_j for which $\Pi(pc_j) \neq (instrType_j, addrA_j, addrB_j, addrC_j)$. Since the transaction PunishInstruction and CommitInstruction have the same hard-coded keys, it must be that $pc_j = pc_\theta$ since this is the only commitment at program counter which satisfies the condition in Algorithm 15, line 3. $\square$

D.4.1. Read bisection game.

Lemma D.10 (A party is inactive during the read bisection game). *Assume that V publishes on-chain the transaction* ChallengeRead *by spending the script* CIScriptRead$_A$ *(or* CIScriptRead$_B$*) of* CommitInstruction.

1) *Scenario 1, V is inactive:* Assume that ReadResponse$_i$, $i \in \{1, ..., \tilde{N}_2\}$ is published on-chain. If ReadChallenge$_i$ is not published on-chain after time Δ, then it is a dominant strategy for P to claim the coins locked in the multisignature. As a result, P's balance account is $(\alpha + \beta + (2\tilde{N}_1 + 4\tilde{N}_2 + 0 - 2i)f)_P$, and V's balance account is $(0)_V$.

2) *Scenario 2, P is inactive:* Assume that a set of transactions $\{tx\} \subseteq \{$ChallengeRead$\} \cup \{$ReadChallenge$_i\}_{i \in \{1,...,\tilde{N}_2\}}$, where $\{tx\} \neq \emptyset$, is published on-chain and one of the following scenarios is true:

   a) ChallengeRead $\in \{tx\}$ (let $j = 0$) and P has not posted ReadResponse$_1$ within Δ,

   b) ReadChallenge$_i \in \{tx\}$ for $i < \tilde{N}_2$ (let $j = i$) and P has not posted ReadResponse$_{i+1}$ within Δ,

   c) ReadChallenge$_{\tilde{N}_2} \in \{tx\}$ (let $j = \tilde{N}_2$) and P has not posted exactly one transaction $tx' \in \{$MerkleRootHash$\} \cup \{$MerkleHash$_i\}_{i \in \{1,...,\tilde{N}_2\}}$ within Δ,

   Then, it is a dominant strategy for V to claim the coins locked in the multisignature. P's balance account is then $\langle 0 \rangle_P$, and V's balance account is $\langle \alpha + \beta + (2\tilde{N}_1 + 4\tilde{N}_2 - 2j + 8)f \rangle_V$.

Proof: First, in every case, since ChallengeRead is published on-chain, P has published on-chain the set of transactions $S_P = $ Setup $\cup$ CommitComputation, $\cup \{$MerkleResponse$_i\}_{i \in \{1,...,\tilde{N}_1\}} \cup$ CommitInstruction and V has published $S_V = $ KickOff $\cup \{$MerkleChallenge$_i\}_{i \in \{1,...,\tilde{N}_1\}} \cup$ ChallengeRead so $(2\tilde{N}_1 + 5)f$ coins have been already spent in transaction fees.

1) In this scenario, P has published on-chain $S_P \cup \{$ReadResponse$_k\}_{k \in \{1,...,i\}}$, and V has published on-chain $S_V \cup S_{V'}$, where $S_{V'} = \{$ReadChallenge$_k\}_{k \in \{1,...,i-1\}}$ if $i > 0$, otherwise $V' = \emptyset$. Therefore, extra $2i - 1$ have been spent in fees. Since ReadResponse$_i$ is published on-chain and V has not published ReadChallenge$_i$ on-chain after Δ, P can activate the timelock to claim the coins locked in the multisignature. To this end, P spends extra f in transaction fees. As a result, his balance account is $u_1 = (\alpha + \beta + (2\tilde{N}_1 + 4\tilde{N}_2 + 9 - 2i)f)$ and V's account is $\langle 0 \rangle_V$. Otherwise, P's balance account is $0 < u_1$, and therefore activating the timelock is a dominant strategy.

2) In all of the scenarios Items 2a to 2c extra $2j$ have been spent in transaction fees. Moreover, since P is inactive, V can activate the timelock to claim the coins locked in the multisignature, spending an extra f in transaction fees. As a result, V's balance account is $u_1 = (\alpha + \beta + (\alpha + \beta + (2\tilde{N}_1 + 4\tilde{N}_2 + 8 - 2j)f)$ and V's account is $\langle 0 \rangle_V$. Otherwise, V's balance account is $0 < u_1$, and therefore activating the timelock is a dominant strategy. $\square$

Lemma D.11 (Read bisection game completes). *Assume that V publishes on-chain the transaction* ChallengeRead *by spending the script* CIScriptRead$_A$ *(or* CIScriptRead$_B$*) of* CommitInstruction.

1) *Scenario 1, P reads an incorrect value from the memory:* Consider $\mathcal{N}$ the number to which V has committed during the dispute bisection game. If P has committed to a correct execution trace element for step $\mathcal{N}$ in the dispute bisection game, i.e., $E_{\mathcal{N}}^P = E_{\mathcal{N}}$, and P has committed to a value $valA_\theta \neq M[addrA_\theta]$ (or to a value $valB_\theta \neq M[addrB_\theta]$), it is a dominant strategy for V to claim the coins in the multisignature. As a result, P's balance account is $\langle 0 \rangle_P$, and V's balance account is $\langle \alpha + \beta + (2\tilde{N}_1 + 2\tilde{N}_2 + 7)f \rangle_V$.

2) *Scenario 2, P follows the protocol specifications:* If P has followed the protocol specifications, P will eventually claim the coins in the multisignature. In that scenario, V's balance account is $\langle 0 \rangle_P$, and P's balance account is $\langle \alpha + \beta + (2\tilde{N}_1 + 2\tilde{N}_2 + 8)f \rangle_V$.

Proof: As explained in Lemma D.10, when ChallengeRead is published on-chain $(2\tilde{N}_1 + 5)f$ coins have been already spent in transaction fees.

