BitVM: Quasi-Turing Complete Computation on Bitcoin

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

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

1. Introduction

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

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

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

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

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

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

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

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

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

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

The contributions of this paper are summarized below.

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

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

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

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

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

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

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

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

2. Model

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

2.1. System model

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

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

4. Configured with 2^{32} memory cells for 32-bit integers, supporting up to 2^{32} instructions and steps.

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

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

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

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

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

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

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

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

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 $w_\mathsf{L}(P, tx) := \mathsf{bal}_{\mathsf{L}_1}(P) - \mathsf{bal}_{\mathsf{L}_2}(P)$, where $\mathsf{L}_1 \prec \mathsf{L}$ is the ledger up to (not including) tx and $\mathsf{L}_2 := \mathsf{L}_1 || tx$. Usually, it is obvious which ledger we refer to, so we omit the subscript. In addition to balances of parties, a ledger state $\mathsf{st}(\mathsf{L})$ can include a string $s \in \{0,1\}^*$, denoted as $s \in \mathsf{st}(\mathsf{L})$, if there exists a transaction $tx \in \mathsf{L}$, such that tx contains the string s.

2.2. Threat model

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

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

2.3. Protocol goals

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

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

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

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

- $\mathsf{setup}(\mathsf{in}_P, \mathsf{in}_V, \Pi, f)$: 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 $v_P + v_V \leq \mathsf{in}_P + \mathsf{in}_V$ and returns an instance $\mathcal{I}$.
- $\mathsf{execute}(\mathcal{I}, x)$: 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 $\mathcal{I} \leftarrow \mathsf{setup}(\mathsf{in}_P, \mathsf{in}_V, \Pi, f)$, and then $\mathsf{execute}(\mathcal{I}, x)$ 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 $\sum_{tx \in \mathcal{T}}(w(tx, A)) \geq v_A$ where $v_A = f_A(\Pi(x))$, for any honest $A \in \{P, V\}$.

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

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 some key primitives our construction builds upon.

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

3.1. Transactions in the UTXO model

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

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

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

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

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

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

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

Tapleaves, form the leaves of a Merkle tree. To spend a UTXO locked by a Taptree locking script, the user must provide a witness for one of the Tapleaves along with proof of inclusion of that leaf in the Taptree.
We denote the Tapleaves of a Taptree locking script as $\langle \mathsf{leaf}_1, \ldots, \mathsf{leaf}_r \rangle$. When a user fulfills the script leaf_i to unlock the j-th output of the transaction $\mathtt{Tx}$, the corresponding input is represented as $(\mathtt{Tx}, j, \langle \mathsf{leaf}_i \rangle)$.
Whenever a user spends a UTXO via a Tapleaf of a Taptree, we assume that they have provided a valid Merkle proof of inclusion for that Tapleaf.
- **Other conditions.** We denote with True (False) a condition that is always fulfilled (can never be fulfilled).

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

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

3.2. Lamport digital signature scheme

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

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

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

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

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

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

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

```
function Lamp.KeyGen(ℓ)
    Let sk_M ← ( x[0,0], ..., x[0, ℓ−1] ; x[1,0], ..., x[1, ℓ−1] ), where every element x[i,j] is sampled uniformly at random from the set X;
    for i = 0,1 and j = 0,...,ℓ−1 do
        y[i,j] ← h(x[i,j]);
    Let pk_M ← ( y[0,0], ..., y[0, ℓ−1] ; y[1,0], ..., y[1, ℓ−1] );
    return (sk_M, pk_M).
```

where in line 2 $sk_\mathcal{M} \leftarrow \begin{pmatrix} x[0,0], \ldots, x[0,\ell-1] \\ x[1,0], \ldots, x[1,\ell-1] \end{pmatrix}$ and in line 5 $pk_\mathcal{M} \leftarrow \begin{pmatrix} y[0,0], \ldots, y[0,\ell-1] \\ y[1,0], \ldots, y[1,\ell-1] \end{pmatrix}$.

