BitVM: Quasi-Turing Complete Computation on Bitcoin

Lukas Aumayr[1,2], Zeta Avarikioti[1,3], Robin Linus[4,5], Matteo Maffei[3], Andrea Pelosi[3], Christos Stefo[3], and Alexei Zamyatin[6]

[1]Common Prefix
[2]University of Edinburgh
[3]TU Wien
[4]Stanford University
[5]ZeroSync
[6]BOB

Abstract

A long-standing question in the blockchain community is which class of computations is 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 an honest majority. We present BitVM, a two-party protocol that realizes a generic virtual machine by combining cryptographic primitives and economic incentives. We conduct a formal analysis of BitVM, characterizing its functionality, system assumptions, and security properties. We further demonstrate the practicality of our approach by implementing a prototype and performing an experimental evaluation: in the optimistic case (i.e., when parties agree), 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. We exemplify the deployment potential of BitVM by building a Bitcoin-sidechain bridge application. 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 are a foundational component of modern blockchain systems, enabling decentralized applications (dApps) and programmable money without the need for trusted intermediaries. These self-executing programs have unlocked a wide range of use cases, spanning finance, governance, and supply chain management, by enforcing agreement logic directly on-chain.

While some blockchains, such as Ethereum, support quasi-Turing complete[1] execution through bytecode languages like the Ethereum Virtual Machine (EVM), others, most notably Bitcoin, opt

[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) [35].

for a minimalist approach. Bitcoin Script deliberately limits expressiveness to minimize complexity and attack surface, trading expressiveness for security and stability.

This trade-off has given rise to a long-standing open question: *Can general-purpose computation be supported on Bitcoin, using only its existing scripting language and consensus rules?* Unlocking such expressiveness could expand Bitcoin's utility for dApps and decentralized finance (DeFi), while preserving its conservative security model.

The prevailing belief in the blockchain community is that Bitcoin Script cannot support general-purpose computation in practice. This belief stems from several structural limitations of the language: it is stateless, lacks loops and recursion, and enforces strict constraints on script and transaction size. These properties make it well-suited for simple functionalities, such as multisignature payments or hashed timelock contracts, but appear to rule out the execution of complex logic on-chain.

Several proposals have attempted to overcome these constraints, either by extending Bitcoin with new opcodes (e.g., covenants [12]), encoding computations into low-level counter machines [13], or relying on trusted execution environments [18, 24] and oracles [32, 19]. However, these approaches either require consensus changes, incur prohibitive on-chain costs, or compromise Bitcoin's trust model. As a result, it has long been assumed that supporting arbitrary computation on Bitcoin would require trade-offs incompatible with its conservative design philosophy.

Contributions. In this work, we challenge this long-standing belief by showing that arbitrary (bounded) computations can be executed securely on Bitcoin in a practical manner. We introduce BitVM[2], a two-party protocol that enables expressive, verifiable off-chain computation using only existing Bitcoin features. BitVM requires no consensus changes, no new opcodes, and no trusted hardware or oracles.

The core idea behind BitVM is to shift computation off-chain while retaining on-chain verifiability through a fraud-proof mechanism. Specifically, a prover submits a claim about the output of a bounded computation. The verifier can either accept the result or, in case of disagreement, initiate an interactive dispute protocol. This protocol relies on a custom virtual machine that encodes computation as an execution trace, enabling the parties to identify a point of disagreement and verify a single step of computation using Bitcoin Script.

When both parties agree, the entire process completes with just three on-chain transactions. In case of dispute, BitVM guarantees that verification incurs only a logarithmic number of additional transactions. This makes BitVM both practical and expressive, enabling Bitcoin to support advanced smart contract functionality without compromising its trust model or requiring protocol changes.

To illustrate how BitVM operates in practice, consider a simple two-party wager: a prover claims to have solved a chess puzzle, and a verifier bets against them. Both parties lock funds into a BitVM contract that verifies the claimed solution. If the verifier agrees, the outcome is settled off-chain. If not, they can initiate an on-chain dispute. The contract then executes the verification step by step, and settles the funds accordingly, using only Bitcoin Script to enforce the outcome. While conceptually simple, this example demonstrates the core functionality of BitVM: enabling trustless agreement on arbitrary computation, with minimal on-chain footprint in the absence of disputes.

The same mechanism underpins more complex applications. A key example, which we outline in this work, is a *trust-minimized bridge between Bitcoin and a sidechain.* In typical designs, a committee of operators handles redemptions from the sidechain to Bitcoin, with security guaranteed only under an honest-majority assumption (t-of-n, where $t > n/2$), e.g., [9, 1]. With BitVM, this

[2]This work extends and formalizes the original BitVM design, which was conceptualized and developed by Robin Linus [38].

trust requirement is minimized to *existential honesty*: security holds as long as *at least one* operator behaves honestly ($t = 1$). Any operator can front coins to a user on Bitcoin and later claim reimbursement by proving correctness; if another committee member disputes the claim, they can initiate a BitVM instance to verify or refute it on-chain. This approach *reduces the trust assumption from honest majority of the bridge committee to existential honesty*, while remaining fully compatible with Bitcoin's existing scripting model. Crucially, such a bridge serves as a foundational building block for a broader ecosystem of decentralized applications: once assets can move between Bitcoin and external execution layers securely and trustlessly, DeFi protocols, from decentralized exchanges to lending platforms, can operate on top of Bitcoin without compromising its security guarantees.

The appeal of the BitVM paradigm is witnessed by the massive investments of the industry (more than 150M USD[3]) and the number of BitVM-based protocols already in development [23, 39, 17, 5].

This paper makes the following contributions:

- We present BitVM, the first protocol to encode quasi-Turing complete computations in Bitcoin Script, requiring no consensus changes or trusted third parties (Section 5).
- We provide a formal analysis of BitVM, characterizing its functionality, system assumptions, and security properties (Section 6).
- To evaluate BitVM's performance, we implement a prototype of BitVM in JavaScript.[4] Optimistic execution completes in three on-chain transactions, costing approximately $5,832$ satoshis[5], equivalent to $4.26 (as of May 2026). In case of disputes, the on-chain footprint grows logarithmically with the computation size. For a virtual machine with 2^{32} memory cells and steps—comparable to a high-end 1990s workstation—settlement requires up to 81 transactions, at a cost of approximately $732,000$ satoshis, equivalent to $534 (Section 7).
- We demonstrate the capabilities of BitVM by constructing a trust-minimized bridge protocol between Bitcoin and a sidechain, reducing the traditional honest-majority assumption to that of a single honest operator (Section 8).

Related work. Several works have attempted to overcome the limited expressiveness of Bitcoin Script and enable more complex smart contracts by combining UTXOs and scripts, effectively splitting functionality across multiple transactions. BitML [14] provides a high-level, domain-specific language for smart contracts with an associated compiler that produces Bitcoin transactions, illustrating Bitcoin's potential for intricate smart contract designs [11]. These methods, however, incur substantial on-chain costs, as compiled programs often result in numerous large transactions that must ultimately be recorded on-chain, and they support only a restricted class of contracts.

To mitigate these costs, some approaches leverage Trusted Execution Environments (TEEs). FastKitten [18] facilitates off-chain computation within a secure hardware enclave, but relies on collateral, rational adversaries, trusted TEE operators, and a limited contract duration. POSE [24] improves upon FastKitten by removing collateral requirements and time constraints, and by enhancing privacy, but it continues to rely on trusted TEE hardware.

A different approach uses Hashed Timelock Contracts (HTLCs) to shift computation off-chain by encoding outcomes in preimages of hash functions [8], similar in spirit to *state channels* on Ethereum [22, 20]. This model underpins constructions such as Discreet Log Contracts (DLCs) [19] and oracle-based conditional payments [32], which depend on (semi-) trusted oracles to attest to

[3] `https://cryptorank.io`

[4] The prototype is available at [3].

[5] A *satoshi* is a fraction of a bitcoin, i.e., $1sat = 10^{-8}$₿.

specific events. A key limitation of these approaches is that all possible outcomes must be known and encoded in advance. Consequently, they cannot support applications like the chess puzzle or the bridge example, where the correct outcome, such as the solution to a puzzle or the identity of the party holding funds on the sidechain, may not be known a priori to any participant.

Unlike prior works, BitVM enables quasi-Turing complete computation on Bitcoin without consensus changes or relying on external trust assumptions, such as TEEs and semi-trusted oracles. It is the first trustless protocol to allow arbitrary, bounded computation on Bitcoin, while incurring logarithmic on-chain cost in the worst case, unlocking a range of potential applications. Table 1 situates BitVM among prior Bitcoin smart contract approaches.

Table 1: Comparison of Bitcoin-based smart contract approaches. n denotes the upper bound on computational steps, and QT refers to Quasi-Turing completeness.

Approach	**Expressiveness**	**Extra Assumptions**	**On-chain cost**
BitML [14, 11]	QT	None	$O(n)$
TEEs [18, 24]	QT	TEE	$O(n)$
Gen. Channels [8]	Bitcoin	None	$O(1)$
Oracles [19, 32]	QT	Trusted oracle	$O(1)$
BitVM (our work)	QT	None	$O(\log(n))$

Follow-up work [29], informally referred to as BitVM2, targets bridges between Bitcoin and Layer-2 systems. It compiles a SNARK verifier into Bitcoin Script, splits it across transactions, and commits to intermediary states on-chain. This supports permissionless dispute resolution but incurs on-chain cost linear in the verifier size, requiring a transaction that fills a 4MB Bitcoin block in case of disputes. Our BitVM is instead better suited to permissioned settings and resolves disputes at logarithmic on-chain cost, with formal security guarantees. Beyond the asymptotic gap, it is also cheaper in practice: Table 2 shows on-chain costs an order of magnitude below BitVM2 in both the optimistic and dispute cases, where for a fair comparison we size BitVM at 2^{32} memory cells, sufficient to encode the SNARK verifier. Finally, BitVM runs RISC-V programs through off-the-shelf compilers, whereas BitVM2 requires compiling the verifier (around 4GB) into Bitcoin Script, a cumbersome and error-prone task.

A recent development, informally termed BitVM3 [41], pushes this paradigm further by trading on-chain cost for off-chain computation: it encodes the BitVM2 SNARK verifier as a garbled circuit. The operator garbles the verifier off-chain and commits to it during setup; a challenger recovers a fraud-proof secret off-chain iff the claim is incorrect, so a dispute is settled by revealing a single secret on-chain. BitPriv [6] and BABE [26] build on this paradigm, the former adding input privacy and the latter reducing verification costs under a permissioned challenger set. The trade-off is steep off-chain cost: each garbled circuit reaches 40GB, yielding terabytes of storage and communication overhead.

Table 2 summarizes the three BitVM paradigms. BitVM is the first to introduce this paradigm; BitVM2 and BitVM3 are follow-ups that trade it off differently, the former for permissionless disputes at linear on-chain cost, the latter for lower on-chain cost at the expense of heavy off-chain garbling.

Table 2: On-chain and off-chain costs comparison between BitVM paradigms.

Approach	**Optimistic ($)**	**Dispute ($)**	**Optimistic (vB)**	**Dispute (vB)**	**Off-chain cost**
BitVM (our work)	4.26	534	1.9k	244k	Low
BitVM2 [30]	38.7	11,680	16.8k	496.7k	Low
BitVM3 [41]	5.26	5.48	2.4k	2.5k	High

Finally, we note that the optimistic verification paradigm behind BitVM is inspired by optimistic

rollup protocols [27]. While optimistic rollups are prominent in the Ethereum ecosystem, adapting this principle to Bitcoin, which offers limited scripting flexibility and strict block size constraints, remained an open question until now.

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[6]. 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 (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., [25, 40, 7]).

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, 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[7]. 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. [25, 40, 7]). Given two sequences A and B, we use $A \preceq B$ to mean that A is a prefix of B.

Definition 2 (Safety) *A distributed ledger protocol is safe, if (i) for any honest party P and any rounds $r_1 \leq r_2$, it holds that $\mathsf{L}_{r_1}^P \preceq \mathsf{L}_{r_2}^P$ and (ii) for any pair of honest parties P_1, P_2 and any pair of rounds r_1, r_2, it holds that $\mathsf{L}_{r_1}^{P_1} \preceq \mathsf{L}_{r_2}^{P_2} \vee \mathsf{L}_{r_2}^{P_2} \preceq \mathsf{L}_{r_1}^{P_1}$.*

[6] Note that P and V can also 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.

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

Definition 3 (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 $\mathsf{L}^{\bigcap}_{r+u}$.*

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 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 $w_{\mathsf{L}}(P, tx) := \mathsf{bal}_{\mathsf{L}_2}(P) - \mathsf{bal}_{\mathsf{L}_1}(P)$, 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 [36] can always dispute a malicious commitment and claim the channel funds, while rational players will always commit to the last agreed-upon state [37].

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

Definition 4 (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* $\mathsf{in}_P \in [0, \mathsf{bal}_{\mathsf{L}}(P)]$ *and* $\mathsf{in}_V \in [0, \mathsf{bal}_{\mathsf{L}}(V)]$ *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* $v_P + v_V \leq \mathsf{in}_P + \mathsf{in}_V$[8] *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 $\mathcal{I} \leftarrow \text{setup}(\mathsf{in}_P, \mathsf{in}_V, \Pi, f)$, 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 $\mathsf{L}^{\bigcap}_{r+u}$ 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 $\sum_{tx \in \mathcal{T}}(w(A, tx)) \geq v_A$ 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 $\sum_{tx \in \mathcal{T}}(w(A, tx)) = v_A$ where $v_A = f_A(\Pi(x))$, for any $A \in \{P, V\}$ and $\Pi(x) \in \mathsf{st}(\mathsf{L}^{\bigcap}_{r+u})$.

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 the 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. For the full notation used throughout the paper, we refer the reader to Appendix A.

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 $\mathtt{Tx}$ 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 $\mathsf{out} := (a\text{Ƀ}, \mathsf{lockScript})$, where $\mathsf{out}.a \in \mathbb{R}_{\geq 0}$ is the amount of coins (expressed in Ƀ) 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

[8] We use an inequality here to account for potential fees paid during protocol execution.

UTXO. We distinguish the already existing transaction outputs (input of a transaction $\texttt{Tx}$) from the newly created outputs, calling them $\texttt{Tx.inputs}$ and $\texttt{Tx.outputs}$, respectively. A transaction input in is defined as $\mathsf{in} := (\texttt{PrevTx}, \mathsf{outIndex}, \mathsf{lockScript})$, where the output being spent is uniquely identified by specifying the transaction $\texttt{PrevTx}$ and an output index $\mathsf{outIndex}$. To improve readability, we also give the locking script $\mathsf{lockScript}$ that is being fulfilled and we specify the different transaction spending conditions.

In this paper, we use the following subset of Bitcoin spending conditions (which we describe in Appendix B): signature locks, multisignature locks, relative timelocks, and taproot trees. We use these conditions as building blocks, combining them with standard Bitcoin Script constructions using logical operators $\wedge$ (and) and $\vee$ (or).

We formally define a transaction as a tuple $\texttt{Tx} := (\mathsf{inputs}, \mathsf{witnesses}, \mathsf{outputs})$ where $\texttt{Tx.inputs} := [\mathsf{in}_1, \ldots, \mathsf{in}_n]$ are the transaction inputs, $\texttt{Tx.outputs} := [\mathsf{out}_1, \ldots, \mathsf{out}_m]$ are the transaction outputs and $\texttt{Tx.witnesses} := [\mathsf{w}_1, \ldots, \mathsf{w}_n]$ 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[9].

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.

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* [28] Lamp consists of a triple of algorithms $(\mathsf{KeyGen}, \mathsf{Sig}, \mathsf{Vrfy})$, 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 $\mathsf{Lamp.KeyGen}$.

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

[9] In this work, we separate the locking script from the witness for readability. In practice, the protocol is implemented using SegWit [31] transactions, where the locking script is included in the witness.

Algorithm 1 The key generation algorithm Lamp.KeyGen for a ℓ-bit message 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.

function Lamp.KeyGen(ℓ)
    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;
    for $i = 0, 1$ and $j = 0, \ldots, \ell-1$ **do**
        $y[i,j] \leftarrow h(x[i,j])$;
    Let $pk_{\mathcal{M}} \leftarrow \begin{pmatrix} y[0,0], \ldots, y[0,\ell-1] \\ y[1,0], \ldots, y[1,\ell-1] \end{pmatrix}$;
    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 $\mathcal{M}$.

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

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

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

different ℓ-bit message $m'' \neq m$, 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 [15]. 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 demonstrated in [3].

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 $x', x'' \in X$.

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

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

3.3 Stateful Bitcoin scripting

Although Bitcoin's scripting language is stateless, clever use of one-time digital signature schemes (e.g., Lamport signatures) enables state preservation across 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 $c_m \leftarrow \mathsf{Comm}_{\mathsf{sk}_{\mathcal{M}}}(m)$.

By hard-coding $\mathsf{CheckComm}_{pk_{\mathcal{M}}}$ 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 $\mathsf{out}_1 := (a\ddot{\mathbb{B}}, \mathsf{CheckComm}_{pk_{\mathcal{M}}})$ and $\mathsf{out}'_1 := (b\ddot{\mathbb{B}}, \mathsf{CheckComm}_{pk_{\mathcal{M}}})$. 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 6).

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 Appendix C.3 and Appendix C.4.

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 $\perp$ indicates termination. Let $\mathcal{OP} := \{f_{\mathsf{OP}} : \mathcal{PC} \times \{0, \ldots, n\} \times \{0, \ldots, n\} \to \mathcal{PC} \times \{0, \ldots, n\} \cup \{\perp\}\}$ be a set of CPU instructions that the VM can execute[10]. The function f_{OP} takes as input a triple (pc, val_A, val_B) and outputs a pair (pc, val_C). 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 $\mathcal{I} := \{(f_{\mathsf{OP}}, addr_A, addr_B, addr_C) \mid addr_A, addr_B, addr_C \in \mathcal{A}, f_{\mathsf{OP}} \in \mathcal{OP}\}$. We can now define the following.