1) Consider the Merkle tree of the memory at step $\mathcal{N}$ with root $MR_{\mathcal{N}}$. Moreover, consider the path π

from the root $MR_\mathcal{N}$ to $M_\mathcal{N}[addrA_\theta]$, i.e., $\pi :=$ $(MR_\mathcal{N}, \ldots, M_\mathcal{N}[addrA_\theta])$.

By assumption, V follows the protocol specification and therefore runs the function DisagreementReadP$(\mathcal{B}, \mathsf{n})$ of Algorithm 21, where the sequence $\mathcal{B}$ of length $n = \tilde{N}_2$ consists of the values of π, i.e., $\forall i \in \{1, \ldots \tilde{N}_2\}, \mathcal{B}[i] = \pi[i]$. P runs the function DisagreementReadP$(\mathcal{A}, \mathsf{n})$ of Algorithm 21, where the sequence $\mathcal{A}$ of length $n = \tilde{N}_2$ is constructed as follows. Let $\mathcal{I}$ be the set of all the nodes to which P commits on-chain (line 7) including the root $(i = 1)$, since P has committed to the root $MR_\mathcal{N}^P$ in the dispute bisection game (in the trace element $E_\mathcal{N}^P$), and the leaf of the path $(i = \tilde{N}_2)$ to which P committed in CommitInstruction, i.e., $M_\mathcal{N}^P[addrA_\theta] = valA_\theta$. For every $i \in \mathcal{I}$, $\mathcal{A}[i] = \pi^P[i]$, where by $\pi^S[i]$ we denote the nodes of level $(i - 1)$ to which P has committed on-chain. For the rest indices, $i \in \{1, \ldots, \tilde{N}_2\} \setminus \mathcal{I}$, without loss of generality we assume that $\mathcal{A}[i] = \pi[i]$, i.e, $\mathcal{A}[i]$ is the correct node of π at level $(i - 1)$. By assumption $\mathcal{A}[1] = \mathcal{B}[1]$ and $\mathcal{A}[\tilde{N}_2] \neq \mathcal{B}[\tilde{N}_2]$, and therefore according to Lemma D.6 the bisection game outputs a step $\mathcal{N}_{Mer}$ such that $\mathcal{A}[\mathcal{N}_{Mer}] = \mathcal{B}[\mathcal{N}_{Mer}]$ and $\mathcal{A}[\mathcal{N}_{Mer} + 1] \neq \mathcal{B}[\mathcal{N}_{Mer} + 1]$ and finishes in $\tilde{N}_2$ steps. Depending on the value of $\mathcal{N}_{Mer}$, P can spend the transaction ReadChallenge$_{\tilde{N}_2}$ as follows.

$\mathcal{N}_{Mer} = 0$. P can unlock the script RootReadScript$_i$ (Algorithm 18) for some $i \in \{1, \ldots, \tilde{N}_1\}$ by providing the commitment of V to $\mathcal{N}_{Mer}$ made in the disagreement phase of the read bisection game (line 3). and the commitment of V to $\mathcal{N}$, the number output in the Identify Disagreement phase (line 5), for which it must hold Count_Zeroes$(\mathsf{N}) = \mathsf{i}$. Since V follows the protocol specifications, the condition $\mathcal{N}_{Mer} = 0$ is true only when V disagrees with every node committed by P, including the node $u = B[2]$ committed in ReadResponse$_{\tilde{N}_2}$ which is on of the children of the root $\mathcal{B}[1] = MR_\mathcal{N}^P$. To unlock the script, P must provide as input three nodes (Npar, Nchild, Nsib) s.t. Npar $= \mathcal{B}[1]$, Nchild $= \mathcal{B}[2]$, and i) if Nchild is the right child of Npar then $H(\mathsf{Nchild}||\mathsf{Nsib}) = \mathsf{Npar}$ (line 16), ii) else $H(\mathsf{Nsib}||\mathsf{Nchild}) = \mathsf{Npar}$ (line 21). We enforce the position of the child Nchild as follows. We take the $\mathcal{N}_{Mer}$-th bit of the binary representation of $addrA_\theta$ which we denote by $b_{\mathcal{N}_{Mer}}$. By construction of a Merkle Tree, $b_{\mathcal{N}_{Mer}}$ defines the position of Nchild, namely if $b_{\mathcal{N}_{Mer}} = 1$ the Nchild is the right child of Npar, else Nchild is the left child child of Npar. P can only provide the wrong position of Nchild only by providing a commitment to $addrA'_\theta \neq addrA_\theta$, i.e., in which case V can prove the equivocation and publish PunishRead3 to claim the coins in the multisignature. That is because the scripts for unlocking CommitInstruction and RootReadScript$_i$ hard-code the same public key $\mathsf{pk}_{addrA_\theta}$ for $addrA_\theta$ (line 11). For the rest of the proof, we assume that P

did not equivocate at this point and that, w.l.o.g., Nchild is the right child of Npar.

Since V has followed the protocol specifications, V knows a node Nsib^V that satisfies this condition, i.e., for the input $x = \mathsf{Nsib}^V||\mathcal{A}[2]$ the hash function H returns $H(x) = \mathcal{A}[1]$. P must find a value Nsib s.t. $x' = \mathsf{Nsib}||\mathcal{B}[2] \neq x$ (since $\mathcal{B}[2] \neq \mathcal{A}[2]$), and $H(x) = \mathcal{B}[1] = H(x')$ (since $\mathcal{B}[1] = \mathcal{A}[1]$,) which can happen only with a negligible probability since H is a collision-resistant function. Therefore, to satisfy the condition, P must equivocate and provide Nchild $= \mathcal{A}[2] \neq \mathcal{B}[2]$ or Npar $= \mathcal{A}[1] \neq \mathcal{B}[1]$. In both cases V proves the equivocation and publish on-chain PunishRead3 to claim the coins in the multisignature, since the scripts ChallScript$_i$ and RootReadScript$_i$ have the same hard-coded public key for $MR_\mathcal{N}^P$ and the scripts ReadChallScript$_5$ and RootReadScript$_i$ have the same hard-coded public key for Node$_{d_0}$.