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 LampSig_{sk_M}(m)
    for i = 0,...,ℓ−1 do
        Let c_m[i] ← sk_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 Lamp.Vrfy_{pk_M}(m, c_m)
    for i = 0,...,ℓ−1 do
        if h(c_m[i]) ≠ pk_M[m[i], i] then
            return False;
    return True.
```

Algorithm 4 The $\mathsf{CheckEquivocation}$ 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 $\mathsf{CheckEquivocation}(pk_{\mathcal{B}}, x', x'')$
  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.

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

3.3. Stateful Bitcoin scripting

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

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

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 $\mathsf{out}_1 := (a\text{B}, \mathsf{CheckComm}_{pk_{\mathcal{M}}})$ and $\mathsf{out}'_1 := (b\text{B}, \mathsf{CheckComm}_{pk_{\mathcal{M}}})$. To unlock both out_1 and out'_1, a Lamport commitment c_m must be provided. Since the same Lamport public key appears in both scripts, every party in the network knows that when U unlocks these scripts, U is assigning a value to the same variable $\mathcal{M}$. Following from *one-time security*, no user other than U can assign a different value to $\mathcal{M}$ without knowing $sk_{\mathcal{M}}$. Moreover, U cannot assign two different values $m_1 \neq m_2$ to $\mathcal{M}$ without equivocating, which is detectable and can be punished on-chain.

4. BitVM **Virtual Machine**

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

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

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

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

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

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

Algorithm 5 State Transition Function f_{ST}.

function $f_{ST}(M, pc, \Pi)$
  $M' \leftarrow M$;
  $(f_{\mathsf{OP}}, addr_A, addr_B, addr_C) \leftarrow \Pi[pc]$;
  $val_A \leftarrow M[addr_A]$;
  $val_B \leftarrow M[addr_B]$;
  $(pc', val_C) \leftarrow f_{\mathsf{OP}}(val_A, val_B, pc)$;
  if $val_C \neq \perp$ **then**
    $M'[addr_C] \leftarrow val_C$;
  return (M', pc', Π).

8. Even though $\mathcal{OP}$ can be arbitrary, we are interested in a Turing-complete instruction set. In particular, we later use ADD, BEQ, and JMP, cf. Algorithm 7 – a well-known Turing-complete instruction set [30].

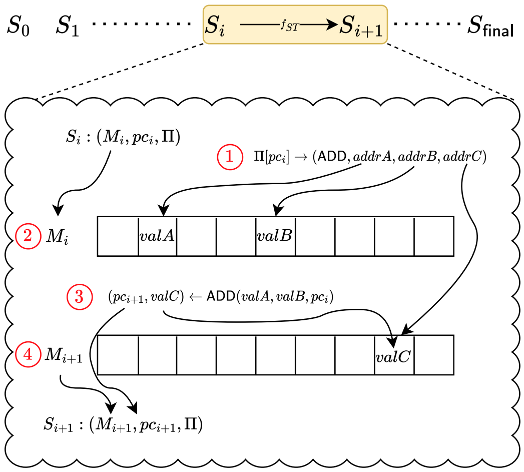

Figure 1. Overview of the state transition function execution f_{ST}. Given a state S_i: (1) instruction $\Pi[pc_i]$ is fetched, (2) values $valA, valB$ are taken from memory at their respective addresses, (3) the instruction is executed, and (4) part of the result (i.e., $valC$) is stored in the memory. The state transition function outputs the new state S_{i+1}.

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

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

We define a VM instance as a tuple

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

We write Γ^A to refer to the VM instance executed by party A. We write S^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 8 (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 $i \in \{0, \dots, \mathsf{final}\}$.

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 consecutive trace elements $ExecTrace := (E_0, \dots, E_{\mathsf{final}})$. We write $ExecTrace^A$ as a short-hand for $(E^A_0, \dots, E^A_{\mathsf{final}})$.

We describe how the VM behaves in Algorithm 6: starting from initial state S_0, it applies the state transition function f_{ST} to the state and records the related trace elements until the program Π ends, namely, once pc is set to be $\perp$. 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 VM_final(S_0)
    stepCount ← 0;
    while stepCount < final do
        E_stepCount ← (MR, pc);
        (M, pc, Π) ← f_ST(M, pc, Π);
        increment stepCount by 1;
    E_stepCount ← (MR, pc);
    ExecTrace ← (E_0, ..., E_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 $n = 2^{32}$.

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

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

5. The BitVM protocol

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

Protocol overview. The protocol proceeds in four phases.

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

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