Definition 5 (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 6 (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, Π)
2:     if pc = ⊥ then
3:         return (M, pc, Π).
4:     M' ← M;
5:     (f_OP, addr_A, addr_B, addr_C) ← Π[pc];
6:     val_A ← M[addr_A];
7:     val_B ← M[addr_B];
8:     (pc', val_C) ← f_OP(pc, val_A, val_B);
9:     if val_C ≠ ⊥ then
10:        M'[addr_C] ← val_C;
11:    return (M', pc', Π).
```

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 $f^i_{ST}(S)$ when we apply the state transition function f_{ST} to a state S exactly i times, $f^i_{ST}(S) := f_{ST}(f_{ST}(\ldots(f_{ST}(S))))$. 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 $S_i = f^i_{ST}(S_0)$. We avoid the subscripts (and refer to 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 $\Pi(S_0) := f^{\mathsf{final}}_{ST}(S_0)$.

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^A_i 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 $S^A_i = S^B_i$, and disagree on S_i otherwise.

Definition 7 (Execution Trace Element) *Let* $(M_i, pc_i, \Pi) := f^i_{ST}(S_0)$*, 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^A_i 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

[10] 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 [21].

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, when $pc = \perp$ and $i = \mathsf{final}$. 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.

function $\mathsf{VM}_{\mathsf{final}}(S_0)$
  $stepCount \leftarrow 0$;
  while $stepCount < \mathsf{final}$ **do**
    $E_{stepCount} \leftarrow (MR, pc)$;
    $(M, pc, \Pi) \leftarrow f_{ST}(M, pc, \Pi)$;
    increment $stepCount$ by 1;
  $E_{stepCount} \leftarrow (MR, pc)$;
  $ExecTrace \leftarrow (E_0, \ldots, E_{\mathsf{final}})$;
  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 $n = 2^{32} - 1$. Furthermore, we assume that the input program Π has $\ell \leq 2^{32}$ number of instructions[11] 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 [21]. 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 enhances Bitcoin's expressiveness by enabling spending conditions based on the result of general-purpose computation—performed off-chain, but verifiable on-chain through an interactive protocol. While Bitcoin Script is not (quasi-)Turing complete, BitVM effectively simulates such computation by using cryptographic commitments, economic incentives, and Script-compatible fraud proofs.

BitVM enables two mutually distrusting parties, the prover (P) and the verifier (V), to agree on the output of a program Π run on input S_0. If both parties agree on the outcome, funds are distributed accordingly with minimal on-chain interaction. In case of disagreement, BitVM executes a dispute resolution protocol that isolates the exact point of divergence and verifies correctness

[11] In the BitVM protocol, we build a Taproot tree where every program instruction is a Tapleaf script. We chose such ℓ since $2^{32} << 2^{128}$, the maximum number of leaf scripts in the current specification of Bitcoin [42].

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 return (pc + 1, ⊥);
7:     else return (pc + 2, ⊥);

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

using Bitcoin Script. All necessary transaction logic is expressible within Bitcoin, utilizing Lamport signatures and other Bitcoin Script features, thus no consensus changes are needed.

In the following, we provide a high-level description of the protocol's logic, and conclude with a discussion on its security guarantees and on-chain performance. The full protocol specification and concrete transaction constructions are deferred to Appendix C.

Protocol Phases. The BitVM protocol proceeds in four phases:

1. **Setup:** P and V agree on:
   - A program Π written for the BitVM virtual machine.
   - An outcome mapping function f, defining fund redistribution based on program output.
   - A time parameter Δ, used to timelock transactions to deter inactivity,
   - and a maximum execution trace length $\mathsf{final} = 2^{32}$.

   They presign all necessary transactions, including outcomes and disputes, and publish a `Setup` transaction locking funds on-chain in a multisig with timeout clauses.

2. **Execution:** P communicates the initial input S_0 to V. Then, both parties compute $\Pi(S_0)$ off-chain, producing an execution trace $ExecTrace = (E_0, \ldots, E_{\mathsf{final}})$, where each E_i is an execution trace element as in Definition 7.

3. **Commitment:** P posts `CommitComputation`, committing to both S_0 and the claimed output $\Pi(S_0)$ using Lamport-based commitments. If V agrees, P finalizes with `Close`, spending funds according to f. Otherwise, a dispute begins.

4. **Dispute Resolution:** If V disagrees with the claimed output, V engages P in an on-chain bisection game over the execution trace to identify a disputed VM step. Once found, this VM step is executed on-chain using Bitcoin Script. If the result is different from what P claimed, their funds are forfeited.

Optimistic Case (Happy Path)

If both parties agree on the result of the off-chain computation, only three transactions are posted on-chain:

1. `Setup` — locks funds in a multisig.
2. `CommitComputation` — P commits to the input and the output of the program using Lamport signatures.
3. `Close` — P spends `CommitComputation`'s transaction output with a result consistent with f, redistributing the funds.

Dispute Resolution (Unhappy Path)

If V disagrees with P's committed result, the protocol enters the dispute phase. Let $ExecTrace := (E_0, \ldots, E_{\mathsf{final}})$ be the off-chain VM execution trace, where each element is defined as in Definition 7. Notably, each successive element in this execution trace results from applying a single VM instruction to the preceding element. Since P and V agree on S_0[12], any disagreement on the result implies a disagreement on some S_{i+1}, while agreeing on S_i. The bisection game locates such an index $\mathcal{N}$, enabling on-chain verification of a single VM step (we formalize the bisection game in Appendix D). Fig. 1 illustrates an overview of the dispute process.

Identify Disagreement. V initiates the dispute by publishing the `KickOff` transaction, starting a 32-round challenge-response game. In each round j, V commits to a bit b_{32-j} via $\texttt{TraceChallenge}_{\texttt{j}}$, directing the search left ($b_{32-j} = 0$) or right ($b_{32-j} = 1$).

P responds with $\texttt{TraceResponse}_{\texttt{j+1}}$, revealing the midpoint E_{n_r}, where $r = 32 - (j+1)$ and $n_r = \sum_{i=r}^{31} b_i \cdot 2^i + 2^r$. After 32 rounds, the final index is $\mathcal{N} = \sum_{i=0}^{31} b_i \cdot 2^i$, with $\mathcal{N}' = \mathcal{N} + 1$. This yields the first divergent pair: agreement on $S_{\mathcal{N}}$, disagreement on $S_{\mathcal{N}'}$.

Instruction Commitment. P must now prove that $S_{\mathcal{N}'} = f_{ST}(S_{\mathcal{N}})$. To do so, they publish a `CommitInstruction` transaction, committing to:

- $pc_\theta = pc_{\mathcal{N}}$ and $pc_{\theta'} = pc_{\mathcal{N}'}$: the program counters of the states $S_{\mathcal{N}}$ and $S_{\mathcal{N}'}$, respectively;
- $insType_\theta \in \mathcal{OP} := \{\mathsf{ADD}, \mathsf{BEQ}, \mathsf{JMP}\}$: 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 $addrA_\theta$, $addrB_\theta$ in $S_{\mathcal{N}}$;
- $valC_\theta$: the memory value at address $addrC_\theta$ in $S_{\mathcal{N}'}$, i.e., after executing $f_{ST}(S_{\mathcal{N}})$.

The Taproot script must enforce that: $(pc_{\theta'}, valC_\theta) = insType_\theta(pc_\theta, valA_\theta, valB_\theta)$. Next, V may challenge this step via several failure cases.

Verifier Challenges. There are five challenge paths to verify the correctness of the transition $S_{\mathcal{N}} \to S_{\mathcal{N}'}$. 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.

[12] They agree by default, since P chooses the initial state S_0 and communicates it to V.

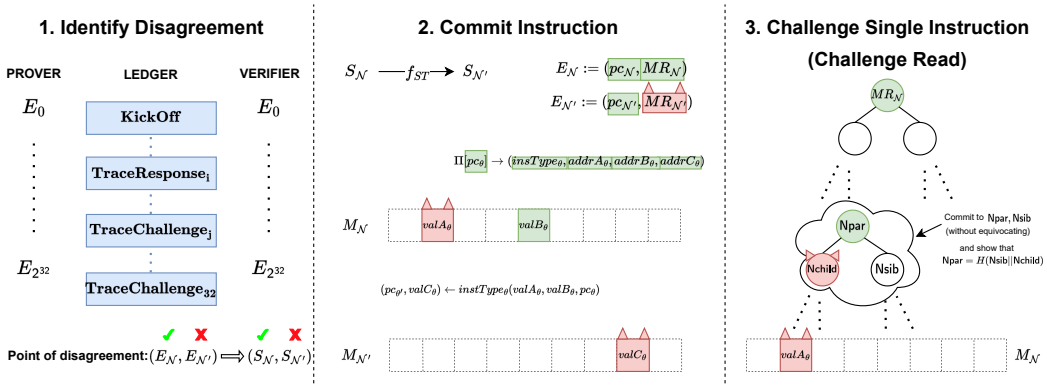

Figure 1: Example of dispute resolution in the BitVM protocol: To resolve a dispute, (1) P and V engage in a bisection game to identify the point of disagreement $(E_{\mathcal{N}}, E_{\mathcal{N}'})$ in their execution traces, indicating a disagreement in the transition from state $S_{\mathcal{N}}$ to $S_{\mathcal{N}'}$. 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.

(i)-(ii) Program Counter (Current/Next). V can dispute that $pc_\theta \neq pc_{\mathcal{N}}$ or $pc_{\theta'} \neq pc_{\mathcal{N}'}$ via `ChallengeCurrPC` or `ChallengeNextPC`. If proven, V claims the funds. If V equivocates on their bitwise commitment to $\mathcal{N}$, P can reveal equivocation via `PunishCurrPC`, claiming all funds instead.

(iii) Instruction Mismatch. Each program line (indexed by pc_θ) is encoded in a tapleaf with the expected $insType_\theta$, $addrA_\theta$, $addrB_\theta$, and $addrC_\theta$. If P commits to inconsistent values, V can reveal this via `DisproveProgram`, spending the corresponding tapleaf. If the program counter matches but any operand or opcode does not, V wins the dispute and claims the funds.

(iv) Read Error. To verify $valA_\theta = M_{\mathcal{N}}[addrA_\theta]$, V initiates a challenge by publishing `ChallengeRead`. A 5-round bisection game over the 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 $M_{\mathcal{N}}[addrA_\theta]$, isolates a disagreement (Npar, Nchild). If P fails to justify it via a sibling hash Nsib (such that H(Nsib||Nchild) = Npar) or equivocates on one of the elements of $\mathcal{P}_R$, V wins via `PunishRead`. Otherwise, P claims the funds after a timeout. The same logic applies to $valB_\theta$ at address $addrB_\theta$.

(v) Write Error. V challenges $valC_\theta \neq M_{\mathcal{N}'}[addrC_\theta]$ by publishing `ChallengeWrite`. The bisection game now runs on two Merkle paths (before and after the write). Again, V identifies a disagreement (Npar, Nchild) and challenges P to provide a consistent sibling node. If P cannot do so, V wins via `PunishWrite`; otherwise, P claims the funds after a timeout.

6 Security Analysis

We now show that BitVM is an *on-chain state verification protocol* that satisfies two key properties: *Balance Security*, which ensures that honest users never lose funds even if their counterparty behaves arbitrarily; and *Rational Correctness*, which guarantees that rational participants always follow the intended optimistic execution path. Our analysis models BitVM as an *Extensive Form Game (EFG)* (see Appendix E), which enables unified reasoning about both Byzantine and rational adversaries. This approach builds on recent work on incentive-compatible Layer-2 protocols [37], and provides a natural way to establish equilibrium guarantees, which are essential in financial settings.[13]

Theorem 6.1 BitVM *is an on-chain state verification protocol that achieves* balance security *and* rational correctness.

Balance Security

We consider two cases: (i) both parties behave honestly, and (ii) one party $A \in \{P, V\}$ deviates at any step. In both scenarios, we prove that the honest party does not lose their funds. We note that if either party deviates during *Setup*, the honest party will refuse to sign the `Setup` transaction, ensuring no coins are locked unless both parties have received all necessary pre-signed transactions (Lemma F.1). Thus, we assume that the setup phase has concluded successfully.

Honest parties. When both parties are honest, BitVM follows an optimistic path: the prover posts the correct computation result on-chain via `CommitComputation`, and after the timelock expires, publishes `Close` to distribute the funds according to f (Lemma F.3).

V honest, P Byzantine. If the prover fails to publish in time `CommitComputation`, either due to inactivity or incorrect computation, or subsequently fails to post `Close`, the verifier can reclaim all the funds locked in the multisignature after the respective timelocks expire (Lemmas F.2, F.3). This mechanism prevents hostage scenarios by ensuring that the verifier can recover their coins in case of non-responsiveness.

If the prover commits to an incorrect result in `CommitComputation`, the verifier initiates the Identify Disagreement phase by publishing `KickOff`. If the prover remains inactive, the verifier can claim the coins after the timelock expires (Lemma F.4). If the phase completes, the verifier obtains a VM step for which the prover has incorrectly committed to the outcome of the state transition function (Algorithm 5) (Lemma F.7).

The prover may have deviated by using an invalid program counter (current or next), performing an incorrect memory read or write, or executing an invalid instruction. For each case, the verifier can post the corresponding on-chain transaction—such as `ChallengeCurrPC`, `ChallengeNextPC`, `ChallengeRead`, `ChallengeWrite`, or `DisproveProgram`—to initiate the appropriate dispute path. This allows the verifier to disprove the prover's computation and claim the funds (Lemma F.14).

P honest, V Byzantine. A malicious verifier may initiate the Dispute Phase by publishing `KickOff` on-chain, even though the prover has correctly committed to the result in `CommitComputation`.

[13] While Universal Composability (UC) is well-suited for modeling arbitrary adversaries, it does not support equilibrium reasoning. Formal tools targeting rational security exist for simple constructions (e.g., Lightning's closing game [16]), but do not scale to BitVM 's combinatorial structure (e.g., bisection-based disputes). Developing general-purpose frameworks for rational security in expressive smart contract systems remains an open challenge.

If the verifier becomes inactive during the Identify Disagreement phase, the prover can claim the funds once the timelock expires (Lemma F.5).

If the phase completes, the verifier must follow up with a challenge by posting one of the transactions `ChallengeCurrPC`, `ChallengeNextPC`, `ChallengeRead`, `ChallengeWrite`, or `DisproveProgram`. Since the prover's commitment is correct, the verifier cannot produce a valid inconsistency and ultimately fails to disprove the computation. In this case, the prover reclaims the funds (Lemma F.15).

Rational Correctness

We now establish that rational participants are incentivized to follow the optimistic execution path of BitVM. Specifically, in Theorem F.16, we prove that the honest strategy profile constitutes a *Subgame Perfect Nash Equilibrium (SPNE)*. The proof proceeds by backward induction on the game tree: in every subgame, deviation results in strictly lower utility, since the honest counterparty can either recover funds via timeouts or successfully disprove incorrect behavior on-chain. These consequences are established through the same mechanisms formalized in Lemmas F.2–F.15.