$\mathcal{N}_{Mer} \neq 0$. To spend the transaction ReadChallenge$_{\tilde{N}_2}$ P must unlock the script ValueAScript if $\mathcal{N}_{Mer} = \tilde{N}_2 - 1$, otherwise unlock the script HashReadScript$_i$ (Algorithm 16) for some i s.t. Count_Zero$(\mathcal{N}_{Mer}) = \mathsf{i}$. In both scenarios, P must provide as input three nodes in the path (Npar, Nchild, Nsib) s.t. Npar $= \mathcal{B}[j]$, Nchild $= \mathcal{B}[j + 1]$, and i) if Nchild is the right child of Npar then $H(\mathsf{Nsib}||\mathsf{Nchild}) = \mathsf{Npar}$, ii) else $H(\mathsf{Nchild}||\mathsf{Nsib}) = \mathsf{Npar}$. Again, we enforce the position of Nchild using the $\mathcal{N}_{Mer}$-th bit of the binary representation of $addrA_\theta$. W.l.o.g., we assume that Nchild is the right child of Npar.

Since V has followed the protocol specifications V knows a node Nsib^V s.t. for the input $x = \mathsf{Nsib}^V||\mathcal{A}[j+1]$ the hash function H returns $H(x) = \mathcal{A}[j]$. P must find a value Nsib s.t. $x' = \mathsf{Nsib}||\mathcal{B}[j + 1] \neq x$ (since $\mathcal{B}[j + 1] \neq \mathcal{A}[j + 1]$), and $H(x) = \mathcal{B}[j] = H(x')$ (since $\mathcal{B}[j] = \mathcal{A}[j]$,) which can happen only with a negligible probability since we assume a collision-resistant function H.

To provide such a pair P has to equivocate and therefore present a pair s.t. at least Npar $\neq \mathcal{B}[j]$ or Nchild $\neq \mathcal{B}[j + 1]$. More specifically, we have the following scenarios:

- $\mathcal{N}_{Mer} \in \{1, \ldots, \tilde{N}_2 - 2\}$: In this scenario, V can show the equivocation because the hard-coded public keys for the pair Nchild, Npar corresponding to HashReadScript$_i$ are the same to which P commits during the disagreement phase of the read bisection game.
- $\mathcal{N}_{Mer} = \tilde{N}_2 - 1$: If P equivocates on Npar, V can prove the equivocation as explained for $\mathcal{N}_{Mer} \in \{1, \ldots, \tilde{N}_2 - 2\}$. The extra condition in this situation is that the hard-coded public key of Nchild is the same with $valA_\theta$ in CommitInstruction. Therefore, if P equivocates on Nchild, V can again provide the conflicting commitments.

In the worst case, the set of transactions $\{\texttt{ReadChallenge}_i, \texttt{WriteChallenge}_i\}_{i \in \{1, \ldots, \tilde{N}_2\}}$

is published on-chain along with two extra transactions, when P equivocate while. Thus, $(2\tilde{N}_2 + 2)f$ have been spent in transaction fees. Therefore, if V disproves P, V's balance account is $\langle \alpha + \beta + (2\tilde{N}_1 + 2\tilde{N}_2 + 7)f\rangle_V$ and P's balance account is $\langle 0\rangle_P$. If V does not disprove P the respective balance account is $\langle 0\rangle_P$, and thus it is a dominant strategy to disprove P.

2) For any number $\mathcal{N}$ and $\mathcal{N}_{Mer}$ committed by V during the dispute bisection game and the read bisection game, P will use the respective script (either ValueAScript or HashReadScript$_i$ or RootReadScript$_i$ for some $i \in \{1,...,\tilde{N}_1\}$), to spend the transaction ReadChallenge$_{\tilde{N}_2}$. In the worst case, one less transaction is published than Scenario 1 since V cannot prove an equivocation when P provides the required triple of nodes. Therefore, if P claim the deposits P's balance account is $\langle \alpha + \beta + (2\tilde{N}_1 + 2\tilde{N}_2 + 8)f\rangle_P$ and V's balance account is $\langle 0\rangle_V$. This is the dominant strategy for P, since otherwise the respective balance account is $\langle 0\rangle_P$. $\square$

D.4.2. Write bisection game.

Lemma D.12 (A party is inactive during the Write bisection game). Assume that V spends the script CIScriptWrite$_C$ of CommitInstruction to publish on-chain the transaction ChallengeWrite. The following statements hold for the Write bisection game.

1) *V is inactive:* Assume WriteResponse$_i$, $i \in \{1,...,\tilde{N}_2\}$ is published on-chain. If WriteChallenge$_i$ is not published on-chain after time Δ, then it is a dominant strategy for P to claim the coins locked in the multisignature. As a result, P's balance account is $(\alpha + \beta + (2\tilde{N}_1 + 4\tilde{N}_2 + 0 - 2i)f)_P$, and V's balance account is $(0)_V$.

2) *P is inactive:* Assume that a set of transactions $\{tx\} \subseteq \{\text{ChallengeValueC}\} \cup \{\text{WriteChallenge}_i\}_{i\in\{1,...,\tilde{N}_2\}}$, where $\{tx\} \neq \emptyset$, is published on-chain and one of the following scenarios is true:
   a) ChallengeValueC $\in \{tx\}$ (let $j = 0$) and P has not posted WriteResponse$_1$ within Δ,
   b) WriteChallenge$_i \in \{tx\}$ for $i < \tilde{N}_2$ (let $j = i$) and P has not posted WriteResponse$_{i+1}$ within Δ,
   c) WriteChallenge$_{\tilde{N}_2} \in \{tx\}$ (let $j = \tilde{N}_2$) and P has not posted exactly one transaction $tx' \in \{\text{MerkleRootHash}\} \cup \{\text{MerkleHash}_i\}_{i\in\{1,...,\tilde{N}_2\}}$ within Δ,

   Then, it is a dominant strategy for V to claim the coins locked in the multisignature. P's balance account is then $\langle 0\rangle_P$, and V's balance account is $\langle \alpha + \beta + (2\tilde{N}_1 + 4\tilde{N}_2 - 2j + 8)f\rangle_V$.