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

function ADD(pc, val_A, val_B)
  if $pc = \perp$ **then return** $(\perp, \perp)$;
  return $(pc + 1, val_A + val_B)$.

function BEQ(pc, val_A, val_B)
  if $pc = \perp$ **then return** $(\perp, \perp)$;
  if $val_A = val_B$ **then**
    return $(pc + 1, \perp)$.
  else
    return $(pc + 2, \perp)$.

function JMP(pc, val_A, val_B)
  if $pc = \perp$ **then return** $(\perp, \perp)$;
  return $(val_A, \perp)$.

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

2) In the *VM execute* phase, performed off-chain, P and V generate the input S_0 for Π (e.g., the chess moves starting from a given chessboard state) and compute $\Pi(S_0)$.
3) In the *commit* phase, P posts the `CommitComputation` transaction on-chain, i.e., a transaction committing to the input S_0 and the result $\Pi(S_0)$ using Lamport signatures. Upon seeing this, V can either accept the claim as correct and wait for P to settle according to $f(\Pi(S_0))$ (via publishing a `Close` transaction), or, if the committed result does not match $\Pi(S_0)$, initiate a dispute.
4) In the event of a dispute, they enter the *resolve dispute* phase. This phase is the main technical challenge of the BitVM protocol, as it requires an on-chain mechanism to verify whether P's claimed result $\Pi(S_0)$ is correct, while remaining within Bitcoin's scripting limitations.

5.1. Resolving Disputes

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

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

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

Identify Disagreement. Recall that the total length of the execution trace is final. V initiates the bisection game by publishing a `Kickoff` transaction, forcing P to respond with a $\mathtt{TraceResponse_1}$ transaction that reveals the middle trace element, $E_{n_{31}}$, where $n_{31} = 2^{31} = \mathsf{final}/2$.

The loop then begins, where V forces P into progressively smaller sequences of the trace by publishing a $\mathtt{TraceChallenge_1}$ transaction, Lamport committing to a value b_{31} in its witness.

- If V agrees with $E_{n_{31}}$, V commits to $b_{31} = 1$, indicating that P should reveal a trace element in the right half of the sequence.
- If V disagrees with $E_{n_{31}}$, V commits to $b_{31} = 0$, indicating that P should reveal a trace element in the left half of the sequence.
- Based on b_{31}, P responds by publishing a $\mathtt{TraceResponse_2}$ transaction, revealing the middle element of the next sequence, $E_{n_{30}}$, where $n_{30} = b_{31} \cdot 2^{31} + 2^{30}$.

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

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

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

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

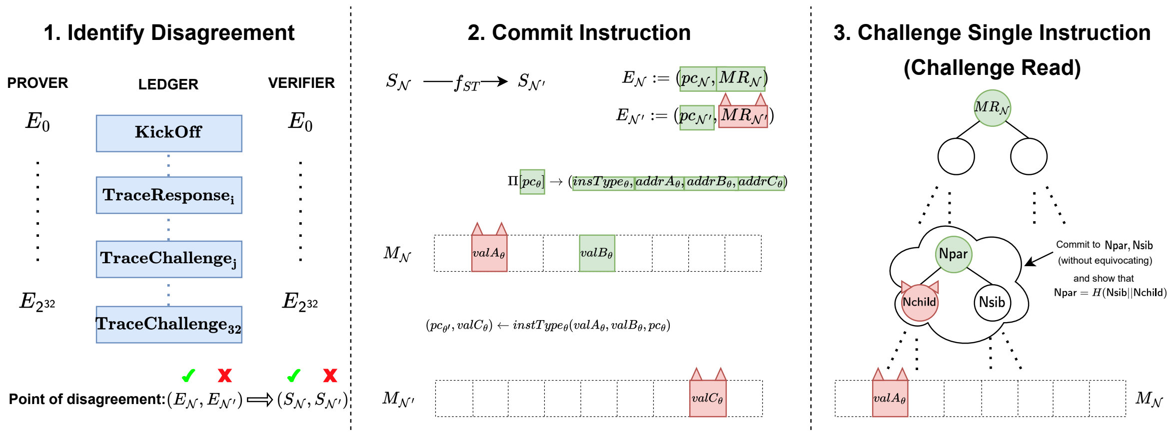

Figure 2. Example of dispute resolution in the BitVM protocol: To resolve a dispute, (1) P and V engage in a bisection game to identify the point of disagreement $(E_{\mathcal{N}}, E_{\mathcal{N}'})$ in their execution traces, indicating a disagreement in the transition from state $S_{\mathcal{N}}$ to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

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

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

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

Since V already committed to $\mathcal{N}$ bit-by-bit during the bisection game with P to locate a disagreement point, V could attempt to equivocate on one of the bits when recommitting to $\mathcal{N}$ in the punishment phase. If equivocation occurs, P publishes a `PunishCurrPC` transaction, revealing two different bitwise commitments to $\mathcal{N}$ linked to the verifier's public key. In this scenario, P claims all funds in the multisignature, thereby terminating the BitVM protocol.

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

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

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

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