In particular, if the prover deviates by omitting `CommitComputation`, failing to post `Close`, or committing an invalid transition, the verifier can either reclaim their funds or win the dispute. Conversely, if the verifier initiates an unwarranted dispute, they will be unable to disprove the prover and ultimately forfeit their claim. As any unilateral deviation leads to lower utility, both parties are incentivized to behave honestly. This makes BitVM incentive-compatible: rational players follow the optimistic path without invoking a dispute.

7 Implementation and Evaluation

To show the feasibility of our approach, we implement a prototype of BitVM in JavaScript [3]. In addition to showing how BitVM can be implemented practically, we use it to compute the transaction fees for both an optimistic run and the most expensive dispute branch of BitVM.

We assume constant transaction fees of $3sat/vB$[14], a Bitcoin price of $\$73,000$ (as of May 2026). To make the prototype more efficient, we realize the one-time signatures with Winternitz signatures [33] instead of Lamport signatures. Using Winternitz signatures, both the size of a signature and the size of a public key are around $5vB$ per message bit. As hash function, we use the Bitcoin Script primitive OP_HASH160 [35]. 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 time evaluation accordingly.

Optimistic case. In the optimistic case, three on-chain transactions are published: `Setup`, `CommitComputation`, and `Close`, totaling $1,944vB$[15]. The protocol's execution cost is $5,832$ sat ($4.26). In terms of execution time, once `Setup` is published, BitVM completes in at most 2Δ time, corresponding to 1 day.

Dispute case. We focus on the most expensive path in terms of fees, the *Write Error* path. Overall, 81 transactions are posted on-chain; the path weighs $244,040vB$. The total protocol execution cost is $732ksat$, or about $534, and, once the `Setup` transaction is published on-chain, it takes

[14] In Bitcoin, the size of a SegWit [31] 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.

[15] Verifying an arbitrary program with BitVM is $5\times$ more expensive than a 2-input standard pay-to-public-key-hash transaction ($374vB$).

at most $80 \times \Delta = 40$ days to complete its execution. In case of a dispute, all the fees needed to run the protocol on-chain are covered by the misbehaving party[16].

These costs are acceptable for a market deployment, as witnessed by the massive investments of the industry in the BitVM paradigm [23, 39, 17].

8 Bridge Application

In this section, we leverage BitVM to instantiate a bridge application between the Bitcoin ledger and a sidechain system running a distributed ledger protocol, as defined in Definition 1, which satisfies *safety* and *liveness*. This bridge enables users to mint (wrapped) Bitcoin tokens on the sidechain and later redeem them back on the Bitcoin blockchain and is secure, assuming only existential honesty of the participants.

We first outline the bridge protocol, then present some high-level security arguments and conclude this section with an evaluation.

8.1 A BitVM-based Bridge Design

Consider two users, Alice and Bob: Alice mints tokens on the sidechain, transfers them to Bob, who then redeems the equivalent amount back to Bitcoin. The protocol assumes a committee of n members active during the BridgeSetup phase[17], with at least one assumed honest during execution. We illustrate the bridge protocol in Fig. 2.

During the *BridgeSetup* phase, the committee members pre-sign the transactions required for the protocol execution. The core procedures are *PegIn* (Alice mints tokens) and *PegOut* (Bob redeems them). To enable `PegOut`, a committee member fronts coins to Bob and later reclaims them in the *reimbursement* phase[18]. We refer to this member as the *operator*. For each operator, the remaining committee members act like *challengers*: they ensure that only honest operators can reclaim funds in the reimbursement phase by disproving any member falsely claiming to have fronted coins to Bob.

BridgeSetup. In that phase, the committee pre-signs, before the PegIn procedure is executed, specific transactions to reimburse an honest operator or punish a misbehaving one.

First, for each distinct pair of committee members denoted c_i and c_j, c_i pre-signs and forwards the transaction $\texttt{Slash}_{(\texttt{i,j})}$ to c_j, which c_j can later publish to punish c_i upon misbehavior, as explained below. Second, for each c_i, all the committee members pre-sign the $\texttt{Take}_{\texttt{i}}$ transaction that refunds c_i, as described below. If a committee member stops collaborating during setup, it is kicked out and the procedure restarts.

PegIn. Alice deposits u coins on Bitcoin via a `PegIn` transaction. This transaction has a single output that deposits the u coins into the multi-signature wallet controlled by the n committee members. The sidechain verifies the inclusion of the `PegIn` transaction in the Bitcoin blockchain using a Bitcoin light client. Once confirmed, the sidechain mints u wrapped tokens to Alice's account.

[16] Since we do not know beforehand if a party will misbehave, both P and V need to put enough collateral to cover for the fees. To keep low the opportunity cost of running BitVM, we aim to have a protocol with minimal on-chain footprint.

[17] Otherwise, the committee is reshuffled until BridgeSetup completes.

[18] To coordinate among committee members, we can define an operator schedule, for example, in a round-robin fashion.

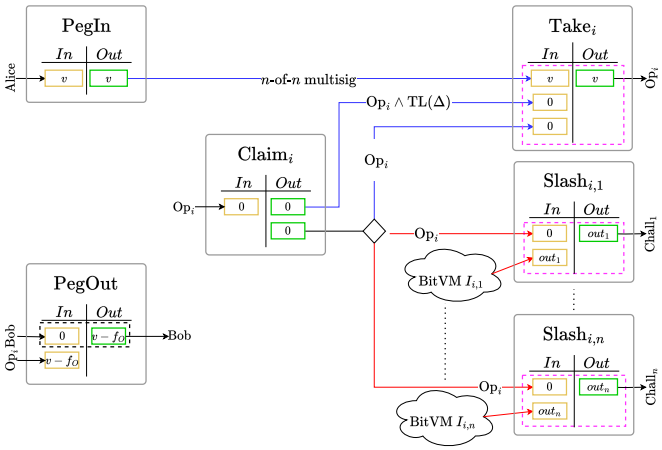

Figure 2: Illustration of BitVM-based bridge. For readability, we represent transactions as grey boxes; the value carried by transaction inputs and outputs is written inside a yellow and green rectangle, respectively. Pink dashed rectangles around the inputs and outputs of `Take` and `Slash` denote that the transaction is pre-signed during BridgeSetup, while the black dashed rectangle indicates which portion of the PegOut transaction Bob signed at PegOut time. Above the arrows, we denote the condition that unlocks the output from which the arrow starts. The arrows are blue if they represent a path taken by the operator, and red if the path is taken by one of the other committee members. The amount out_j, output of the BitVM instance I_i, j, depends on the funds still present after the BitVM dispute phase.

PegOut. To withdraw the u coins, Bob first publishes a `Burn` transaction on the sidechain. Then, he constructs a `PegOut` transaction with a single output allocating $u - f_0$ coins to his Bitcoin address, where f_0 is a service fee. Bob broadcasts this `PegOut` request to the committee members. At least one committee member effectively fronts $u - f_0$ coins from their own funds to fulfill the withdrawal, in exchange for f_0.

Reimbursement. To facilitate the reimbursement of the committee member that fronted its coins to Bob, we proceed with the following construction.

Claim. Each committee member c_i can claim that they fronted coins to Bob by posting the transaction $\texttt{Claim}_\texttt{i}$ on-chain. $\texttt{Claim}_\texttt{i}$ takes as input an empty transaction from the committee member c_i and has an empty output that can be spent after a timelock $\mathsf{TL}(\Delta)$ expires, along with a *connector* output, i.e., a transaction output given as input to different transactions to guarantee that only one of them will appear on-chain.

Slashing Mechanism. If c_i falsely claims to have fronted coins to Bob, any c_j can later publish on-chain $\texttt{Slash}_{(\texttt{i},\texttt{j})}$ to punish c_i. To achieve that, we employ a number of $O(n^2)$ BitVM instances as follows. For every pair of distinct committee members c_i and c_j, we consider the instance $I_{i,j}$ where c_i acts as the prover and c_j as the verifier.[19]

For a fixed i, the program Π, hardcoded into each instance $I_{i,j}$, verifies the following conditions: a) Bob's Burn transaction exists on the sidechain, and, b) some `PegOut` transaction, where operator

[19]The only difference from the current protocol is that, in this case, the verifier commits the `BridgeSetup` on-chain to initiate a dispute.

c_i fronts the coins to Bob, exists on the Bitcoin ledger. For example, the program Π could encode a zk-SNARK verifier that verifies a proof of (1) and (2), making use of a light client of both the sidechain and the Bitcoin ledger, as shown in, e.g., [30, 10, 4].

The transaction $\texttt{Slash}_{(\texttt{i,j})}$ takes as input: i) the connector output of $\texttt{Claim}_\texttt{i}$ and ii) an input from any transaction where c_j wins the instance $I_{i,j}$ (i.e., any transaction where all the coins of the multi-signature are attributed to c_j) and has a single output where all the remaining coins locked in the instance $I_{i,j}$ are given to c_j.

Take. The committee member c_i that fronted coins to Bob, can finally retrieve their funds by publishing transaction $\texttt{Take}_\texttt{i}$ on-chain. The transaction $\texttt{Take}_\texttt{i}$ has the following inputs: i) the output of PegIn, ii) the first output of $\texttt{Claim}_\texttt{i}$, and iii) the connector output of $\texttt{Claim}_\texttt{i}$, and has a single output where the coins of the first input are transferred to c_i.

For each operator c_i, all committee members presign $\texttt{Take}_\texttt{i}$ only if: (i) $\texttt{Claim}_\texttt{i}$ is constructed according to the protocol, and (ii) they have received all the pre-signed transactions related to the BitVM instance $I_{i,j}$.

8.2 Security Arguments

Since the sidechain ensures `PegIn` succeeds, our goal is to guarantee that a successful `PegIn` implies a successful `PegOut`: if Alice mints and transfers coins to Bob, Bob can eventually reclaim the equivalent amount on Bitcoin. This relies on two properties:

Bridge Safety (no false claims): A committee member cannot falsely claim to have fronted coins to Bob. Otherwise, funds in the multisig are stolen, and future honest operators are unable to reclaim their coins.

Bridge Liveness (honest redemption): Eventually, an honest operator will front coins and reclaim them.

Safety holds because if a malicious member c_i posts $\texttt{Claim}_\texttt{i}$ without first posting `PegOut`, an honest challenger c_j can initiate BitVM instance $I_{i,j}$ and post $\texttt{Slash}_{\texttt{i,j}}$, which spends the connector output of $\texttt{Claim}_\texttt{i}$ and invalidates $\texttt{Take}_\texttt{i}$. Liveness holds because an honest member c_i who posts `PegOut` is protected: no malicious c_j can succeed with $\texttt{Slash}_{\texttt{i,j}}$, and after the $\texttt{Claim}_\texttt{i}$ timelock, c_i can post $\texttt{Take}_\texttt{i}$ to reclaim their funds.

8.3 Evaluation

Table 3 summarizes the transaction costs associated with each phase of the bridge protocol (to see how they are computed, refer to Appendix G). Specifically, in the optimistic scenario, where all committee members behave honestly, executing an instance of our bridge protocol requires only four on-chain transactions—`PegIn`, `PegOut`, `Claim`, and `Take`—resulting in a minimal total cost of approximately \$2.04, which can be covered by the application fee f_0 paid by Bob.

On the contrary, if a challenger opens up a dispute, we have the following cases:

1. if the operator was honest, they only pay fees for `PegIn`, `PegOut`, `Claim`, and `Take` (approximately \$2.04, covered by Bob's fee f_0). The malicious challenger pays transaction fees associated with the execution of a BitVM instance (i.e., \$534);
2. if the operator was illegitimately trying to reclaim coins by publishing `Claim`, they will be slashed by the challenger and will pay the cost of a BitVM instance and the fees of `Slash` and `Claim`. This amounts to \$534.

Table 3: Evaluation of a BitVM-based bridge instance w.r.t. transaction sizes and on-chain costs.

Tx	Size (vB)	On-chain cost ($)
PegIn	117	0.26
PegOut	180	0.39
Claim	160	0.35
Slash	212	0.46
Take	475	1.04

9 Conclusion and Future Work

This work presents BitVM, the first protocol to enable general-purpose, trustless computation on Bitcoin without requiring consensus changes, or additional assumptions, e.g., trusted hardware or semi-trusted oracles. By combining off-chain execution with a Bitcoin-compatible dispute resolution protocol, BitVM achieves quasi-Turing completeness using existing scripting capabilities. To demonstrate the applicability of BitVM, we construct a trust-minimizing bridge and provide a prototype implementation with a concrete cost analysis.

BitVM extends the design space for Bitcoin-based applications, enabling programmable logic and verifiable off-chain computation in a trustless setting. Its ability to condition Bitcoin transactions on arbitrary program outputs opens new avenues for decentralized infrastructure anchored in Bitcoin's security model.

Several directions remain for future work. First, while BitVM currently supports two-party interactions, generalizing the protocol to support multiparty or permissionless settings—such as decentralized oracle networks or bridges—is an important next step. Second, further reducing on-chain cost through improved dispute resolution mechanisms, such as more efficient encodings or batched verifications, could improve scalability. Third, building higher-level tooling, including compilers or domain-specific languages, would help lower the barrier to adoption and enable broader experimentation with complex BitVM-based applications.

We believe BitVM marks a foundational step in unlocking the next generation of Bitcoin-native applications.

References

[1] tbtc. https://tbtc.network.

[2] Bitcoin transactions. https://en.bitcoin.it/wiki/Transaction, 2025. Accessed: 2025-04.

[3] Bitvm poc implementation. https://github.com/BitVM/bitvm-js, 2025.

[4] Bob hybrid l2 technical blueprint. https://blog.gobob.xyz/posts/bob-hybrid-l2-technical-blueprint, 2025. Accessed: 2025-04.

[5] What are trustless bitcoin vaults?, 2025. Accessed: 2025-11. URL: https://babylonlabs.io/learn/what-are-trustless-bitcoin-vaults.

[6] Ioannis Alexopoulos, Zeta Avarikioti, Paul Gerhart, Matteo Maffei, and Dominique Schröder. BitPriv: A privacy-preserving protocol for DeFi applications on bitcoin. Cryptology ePrint Archive, Paper 2025/1575, 2025. URL: https://eprint.iacr.org/2025/1575.

[7] Lukas Aumayr, Zeta Avarikioti, Matteo Maffei, Giulia Scaffino, and Dionysis Zindros. Blink: An optimal proof of proof-of-work. In *Financial Cryptography and Data Security*, 2025.

[8] Lukas Aumayr, Oguzhan Ersoy, Andreas Erwig, Sebastian Faust, Kristina Hostáková, Matteo Maffei, Pedro Moreno-Sanchez, and Siavash Riahi. Generalized channels from limited blockchain scripts and adaptor signatures. In *ASIACRYPT*, 2021.

[9] Adam Back, Matt Corallo, Luke Dashjr, Mark Friedenbach, Gregory Maxwell, Andrew Miller, Andrew Poelstra, Jorge Timón, and Pieter Wuille. Enabling blockchain innovations with pegged sidechains, 2014. `https://blockstream.com/sidechains.pdf`.