Proof: Since ChallengeWrite is published on-chain, P has published on-chain the set of transactions $P = \text{Setup} \cup$

CommitComputation, $\cup\{\text{TraceResponse}_i\}_{i\in\{1,...,\tilde{N}_1\}}$ $\cup$ CommitInstruction and V has published $V = \text{KickOff} \cup \{\text{TraceChallenge}_i\}_{i\in\{1,...,\tilde{N}_1\}}$ $\cup$ ChallengeRead so $(2\tilde{N}_1 + 5)f$ coins have been already spent in transaction fees.

1) In this scenario, P has published on-chain $P \cup \{\text{WriteResponse}_k\}_{k\in\{1,...,i\}}$, and V has published on-chain $V \cup V'$, where $V' = \{\text{WriteChallenge}_k\}_{k\in\{1,...,i-1\}}$ if $i > 0$, otherwise $V' = \emptyset$. Therefore, extra $2i - 1$ have been spent in fees. Since WriteResponse$_i$ is published on-chain and V has not published WriteChallenge$_i$ on-chain after Δ, P can activate the timelock to claim the coins locked in the multisignature. To this end, P spends extra f in transaction fees. As a result, his balance account is $u_1 = (\alpha + \beta + (2\tilde{N}_1 + 4\tilde{N}_2 + 9 - 2i)f)$ and V's account is $\langle 0\rangle_V$. Otherwise, P's balance account is $0 < u_1$, and therefore activating the timelock is a dominant strategy.

2) In all of the scenarios Items 2a to 2c extra $2j$ have been spent in transaction fees. Moreover, since P is inactive, V can activate the timelock to claim the coins locked in the multisignature, spending an extra f in transaction fees. As a result, V's balance account is $u_1 = (\alpha + \beta + (\alpha + \beta + (2\tilde{N}_1 + 4\tilde{N}_2 + 8 - 2j)f))$ and V's account is $\langle 0\rangle_V$. Otherwise, V's balance account is $0 < u_1$, and therefore activating the timelock is a dominant strategy. $\square$

Lemma D.13 (The Write bisection game completes). Assume that V spends the script CIScriptWrite$_C$ of CommitInstruction to publish on-chain the transaction ChallengeWrite. The following statements hold for the Write Bisection game.

1) *Scenario 1, P has written incorrect values in the memory:* Consider the number $\mathcal{N}$ the number to which V has committed during the dispute bisection game. If P has committed to two execution trace elements for steps $\mathcal{N}$ and $\mathcal{N} + 1$ s.t. $E^P_{\mathcal{N}} = E_{\mathcal{N}}$ and $E^P_{\mathcal{N}+1} \neq E_{\mathcal{N}+1}$ and P has committed only correct values in CommitInstruction, then it is a dominant strategy for V to claim the coins in the multisignature. As a result, P's balance account is $\langle 0\rangle_P$, and V's balance account is $\langle \alpha + \beta + (2\tilde{N}_1 + 2\tilde{N}_2 + 7)f\rangle_V$.

2) *Scenario 2, P follows the protocol specifications:* If P has followed the protocol specifications, P will eventually claim the coins in the multisignature. As a result, P's In that scenario, V's balance account is $\langle 0\rangle_P$, and P's balance account is $\langle \alpha + \beta + (2\tilde{N}_1 + 2\tilde{N}_2 + 8)f\rangle_V$.

Proof: First, in both scenarios, since ChallengeWrite is published on-chain $(2\tilde{N}_1 + 5)f$ coins have been already spent in transaction fees as explained in Lemma D.12.

1) Consider the Merkle trees of the memory at steps $\mathcal{N}$, $\mathcal{N} + 1$ with the respective roots $MR_{\mathcal{N}}, MR_{\mathcal{N}+1}$. Moreover, consider the path π from the root $MR_{\mathcal{N}}$ to $M_{\mathcal{N}}[addrC_\theta]$ and the path π' from the root $MR_{\mathcal{N}+1}$

to $M_{\mathcal{N}+1}[addrC_\theta]$, where $addrC_\theta$ was committed by P in `CommitInstruction` .

By assumption, V follows the protocol specifications and therefore runs function $\text{DisagreeWriteV}(\mathcal{B}_1, \mathcal{B}_2, \tilde{N}_2)$ of Algorithm 23, where the pair of sequences $(\mathcal{B}_1, \mathcal{B}_2)$ consists of the values of π and π' respectively, i.e., $\forall i \in \{1, ..., \tilde{N}_2\}, (\mathcal{B}_1[i], \mathcal{B}_2[i]) = (\pi[i], \pi'[i])$. On the other side, P runs function $\text{DisagreeWriteP}(\mathcal{A}_1, \mathcal{A}_2, \tilde{N}_2)$ of Algorithm 23 where the pair of sequences $(\mathcal{A}_1, \mathcal{A}_2)$ is constructed as follows. Let $\mathcal{I}$ be the set of all the nodes to which P commits on-chain (line 7) including the root $(i = 1)$ for which $\mathcal{A}_1[1] = MR_{\mathcal{N}}^P, \mathcal{A}_2[1] = MR_{\mathcal{N}+1}^P$ to which P committed via the respective execution trace elements in the dispute bisection game, and the leaf of the paths $(i = \tilde{N}_2)$, $\mathcal{A}_1[1] = MR_{\mathcal{N}}^P, \mathcal{A}_2[1] = MR_{\mathcal{N}+1}^P$ which are the values P committed in `CommitInstruction`. For every $i \in \mathcal{I}$, the pair $(\mathcal{A}_1[i], \mathcal{A}_2[i])$ consists of the nodes to which P has committed on-chain. For the rest indices, $i \in \{1, ..., \tilde{N}_2\} \setminus \mathcal{I}$, without loss of generality, we assume that P's pair of sequences holds the correct nodes of the paths, i.e., $(\mathcal{A}_1[i], \mathcal{A}_2[i]) = (\pi[i], \pi'[i])$. We will show that Algorithm 23 pinpoints a step $\mathcal{N}_{Mer}$ such that $\mathcal{A}_i[\mathcal{N}_{Mer}] = \mathcal{B}_i[\mathcal{N}_{Mer}]$ and $\mathcal{A}_i[\mathcal{N}_{Mer}+1] \neq \mathcal{B}_i[\mathcal{N}_{Mer}+1]$ for at least one $i \in \{1, 2\}$. To this end, we decompose the result of the execution of Algorithm 23 in the following cases:

- *There is a step of the bisection game i s.t. for some $j \in \{1, ..., \tilde{N}_2\}$ s.t. $\mathcal{A}_1[j] \neq \mathcal{B}_1[j]$:* V will set its local variable $flag$ to $True$ (line 25. In that situation, starting from the next iteration $i + 1$, V will always skips the lines 24-31. The remaining code that V and P run, given that the sequences $\mathcal{A}_2$ and $\mathcal{B}_2$ do not affect the execution, is similar to running Section B.2 where V has the sequence $\mathcal{A}_1[1 : j]$ and P has the sequence $\mathcal{B}_1[1 : j]$. Therefore with a proof similar to Theorem D.7, we can show that Algorithm 23 pinpoints a step $\mathcal{N}_{Mer}$ s.t. $A_2[\mathcal{N}_{Mer}] = B_2[\mathcal{N}_{Mer}]$ and $A_2[\mathcal{N}_{Mer} + 1] \neq B_2[\mathcal{N}_{Mer} + 1]$.
  Otherwise: V's local variable $flag$ is always $False$. In this case, V will always skips the lines 32-36. For the remaining code that V and P run the sequences $\mathcal{A}_2$ and $\mathcal{B}_2$ do not affect the execution. We can prove that since $A_2[1] \neq B_2[1]$ and $A_2[\tilde{N}_2] = B_2[\tilde{N}_2]$ Algorithm 23 pinpoints a step $\mathcal{N}_{Mer}$ s.t. $A_2[\mathcal{N}_{Mer}] = B_2[\mathcal{N}_{Mer}]$ and $A_2[\mathcal{N}_{Mer} + 1] \neq B_2[\mathcal{N}_{Mer} + 1]$ with a proof similar to Theorem D.7.

In any case, since the point of disagreement is identified, we can prove that V will eventually manage to disprove P similar to the Read Bisection game (Lemma D.11).

2) Similar to the Read Bisection game (Lemma D.11), since P has committed to only correct values, V cannot disprove the computation. Therefore, P will eventually claim the coins in the multisignature. □

D.5. Concluding Lemmas

***Lemma D.14** (P **has committed to the wrong state and** `CommitInstruction` **is published on-chain**).* Assume that P has committed to the execution trace E_{final}^P in `CommitComputation` s.t. $E_{final}^P \neq E_{final}^*$, and P has also published on-chain `CommitInstruction` committing to the values pc_θ, $pc_{\theta'}$, $insType_\theta$, $addrA_\theta$, $addrB_\theta$, $addrC_\theta$, $valA_\theta$, $valB_\theta$, $valC_\theta$. It is a feasible and dominant strategy for V to prove the misbehavior. As a result, V's balance account will be $(u)_V$, where $u \geq \alpha + \beta + (2\tilde{N}_1 + 2\tilde{N}_2 + 7)f$ and P's balance account $(0)_P$.

Proof: We will prove that V will eventually claim the coins in the multisignature by following the protocol specifications. We will also show that this is the dominant strategy for V.

`CommitInstruction` is published on-chain which means that V initiated the dispute bisection game. That is because `CommitInstruction` spends $\text{TraceChallenge}_{\tilde{N}_1}$ which can only be published on chain if the set of transactions $\{\text{KickOff}\} \cup \{\text{TraceChallenge}_i\}_{i \in \{1, ..., \tilde{N}_1 - 1\}} \cup \{\text{TraceResponse}_i\}_{i \in \{1, ..., \tilde{N}_1\}}$ is already on-chain. Since follows the protocol specifications, V holds a sequence consisting of the correct execution trace elements during the dispute bisection game, i.e., $E_i^V = E_i^*, \forall i \in \{1, ..., final\}$.

By assumption, P and V agree on the initial execution trace, i.e., $E_0^P = E_0^V = E_0^*$, and disagree in the execution trace of the final step, i.e., $E_{final}^P \neq E_{final}^V = E_{final}^*$. According to Lemma D.7, the Identify Disagreement phase outputs a step $\mathcal{N}$ for which the following condition holds: for the VM execution steps $\mathcal{N}$ and $\mathcal{N}+1$, P has committed to the execution trace elements $E_{\mathcal{N}}^P = E_{\mathcal{N}}^V = E_{\mathcal{N}}^*$ and $E_{\mathcal{N}+1}^P \neq E_{\mathcal{N}+1}^V = E_{\mathcal{N}+1}^*$.

Since $E_{\mathcal{N}}^P = E_{\mathcal{N}}^*$ and $E_{\mathcal{N}+1}^P \neq E_{\mathcal{N}+1}^*$, P has executed the state transition of the VM at step $\mathcal{N}+1$ (Algorithm 6, line 5) incorrectly. The possible ways that P has run incorrectly Algorithm 5 at step $\mathcal{N}+1$, are the following:

- *Using incorrect inputs:*
  - *P uses an incorrect program counter:* Since $E_{\mathcal{N}}^P = (MR_{\mathcal{N}}^P, pc_{\mathcal{N}}^P) = E_{\mathcal{N}}^*$, the program counter to which P committed during the dispute bisection game, i.e., $pc_{\mathcal{N}}^P$, is correct. However, P can use a different program counter at step $\mathcal{N}+1$ in lines 3 6. P commits to the program counter of the program at step $\mathcal{N}$ in `CommitInstruction`, so P can commit to $pc_\theta^P \neq pc_{\mathcal{N}}^P$. Then, according to Lemma D.8, it is a dominant strategy for V to claim the coins in the multisignature. P's balance account will be $\langle 0 \rangle_P$, and V's balance account $\langle \alpha + \beta + (2\tilde{N}_1 + 4\tilde{N}_2 + 9)f \rangle_V$.
  - *P sets a program instruction which is either invalid or does not correspond to the instruction of Π at the program counter $pc_{\mathcal{N}}$:* P can set an incorrect program instruction in line 3. However, in that case, P commits to a program instruction such that