The parties engage in a bisection game on the memory Merkle path $\mathcal{P}_R := (MR_{\mathcal{N}}, \ldots, M_{\mathcal{N}}[addrA_\theta])$, i.e., the path from the root $MR_{\mathcal{N}}$ to the leaf $M_{\mathcal{N}}[addrA_\theta]$. The goal is to isolate a point of disagreement between V and P. Over 5 rounds of challenge-response (given $\mathsf{MemLen} = 2^{32}$), they progressively narrow down the path until they identify a pair of nodes ($\mathsf{Npar}, \mathsf{Nchild}$) where they agree on Npar (i.e., the parent node) but disagree on Nchild (i.e., the child node). This bisection game proceeds in a similar fashion as the one for finding the disagreement in the execution trace.

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

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

5.2. Honest Closure

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

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

6. The BitVM full protocol

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

6.1. Setup

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

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

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

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

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

6.2. VM Execute

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

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

6.3. Commit

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

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

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

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.

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, $\mathtt{Close_1}, \ldots, \mathtt{Close_m}$. Each of these transactions distributes the funds according to the outcome mapping function f, applied to one of the possible results of the computation[10]. If P does not publish any $\mathtt{Close_i}$ 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 $\mathtt{Close_i}$ (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.

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

Algorithm 9 The script 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 $\mathsf{Equivocation}(\mathsf{pk}_{MR_{\mathsf{final}}[i]}, c_0, c_1)$ = 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.

$$
\begin{aligned}
&\mathtt{KickOff} := \\
&\Big([in = (\mathtt{CommitComputation}, 0, \mathsf{CheckMSig}_{\mathsf{pk}_{PV}})], \\
&\ wit = [(\sigma_{PV})], \\
&\ out = [(d\text{₿}; \langle \mathsf{ChallScript_1}, \mathsf{TL}(\Delta) \wedge \mathsf{CheckSig}_{\mathsf{pk}_V}\rangle)]\Big).
\end{aligned}
\tag{4}
$$

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

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

The parties engage in an on-chain interactive protocol known as dispute bisection game (cf. [31, Appendix B.1]): the game is played over the VM execution trace $ExecTrace := (E_0, \ldots, E_{\mathsf{final}})$ and has the goal to determine a pair of consecutive VM trace elements $(E_{\mathcal{N}}, E_{\mathcal{N}'})$, where $\mathcal{N}' := \mathcal{N} + 1$, such that they agree on $E_{\mathcal{N}}$ and disagree on $E_{\mathcal{N}'}$.

After that, V initiates the bisection game by publishing the `Kickoff` transaction, P responds by publishing the $\mathtt{TraceResponse_1}$ transaction (cf. Eq. (5)), committing to $E_{n_{31}}$ in the witness, where $n_{31} = 1 \cdot 2^{31}$.

$$
\begin{aligned}
&\mathtt{TraceResponse_1} := \\
&\Big(in = [(\mathtt{KickOff}, 0, \mathsf{CheckMSig}_{\mathsf{pk}_{PV}} \wedge \\
&\ \mathsf{CheckComm}_{\mathsf{pk}_{E_{n_{31}}}})], \\
&\ wit = [(\sigma_{PV}, E_{n_{31}}, c_{E_{n_{31}}})], \\
&\ out = [(d\text{₿}; \langle \mathsf{RespScript_1}, \mathsf{TL}(\Delta) \wedge \mathsf{CheckSig}_{\mathsf{pk}_P}\rangle)]\Big).
\end{aligned}
\tag{5}
$$

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

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

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

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

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

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

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

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

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

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

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

To unlock the $\texttt{TraceChallenge}_{32}$ 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 A.

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