[10] Ekrem Bal, Lukas Aumayr, Atacan İyidoğan, Giulia Scaffino, Hakan Karakuş, Cengiz Eray Aslan, and Orfeas Stefanos Thyfronitis Litos. Clementine: A collateral-efficient, trust-minimized, and scalable bitcoin bridge. Cryptology ePrint Archive, Paper 2025/776, 2025. URL: `https://eprint.iacr.org/2025/776`.

[11] Massimo Bartoletti, Tiziana Cimoli, and Roberto Zunino. Fun with bitcoin smart contracts. In *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*, pages 432–449, Cham, 2018.

[12] Massimo Bartoletti, Stefano Lande, and Roberto Zunino. Bitcoin covenants unchained. In *Leveraging Applications of Formal Methods, Verification and Validation: Applications*, pages 25–42, 2020.

[13] Massimo Bartoletti, Riccardo Marchesin, and Roberto Zunino. Secure compilation of rich smart contracts on poor UTXO blockchains. In *9th IEEE European Symposium on Security and Privacy, EuroS&P*, 2024. URL: `https://doi.org/10.1109/EuroSP60621.2024.00021`, `doi:10.1109/EUROSP60621.2024.00021`.

[14] Massimo Bartoletti and Roberto Zunino. Bitml: A calculus for bitcoin smart contracts. In *ACM CCS*, page 83–100, 2018. `doi:10.1145/3243734.3243795`.

[15] Dan Boneh and Victor Shoup. A graduate course in applied cryptography. *Draft 0.6*, 2023. `https://toc.cryptobook.us/`.

[16] Lea Salome Brugger, Laura Kovács, Anja Petkovic Komel, Sophie Rain, and Michael Rawson. Checkmate: automated game-theoretic security reasoning. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 1407–1421, 2023.

[17] Citrea. Bitcoin settlement trust-minimized btc bridge: Bitvm, 2024. Accessed: 2025-11. URL: `https://docs.citrea.xyz/technical-specs/characteristics/bitcoin-settlement-trust-minimized-btc-bridge/bitvm`.

[18] Poulami Das, Lisa Eckey, Tommaso Frassetto, David Gens, Kristina Hostáková, Patrick Jauernig, Sebastian Faust, and Ahmad-Reza Sadeghi. FastKitten: Practical smart contracts on bitcoin. In *USENIX Security 19*, 2019.

[19] Contributor DLC. Discreet log contracts: An overview. 2018. URL: `https://adiabat.github.io/dlc.pdf`.

[20] Stefan Dziembowski, Sebastian Faust, and Kristina Hostáková. General state channel networks. In *ACM CCS*, page 949–966, 2018. `doi:10.1145/3243734.3243856`.

[21] Esolangs. Addleq, 2025. `https://esolangs.org/wiki/Addleq`.

[22] Ethereum Foundation. State channels, 2024. Accessed: 2025-04. URL: https://ethereum.org/en/developers/docs/scaling/state-channels/.

[23] FairGate Labs. Bitvmx & bitvm, 2024. Accessed: 2025-11. URL: https://fairgate.io.

[24] Tommaso Frassetto, Patrick Jauernig, David Koisser, David Kretzler, Benjamin Schlosser, Sebastian Faust, and Ahmad-Reza Sadeghi. Pose: Practical off-chain smart contract execution. In *NDSS Symposium*. Internet Society, 2023. URL: http://dx.doi.org/10.14722/ndss.2023.23118, doi:10.14722/ndss.2023.23118.

[25] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. *J. ACM*, 71(4), August 2024. doi:10.1145/3653445.

[26] Sanjam Garg, Dimitris Kolonelos, Mikhail Sergeevitch, Srivatsan Sridhar, and David Tse. BABE: Verifying proofs on bitcoin made 1000x cheaper. Cryptology ePrint Archive, Paper 2026/065, 2026. URL: https://eprint.iacr.org/2026/065.

[27] Harry Kalodner, Steven Goldfeder, Xiaoqi Chen, S Matthew Weinberg, and Edward W Felten. Arbitrum: Scalable, private smart contracts. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1353–1370, 2018.

[28] Leslie Lamport. Constructing digital signatures from a one way function. Technical Report CSL-98, October 1979. https://www.microsoft.com/en-us/research/publication/constructing-digital-signatures-one-way-function/.

[29] Robin Linus, Lukas Aumayr, Zeta Avarikioti, Matteo Maffei, Andrea Pelosi, Orfeas Thyfronitis Litos, Christos Stefo, David Tse, and Alexei Zamyatin. Bridging bitcoin to second layers via BitVM2. In *35th USENIX Security Symposium (USENIX Security 26)*, 2026.

[30] Robin Linus, Lukas Aumayr, Alexei Zamyatin, Andrea Pelosi, Zeta Avarikioti, and Matteo Maffei. BitVM2: Bridging bitcoin to second layers, 2024. Accessed: 2025-04. URL: https://bitvm.org/bitvm_bridge.pdf.

[31] Eric Lombrozo and Pieter Wuille. Bip 141, segregated witness, 2015. Accessed: 2025-04. URL: https://github.com/bitcoin/bips/blob/master/bip-0141.mediawiki.

[32] Varun Madathil, Sri AravindaKrishnan Thyagarajan, Dimitrios Vasilopoulos, Lloyd Fournier, Giulio Malavolta, and Pedro Moreno-Sanchez. Cryptographic oracle-based conditional payments. In *NDSS Symposium*, 2023. URL: https://dx.doi.org/10.14722/ndss.2023.24024, doi:10.14722/ndss.2023.24024.

[33] Ralph C Merkle. A certified digital signature. In *Conference on the Theory and Application of Cryptology*, pages 218–238. Springer, 1989.

[34] Martin J. Osborne and Ariel Rubinstein. *A Course in Game Theory*. MIT Press, Cambridge, MA, 1994.

[35] Flavius Plesu. Turing completeness in evm machines, 2023. Accessed: 2025-04. URL: https://www.flavius.io/media/turing-completeness-in-evm-machines.

[36] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments. 2016.

[37] Sophie Rain, Georgia Avarikioti, Laura Kovács, and Matteo Maffei. Towards a game-theoretic security analysis of off-chain protocols. In *IEEE CSF*, pages 107–122, 2023. doi:10.1109/CSF57540.2023.00003.

[38] Robin Linus. Bitvm: Compute anything on bitcoin, 2023. Accessed: 2025-11. URL: https://bitvm.org/bitvm.pdf.

[39] Sovryn. Bitcoinos, 2024. Accessed: 2025-11. URL: https://sovryn.com/bitcoinos.

[40] Apostolos Tzinas, Srivatsan Sridhar, and Dionysis Zindros. On-chain timestamps are accurate. Cryptology ePrint Archive, Paper 2023/1648, 2023. URL: https://eprint.iacr.org/2023/1648.

[41] Robin Linus Woll, Ioannis Alexopoulos, Lukas Aumayr, Zeta Avarikioti, Matteo Maffei, and David Tse. BitVM3: Efficient bitcoin bridges via garbled circuits. In *ACM Conference on Computer and Communications Security (CCS)*, 2026.

[42] Pieter Wuille, Jonas Nick, and Anthony Towns. Bip 0341, taproot: Segwit version 1 spending rules, jan 2020. https://en.bitcoin.it/wiki/BIP_0341.

A Table of Notation

For convenience, we summarize in Table 4 the symbols used throughout the paper, grouped by topic. We refer the reader to Appendix C for the transactions and the locking scripts of the protocol, and to the corresponding algorithms for their pseudocode.

Table 4: Summary of the notation used in this paper.

Symbol	Description
General notation	
$A[i]$	i-th element of a sequence $A := (a_1, \ldots, a_n)$
$A[i:j]$	Subsequence $(a_i, \ldots, a_j)$ of A
$\lvert A\rvert$	Length of the sequence A
$\lvert s\rvert_{\mathsf{bit}}$	Bit length of a string $s \in \{0,1\}^*$
$A \preceq B$	Sequence A is a prefix of sequence B
$\wedge$, $\vee$	Logical conjunction and disjunction, used to combine spending conditions
$\bot$	Undefined value; also denotes VM termination
$*$	Generic transaction input, witness, or output not relevant to the protocol
κ	Security parameter
$H : \{0,1\}^* \to \{0,1\}^\kappa$	Hash function, modeled as a random oracle
System model and distributed ledger	
P, V	Prover and verifier, the two parties running BitVM

continued on next page

Table 4 (continued)

Symbol	Description
L	Transaction ledger output by `read()`
L_r^P	Ledger output by `read()` at party P at the end of round r
u	Liveness parameter: number of rounds within which a written transaction appears in every honest ledger
$\mathsf{st}(L)$	Ledger state induced by L
$s \in \mathsf{st}(L)$	String s contained in some transaction of L
$\mathsf{bal}_L(P)$	Balance of party P in the state induced by L
in_P, in_V	Monetary inputs (deposits) of P and V
$w_L(P, \mathsf{tx})$	On-chain monetary utility of transaction tx for party P
Δ	Time parameter used in the relative timelocks of the protocol
On-chain state verification protocol	
$\Pi : \mathcal{S} \to \mathcal{O}$	Program (computable function) mapping states to outcomes
$\mathcal{S}$, $\mathcal{O}$	Set of states and set of outcomes of Π
$x \in \mathcal{S}$	Input of the program Π
$f : \mathcal{O} \to \mathbb{R}^2_{\geq 0}$	Outcome mapping function, assigning a pair of utilities to each outcome
(v_P, v_V)	Pair of utilities returned by f, with $v_P + v_V \leq \mathsf{in}_P + \mathsf{in}_V$
$f_A(\Pi(x))$	Utility of party $A \in \{P, V\}$ in $f(\Pi(x))$
I	Instance returned by $\mathsf{setup}(\mathsf{in}_P, \mathsf{in}_V, \Pi, f)$
T	Set of transactions included in the ledger as a result of an execution
Keys, signatures, and units	
$(\mathsf{pk}_U, \mathsf{sk}_U)$	Public/secret key pair identifying user U
$\sigma_U(m)$	Digital signature of U over message m
σ_{PV}	2-of-2 multisignature of P and V over the transaction body
B, sat	Bitcoin and satoshi, $1\,\mathsf{sat} = 10^{-8}\mathsf{B}$
vB	Virtual byte, the unit of SegWit transaction size
d	Amount of coins carried by the multisignature output along the dispute chain
Lamport commitments	
$h : X \to Y$	One-way function, $X = \{0,1\}^*$, $Y = \{0,1\}^\lambda$
λ	Security parameter of the one-way function h
$\mathcal{M} = \{0,1\}^\ell$	Message space of ℓ-bit messages
$(\mathsf{pk}_\mathcal{M}, \mathsf{sk}_\mathcal{M})$	Lamport key pair for the message space $\mathcal{M}$
$x[i,j]$, $y[i,j]$	Secret preimage and its image at row i, column j of $\mathsf{sk}_\mathcal{M}$, $\mathsf{pk}_\mathcal{M}$
c_m	Lamport commitment (signature) on the message m

continued on next page

Table 4 (continued)

Symbol	Description
c_0, c_1	Conflicting commitments on the same bit, witnessing an equivocation
Virtual machine	
$\Gamma := \langle \Pi, \mathsf{MemLen}, n, \mathsf{final} \rangle$	VM instance
Γ^A	VM instance executed by party $A \in \{P, V\}$
MemLen	Number of VM memory cells (2^{32} in our instantiation)
n	Largest value storable at a memory address ($2^{32} - 1$ in our instantiation)
$\mathcal{A} = \{0, \ldots, \mathsf{MemLen} - 1\}$	Set of VM memory addresses
$addr$	VM memory address, $addr \in \mathcal{A}$
$M \in \mathcal{M}$	VM memory, $\mathcal{M} := \{0, \ldots, n\}^{\mathsf{MemLen}}$
$M[addr]$	Value stored in the memory at address $addr$
$pc \in \mathcal{PC}$	Program counter, $\mathcal{PC} := \{0, \ldots, \ell - 1\} \cup \{\perp\}$
ℓ	Maximum program length ($\ell \leq 2^{32}$)
$\mathcal{OP}$	Set of CPU instructions executable by the VM
$f_{\mathcal{OP}}$	CPU instruction, mapping $(pc, valA, valB)$ to $(pc, valC)$
ADD, BEQ, JMP	The Turing-complete instruction set used in our VM instance
$\mathcal{I}$	Set of VM instructions $(f_{\mathcal{OP}}, addrA, addrB, addrC)$
$\Pi \in \mathcal{I}^\ell$	VM program, an ordered sequence of ℓ instructions
$\Pi[pc]$	Instruction of Π at program counter pc
$S := (M, pc, \Pi)$	VM state
$\mathcal{S} := \mathcal{M} \times \mathcal{PC} \times \mathcal{I}^\ell$	Set of all VM states
$S_0 := (M, 0, \Pi)$	Initial VM state
S_i	VM state at step i; correct iff $S_i = f_{ST}^i(S_0)$
S_i^A	VM state at step i claimed by party A
f_{ST}	State transition function
f_{ST}^i	i-fold application of f_{ST}
final	Maximum number of state transitions (2^{32} in our instantiation)
$\Pi(S_0) := f_{ST}^{\mathsf{final}}(S_0)$	Outcome (final state) of the computation
MR_i	Root of the Merkle tree whose leaves are the entries of M_i
$\mathsf{MerkleTree}_{M_i}$	Merkle tree built over the memory M_i
$E_i := (MR_i, pc_i)$	i-th execution trace element
E_i^A	i-th trace element claimed by party A
$\mathsf{ExecTrace} := (E_0, \ldots, E_{\mathsf{final}})$	VM execution trace
$\mathsf{ExecTrace}^A$	Execution trace claimed by party A
Bisection games	
A_P, A_V	Sequences held by the prover and the verifier in a bisection game

continued on next page

Table 4 (continued)

Symbol	Description
$(A[i], A[i+1])$	Point of disagreement: consecutive elements on which the parties agree, resp. disagree
l, r, m	Left boundary, right boundary, and midpoint of the search interval
b_i	Bit committed by V at round i, directing the search left (0) or right (1)
Dispute resolution	
$\mathcal{N}$	Index of the last trace element the parties agree on, $\mathcal{N} = \sum_{i=0}^{31} b_i 2^i$
$\mathcal{N}'$	First index of disagreement, $\mathcal{N}' = \mathcal{N} + 1$
n_r	Midpoint index revealed at round r, $n_r = \sum_{i=r}^{31} b_i 2^i + 2^r$
pc_θ, $pc_{\theta'}$	Program counters at steps $\mathcal{N}$ and $\mathcal{N}'$ committed by P
$insType_\theta$	Instruction type at $\Pi[pc_\theta]$, $insType_\theta \in \mathcal{OP}$
$addrA_\theta$, $addrB_\theta$, $addrC_\theta$	Memory addresses referenced by $\Pi[pc_\theta]$
$valA_\theta$, $valB_\theta$	Values read at $addrA_\theta$, $addrB_\theta$ in $S_\mathcal{N}$
$valC_\theta$	Value written at $addrC_\theta$ in $S_{\mathcal{N}'}$
$\mathcal{P}_R$	Merkle path $(MR_\mathcal{N}, \ldots, M_\mathcal{N}[addrA_\theta])$, used in the read bisection game
$\mathcal{P}_W$, $\mathcal{P}'_W$	Merkle paths to $M_\mathcal{N}[addrC_\theta]$ and $M_{\mathcal{N}'}[addrC_\theta]$, used in the write bisection game
b'_i	Bit committed by V at round i of the read/write bisection game
$\mathcal{N}_{Mer}$	Index of the point of disagreement on a Merkle path, $\mathcal{N}_{Mer} = \sum_{k=0}^{4} b'_k 2^k$
d_i	Index of the Merkle-path element revealed at a given round
Node_{d_i}, Node'_{d_i}	Merkle-path nodes committed by P in $\mathcal{P}_R/\mathcal{P}_W$ and in $\mathcal{P}'_W$
Npar, Nchild	Parent and child node forming the point of disagreement
Npar′, Nchild′	Corresponding nodes on the post-write path $\mathcal{P}'_W$
Nsib	Sibling node provided by P, s.t. $H(\mathsf{Nsib}\|\mathsf{Nchild}) = \mathsf{Npar}$
v_{pos}	Bit encoding whether Nchild is the left (0) or right (1) child of Npar
Extensive form games	
$G = (N, H, P, u)$	Extensive form game (EFG)
N	Set of players of the EFG
H, $T \subseteq H$	Set of histories and set of terminal histories
$P(h)$	Next-player function
u	Utility function
$A(h)$	Set of actions available after history h
σ_p	Strategy of player p

continued on next page

Table 4 (continued)

Symbol	Description
$\sigma = (\sigma_1, \ldots, \sigma_n)$	Joint strategy (strategy profile)
$\varphi(h)$, $\sigma\|_h$	Subgame rooted at h, and strategy restricted to it
Γ, Γ'	Game trees representing BitVM
$\gamma \subseteq \Gamma$	Subtree of Γ describing executions in which one party is honest
SPNE	Subgame Perfect Nash Equilibrium
Security analysis	
N_1, N_2	Number of VM execution steps and size of the VM memory
$\tilde{x}$, $\tilde{\tilde{x}}$	$\log(x)$ and $\log(\log(x))$
f	Constant transaction fee
α, β	Collateral of P and V, net of the fees reserved for the longest execution path
$\langle u \rangle_A$	Balance account of party A, holding u coins
u_1, u_2	Utilities of the alternative strategies compared in a lemma
f_P, f_V	Coins assigned to P and V by the outcome mapping function
Bridge application	
n (bridge)	Size of the bridge committee
c_i, c_j	Committee members: the operator, resp. a challenger
$I_{i,j}$	BitVM instance with c_i as prover and c_j as verifier
t-of-n	Threshold assumption; BitVM reduces it to $t = 1$ (existential honesty)
u (bridge)	Amount of coins pegged in by Alice
f_0	Service fee paid by Bob to the operator
out_j	Funds remaining in instance $I_{i,j}$ after the dispute phase