$\Pi(pc_\theta) \neq (insType_\theta, addr A_\theta, addr B_\theta, addr C_\theta)$ in `CommitInstruction`. Following from Lemma D.9, if P commits such an invalid program instruction, it is a dominant strategy for V to claim the coins in the multisignature. P's balance account is $\langle 0 \rangle_P$, and V's balance account $\langle \alpha + \beta + (2\tilde{N}_1 + 4\tilde{N}_2 + 9)f \rangle_V$.

– *P reads incorrect values from the memory:* P can read incorrect values (val_A or val_B) from the memory ($M^{\mathcal{N}}[addr A]$ or $M^{\mathcal{N}}[addr B]$) at step $\mathcal{N}$ (lines 4, 5). However, P has committed to the correct memory root at step $\mathcal{N}$ (memory output by executing Algorithm 6 correctly), $MR_{\mathcal{N}}^P = MR_{\mathcal{N}}^*$ of the dispute bisection game (since $E_{\mathcal{N}}^P = E_{\mathcal{N}}^*$). In Lemma D.10 we show that if P remains inactive in the Read bisection game, it is a dominant strategy V to claim the coins in the multisignature. Similarly, in Lemma D.11, we show that if $val_A \neq M[addr_A]$ or $val_B \neq M[addr_B]$ and the Read bisection game completes, it is again a dominant strategy for V to claim the coins. Moreover, P's balance account is $\langle 0 \rangle_P$, and V's balance account is in the worst case $\langle \alpha + \beta + (2\tilde{N}_1 + 2\tilde{N}_2 + 7)f \rangle_V$.

- *Using correct inputs but executing incorrectly the algorithm.* Here we assume that P has provided the correct inputs i.e., the inputs when executing Algorithm 6 correctly. Since `CommitInstruction` is successfully published on-chain it must be that $(pc_{\theta'}, valC_\theta) = insType_\theta(pc_\theta, valA_\theta, valB_\theta)$ since this is a necessary condition to unlock the script of `TraceChallenge`$_{32}$. Therefore the values related to the execution of step $\mathcal{N}+1$ that P committed in `CommitInstruction` are correct, i.e., $(pc_{\theta'}, valC_\theta) = ((pc')^*, val_C^*)$. However, since P has committed to a wrong execution trace element for step $\mathcal{N}+1$ in the dispute bisection game, i.e., $E_{\mathcal{N}+1}^P = (MR_{\mathcal{N}+1}^P, pc_{\mathcal{N}+1}^P) \neq E_{\mathcal{N}+1}^*$, one of the following conditions hold:

  – $pc_{\theta'}^P \neq pc_{\mathcal{N}+1}^P$: Then, according to Lemma D.8, it is a dominant strategy for V to claim the coins in the multisignature. P's balance account will be $\langle 0 \rangle_P$, and V's balance account $\langle \alpha + \beta + (2\tilde{N}_1 + 4\tilde{N}_2 + 9)f \rangle_V$.

  – *P has committed to all the correct values in* `CommitInstruction` *but* $E_{\mathcal{N}+1}^P \neq E_{\mathcal{N}+1}^*$ *or is inactive during the Write bisection game*: according to Lemmas D.13, D.12 accordingly, V can show that P has written a wrong value in the memory and claim the coins in the multisignature. P's balance account is $\langle 0 \rangle_P$, and V's balance account is in the worst case $\langle \alpha + \beta + (2\tilde{N}_1 + 2\tilde{N}_2 + 7)f \rangle_V$.

In any case, it is a dominant strategy for V to prove P's misbehavior and claim the coins in the multisig. As a result, V's balance account is $(u)_V$, where $u \geq \alpha + \beta + (2\tilde{N}_1 + 2\tilde{N}_2 + 7)f$ and P's balance account is in any case $(0)_P$. $\square$

Lemma D.15 (P follows the protocol specifications and `CommitInstruction` ***is published on-chain).*** Assume that P

has published on-chain `CommitInstruction` committing to the values $pc_\theta, pc_{\theta'}, insType_\theta, addr B_\theta, addr C_\theta, valA_\theta, valB_\theta, valC_\theta$. If P follows the protocol specifications, P will eventually claim the coins in the multisignature. As a result, P's balance account will be $(u)_P$, where $u \geq \alpha + \beta + (2\tilde{N}_1 + 2\tilde{N}_2 + 8)f$ and V's balance account $(0)_P$.

Proof: `CommitInstruction` is posted on-chain which means that V initiated the dispute bisection game (as explained in D.14). Since P follows the protocol specifications, P has executed the VM algorithm (Algorithm 6) correctly. Therefore, for any step i s.t. P committed to an execution trace element during the dispute bisection game it holds that $E_i^P = E_i$. Moreover, the values that P has committed in `CommitInstruction` are correct (they are derived by executing Algorithm 6 correctly).

The possible ways for V to spend `CommitInstruction` is publishing on-chain one of the following transactions:

- V publishes on-chain `PunishFaultyProgramCounter` claiming that $pc_\theta \neq pc_{\mathcal{N}}^P$ (or $pc_{\theta'} \neq pc_{\mathcal{N}}^P + 1$): By assumption, $pc_\theta = pc_{\mathcal{N}}^P$ and $pc_{\theta'} \neq pc_{\mathcal{N}}^P + 1$, and according to Lemma D.8, it is a dominant strategy for P to claim the deposits. P's balance account will be in the worst case $\langle \alpha + \beta + (2\tilde{N}_1 + 4\tilde{N}_2 + 8)f \rangle_P$, and V's balance account $\langle 0 \rangle_V$.