function $\mathsf{ADDScript}(\sigma_{PV}, pc_\theta, c_{pc_\theta}, pc_{\theta'}, c_{pc_{\theta'}}, insType_\theta, c_{insType_\theta}, addrA_\theta, c_{addrA_\theta}, addrB_\theta, c_{addrB_\theta}, addrC_\theta, c_{addrC_\theta}, valA_\theta, c_{valA_\theta}, valB_\theta, c_{valB_\theta}, valC_\theta, c_{valC_\theta})$
  $\mathsf{CheckMSigVerify}_{\mathsf{pk}_{PV}}(\sigma_{PV})$;
  $\mathsf{CheckCommVerify}_{\mathsf{pk}_{pc_\theta}}(pc_\theta, c_{pc_\theta})$;
  $\mathsf{CheckCommVerify}_{\mathsf{pk}_{pc_{\theta'}}}(pc_{\theta'}, c_{pc_{\theta'}})$;
  $\mathsf{CheckCommVerify}_{\mathsf{pk}_{insType_\theta}}(insType_\theta, c_{insType_\theta})$;
  $\mathsf{CheckCommVerify}_{\mathsf{pk}_{addrA_\theta}}(addrA_\theta, c_{addrA_\theta})$;
  $\mathsf{CheckCommVerify}_{\mathsf{pk}_{addrB_\theta}}(addrB_\theta, c_{addrB_\theta})$;
  $\mathsf{CheckCommVerify}_{\mathsf{pk}_{addrC_\theta}}(addrC_\theta, c_{addrC_\theta})$;
  $\mathsf{CheckCommVerify}_{\mathsf{pk}_{valA_\theta}}(valA_\theta, c_{valA_\theta})$;
  $\mathsf{CheckCommVerify}_{\mathsf{pk}_{valB_\theta}}(valB_\theta, c_{valB_\theta})$;
  $\mathsf{CheckCommVerify}_{\mathsf{pk}_{valC_\theta}}(valC_\theta, c_{valC_\theta})$;
  if $insType_\theta = \mathsf{ADD} \wedge \mathsf{ADD}(pc_\theta, valA_\theta, valB_\theta) = (pc_{\theta'}, valC_\theta)$ **then**
    return True;
  else
    return False.

7. Security Analysis

In this section, we show that BitVM is an *on-chain state verification protocol* that satisfies *balance security* and *rational correctness* under our model. Due to space constraints, we only provide proof sketches and defer the full proofs to the extended version of the paper [31], where we model BitVM as an Extensive Form Game (EFG).

Theorem 7.1. BitVM is an on-chain state verification protocol that achieves *balance security* and *rational correctness* as defined in Definition 5.

7.1. Balance Security

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

Setup. If either party deviates during Setup, the honest party will refuse to sign the `Setup` transaction, ensuring no coins are locked unless both parties have received all necessary pre-signed transactions (cf. [31, Lemma D.1]).

Both parties honest. When both parties are honest, BitVM follows an optimistic path: P posts the correct computation result on-chain in `CommitComputation` and, after a timelock Δ, publishes `Close` to distribute funds according to the outcome function f (cf. [31, Lemma D.3]).

V **is honest and** P **is Byzantine.** If P fails to publish `CommitComputation`, because of either being inactive or executing invalid computations, or subsequently fails to post `Close`, V can claim the coins after the relevant timelocks expire (Δ or 2Δ) (cf. [31, Lemmas D.2, D.3]). This mechanism prevents hostage scenarios by enabling V to reclaim funds in case of non-responsiveness.

If P has committed an incorrect computation result in `CommitComputation`, V publishes `KickOff` (Identify Disagreement phase). As shown in (cf. [31, Lemma D.4]), if P is inactive, V can claim the coins after the relevant timelock expires. If the Identify Disagreement phase completes, V knows a step for which P has committed on-chain to an execution of the VM state transition function (Algorithm 5) incorrectly (cf. [31, Lemma D.7]).

P can deviate in the following ways: by using an incorrect program counter (current or next), making an incorrect memory read or write, or using invalid instructions. For each of these deviations, V can post the corresponding transaction on-chain (e.g., `ChallengeCurrPC`, `ChallengeNextPC`, `ChallengeRead`, `ChallengeWrite`, `PunishInstruction`). Following the respective dispute path, V is able to disprove P 's computation and claim the coins, as we conclude in [31, Lemma D.14]).

P is honest and V is Byzantine. V can misbehave by publishing `KickOff` on-chain to initiate the Dispute Phase, although P has committed to the correct output of the execution in `CommitComputation`. We show that if V is inactive during the Identify Disagreement phase, P can claim the coins after the timelock expires (cf. [31, Lemma D.5]) Otherwise, if the Identify Disagreement phase completes, V has to publish one of the transactions `ChallengeCurrPC`, `ChallengeNextPC`, `ChallengeRead`, `ChallengeWrite`, `PunishInstruction`. Since P has committed to the correct values, V cannot disprove P, as we conclude in [31, Lemma D.15]).

7.2. Rational Correctness

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