B 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 [42], also known as Taptrees, enable a UTXO to be spent by satisfying one of several possible spending conditions. These conditions, referred to as 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), and with $h(x)$ the hash of x.

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.

C The BitVM protocol specification

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.

C.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 Appendix C, except `Setup`. Whenever such a transaction contains a new Lamport public key, the corresponding party creates one using $\mathsf{Lamp.KeyGen}$ 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 in_P₿) and V (holding (in_V₿), 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).

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

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

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

C.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*.

C.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 $E_{\mathsf{final}} := (MR_{\mathsf{final}}, pc_{\mathsf{final}})$.

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

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

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 $\texttt{Close}_1, \ldots,$

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

function $\mathsf{CloseScript_i}(\sigma_{PV},\ MR_{\mathsf{final}},\ c_{MR_{\mathsf{final}}})$
  $\mathsf{TL}(\Delta)$;
  $\mathsf{CheckMSigVerify}_{\mathsf{pk}_{PV}}(\sigma_{PV})$;
  $\mathsf{CheckCommVerify}_{\mathsf{pk}_{MR_{\mathsf{final}}}}(MR_{\mathsf{final}}, c_{MR_{\mathsf{final}}})$;
  if $MR_{\mathsf{final}} = \mathcal{MR}_i$ **then**
    return True;
  return False.

$\texttt{Close}_{\texttt{m}}$. Each of these transactions distributes the funds according to the outcome mapping function f, applied to one of the possible results of the computation[20]. If P does not publish any $\texttt{Close}_{\texttt{i}}$ transaction after that $\mathsf{TL}(2\Delta)$ expires after the publication of `CommitComputation` transaction, V can unlock `CommitComputation` output with their signature and claim all the funds.

Transaction $\texttt{Close}_{\texttt{i}}$ (cf. Eq. (3)) spends the output of `CommitComputation` by unlocking $\mathsf{CloseScript_i}$ and creates two outputs. The first output carries o_P₿ and can be 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₿ and can be unlocked by V.

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

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

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

[20]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.

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

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

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

The parties engage in an on-chain interactive protocol known as dispute bisection game (cf. Appendix D.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 $n_{31} = 1 \cdot 2^{31}$.

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

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

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

Next, V publishes the $\texttt{TraceChallenge}_1$ transaction (cf. Eq. (6)), committing to bit b_{31} in the witness.

$$
\begin{aligned}
&\texttt{TraceChallenge}_1 := \\
&\Big(in = [(\texttt{TraceResponse}_1, 0, \mathsf{RespScript}_1)], \\
&\ wit = [(\sigma_{PV}, b_{31}, c_{b_{31}})], \\
&\ out = [(d\ddot{\mathrm{B}}; \langle \mathsf{ChallScript}_2, \mathsf{TL}(\Delta) \wedge \mathsf{CheckSig}_{\mathsf{pk}_V} \rangle)]\Big).
\end{aligned} \tag{6}
$$

During the dispute bisection game, P publishes transactions $\texttt{TraceResponse}_\mathsf{i}$ (cf. Eq. (7)), with $\mathsf{i} = 1, \ldots, 32$, and V publishes transactions $\texttt{TraceChallenge}_\mathsf{j}$ (cf. Eq. (8)), with $\mathsf{j} = 1, \ldots, 31$.

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

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

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

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

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

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 $\mathsf{ADD}(pc_\theta, valA_\theta, valB_\theta) = (pc_{\theta'}, valC_\theta)$ 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 C.4.

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: $\quad\mathsf{CheckMSigVerify}_{\mathsf{pk}_{PV}}(\sigma_{PV})$;
3: $\quad\mathsf{CheckCommVerify}_{\mathsf{pk}_{pc_\theta}}(pc_\theta, c_{pc_\theta})$;
4: $\quad\mathsf{CheckCommVerify}_{\mathsf{pk}_{pc_{\theta'}}}(pc_{\theta'}, c_{pc_{\theta'}})$;
5: $\quad\mathsf{CheckCommVerify}_{\mathsf{pk}_{insType_\theta}}(insType_\theta, c_{insType_\theta})$;
6: $\quad\mathsf{CheckCommVerify}_{\mathsf{pk}_{addrA_\theta}}(addrA_\theta, c_{addrA_\theta})$;
7: $\quad\mathsf{CheckCommVerify}_{\mathsf{pk}_{addrB_\theta}}(addrB_\theta, c_{addrB_\theta})$;
8: $\quad\mathsf{CheckCommVerify}_{\mathsf{pk}_{addrC_\theta}}(addrC_\theta, c_{addrC_\theta})$;
9: $\quad\mathsf{CheckCommVerify}_{\mathsf{pk}_{valA_\theta}}(valA_\theta, c_{valA_\theta})$;
10: $\quad\mathsf{CheckCommVerify}_{\mathsf{pk}_{valB_\theta}}(valB_\theta, c_{valB_\theta})$;
11: $\quad\mathsf{CheckCommVerify}_{\mathsf{pk}_{valC_\theta}}(valC_\theta, c_{valC_\theta})$;
12: $\quad$**if** $insType_\theta = \mathsf{ADD} \wedge \mathsf{ADD}(pc_\theta, valA_\theta, valB_\theta) = (pc_{\theta'}, valC_\theta)$ **then**
13: $\quad\quad$**return** True;
14: $\quad$**else**
15: $\quad\quad$**return** False.

C.4 Dispute Resolution

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

$$
\begin{aligned}
&\texttt{CommitInstruction} := \\
&\Big(in = [(\texttt{TraceChallenge}_{32}, 0, \mathsf{OPScript})], \\
&\ wit = [(\sigma_{PV}, pc_\theta, c_{pc_\theta}, pc_{\theta'}, c_{pc_{\theta'}}, insType_\theta, \\
&\ c_{insType_\theta}, addrA_\theta, c_{addrA_\theta}, addrB_\theta, c_{addrB_\theta}, addrC_\theta, \\
&\ c_{addrC_\theta}, valA_\theta, c_{valA_\theta}, valB_\theta, c_{valB_\theta}, valC_\theta, c_{valC_\theta})], \\
&\ out = [(d\text{Ƀ}; \langle \mathsf{CheckMSig}_{\mathsf{pk}_{PV}}, \{\mathsf{CIScriptPCCurr_i}\}_{\mathsf{i}\in\{1,\ldots,32\}}, \\
&\ \{\mathsf{CIScriptPCNext_i}\}_{\mathsf{i}\in\{1,\ldots,32\}}, \{\mathsf{CIScriptInstr_j}\}_{\mathsf{j}\in\{1,\ldots,\ell\}}, \\
&\ \mathsf{TL}(\Delta) \wedge \mathsf{CheckSig}_{\mathsf{pk}_P}\rangle)]\Big).
\end{aligned}
\tag{10}
$$

The tapleaf that P unlocks when publishing `CommitInstruction` is

$$\mathsf{OPScript} \in \{\mathsf{ADDScript}, \mathsf{BEQScript}, \mathsf{JMPScript}\}.$$

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.

C.4.1 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 $\mathsf{CIScriptPCCurr_0}$, ..., $\mathsf{CIScriptPCCurr_{31}}$. 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.

$$
\begin{aligned}
&\texttt{ChallengeCurrPC} := \\
&\Big(in = [(\texttt{CommitInstruction}, 0, \mathsf{CIScriptPCCurr}_{\mathcal{N}})], \\
&\ wit = [(\sigma_{PV}, \mathcal{N}, c_{\mathcal{N}}, pc_{\mathcal{N}}, c_{pc_{\mathcal{N}}}, pc_\theta, c_{pc_\theta})], \\
&\ out = [(d\text{Ƀ}; \langle \mathsf{ChallPCScript}, \mathsf{TL}(\Delta) \wedge \mathsf{CheckSig}_{\mathsf{pk}_V}\rangle)]\Big).
\end{aligned}
\tag{11}
$$

In the `ChallengeCurrPC` transaction, V commits again to $\mathcal{N}$, potentially equivocating. P can punish equivocation by unlocking $\mathsf{ChallPCScript}$ script (cf. Algorithm 13).

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

$$
\begin{aligned}
&\texttt{PunishCurrPC} := \\
&\Big(in = [(\texttt{ChallengeCurrPC}, 0, \mathsf{ChallPCScript})], \\
&\ wit = [(\sigma_{PV}, c_0, c_1)], \\
&\ out = [(d\text{Ƀ}; \mathsf{CheckSig}_{\mathsf{pk}_P})]\Big).
\end{aligned}
\tag{12}
$$

Algorithm 11 The script $\mathsf{CIScriptPCCurr_i}$, for $\mathsf{i} \in \{0, \ldots, 31\}$. 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 $\mathsf{i} \in \{1, \ldots, 31\}$, 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 $\mathsf{CommitComputationScript}$.

function $\mathsf{CIScriptPCCurr_i}(\sigma_{PV}, \mathcal{N}, c_{\mathcal{N}}, pc_i, c_{pc_i}, pc_\theta, c_{pc_\theta})$
  $\mathsf{CheckMSigVerify}_{\mathsf{pk}_{PV}}(\sigma_{PV})$;
  $\mathsf{CheckCommVerify}_{\mathsf{pk}_{\mathcal{N}}}(\mathcal{N}, c_{\mathcal{N}})$;
  if $\mathsf{CountZeroes}(\mathcal{N}) \neq \mathsf{i}$ **then**
    ▷ *Maps $\mathcal{N}$ to one of the 32 program counters $pc_{n_0}, \ldots, pc_{n_{31}}$.* ◁
    return False;
  $\mathsf{CheckCommVerify}_{\mathsf{pk}_{pc_i}}(pc_i, c_{pc_i})$;
  $\mathsf{CheckCommVerify}_{\mathsf{pk}_{pc_\theta}}(pc_\theta, c_{pc_\theta})$;
  if $pc_i \neq pc_\theta$ **then**
    return True;
  else
    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.

function $\mathsf{CountZeroes}(N)$
  $counter \leftarrow 0$;
  $flag \leftarrow \mathsf{False}$;
  for $i = 0, \ldots, |N|_{bit} - 1$ **do**
    if $N[|N|_{bit} - i] = 1$ **then**
      $flag \leftarrow \mathsf{True}$;
      ▷ *Set the flag, stop incrementing the counter.* ◁
    else
      if $flag = \mathsf{False}$ **then**
        $counter \leftarrow counter + 1$;
  return *counter*.

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

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

C.4.2 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 $\mathsf{i} \in \{0, \ldots, 31\}$. 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.

function $\mathsf{CIScriptPCNext_i}(\sigma_{PV}, \mathcal{N}', c_{\mathcal{N}'}, pc_i, c_{pc_i}, pc_{\theta'}, c_{pc_{\theta'}})$
  $\mathsf{CheckMSigVerify}_{\mathsf{pk}_{PV}}(\sigma_{PV})$;
  $\mathsf{CheckCommVerify}_{\mathsf{pk}_{\mathcal{N}}}(\mathcal{N}', c_{\mathcal{N}'})$;
  if $\mathsf{CountZeroes}(\mathcal{N}') \neq \mathsf{i}$ **then**
    ▷ *Maps $\mathcal{N}'$ to one of the 32 program counters $pc_{n_0}, \ldots, pc_{n_{31}}$.* ◁
    return False;
  $\mathsf{CheckCommVerify}_{\mathsf{pk}_{pc_i}}(pc_i, c_{pc_i})$;
  $\mathsf{CheckCommVerify}_{\mathsf{pk}_{pc_{\theta'}}}(pc_{\theta'}, c_{pc_{\theta'}})$;
  if $pc_i \neq pc_{\theta'}$ **then**
    return True;
  else
    return False.

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

In this challenge path, V can equivocate on $\mathcal{N}'$[21]. P can punish equivocation by publishing the `PunishNextPC` transaction (cf. Eq. (14)), which unlocks $\mathsf{ChallPCScript}$ by proving the equivocation. Upon doing so, P redeems all the funds locked in the multisignature.

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

C.4.3 Punish Wrong Instruction

P has committed to a current program counter pc_θ that does not correspond to the correct program instruction, specifically:

$$\Pi[pc_\theta] \neq (insType_\theta, addrA_\theta, addrB_\theta, addrC_\theta).$$

[21] 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.

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

Algorithm 15 The script $\mathsf{CIScriptInstr_j}$, for $\mathsf{j} \in \{1, ..., \ell\}$. In the script $\mathsf{CIScriptInstr_j}$, during the setup phase we hard-code the public keys pk_{pc_θ}, $\mathsf{pk}_{insType_\theta}$, $\mathsf{pk}_{addrA_\theta}$, $\mathsf{pk}_{addrB_\theta}$, $\mathsf{pk}_{addrC_\theta}$, 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 $\mathsf{CIScriptInstr_j}$.

function $\mathsf{CIScriptInstr_j}(\sigma_{PV}, pc_\theta, c_{pc_\theta}, insType_\theta, c_{insType_\theta}, addrA_\theta, c_{addrA_\theta}, addrB_\theta, c_{addrB_\theta}, addrC_\theta, c_{addrC_\theta})$
  $\mathsf{CheckMSigVerify}_{\mathsf{pk}_{PV}}(\sigma_{PV})$;
  $\mathsf{CheckCommVerify}_{\mathsf{pk}_{pc_\theta}}(pc_\theta, c_{pc_\theta})$;
  $\mathsf{CheckCommVerify}_{\mathsf{pk}_{insType_\theta}}(insType_\theta, c_{insType_\theta})$;
  $\mathsf{CheckCommVerify}_{\mathsf{pk}_{addrA_\theta}}(addrA_\theta, c_{addrA_\theta})$;
  $\mathsf{CheckCommVerify}_{\mathsf{pk}_{addrB_\theta}}(addrB_\theta, c_{addrB_\theta})$;
  $\mathsf{CheckCommVerify}_{\mathsf{pk}_{addrC_\theta}}(addrC_\theta, c_{addrC_\theta})$;
  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**
    return True;
  else
    return False.

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

C.4.4 Challenge Read

V starts the challenge by publishing the `ChallengeRead` transaction (cf. Eq. (16)), spending the `CommitInstruction` output[22].

$$
\begin{aligned}
&\texttt{ChallengeRead} := \\
&\Big(in = [(\texttt{CommitInstruction}, 0, \mathsf{CheckMSig}_{\mathsf{pk}_{PV}})], \\
&\ wit = [(\sigma_{PV})], \\
&\ out = [(d\!\!\!\!\Bbbk; \langle \mathsf{ReadChallScript_1}, \mathsf{TL}(\Delta) \wedge \mathsf{CheckSig}_{\mathsf{pk}_V} \rangle)]\Big).
\end{aligned}
\tag{16}
$$

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

$$\mathsf{ReadChallScript_j} := \mathsf{CheckMSig}_{\mathsf{pk}_{PV}} \wedge \mathsf{CheckComm}_{\mathsf{pk}_{\mathsf{Node}_{\mathsf{d}_{5-\mathsf{j}}}}}.$$

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

The parties engage in the read bisection game (cf. Appendix D.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 $\texttt{ReadResponse}_1$ transaction (cf. Eq. (17)), committing to $\mathsf{Node}_{\mathsf{d}_4} := \mathcal{P}_R[\mathsf{d}_4]$ in the witness, where $\mathsf{d}_4 = 1 \cdot 2^4$.

$$\begin{aligned}
&\texttt{ReadResponse}_1 := \\
&\Big(in = [(\texttt{ChallengeRead}, 0, \mathsf{ReadChallScript}_1)], \\
&\ wit = [(\sigma_{PV}, \mathsf{Node}_{\mathsf{d}_4}, c_{\mathsf{Node}_{\mathsf{d}_4}})], \\
&\ out = [(d\ddot{\mathbb{B}}; \langle \mathsf{ReadRespScript}_1, \mathsf{TL}(\Delta) \wedge \mathsf{CheckSig}_{\mathsf{pk}_P} \rangle)]\Big).
\end{aligned} \tag{17}$$

$\mathsf{ReadRespScript_i}$ with $i \in \{1, \ldots, 5\}$ is defined as:

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

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

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

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

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

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

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

Then, V publishes the $\texttt{ReadChallenge}_5$ 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 $\mathcal{N}_{Mer} = \sum_{k=0}^{4} b'_k \cdot 2^k$.

$$
\begin{aligned}
&\texttt{ReadChallenge}_5 := \\
&\Big(in = [(\texttt{ReadResponse}_5, 0, \mathsf{ReadRespScript}_5)], \\
&\ wit = [(\sigma_{PV}, b'_0, c_{b'_0})], \\
&\ out = [(d\ddot{\mathbb{B}}; \langle \mathsf{HashReadScript}_1, \ldots, \mathsf{HashReadScript}_{20}, \\
&\ \mathsf{RootReadScript}_1, \ldots, \mathsf{RootReadScript}_{32}, \mathsf{ValueAScript}, \\
&\ \mathsf{TL}(\Delta) \wedge \mathsf{CheckSig}_{\mathsf{pk}_V} \rangle)]\Big).
\end{aligned} \tag{21}
$$