- V publishes on-chain `ChallengeRead` to claim that $valA_\theta \neq M^{\mathcal{N}}[addr A_\theta]$ (or $valB_\theta \neq M^{\mathcal{N}}[addr B_\theta]$): Then, if i) V remains inactive, or ii) the Read Bisection game finishes, it is a dominant strategy for V to claim the deposits as we prove Lemmas D.10, D.11 accordingly. $\langle \alpha + \beta + (2\tilde{N}_1 + 2\tilde{N}_2 + 8)f \rangle_P$, and V's balance account $\langle 0 \rangle_V$.

- V *publishes* `ChallengeWrite` *to claim that P has written an incorrect value in the memory*: either 1) V remains inactive, or ii) the Write bisection game completes, we show in Lemmas D.12, D.13 that V will eventually claim the coins in the multisignature. Therefore, P's balance account is in the worst case $\langle \alpha + \beta + (2\tilde{N}_1 + 4\tilde{N}_2 + 8)f \rangle_P$, and V's balance account $\langle 0 \rangle_V$.

To summarize, P's balance account is $\langle u \rangle_V$, where $u \geq (2\tilde{N}_1 + 2\tilde{N}_2 + 8)f$ and V's balance account is in any case $\langle u \rangle_V$. $\square$

D.6. Theorems

Theorem D.16. The strategy profile representing the optimistic path of BitVM forms a Subgame Perfect Nash Equilibrium.

Proof: We prove that by backward induction on Γ, which we illustrate in Figs. 4 and 5. $\square$

Theorem D.17. (Balance Security) BitVM satisfies Balance Security.

Proof: Let us fix one party $A \in \{P, V\}$ and assume that p follows the protocol specifications. We prove that no matter what strategy the other party p' chooses, p will eventually claim at least $f_A(S^*_{final})$ coins, i.e., the coins which A should receive according to the outcome mapping function.

To prove that, we only consider the subtree $\gamma \subseteq \Gamma$, which gives a comprehensive description of BitVM given that party A follows the protocol specification. Below, we consider the respective scenarios where P or V follow the protocol specifications.

- *Case 1: P follows the protocol specifications.* We consider the subtree $\gamma \subseteq \Gamma$, which we derive as follows. First, consider the subtree γ' derived by Γ with the following changes. After P posts `Setup` on-chain, the only possible action is to then commit to the correct final result (by posting `CommitComputation` on-chain). Moreover, after have posted the correct result, in the case where P has not disputed the result, the only possible action for P is to publish on-chain the corresponding Close transaction. Then, we derive γ by deleting any action (or edge) in γ' where P remains inactive. The subtree γ gives a comprehensive description of BitVM given that P follows the protocol specification. For any node $u \in \gamma$, there is a path leading to a leaf node where P claims at least $(\alpha + \beta + (2\tilde{N}_1 + 2\tilde{\tilde{N}}_2 + 8)f) \geq f_P$ by assumption. Since P is rational, they follow one of these paths.

- *Case 2: V follows the protocol specifications.* Now consider the subtree $\gamma \subseteq \Gamma$, which we derive as follows. First, if P has posted the correct final result, V does not initiate a dispute. Moreover, we delete the actions where V remains inactive. The subtree γ gives a comprehensive description of BitVM given that V follows the protocol specification. For any node $u \in \gamma$, there is a path leading to a leaf node where V claims at least $(\alpha + \beta + (2\tilde{N}_1 + 2\tilde{\tilde{N}}_2 + 5.5)f) \geq f_V$ by assumption. Since V is rational, they follow one of these paths. $\square$

Appendix E.
Transaction estimation

In this section, we provide a detailed overview of how we compute the size of the transactions published on the Bitcoin blockchain during the execution of the BitVM protocol.

As shown in [32], computing the size of a SegWit [25] transaction requires estimating both its non-witness and witness components. For the non-witness portion, each Byte counts as a vByte, whereas in the witness, 4 Bytes count as a vByte.

The non-witness portion consists of three main parts(for fields with variable sizes, we fix the vByte count based on the transaction that we have in our protocol):

Overhead
- The transaction version number ($4vB$).
- The input count ($1vB$) and the output count ($1vB$).
- The timestamp until which the transaction is locked ($4vB$).

- SegWit transaction flag ($1vB$).

Input
- The previous transaction ID and index of the output being spent in the previous transaction ($36vB$)

Output
- The amount of ₿ being transferred ($8vB$).
- The length of the scriptPubKey field ($1vB$).
- The scriptPubKey (it varies, we upper bound this with the PayToTaproot(P2TR) vbyte size that is $34vB$).

For witnesses, we categorize them based on the type of scriptPubKey they unlock:

We can distinguish between two kinds of witnesses, according to which scriptPubKey they unlock:

- PayToWitnessPublicKeyHash(P2WPKH): witness size is approximately $27vB$.
- PayToTaproot: includes a control block, a script, and the script data. The witness size varies, and transaction witnesses can be grouped into four size categories where each group has a similar weight.

Note that the elements that most significantly impact the size of a witness are the 2-of-2 public key and its multisignature and the Lamport public keys and commitments. Consequently, we categorize all transactions that spend a P2TR output into four groups, (each transaction in these groups verifies the validity of a multisignature):

1) Transactions where there is one Lamport commitment verification: we upper bound their weight with $1112vB$. The transactions `KickOff`, `TraceChallenge`$_1$, ..., `TraceChallenge`$_{32}$, `TraceResponse`$_1$, ..., `TraceResponse`$_{32}$, `ChallengeWrite`, `WriteChallenge`$_1$, ..., `WriteChallenge`$_5$ belong to this group.

2) Transactions where there are two Lamport commitment verifications: we upper bound their weight with $2093vB$. The transactions `CommitComputation`, `WriteResponse`$_1$, ..., `WriteResponse`$_5$ belong to this group.

3) Transactions where there are four Lamport commitment verifications: we upper bound their weight with $7268vB$. The transactions `CommitWrite1` and `PunishWrite1` belong to this group.