The integer $\mathcal{N}_{Mer}$, chosen by V, conditions which Tapleaves P can unlock to spend the $\texttt{ReadChallenge}_5$ 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 $\texttt{ReadChallenge}_5$ output. To do so, P provides a witness that unlocks one of the scripts $\mathsf{HashReadScript}_1$, ..., $\mathsf{HashReadScript}_{20}$. Each script hard-codes the public key of a pair of nodes belonging to $\{\mathsf{Node}_{\mathsf{d}_0}, \ldots, \mathsf{Node}_{\mathsf{d}_4}\}$, the first being the parent node in $MerkleTree_{MR_{\mathcal{N}}}$ and the second being the child node[23]. 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 $\mathsf{HashReadScript}_1$ in Algorithm 16. The scripts $\mathsf{HashReadScript}_2$, ..., $\mathsf{HashReadScript}_{20}$ 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.

$$
\begin{aligned}
&\texttt{CommitRead1} := \\
&\Big(in = [(\texttt{ReadChallenge}_5, 0, \mathsf{HashReadScript}_\mathsf{i})], \\
&\ wit = [(\sigma_{PV}, \mathcal{N}_{Mer}, c_{\mathcal{N}_{Mer}}, v_{pos}, c_{v_{pos}}, c_{addrA_\theta}, \mathsf{Nsib}, \\
&\ \mathsf{Npar}, c_{\mathsf{Npar}}, \mathsf{Nchild}, c_{\mathsf{Nchild}})], \\
&\ out = [(d\ddot{\mathbb{B}}; \langle \mathsf{CommitRead1Script}, \mathsf{TL}(\Delta) \wedge \mathsf{CheckSig}_{\mathsf{pk}_P} \rangle)]\Big).
\end{aligned} \tag{22}
$$

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

$$
\begin{aligned}
&\texttt{PunishRead1} := \\
&\Big(in = [(\texttt{CommitRead1}, 0, \mathsf{CommitRead1Script})], \\
&\ wit = [(\sigma_{PV}, c_0, c_1)], out = [(d\ddot{\mathbb{B}}; \mathsf{CheckSig}_{\mathsf{pk}_V}]\Big).
\end{aligned} \tag{23}
$$

(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, $\mathsf{Node}_{\mathsf{d}_0}$, and $valA_\theta$; To spend the $\texttt{ReadChallenge}_5$ output, P unlocks $\mathsf{ValueAScript}$. $\mathsf{ValueAScript}$ is analogous to $\mathsf{HashReadScript}_\mathsf{i}$ with the following differences: (i) $\mathsf{CountZeroes}(\mathcal{N}_{Mer}) = 0$; (ii) the parent node is $\mathsf{Node}_{\mathsf{d}_0}$; (iii) the child node is not one of the nodes $\mathsf{Node}_{\mathsf{d}_4}$, ..., $\mathsf{Node}_{\mathsf{d}_0}$, but $valA_\theta$ instead.

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

Algorithm 16 The script $\mathsf{HashReadScript}_1$. The bit $v_{pos} \in \{0,1\}$ represents the position of the child node ($v_{pos} = 0$ means that $\mathsf{Node}_{\mathsf{d}_0}$ is the left child of $\mathsf{Node}_{\mathsf{d}_4}$, $v_{pos} = 1$ means the opposite. Nsib is the sibling node of $\mathsf{Node}_{\mathsf{d}_0}$ that P presents. In the setup phase, the public keys $\mathsf{pk}_{\mathcal{N}_{Mer}}$, $\mathsf{pk}_{addrA_\theta}$, $\mathsf{pk}_{\mathsf{Node}_{\mathsf{d}_4}}$, $\mathsf{pk}_{\mathsf{Node}_{\mathsf{d}_0}}$, are hard-coded in the script.

function $\mathsf{HashReadScript}_1(\sigma_{PV}, \mathcal{N}_{Mer}, c_{\mathcal{N}_{Mer}}, v_{pos}, c_{v_{pos}}, c_{addrA_\theta}, \mathsf{Nsib}, \mathsf{Node}_{\mathsf{d}_4}, c_{\mathsf{Node}_{\mathsf{d}_4}}, \mathsf{Node}_{\mathsf{d}_0}, c_{\mathsf{Node}_{\mathsf{d}_0}})$
  $\mathsf{CheckMSigVerify}_{\mathsf{pk}_{PV}}(\sigma_{PV})$;
  $\mathsf{CheckCommVerify}_{\mathsf{pk}_{\mathcal{N}_{Mer}}}(\mathcal{N}_{Mer}, c_{\mathcal{N}_{Mer}})$;
  ▷ *Since V committed to $\mathcal{N}_{Mer}$, P does not know $\mathsf{sk}_{\mathcal{N}_{Mer}}$. Therefore, to satisfy this guard, has to provide the commitment that V made* ◁
  if $\mathsf{CountZeroes}(\mathcal{N}_{Mer}) \neq 5 - 1$ **then**
    ▷ *Since $\mathsf{Node}_{\mathsf{d}_4}$ is the parent node here, $\mathsf{CountZeroes}(\mathcal{N}_{Mer})$ should be 4.* ◁
    return False;
  $\mathsf{CheckCommVerify}_{\mathsf{pk}_{addrA_\theta[\mathcal{N}_{Mer}]}}(v_{pos}, c_{addrA_\theta[\mathcal{N}_{Mer}]})$;
  ▷ *The whole public key $\mathsf{pk}_{addrA_\theta}$ is hard-coded in the script, but only the the $\mathcal{N}_{Mer}$-th entry is used* ◁
  $\mathsf{CheckCommVerify}_{\mathsf{pk}_{\mathsf{Node}_{\mathsf{d}_4}}}(\mathsf{Node}_{\mathsf{d}_4}, c_{\mathsf{Node}_{\mathsf{d}_4}})$; ▷ *Parent node*
  $\mathsf{CheckCommVerify}_{\mathsf{pk}_{\mathsf{Node}_{\mathsf{d}_0}}}(\mathsf{Node}_{\mathsf{d}_0}, c_{\mathsf{Node}_{\mathsf{d}_0}})$; ▷ *Child node*
  if $v_{pos} = 0$ **then**
    if $H(\mathsf{Node}_{\mathsf{d}_0}||\mathsf{Nsib}) = \mathsf{Node}_{\mathsf{d}_4}$ **then**
      return True;
    else
      return False;
  else
    if $H(\mathsf{Nsib}||\mathsf{Node}_{\mathsf{d}_0}) = \mathsf{Node}_{\mathsf{d}_4}$ **then**
      return True;
    else
      return False.

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

function $\mathsf{CommitRead1Script}(\sigma_{PV}, c_0, c_1)$
  $\mathsf{CheckMSigVerify}_{\mathsf{pk}_{PV}}(\sigma_{PV})$;
  for $i = 1, \ldots, |\mathsf{Npar}|_{bit}$ **do**
    if $\mathsf{Equivocation}(\mathsf{pk}_{\mathsf{Npar}[i]}, c_0, c_1) = \mathsf{True} \vee \mathsf{Equivocation}(\mathsf{pk}_{\mathsf{Nchild}[i]}, c_0, c_1) = \mathsf{True} \vee \mathsf{Equivocation}(\mathsf{pk}_{addrA_\theta[i]}, c_0, c_1) = \mathsf{True}$ **then**
      return True;
  return False.

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, $\mathsf{Node}_{\mathsf{d}_0}$, and $MR_{\mathcal{N}}$. P unlocks one of the leaves $\mathsf{RootReadScript}_1$, ..., $\mathsf{RootReadScript}_{32}$, according to which number $\mathcal{N}$ V committed at the end of the dispute bisection game. We provide $\mathsf{RootReadScript}_\mathsf{i}$ in Algorithm 18.

Algorithm 18 The script $\mathsf{RootReadScript}_\mathsf{i}$. In the setup phase, the public keys $\mathsf{pk}_{\mathcal{N}_{Mer}}$, $\mathsf{pk}_{\mathcal{N}}$, $\mathsf{pk}_{\mathsf{Node}_{\mathsf{d}_0}}$, pk_{MR_i} are hard-coded in the script.

function $\mathsf{RootReadScript}_\mathsf{i}(\sigma_{PV}, \mathcal{N}_{Mer}, c_{\mathcal{N}_{Mer}}, v_{pos}, c_{v_{pos}}, c_{addrA_\theta}, \mathsf{Nsib}, \mathcal{N}, c_{\mathcal{N}}, \mathsf{Node}_{\mathsf{d}_0}, c_{\mathsf{Node}_{\mathsf{d}_0}}, MR_i, c_{MR_i})$
  $\mathsf{CheckMSigVerify}_{\mathsf{pk}_{PV}}(\sigma_{PV})$;
  $\mathsf{CheckCommVerify}_{\mathsf{pk}_{\mathcal{N}_{Mer}}}(\mathcal{N}_{Mer}, c_{\mathcal{N}_{Mer}})$;
  ▷ *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* ◁
  $\mathsf{CheckCommVerify}_{\mathsf{pk}_{\mathcal{N}}}(\mathcal{N}, c_{\mathcal{N}})$;
  ▷ *P has to provide the commitment that V made in the dispute phase* ◁
  if $\mathsf{CountZeroes}(\mathcal{N}) \neq \mathsf{i}$ **then**
    return False;
  if $\mathsf{CountZeroes}(\mathcal{N}_{Mer}) \neq 5$ **then** ▷ *$\mathcal{N}_{Mer}$ must be equal to 0*
    return False;
  $\mathsf{CheckCommVerify}_{\mathsf{pk}_{addrA_\theta[\mathcal{N}_{Mer}]}}(v_{pos}, c_{addrA_\theta[\mathcal{N}_{Mer}]})$;
  $\mathsf{CheckCommVerify}_{\mathsf{pk}_{\mathsf{Node}_{\mathsf{d}_0}}}(\mathsf{Node}_{\mathsf{d}_0}, c_{\mathsf{Node}_{\mathsf{d}_0}})$;
  ▷ *for any $\mathsf{RootReadScript}_\mathsf{i}$, $\mathsf{Node}_{\mathsf{d}_0}$ is always the child node* ◁
  $\mathsf{CheckCommVerify}_{\mathsf{pk}_{MR_i}}(\mathsf{Node}_{\mathsf{d}_0}, c_{\mathsf{Node}_{\mathsf{d}_0}})$;
  if $v_{pos} = 0$ **then**
    if $H(\mathsf{Node}_{\mathsf{d}_0}||\mathsf{Nsib}) = MR_i$ **then**
      return True;
    else
      return False;
  else
    if $H(\mathsf{Nsib}||\mathsf{Node}_{\mathsf{d}_0}) = MR_i$ **then**
      return True;
    else
      return False.

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

$$\begin{aligned}&\texttt{CommitRead3} :=\\&\Big(in = [(\mathsf{ReadChallenge}_5, 0, \mathsf{ReadRootScript}_\mathsf{i})],\\&\ wit = [(\sigma_{PV}, \mathcal{N}_{Mer}, c_{\mathcal{N}_{Mer}}, \mathcal{N}, c_{\mathcal{N}}, \mathsf{Node}_{\mathsf{d}_0}, c_{\mathsf{Node}_{\mathsf{d}_0}})],\\&\ out = [(d\ddot{\mathbb{B}}; \langle \mathsf{CommitRead3Script}, \mathsf{TL}(\Delta) \wedge \mathsf{CheckSig}_{\mathsf{pk}_P}\rangle)]\Big).\end{aligned} \tag{24}$$

V can punish P if they equivocate on $\mathsf{Node}_{\mathsf{d}_0}$, MR_i or $addrA_\theta$ by publishing `PunishRead3` (cf. Eq. (25)), which unlocks CommitRead3Script, analogous to CommitRead1Script but with pk_{MR_i}, $\mathsf{pk}_{\mathsf{Node}_{\mathsf{d}_0}}$ instead of $\mathsf{pk}_{\mathsf{Npar}}$, $\mathsf{pk}_{\mathsf{Nchild}}$.

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

C.4.5 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[24]. As a result, the memory root $MR_{\mathcal{N}'}$ is incorrect.

The parties engage in the write bisection game (cf. Appendix D.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 $\texttt{WriteResponse}_\texttt{i}$ transaction is analogous to $\texttt{ReadResponse}_\texttt{i}$ 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 $\texttt{WriteChallenge}_\texttt{j}$ transaction (analogously to $\texttt{ReadChallenge}_\texttt{j}$), 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 $\{(\mathsf{Node}_{\mathsf{d}_4}, \mathsf{Node}'_{\mathsf{d}_4}), \ldots, (\mathsf{Node}_{\mathsf{d}_0}, \mathsf{Node}'_{\mathsf{d}_0})\}$, where $\mathsf{Node}_{\mathsf{d}_4}, \ldots, \mathsf{Node}_{\mathsf{d}_0} \in \mathcal{P}_W$ and $\mathsf{Node}'_{\mathsf{d}_4}, \ldots, \mathsf{Node}'_{\mathsf{d}_0} \in \mathcal{P}'_W$. 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 $\texttt{WriteChallenge}_5$. 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 $\texttt{WriteChallenge}_5$.

(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 $\texttt{CommitWrite1}$ transaction (cf. Eq. (26)).

[24] We assume P commits correctly to $valC_\theta$ in the witness of the $\texttt{CommitInstruction}$ 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 $\mathsf{HashWriteScript}_1$. The bit $v_{pos} \in \{0,1\}$ represents the position of the child nodes ($v_{pos} = 0$ means that $\mathsf{Node}_{\mathsf{d}_0}$ and $\mathsf{Node}'_{\mathsf{d}_0}$ are the left childs of $\mathsf{Node}_{\mathsf{d}_4}$ and $\mathsf{Node}'_{\mathsf{d}_4}$, respectively). Nsib is the sibling node of $\mathsf{Node}_{\mathsf{d}_0}$ in $\mathcal{P}_W$ and of $\mathsf{Node}'_{\mathsf{d}_0}$ in $\mathcal{P}'_W$. In the setup phase, the public keys $\mathsf{pk}_{\mathcal{N}_{Mer}}$, $\mathsf{pk}_{addrC_\theta}$, $\mathsf{pk}_{\mathsf{Node}_{\mathsf{d}_4}}$, $\mathsf{pk}_{\mathsf{Node}_{\mathsf{d}_0}}$, $\mathsf{pk}_{\mathsf{Node}'_{\mathsf{d}_4}}$, $\mathsf{pk}_{\mathsf{Node}'_{\mathsf{d}_0}}$ are hard-coded in the script.

1: **function** $\mathsf{HashWriteScript}_1(\sigma_{PV}, \mathcal{N}_{Mer}, c_{\mathcal{N}_{Mer}}, v_{pos}, c_{v_{pos}}, c_{addrC_\theta}, \mathsf{Nsib}, \mathsf{Node}_{\mathsf{d}_4}, c_{\mathsf{Node}_{\mathsf{d}_4}}, \mathsf{Node}_{\mathsf{d}_0}, c_{\mathsf{Node}_{\mathsf{d}_4}}, \mathsf{Node}'_{\mathsf{d}_4}, c_{\mathsf{Node}'_{\mathsf{d}_4}}, \mathsf{Node}'_{\mathsf{d}_0}, c_{\mathsf{Node}'_{\mathsf{d}_4}})$
2: $\quad \mathsf{CheckMSigVerify}_{\mathsf{pk}_{PV}}(\sigma_{PV})$;
3: $\quad \mathsf{CheckCommVerify}_{\mathsf{pk}_{\mathcal{N}_{Mer}}}(\mathcal{N}_{Mer}, c_{\mathcal{N}_{Mer}})$;
4: $\quad \triangleright$ *Since V committed to $\mathcal{N}_{Mer}$, P does not know $\mathsf{sk}_{\mathcal{N}_{Mer}}$. Therefore, to satisfy this guard, has to provide the commitment that V made* $\triangleleft$
5: $\quad$ **if** $\mathsf{CountZeroes}(\mathcal{N}_{Mer}) \neq 5 - 1$ **then**
6: $\quad\quad \triangleright$ *Since $\mathsf{Node}_{\mathsf{d}_4}$, $\mathsf{Node}'_{\mathsf{d}_4}$ are the parent nodes here, $\mathsf{CountZeroes}(\mathcal{N}_{Mer})$ should be 4.* $\triangleleft$
7: $\quad\quad$ **return** False;
8: $\quad \mathsf{CheckCommVerify}_{\mathsf{pk}_{addrC_\theta[\mathcal{N}_{Mer}]}}(v_{pos}, c_{addrC_\theta[\mathcal{N}_{Mer}]})$;
9: $\quad \triangleright$ *The whole public key $\mathsf{pk}_{addrC_\theta}$ is hardcoded in the script, but only the the $\mathcal{N}_{Mer}$-th entry is used* $\triangleleft$
10: $\quad \mathsf{CheckCommVerify}_{\mathsf{pk}_{\mathsf{Node}_{\mathsf{d}_4}}}(\mathsf{Node}_{\mathsf{d}_4}, c_{\mathsf{Node}_{\mathsf{d}_4}})$; $\triangleright$ *Parent node in $\mathcal{P}_W$*
11: $\quad \mathsf{CheckCommVerify}_{\mathsf{pk}_{\mathsf{Node}_{\mathsf{d}_0}}}(\mathsf{Node}_{\mathsf{d}_0}, c_{\mathsf{Node}_{\mathsf{d}_0}})$; $\triangleright$ *Child node in $\mathcal{P}_W$*
12: $\quad \mathsf{CheckCommVerify}_{\mathsf{pk}_{\mathsf{Node}'_{\mathsf{d}_4}}}(\mathsf{Node}'_{\mathsf{d}_4}, c_{\mathsf{Node}'_{\mathsf{d}_4}})$; $\triangleright$ *Parent node in $\mathcal{P}'_W$*
13: $\quad \mathsf{CheckCommVerify}_{\mathsf{pk}_{\mathsf{Node}'_{\mathsf{d}_0}}}(\mathsf{Node}'_{\mathsf{d}_0}, c_{\mathsf{Node}'_{\mathsf{d}_0}})$; $\triangleright$ *Child node in $\mathcal{P}'_W$*
14: $\quad$ **if** $v_{pos} = 0$ **then**
15: $\quad\quad$ **if** $H(\mathsf{Node}_{\mathsf{d}_0}||\mathsf{Nsib}) = \mathsf{Node}_{\mathsf{d}_4} \wedge H(\mathsf{Node}'_{\mathsf{d}_0}||\mathsf{Nsib}) = \mathsf{Node}'_{\mathsf{d}_4}$ **then**
16: $\quad\quad\quad$ **return** True
17: $\quad\quad$ **else**
18: $\quad\quad\quad$ **return** False
19: $\quad$ **else**
20: $\quad\quad$ **if** $H(\mathsf{Nsib}||\mathsf{Node}_{\mathsf{d}_0}) = \mathsf{Node}_{\mathsf{d}_4} \wedge H(\mathsf{Nsib}||\mathsf{Node}'_{\mathsf{d}_0}) = \mathsf{Node}'_{\mathsf{d}_4}$ **then**
21: $\quad\quad\quad$ **return** True
22: $\quad\quad$ **else**
23: $\quad\quad\quad$ **return** False

$$
\begin{aligned}
&\texttt{CommitWrite1} := \\
&\Big(in = [(\texttt{WriteChallenge}_5, 0, \mathsf{HashWriteScript_i})], \\
&\ wit = [(\sigma_{PV}, \mathcal{N}_{Mer}, c_{\mathcal{N}_{Mer}}, v_{pos}, c_{v_{pos}}, c_{addrC_\theta}, \mathsf{Nsib}, \\
&\ \mathsf{Npar}, c_{\mathsf{Npar}}, \mathsf{Nchild}, c_{\mathsf{Nchild}}, \mathsf{Npar}', c_{\mathsf{Npar}'}, \mathsf{Nchild}', c_{\mathsf{Nchild}'})], \\
&\ out = [(d\ddot{\mathsf{B}}; \langle \mathsf{CommitWrite1Script}, \mathsf{TL}(\Delta) \wedge \mathsf{CheckSig}_{\mathsf{pk}_P} \rangle)]\Big);
\end{aligned}
\tag{26}
$$

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 $\mathsf{Npar}, \mathsf{Nchild}, \mathsf{Npar}', \mathsf{Nchild}'$, V can claim all the coins locked in the multisignature.

(B) Commit Value C. $\mathcal{N}_{Mer} = 31$, the point of disagreement is between $\mathsf{Node}_{\mathsf{d}_0}$ and $valC_\theta$ or between $\mathsf{Node}'_{\mathsf{d}_0}$ and $valC_\theta$. This case is analogous to the "commit value A" case of the *Challenge Read* branch. For ValueCScript, the difference with $\mathsf{HashWriteScript_i}$, is that: (i) $\mathsf{CountZero} = 0$; (ii) the parent nodes are $\mathsf{Node}_{\mathsf{d}_0}$ and $\mathsf{Node}'_{\mathsf{d}_0}$, 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 $\mathsf{Node}_{\mathsf{d}_0}$ or between $MR_{\mathcal{N}'}$ and $\mathsf{Node}'_{\mathsf{d}_0}$. This case is analogous to the "commit read root" case of the *Challenge Read* branch. The script $\mathsf{RootWriteScript_i}$, with $\mathsf{i} \in \{0, \ldots, 31\}$ is the same as $\mathsf{RootReadScript_i}$ but takes as additional inputs $\mathsf{Node}'_{\mathsf{d}_0}$, 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 $\mathsf{RootWriteScript_i}$, 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 $\mathsf{Node}_{\mathsf{d}_0}$, $\mathsf{Node}'_{\mathsf{d}_0}$, MR_i, MR_{i+1}, $addrC_\theta$.

Algorithm 20 The script $\mathsf{RootWriteScript_i}$. In the setup phase, the public keys $\mathsf{pk}_{\mathcal{N}_{Mer}}$, $\mathsf{pk}_{\mathcal{N}}$, $\mathsf{pk}_{\mathsf{Node}_{\mathsf{d}_0}}$, pk_{MR_i} are hard-coded in the script.

```
1: function RootWriteScript_i
2:     ▷ ...                                                              ◁
3:     if v_pos = 0 then
4:         if H(Node_d0||Nsib) = MR_i ∧ H(Node'_d0||Nsib) = MR_{i+1} then
5:             return True;
6:         else
7:             return False;
8:     else
9:         if H(Nsib||Node_d0) = MR_i ∧ H(Nsib||Node'_d0 = MR_{i+1}) then
10:            return True;
11:        else
12:            return False.
```

D 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 $A^P[i] = A^V[i] \wedge A^P[i+1] \neq A^V[i+1]$ or $A^P[i] \neq A^V[i] \wedge A^P[i+1] = A^V[i+1]$ (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.

D.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).

D.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 $M_{\mathcal{N}}[addrA_\theta]$ 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 $\mathcal{P}_R^P := (MR_{\mathcal{N}}^P, \ldots, M_{\mathcal{N}}^P[addrA_\theta])$. The algorithm outputs the index of a point of disagreement. V runs $\mathsf{DisagreeV}(\mathcal{P}_R^V, |\mathcal{P}_R^V|)$ (cf. Algorithm 21, lines 15 to 32). The algorithm outputs the index of a point of disagreement.

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 Npar^P, 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 $\mathsf{CommitRead}^P$.

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.

function DisagreeP(I_P, n)
  $l \leftarrow 1$; ▷ *Left search boundary*
  $r \leftarrow n$; ▷ *Right search boundary*
  $i \leftarrow 1$; ▷ *Counter*
  while $l + 1 < r$ **do**
    $m \leftarrow \lfloor \frac{l+r}{2} \rfloor$;
    Publish $\mathsf{Tx}_i^P[\{I_P[m]\}]$ on L;
    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 $\mathsf{Tx}_i^V[\{b_i\}]$;
    if $b_i = 0$ **then**
      $r \leftarrow m$;
    else
      $l \leftarrow m$;
    $i \leftarrow i + 1$;
  return $r - 1$.

function DisagreeV(I_V, n)
  $l \leftarrow 1$; ▷ *Left search boundary*
  $r \leftarrow n$; ▷ *Right search boundary*
  $i \leftarrow 1$; ▷ *Counter*
  while $l + 1 < r$ **do**
    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 $\mathsf{Tx}_i^P[\{I_P[m]\}]$;
    $m \leftarrow \lfloor \frac{l+r}{2} \rfloor$;
    if $I_P[m] \neq I_V[m]$ **then** ▷ *Disagreement*
      $b_i \leftarrow 0$;
    else
      $b_i \leftarrow 1$;
    Publish $\mathsf{Tx}_i^V[\{b_i\}]$ on L;
    if $b_i = 0$ **then**
      $r \leftarrow m$; ▷ *Challenge the left half of at the next step*
    else
      $l \leftarrow m$; ▷ *Challenge the right half of at the next step*
    $i \leftarrow i + 1$;
  return $r - 1$.

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 $M^P_{\mathcal{N}}[addrA_\theta]$ is really the $addrA_\theta$-th leaf of the Merkle tree with root $MR^P_{\mathcal{N}}$.

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

8: **function** SolveReadV(.)
9: Wait until $\mathsf{CommitRead}^P[\{\mathsf{Npar}, \mathsf{Nchild}, v_{pos}\}, \mathsf{Nsib}]$ appears in L, where $\mathsf{Npar}, \mathsf{Nchild}, v_{pos}, \mathsf{Nsib}$ is part of the witness of transaction $\mathsf{CommitRead}^P$ 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 $\mathsf{PunishRead}^V[c_0, c_1]$ on L.

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

D.3 Write bisection game

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

P runs $\mathsf{DisagreeWriteP}(\mathcal{P}^P_W, {\mathcal{P}'_W}^P, |\mathcal{P}^P_W|)$, which returns the index of a point of disagreement. V runs $\mathsf{DisagreeWriteV}(\mathcal{P}^V_W, {\mathcal{P}'_W}^V, |\mathcal{P}^V_W|)$, 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 $(\mathsf{Npar}, \mathsf{Nchild})$ or the pair $(\mathsf{Npar}', \mathsf{Nchild}')$. P runs $\mathsf{SolveWriteP}(\mathsf{Npar}^P, \mathsf{Nchild}^P, {\mathsf{Npar}'}^P, {\mathsf{Nchild}'}^P, \mathsf{Nsib}, v_{pos})$ (cf. Algorithm 24, Lines 1 to 7). In the algorithm, P asserts that Nchild^P and ${\mathsf{Nchild}'}^P$ are the left or right child of the nodes Npar^P and ${\mathsf{Npar}'}^P$, 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 ${\mathsf{Nchild}'}^P$. P publishes the transaction $\mathsf{CommitWrite}^P$, where they provide a commitment for $\mathsf{Npar}^P, \mathsf{Nchild}^P, {\mathsf{Npar}'}^P, {\mathsf{Nchild}'}^P, v^P_{pos}$. Intuitively, the prover can commit to all

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.

function DisagreeWriteP(I_P, I'_P, n)
  $l \leftarrow 1$; ▷ *Left search boundary*
  $r \leftarrow n$; ▷ *Right search boundary*
  $i \leftarrow 1$; ▷ *Counter*
  while $l + 1 < r$ **do**
    $m \leftarrow \lfloor \frac{l+r}{2} \rfloor$;
    Publish $\mathsf{Tx}_i^P[\{I_P[m], I'_P[m]\}]$ on L;
    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 $\mathsf{Tx}_i^V[\{b_i\}]$;
    if $b_i = 0$ **then**
      $r \leftarrow m$;
    else
      $l \leftarrow m$;
    $i \leftarrow i + 1$;
  return $r - 1$.

function DisagreeWriteV(I_V, I'_V, n)
  $l \leftarrow 1$; ▷ *Left search boundary*
  $r \leftarrow n$; ▷ *Right search boundary*
  $i \leftarrow 1$; ▷ *Counter*
  $flag \leftarrow \mathsf{False}$;
  while $l + 1 < r$ **do**
    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 $\mathsf{Tx}_i^P[\{I_P[m], I'_P[m]\}]$;
    $m \leftarrow \lfloor \frac{l+r}{2} \rfloor$;
    if $flag = \mathsf{False}$ **then**
      if $I_P[m] \neq I_V[m]$ **then**
        $flag = \mathsf{True}$; ▷ *From now on, look only for disagreement on I_V*
        $b_i \leftarrow 0$;
      else
        if $I'_P[m] = I'_V[m]$ **then**
          $b_i \leftarrow 0$;
        else
          $b_i \leftarrow 1$;
    else
      if $I_P[m] \neq I_V[m]$ **then**
        $b_i \leftarrow 0$;
      else
        $b_i \leftarrow 1$;
    Publish $\mathsf{Tx}_i^V[\{b_i\}]$ on L;
    if $b_i = 0$ **then**
      $r \leftarrow m$; ▷ *Challenge the left half of at the next step*
    else
      $l \leftarrow m$; ▷ *Challenge the right half of at the next step*
    $i \leftarrow i + 1$;
  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 $\mathsf{Response}_i$ transaction within Δ rounds after $\mathsf{Challenge}_i$ 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 $\mathsf{Challenge}_i$ 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 $\mathsf{Challenge}_i$ transaction within Δ rounds after $\mathsf{Response}_i$ 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).

the aforementioned elements without equivocating only if they previously committed to $MR^P_{\mathcal{N}'}$ and $M^P_{\mathcal{N}'}[addrC_\theta]$ such that $M^P_{\mathcal{N}'}[addrC_\theta]$ is really the $addrC_\theta$-th leaf of the Merkle tree with root $MR^P_{\mathcal{N}'}$.

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

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.

E 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 8 (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 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 $A(h) = \{a : (h, a) \in H\}$.*

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}.

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

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

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 9 (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 10) [34]. 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 10 (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 $A_{|h}(h')$.*

F 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 until V claims the remaining funds, V will not lose money even if $\beta = 0$. That is because, as we will show in Lemma F.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 $v_P + v_V \leq \mathsf{in}_P + \mathsf{in}_V - (2\tilde{N_1} + 2\tilde{\tilde{N}_2} + 7)f$. We interpret that cost as application fees, which is again analogous to the longest path of the execution.

F.1 Agreement phase

Lemma F.1 (Correctness of the setup) *Let $Presigned_P$ be the set of presigned transactions V handovers to P and $Presigned_V$ be the set of presigned transactions P handovers to V during the Setup phase. If* `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 `Setup` is accepted by the miners, both P and V must have signed `Setup`. Given that, the following holds:

1. P signed `Setup` only after receiving the set of transactions in $Presigned_P$ signed by V. Similarly, V signed `Setup` only after receiving the transactions in $Presigned_V$ signed by P.
2. `Setup` has an output of $\alpha + \beta + (4\tilde{N_1} + 4\tilde{\tilde{N}_2} + 14)$ to the multisig σ_{PV} of P and V.

□

F.2 Execution phase

Lemma F.2 (P does not post `CommitComputation`) *If* `Setup` *is published on chain and* `CommitComputation` *is not published on chain within the timelock Δ, it is a dominant strategy for V to claim the output of* `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 `Setup` by publishing `CommitComputation` on chain. If P does not publish `CommitComputation` after Δ when the timelock in `Setup` expires, V can claim the output of `Setup`. If V claims the collateral, she spends f coins in transaction fees. Moreover, since P has already posted `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, the respective 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 `Setup` when the timelock expires. □

F.3 Identify Disagreement phase

F.3.1 Normal closing

Lemma F.3 (V does not publish `KickOff` to initiate a dispute. P is supposed publish a `Close` transactio *Assume that* `CommitComputation` *is published on-chain and* `KickOff` *is not published on-chain within the timelock Δ. Moreover, consider $f(R_{final}) = (f_P, f_V)$ where R_{final} the final state that uniquely corresponds to MR_{final} which P has committed in* `CommitComputation`*, f is the outcome mapping function and* $\texttt{Close}_\texttt{i}$ *for some $i \in \{1, ..., m\}$ is the corresponding transaction that distributes the funds accordingly. Then, the following statements hold:*

- *If P publishes on-chain* $\texttt{Close}_\texttt{i}$*, P's balance account will be $\langle f_P + (2\tilde{N}_1 + 2\tilde{\tilde{N}}_2 + 5.5)f \rangle_P$, and V's balance account will be $\langle f_V + (2\tilde{N}_1 + 2\tilde{\tilde{N}}_2 + 5.5)f \rangle_V$.*
- *If P publishes on-chain* $\texttt{Close}_\texttt{j}$*, 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 $\langle \alpha + \beta + (4\tilde{N}_1 + 4\tilde{\tilde{N}}_2 + 11)f \rangle_P$ and V's $\langle 0 \rangle_v$.*
- *If none of transactions in the set $\mathcal{S} = \cup_{\{1,...,m\}} \texttt{Close}_\texttt{i}$ is published on-chain within 2Δ since* `CommitInstruction` *was published, it is a dominant strategy for V to claim the coins in the multisignature. In that scenario, V's balance account is $\langle \alpha + \beta + (4\tilde{N}_1 + 4\tilde{\tilde{N}}_2 + 11)f \rangle_V$ 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.

- $\texttt{Close}_\texttt{i}$ redistributes the rest of the coins to the parties according to the outcomes mapping function f, namely $(f_P + (2\tilde{N}_1 + 2\tilde{\tilde{N}}_2 + 5.5)f)$ coins to P and $(f_P + (2\tilde{N}_1 + 2\tilde{\tilde{N}}_2 + 5.5)f)$ 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 $\texttt{Close}_\texttt{i}$.
- Since $i \neq j$ and each transaction in $\mathcal{S}$ uniquely corresponds to an outcome of the computation, in $\texttt{Close}_\texttt{j}$ P commits to an $MR'_{\mathsf{final}} \neq MR_{\mathsf{final}}$. V can show the equivocation of P by providing the conflicting commitments since for each $k \in \{1, ..., m\}$ the Script $\mathsf{CloseScript}_\mathsf{i}$ (Algorithm 8) has the same hard-code keys with `CommitComputationScript`. As a result P will have a balance $u_1 = \langle \alpha + \beta + (4\tilde{N}_1 + 4\tilde{\tilde{N}}_2 + 10)f \rangle_V$, since $4f$ are spent in the transaction fees for `Setup`, `CommitComputation`, $\texttt{Close}_\texttt{j}$ 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 $u_1 = \langle \alpha+\beta+(4\tilde{N_1}+4\tilde{\tilde{N}_2}+11)f \rangle_V$, 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.

□

F.3.2 Dispute bisection game

Lemma F.4 (P is inactive during the challenge path) *Consider a set of transactions $\{tx\} \subseteq \{\texttt{KickOff}\} \cup \{\texttt{TraceChallenge}_\texttt{i}\}_{i \in \{1,\dots,\tilde{N_1}\}}$, 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* $\texttt{TraceResponse}_1$ *within Δ,*
2. $\texttt{TraceChallenge}_i \in \{tx\}$ *for $i < \tilde{N_1}$ (let $j = i$) and P has not posted* $\texttt{TraceResponse}_{i+1}$ *within Δ,*
3. $\texttt{TraceChallenge}_{\tilde{N_1}} \in \{tx\}$ *(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 $\langle \alpha + \beta + (4\tilde{N_1} + 4\tilde{\tilde{N}_2} - 2j + 10)f \rangle_V$.

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 $u_1 = \langle \alpha + \beta + (4\tilde{N_1} + 4\tilde{\tilde{N}_2} + 10) \rangle_V$, 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 $\{\texttt{TraceResponse}_\texttt{k}\}_{k \in \{1,\dots,j\}}$ paying extra jf in transaction fees, and V, in turn, has posted $\{\texttt{TraceChallenge}_\texttt{k}\}_{k \in \{1,\dots,j\}}$ paying jf in fees too. Now, if V utilizes the timelock, her balance account is $\langle u_1 = \alpha + \beta + (4\tilde{N_1} + 4\tilde{\tilde{N}_2} - 2j + 10)f \rangle_V$ 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. □

Lemma F.5 (V is inactive during the challenge path) *Consider a set of transactions $\{tx\} \subseteq \{\texttt{TraceChallenge}_\texttt{i}\}_{i \in \{1,\dots,\tilde{N_1}\}} \cup \{\texttt{CommitInstruction}\}$, where $\{tx\} \neq \emptyset$, is published on-chain and one of the following scenarios is true:*

1. $\texttt{TraceResponse}_i \in \{tx\}$ *for some $i \leq \tilde{N_1}$ (let $j = i$), and V has not posted* $\texttt{TraceChallenge}_i$ *within Δ,*

2. $\texttt{CommitInstruction} \in \{tx\}$ *(let* $j = \tilde{N}_1 + 1$*) and* V *has not posted any transaction* $tx' \in \{\texttt{ChallengeCurrPC}, \texttt{PunishInstruction}, \texttt{ChallengeRead}, \texttt{ChallengeWrite}\}$ *within* Δ.

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

Proof: Since $\texttt{TraceResponse}_i$ is posted on-chain, `Setup`, `Close`, `KickOff` are posted on-chain as well. Moreover, if $i > 1$, $\{\texttt{TraceResponse}_k\}_{k\in\{1,\ldots,i-1\}}$ and $\{\texttt{TraceChallenge}_k\}_{k\in\{1,\ldots,i-1\}}$ are also published on-chain. More specifically, first, P committed the `Setup` and the `Close`. Then, V posted `KickOff`. Furthermore, if $i > 1$, P has also published on-chain $\{\texttt{TraceResponse}_k\}_{k\in\{1,\ldots,i-1\}}$ and V the respective $\{\texttt{TraceChallenge}_k\}_{k\in\{1,\ldots,i-1\}}$. 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 `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 $u_1 = \langle \alpha+\beta+(4\tilde{N}_1+4\tilde{\tilde{N}}_2-2j+11)\rangle_P$. 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. □

Lemma F.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* $O(logn)$ *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: $\mathcal{A}[1] = \mathcal{B}[1]$, and $\mathcal{A}[n] \neq \mathcal{B}[n]$, 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 $r^V_{i+1} \leftarrow m$ (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 $r^P_{i+1} \leftarrow m$ (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, $r^P_{i+1} = m = \lfloor \frac{l_i + r_i}{2} \rfloor \geq \lfloor \frac{2l_i+2}{2} \rfloor = l_i + 1 = l^P_{i+1} + 1$.

  Therefore, since $l_{i+1} = l^P_{i+1} = l^V_{i+1} = l_i$, $r_{i+1} = r^P_{i+1} = r^l_{i+1} = m$, $r_{i+1} > l_{i+1}$, $i^V = i^P = i+1$, 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 $l^P_{i+1} = l^V_{i+1} = l_{i+1}$, $r^P_{i+1} = r^l_{i+1} = r_{i+1}$, $r_{i+1} > l_{i+1}$. Since, the subsequences are decreasing to the half after every step i and $r_{i+1} > l_{i+1}$, after $O(logn)$ steps the algorithm will pinpoint the point of disagreement. □

Lemma F.7 (P has committed to the wrong state and V initiates a dispute) *Assume that P has committed to an execution trace E^P_{final} in* `CommitComputation` *different than the VM execution trace element at step $final$, i.e., $E^P_{final} \neq E_{final}$. Assume that V follows the protocol specifications, publishes on-chain* `KickOff` *and P and V run Algorithm 21. Algorithm 21 outputs a step $\mathcal{N}$ such that P has committed to the execution traces $E^P_{\mathcal{N}} = E_{\mathcal{N}}$ at step $\mathcal{N}$ and $E^P_{\mathcal{N}+1} \neq E_{\mathcal{N}+1}$ 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^P_0], and $i = final$ for which P has committed to E^P_{final} in `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 $i \in \mathcal{I}$, $\mathcal{A}[i] = E^P_{final}$, 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 F.6 the bisection game outputs a step $\mathcal{N}$ such that $\mathcal{A}[\mathcal{N}] = \mathcal{B}[\mathcal{N}]$ and $\mathcal{A}[\mathcal{N}+1] \neq \mathcal{B}[\mathcal{N}+1]$. □

F.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 $\texttt{TraceChallenge}\tilde{\texttt{N}}_1$, which can only exist on-chain if `Setup` has previously been published on-chain too. Therefore, according to Lemma F.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 F.1.

Lemma F.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.*

- 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 $\langle \alpha+\beta+(2\tilde{N_1}+4\tilde{\tilde{N}}_2+8)f\rangle_P$, 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 $\langle \alpha+\beta+(2\tilde{N_1}+4\tilde{\tilde{N}}_2+9)f\rangle_V$.*

Proof: Since `ChallengeCurrPC` is published on-chain, P has published on chain the set of transactions $\texttt{Setup} \cup \texttt{CommitComputation} \cup \{\texttt{TraceResponse}_\texttt{i}\}_{i\in\{1,...,\tilde{N_1}\}} \cup \texttt{CommitInstruction}$ and V has published $\texttt{KickOff} \cup \{\texttt{TraceChallenge}_\texttt{i}\}_{i\in\{1,...,\tilde{N_1}\}} \cup \texttt{ChallengeCurrPC}$ 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 $\mathsf{CountZeroes}(N') = i$.

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. □

Lemma F.9 (P claims an incorrect 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 $\langle \alpha + \beta + (2\tilde{N}_1 + 4\tilde{\tilde{N}}_2 + 9)f \rangle_V$.*

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

We remind that the pc_θ-th spending condition of $\mathsf{CIScriptPCCurr_i}$ 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 $\mathsf{CIScriptPCCurr_i}$.

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 $\langle \alpha + \beta + (2\tilde{N}_1 + 4\tilde{\tilde{N}}_2 + 9)f \rangle_V$ 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. □

F.4.1 Read bisection game

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

1. Scenario 1, V is inactive: *Assume that* $\texttt{ReadResponse}_\texttt{i}$*, $i \in \{1, ..., \tilde{\tilde{N}}_2\}$ is published on-chain. If* $\texttt{ReadChallenge}_\texttt{i}$ *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 $(\alpha + \beta + (2\tilde{N}_1 + 4\tilde{\tilde{N}}_2 + 0 - 2i)f)_P$, and V's balance account is $(0)_V$.*
2. Scenario 2, P is inactive: *Assume that a set of transactions $\{tx\} \subseteq \{\texttt{ChallengeRead}\} \cup \{\texttt{ReadChallenge}_\texttt{i}\}_{i \in \{1,...,\tilde{\tilde{N}}_2\}}$, where $\{tx\} \neq \emptyset$, is published on-chain and one of the following scenarios is true:*