4) Transactions where there are nine Lamport commitment verifications: only the `CommitInstruction` transaction belongs to this group. We estimate its size being $2751vB$.

Although `CommitInstruction` verifies more Lamport commitments than `CommitWrite1`, it has a smaller size because it verifies commitments for elements that are at most $4B$, while `CommitWrite1` verifies the commitment of four $20B$ elements.
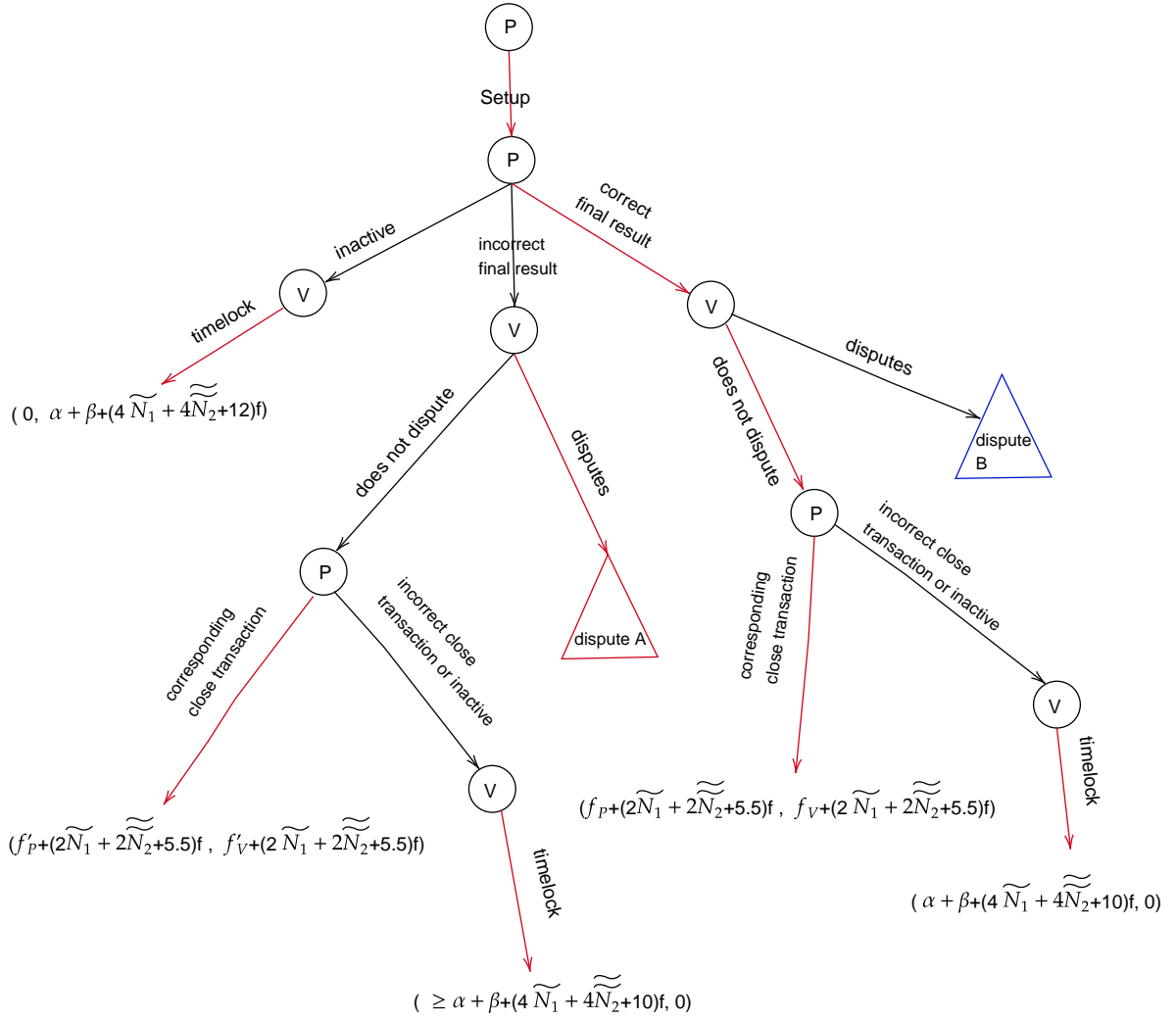
Figure 4. Tree representation Γ of the EFG describing BitVM. We underline with red the actions each parties takes at each step. The subtrees dispute A and dispute B are depicted in Fig. 5.

common subtree
punishment subtree P
punishment subtree V

V

Kickoff

P

Res1   inactive

V   V
                    timelock
inactive
         Chal1 $(0, \alpha + \beta + (4\ \widetilde{N_1} + 4\widetilde{N_2} + 10)f)$

P

P

$(\alpha + \beta + (4\ \widetilde{N_1} + 4\widetilde{N_2} + 9)f, 0)$

timelock

...

V

chalN$_1$ inactive

P   P
                    timelock
inactive
         CommitInstructionTx $(\alpha + \beta + (2\ \widetilde{N_1} + 4\widetilde{N_2} + 11)f, 0)$

V   V

timelock   inactive

$(0, \alpha + \beta + (2\ \widetilde{N_1} + 4\widetilde{N_2} + 10)f)$

P
                    timelock

punishment   punishment
subtree P   subtree V $(0, \alpha + \beta + (2\ \widetilde{N_1} + 4\widetilde{N_2} + 9)f)$

$(0, \alpha + \beta + (2\ \widetilde{N_1} + 4\widetilde{N_2} + 7)f)$ $(\ \geq \alpha + \beta + (2\ \widetilde{N_1} + 2\widetilde{N_2} + 8)f, 0)$

Figure 5. The tree Γ' illustrates Subtree A and Subtree B as depicted in Fig. 4. Subtree A, initiated by an honest V to disprove a malicious P, consists of the "Common Subtree" and "Punishment subtree P". Subtree B, initiated by a malicious V trying to a correct P, consists of the "Common Subtree" and "Punishment subtree V".