   (a) $\texttt{ChallengeRead} \in \{tx\}$ *(let $j = 0$) and P has not posted* $\texttt{ReadResponse}_1$ *within Δ,*
   (b) $\texttt{ReadChallenge}_i \in \{tx\}$ *for $i < \tilde{\tilde{N}}_2$ (let $j = i$) and P has not posted* $\texttt{ReadResponse}_{i+1}$ *within Δ,*

(c) $\texttt{ReadChallenge}_{\tilde{\tilde{N}}_2} \in \{tx\}$ *(let* $j = \tilde{\tilde{N}}_2$*) and* P *has not posted exactly one transaction* $tx' \in \{\texttt{MerkleRootHash}\} \cup \{\texttt{MerkleHash}_\texttt{i}\}_{i\in\{1,\ldots,\tilde{\tilde{N}}_2\}}$ *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* $\langle \alpha+\beta+(2\tilde{N}_1+4\tilde{\tilde{N}}_2-2j+8)f\rangle_V$*.*

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

1. In this scenario, P has published on-chain $S_P\cup\{\texttt{ReadResponse}_\texttt{k}\}_{k\in\{1,\ldots,i\}}$, and V has published on-chain $S_V \cup S_{V'}$, where $S_{V'} = \{\texttt{ReadChallenge}_\texttt{k}\}_{k\in\{1,\ldots,i-1\}}$ if $i > 0$, otherwise $V' = \emptyset$. Therefore, extra $2i-1$ have been spent in fees. Since $\texttt{ReadResponse}_\texttt{i}$ is published on-chain and V has not published $\texttt{ReadChallenge}_\texttt{i}$ 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 $u_1 = (\alpha+\beta+(2\tilde{N}_1+4\tilde{\tilde{N}}_2+9-2i)f)$ 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 $u_1 = (\alpha+\beta+(\alpha+\beta+(2\tilde{N}_1+4\tilde{\tilde{N}}_2+8-2j)f)$ 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. □

Lemma F.11 (Read bisection game completes) *Assume that* V *publishes on-chain the transaction* `ChallengeRead` *by spending the script* $\mathsf{CIScriptRead}_A$ *(or* $\mathsf{CIScriptRead}_B$*) 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.,* $E^P_{\mathcal{N}} = E_{\mathcal{N}}$*, 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* $\langle \alpha+\beta+(2\tilde{N}_1+2\tilde{\tilde{N}}_2+7)f\rangle_V$*.*

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* $\langle \alpha+\beta+(2\tilde{N}_1+2\tilde{\tilde{N}}_2+8)f\rangle_V$*.*

Proof: As explained in Lemma F.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 $\mathsf{DisagreementReadP}(\mathcal{B},\mathsf{n})$ 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 $\mathsf{DisagreementReadP}(\mathcal{A},\mathsf{n})$ 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 $MR^P_{\mathcal{N}}$ in the dispute bisection game (in the trace element $E^P_{\mathcal{N}}$), and the leaf of the path ($i = \tilde{N}_2$) to which P committed in `CommitInstruction`, i.e., $M^P_{\mathcal{N}}[addrA_\theta] = valA_\theta$. 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, $i \in \{1, ..., \tilde{N}_2\} \setminus \mathcal{I}$, 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 F.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 $\tilde{\tilde{N}}_2$ steps. Depending on the value of $\mathcal{N}_{Mer}$, P can spend the transaction $\texttt{ReadChallenge}_{\tilde{\tilde{N}}_2}$ as follows.

$\mathcal{N}_{Mer} = 0$ P can unlock the script $\mathsf{RootReadScript_i}$ (Algorithm 18) for some $i \in \{1, ..., \tilde{N}_1\}$ 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 $\mathsf{Count\_Zeroes(N)} = \mathsf{i}$. 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 $\texttt{ReadResponse}_{\tilde{\tilde{N}}_2}$ which is on of the children of the root $\mathcal{B}[1] = MR^P_{\mathcal{N}}$. To unlock the script, P must provide as input three nodes $(\mathsf{Npar}, \mathsf{Nchild}, \mathsf{Nsib})$ s.t. $\mathsf{Npar} = \mathcal{B}[1]$, $\mathsf{Nchild} = \mathcal{B}[2]$, 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 $\texttt{RootReadScript}_\mathsf{i}$ 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 $\mathsf{Nchild} = \mathcal{A}[2] \neq \mathcal{B}[2]$ or $\mathsf{Npar} = \mathcal{A}[1] \neq \mathcal{B}[1]$. In both cases V proves the equivocation and publish on-chain `PunishRead3` to claim the coins in the multisignature, since the scripts $\mathsf{ChallScript_i}$ and $\mathsf{RootReadScript_i}$ have the same hard-coded public key for $MR^P_{\mathcal{N}}$ and the scripts $\mathsf{ReadChallScript_5}$ and $\texttt{RootReadScript}_\mathsf{i}$ have the same hard-coded public key for $\mathsf{Node_{d_0}}$.

$\mathcal{N}_{Mer} \neq 0$ To spend the transaction $\texttt{ReadChallenge}_{\tilde{N}_2}$ P must unlock the script $\mathsf{ValueAScript}$ if $\mathcal{N}_{Mer} = \tilde{N}_2 - 1$, otherwise unlock the script $\mathsf{HashReadScript_i}$ (Algorithm 16) for some i s.t. $\mathsf{Count\_Zero}(\mathcal{N}_{\mathsf{Mer}}) = \mathsf{i}$. In both scenarios, P must provide as input three nodes in the path $(\mathsf{Npar}, \mathsf{Nchild}, \mathsf{Nsib})$ s.t. $\mathsf{Npar} = \mathcal{B}[j]$, $\mathsf{Nchild} = \mathcal{B}[j+1]$, 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 $\mathsf{Npar} \neq \mathcal{B}[j]$ or $\mathsf{Nchild} \neq \mathcal{B}[j+1]$. More specifically, we have the following scenarios:

- $\mathcal{N}_{Mer} \in \{1, ..., \tilde{N}_2 - 2\}$: In this scenario, V can show the equivocation because the hard-coded public keys for the pair $\mathsf{Nchild}, \mathsf{Npar}$ corresponding to $\mathsf{HashReadScript_i}$ 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 $\mathcal{N}_{Mer} \in \{1, ..., \tilde{N}_2 - 2\}$. 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 $\{\mathtt{ReadChallenge_i}, \mathtt{WriteChallenge_i}\}_{i \in \{1,...,\tilde{\tilde{N}}_2\}}$ is published on-chain along with two extra transactions, when P equivocate while. Thus, $(2\tilde{\tilde{N}}_2 + 2)f$ have been spent in transaction fees. Therefore, if V disproves P, V's balance account is $\langle \alpha + \beta + (2\tilde{N}_1 + 2\tilde{\tilde{N}}_2 + 7)f \rangle_V$ 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 $\mathsf{ValueAScript}$ or $\mathsf{HashReadScript_i}$ or $\mathsf{RootReadScript_i}$ for some $i \in \{1, ..., \tilde{N}_1\}$), to spend the transaction $\mathtt{ReadChallenge}_{\tilde{\tilde{N}}_2}$. 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 $\langle \alpha + \beta + (2\tilde{N}_1 + 2\tilde{\tilde{N}}_2 + 8)f \rangle_P$ and V's balance account is $\langle 0 \rangle_V$. This is the dominant strategy for P, since otherwise the respective balance account is $\langle 0 \rangle_P$. □

F.4.2 Write bisection game

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

1. *V* is inactive: *Assume $\mathtt{WriteResponse_i}$, $i \in \{1, ..., \tilde{\tilde{N}}_2\}$ is published on-chain. If $\mathtt{WriteChallenge_i}$ 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 $(\alpha + \beta + (2\tilde{N}_1 + 4\tilde{\tilde{N}}_2 + 0 - 2i)f)_P$, and V's balance account is $(0)_V$.*
2. *P is inactive: Assume that a set of transactions $\{tx\} \subseteq \{\mathtt{ChallengeValueC}\} \cup \{\mathtt{WriteChallenge_i}\}_{i \in \{1,...,\tilde{\tilde{N}}_2\}}$, where $\{tx\} \neq \emptyset$, is published on-chain and one of the following scenarios is true:*
   (a) *$\mathtt{ChallengeValueC} \in \{tx\}$ (let $j = 0$) and P has not posted $\mathtt{WriteResponse}_1$ within Δ,*
   (b) *$\mathtt{WriteChallenge}_i \in \{tx\}$ for $i < \tilde{\tilde{N}}_2$ (let $j = i$) and P has not posted $\mathtt{WriteResponse}_{i+1}$ within Δ,*

(c) $\texttt{WriteChallenge}_{\tilde{\tilde{N}}_2} \in \{tx\}$ *(let* $j = \tilde{\tilde{N}}_2$*) and* P *has not posted exactly one transaction* $tx' \in \{\texttt{MerkleRootHash}\} \cup \{\texttt{MerkleHash}_\texttt{i}\}_{i\in\{1,\ldots,\tilde{\tilde{N}}_2\}}$ *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* $\langle \alpha+\beta+(2\tilde{N}_1+4\tilde{\tilde{N}}_2-2j+8)f\rangle_V$*.*

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

1. In this scenario, P has published on-chain $P \cup \{\texttt{WriteResponse}_\texttt{k}\}_{k\in\{1,\ldots,i\}}$, and V has published on-chain $V \cup V'$, where $V' = \{\texttt{WriteChallenge}_\texttt{k}\}_{k\in\{1,\ldots,i-1\}}$ if $i > 0$, otherwise $V' = \emptyset$. Therefore, extra $2i-1$ have been spent in fees. Since $\texttt{WriteResponse}_\texttt{i}$ is published on-chain and V has not published $\texttt{WriteChallenge}_\texttt{i}$ 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 $u_1 = (\alpha+\beta+(2\tilde{N}_1+4\tilde{\tilde{N}}_2+9-2i)f)$ 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 $u_1 = (\alpha+\beta+(\alpha+\beta+(2\tilde{N}_1+4\tilde{\tilde{N}}_2+8-2j)f)$ 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. □

Lemma F.13 (The Write bisection game completes) *Assume that* V *spends the script* $\mathsf{CIScriptWrite}_C$ *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* $\langle \alpha+\beta+(2\tilde{N}_1+2\tilde{\tilde{N}}_2+7)f\rangle_V$*.*

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* $\langle \alpha+\beta+(2\tilde{N}_1+2\tilde{\tilde{N}}_2+8)f\rangle_V$*.*

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 F.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 $\mathsf{DisagreeWriteV}(\mathcal{B}_1, \mathcal{B}_2, \tilde{N}_2)$ 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., $\forall i \in \{1, ..., \tilde{N_2}\}$, it holds that $(\mathcal{B}_1[i], \mathcal{B}_2[i]) = (\pi[i], \pi'[i])$. On the other side, P runs function $\mathsf{DisagreeWriteP}(\mathcal{A}_1, \mathcal{A}_2, \tilde{N_2})$ 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 $\mathcal{A}_1[1] = MR^P_{\mathcal{N}}, \mathcal{A}_2[1] = MR^P_{\mathcal{N}+1}$ to which P committed via the respective execution trace elements in the dispute bisection game, and the leaf of the paths ($i = \tilde{N_2}$), $\mathcal{A}_1[1] = MR^P_{\mathcal{N}}, \mathcal{A}_2[1] = MR^P_{\mathcal{N}+1}$ 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 Appendix D.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 F.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 F.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 F.11).

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

F.5 Concluding Lemmas

Lemma F.14 (P has committed to the wrong state and `CommitInstruction` is published on-chain) *Assume that P has committed to the execution trace E^P_{final} in `CommitComputation` s.t. $E^P_{final} \neq E_{final}$, 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 $u \geq \alpha + \beta + (2\tilde{N_1} + 2\tilde{\tilde{N}}_2 + 7)f$ 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 $\texttt{TraceChallenge}_{\tilde{N_1}}$ which can only be published on chain if the set of transactions $\{\texttt{KickOff}\} \cup \{\texttt{TraceChallenge}_i\}_{i \in \{1,...,\tilde{N_1}-1\}} \cup \{\texttt{TraceResponse}_i\}_{i \in \{1,...,\tilde{N_1}\}}$

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 F.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 5 8. 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 F.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 $\langle \alpha + \beta + (2\tilde{N}_1 + 4\tilde{\tilde{N}}_2 + 9)f \rangle_V$.
  - *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 5. However, in that case, P commits to a program instruction such that $\Pi(pc_\theta) \neq (insType_\theta, addrA_\theta, addrB_\theta, addrC_\theta)$ in `CommitInstruction`. Following from Lemma F.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 $\langle \alpha + \beta + (2\tilde{N}_1 + 4\tilde{\tilde{N}}_2 + 9)f \rangle_V$.
  - *P reads incorrect values from the memory:* P can read incorrect values (val_A or val_B) from the memory ($M^{\mathcal{N}}[addrA]$ or $M^{\mathcal{N}}[addrB]$) at step $\mathcal{N}$ (lines 6, 7). 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 F.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 F.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 $\langle \alpha + \beta + (2\tilde{N}_1 + 2\tilde{\tilde{N}}_2 + 7)f \rangle_V$.
- *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 $\texttt{TraceChallenge}_{32}$. Therefore the values related to the execution of step $\mathcal{N}+1$ that P committed in `CommitInstruction` are correct. 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 F.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 $\langle \alpha + \beta + (2\tilde{N}_1 + 4\tilde{\tilde{N}}_2 + 9)f \rangle_V$.

– *P has committed to all the correct values in* `CommitInstruction` *but $E^P_{\mathcal{N}+1} \neq E_{\mathcal{N}+1}$ or is inactive during the Write bisection game*: according to Lemmas F.13, F.12, 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 $\langle \alpha + \beta + (2\tilde{N}_1 + 2\tilde{\tilde{N}}_2 + 7)f \rangle_V$.

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 $u \geq \alpha + \beta + (2\tilde{N}_1 + 2\tilde{\tilde{N}}_2 + 7)f$ and P's balance account is in any case $(0)_P$. □

Lemma F.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_θ, $pc_{\theta'}$, $insType_\theta$, $addrB_\theta$, $addrC_\theta$, $valA_\theta$, $valB_\theta$, $valC_\theta$. 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 $u \geq \alpha + \beta + (2\tilde{N}_1 + 2\tilde{\tilde{N}}_2 + 8)f$ 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 F.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 $E^P_i = E_i$. 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 $pc_\theta \neq pc^P_{\mathcal{N}}$ (or $pc_{\theta'} \neq pc^P_{\mathcal{N}} + 1$): By assumption, $pc_\theta = pc^P_{\mathcal{N}}$ and $pc_{\theta'} = pc^P_{\mathcal{N}+1}$, and according to Lemma F.8, it is a dominant strategy for P to claim the deposits. P's balance account will be in the worst case $\langle \alpha + \beta + (2\tilde{N}_1 + 4\tilde{\tilde{N}}_2 + 8)f \rangle_P$, and V's balance account $\langle 0 \rangle_V$.
- V publishes on-chain `ChallengeRead` to claim that $valA_\theta \neq M^{\mathcal{N}}[addrA_\theta]$ (or $valB_\theta \neq M^{\mathcal{N}}[addrB_\theta]$): 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 F.10, F.11 accordingly. $\langle \alpha + \beta + (2\tilde{N}_1 + 2\tilde{\tilde{N}}_2 + 8)f \rangle_P$, 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 F.12, F.13 that V will eventually claim the coins in the multisignature. Therefore, P's balance account is in the worst case $\langle \alpha + \beta + (2\tilde{N}_1 + 4\tilde{\tilde{N}}_2 + 8)f \rangle_P$, and V's balance account $\langle 0 \rangle_V$.

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

F.6 Theorems

To prove that BitVM satisfies *Rational Validity* and *Balance Security*, we first represent BitVM as an EFG which we illustrate in Figs. 4 and 5.

BitVM **as an EFG.** We represent BitVM as an extensive-form game, where the players are P and V. The state of a node in the game tree is defined by the pair (A, B), where A represents the balance account of P and B represents the balance account of V.

The game begins after `Setup` is posted on-chain. The action set is the following. Initially, P has the possible actions: i) not post a `CommitComputation` transaction on-chain, ii) post a `CommitComputation` and commit to the correct result, or iii) post a `CommitComputation` but commit to an incorrect result. In case i), it is a dominant strategy for V to claim the funds after the timelock expires (cf. Lemma F.2). In the other cases (ii and iii), V must decide whether to post a `KickOff` transaction, initiating the dispute phase, or remain inactive. If V does not respond, P has the following actions: i) post a `Close` transaction corresponding to the result committed to `CommitComputation`, ii) remain inactive, or iii) post a `Close` transaction that does not match the result P committed to `CommitComputation`. In the latter two cases (ii and iii), it is a dominant strategy for V to claim the funds once the timelock expires or to prove the equivocation made by P (cf. Lemma F.3). For convenience, and due to similarity, we combine cases i), ii) in the same node.

On the other side, if V initiates the dispute phase by posting `KickOff`, the game enters the Identify Disagreement phase. During this phase, if either player remains inactive, it is a dominant strategy for the other party to claim the funds (cf. Lemmas F.4, F.5). If the dispute completes, the outcome depends on the correctness of the result to which P committed in `CommitInstruction`. More specifically, we have the following scenarios, i) Dispute A: if P committed to an incorrect result, V will claim the funds in the Punishment subtree A (cf. Lemma F.14), ii) Dispute B: if P committed to the correct result in `CommitComputation`, P will eventually claim the funds in the Punishment subtree B (cf. Lemma F.15).

Theorem F.16 *The strategy profile representing the honest execution of* BitVM *forms a Subgame Perfect Nash Equilibrium.*

Proof: We prove that by backward induction on Γ depicted in Figs. 4 and 5.

If P does not post `CommitComputation` on-chain, V will claim the funds after the timelock expires. If P posts `CommitComputation` committing to an incorrect result of the computation, V will publish `KickOff` and eventually claim the funds in the multisignature. If P commits to the correct result of the computation in `CommitComputation` but does not post the respective `Close` transaction, V will again claim the funds. On the other side, if P posts the correct result of the computation in `CommitComputation` and V posts `KickOff` on-chain, P will eventually claim the coins. □

Theorem F.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 taking as input the correct result of the computation.

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 commit to the correct final result (by posting `CommitComputation` on-chain). Moreover, after P has posted the correct result, in the

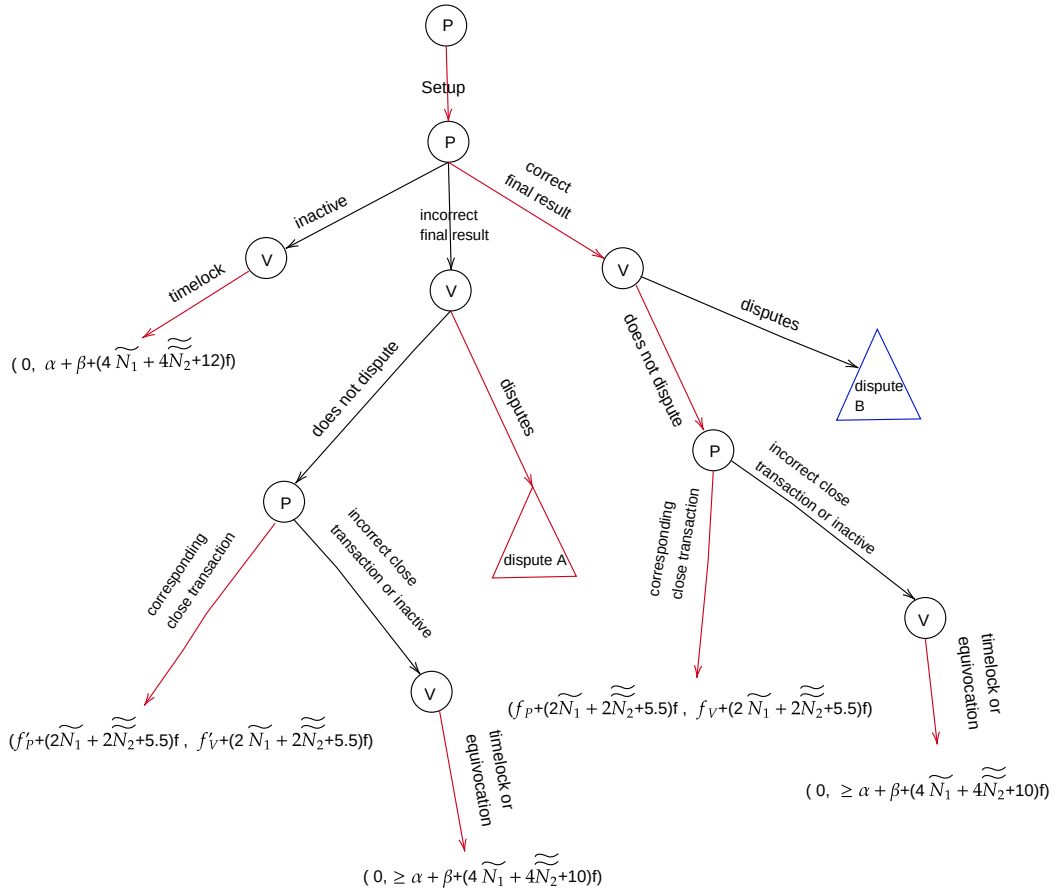

Figure 4: Tree representation Γ of the EFG describing BitVM. We underline with red the actions each parties takes at each step. The subtrees dispute A and dispute B are depicted in Fig. 5.

case where V 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.

- *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 publish 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. □

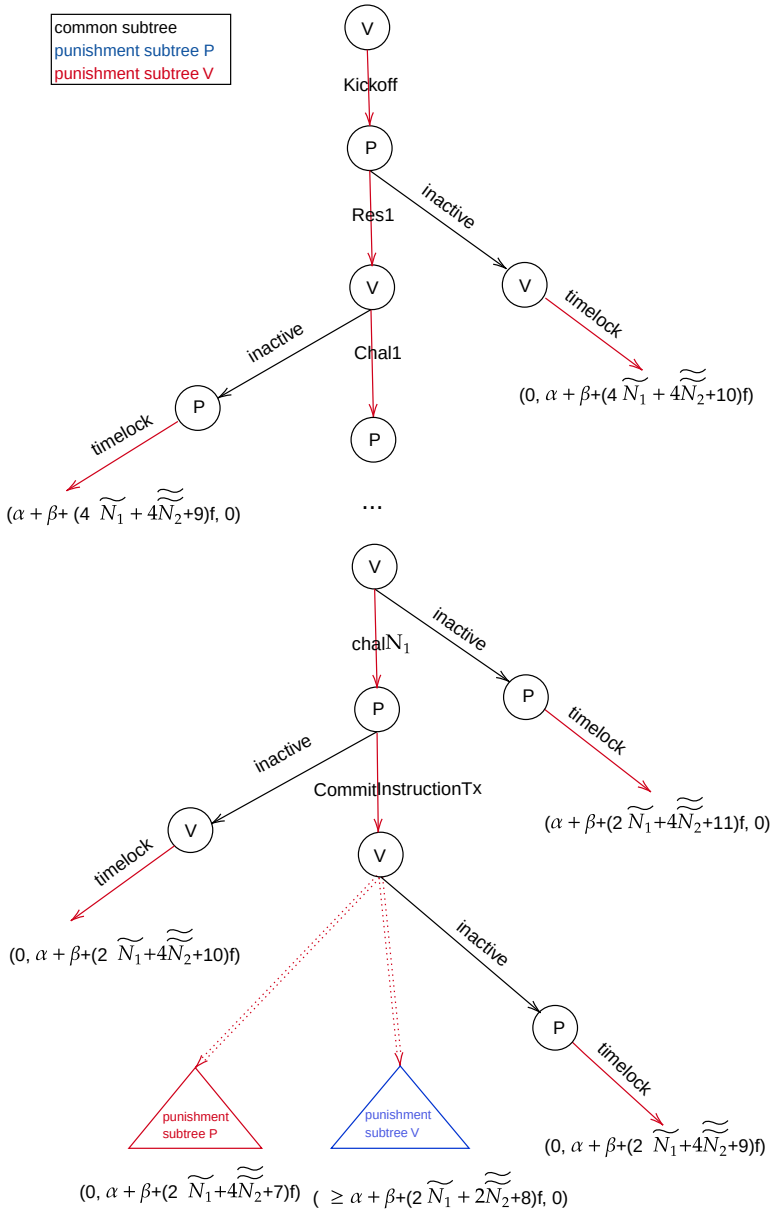

Figure 5: The tree Γ' illustrates Subtree A and Subtree B as depicted in Fig. 4. Subtree A, initiated by an honest V to disprove a malicious P, consists of the "Common Subtree" and "Punishment subtree P ". Subtree B, initiated by a malicious V trying to a correct P, consists of the "Common Subtree" and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

G Transaction computation

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

As shown in [2], computing the size of a SegWit [31] transaction requires computing 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. In our protocol, it can be up to $37vB$: $34vB$ the size of the scriptPubKey for the PayToTaproot(P2TR)), which we use to implement the n-of-n multisignature, $3vB$ for the size of a relative timelock.

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, for a witness of a n-of-n multisignature, the size of the control block is $33B$, the size of the script is $(35 \cdot n + 2)B$ and the size of the script data is $(65 \cdot n)B$, resulting in vByte size equal to $((100 \cdot n) + 35)/4vB$.