8. Discussion

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

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

We assume constant transaction fees of $3sat/vB$[11], a Bitcoin price of $64,300$\$ (as of 24 September 2024). For the Lamport signatures, we set $\lambda = 160$, thus public key and commitment are of length $20B$; for an a-bit message they occupy $a \times$ 20B $= a \times$ 5vB. We also assume that $\Delta = 12$ hours, meaning each timelock expires after half a day. Different concrete values can be chosen, but any such selection would require scaling the evaluation accordingly. For details on transaction size calculations, we refer the reader to the extended version [31, Appendix E].

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

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

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

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

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

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

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

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

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[31] “Bitvm: Quasi-turing complete computation on bitcoin (extended version),” https://bitvm-paper.github.io/.

Appendix A. The BitVM Full Protocol: Resolve Dispute

A.1. Resolve Dispute

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

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

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.

A.1.1. Challenging the Current Program Counter. V is claiming that, by publishing `CommitInstruction`, P is committing to a program counter pc_θ at step $\mathcal{N}$ that differs from the program counter $pc_\mathcal{N}$ (previously committed by P during the dispute bisection game). V challenges the current program counter pc_θ by unlocking one of the leaves $\mathsf{ClScriptPCCurr_i}$ (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 $\mathsf{ClScriptPCCurr_0}$, …, $\mathsf{ClScriptPCCurr_{31}}$. When V unlocks leaf $\mathsf{ClScriptPCCurr_i}$, they challenge the program counter of the $(32-i)$-th challenge-response round of the dispute bisection game.

Algorithm 11 The script $\mathsf{ClScriptPCCurr_i}$, for $\mathsf{i} \in \{0,\ldots,31\}$. For each $\mathsf{ClScriptPCCurr_i}$, in the setup phase, we hard-code the public keys pk_{pc_θ}, $\mathsf{pk}_{\mathcal{N}}$. For each $\mathsf{ClScriptPCCurr_i}$, for $\mathsf{i} \in \{1,\ldots,31\}$, we hard-code the same public key pk_{pc_i} hard-coded in $\mathsf{ChallScript_i}$. For $\mathsf{ClScriptPCCurr_0}$, we hard-code the same public key pk_{pc_0} hard-coded in $\mathsf{CommitComputationScript}$.

function $\mathsf{ClScriptPCCurr_i}(\sigma_{PV}, \mathcal{N}, c_{\mathcal{N}}, pc_i, c_{pc_i}, pc_\theta, c_{pc_\theta})$
  $\mathsf{CheckMSigVerify}_{\mathsf{pk}_{PV}}(\sigma_{PV})$;
  $\mathsf{CheckCommVerify}_{\mathsf{pk}_{\mathcal{N}}}(\mathcal{N}, c_{\mathcal{N}})$;
  if $\mathsf{CountZeroes}(\mathcal{N}) \neq \mathsf{i}$ **then**
    ▷ *Maps $\mathcal{N}$ to one of the 32 program counters pc_{n_0}, …, $pc_{n_{31}}$.* ◁
    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 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 $\mathsf{CountZeroes}(N)$
  $counter \leftarrow 0$;
  $flag \leftarrow$ False;
  for $i = 0, \ldots, |N|_{bit} - 1$ **do**
    if $N[|N|_{bit} - i] = 1$ **then**
      $flag \leftarrow$ True;
      ▷ *Set the flag, stop incrementing the counter.* ◁
    else
      if $flag =$ False **then**
        $counter \leftarrow counter + 1$;
  return $counter$.

`ChallengeCurrPC` :=

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

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.

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 $\mathsf{Equivocation}(\mathsf{pk}_{\mathcal{N}[i]}, c_0, c_1) =$ True **then**
      return True;
  return False.

`PunishCurrPC` :=

$$\Big(in = [(\mathsf{ChallengeCurrPC}, 0, \mathsf{ChallPCScript})],\; wit = [(\sigma_{PV}, c_0, c_1)],\; out = [(d\text{₿}; \mathsf{CheckSig}_{\mathsf{pk}_P})]\Big). \tag{12}$$

Challenging the next program counter works analogous, thus deferred to the extended version of this paper [31].

A.1.2. Punish Wrong Instruction. P has committed to a current program counter pc_θ that does not correspond to the correct program instruction, specifically: $\Pi[pc_\theta] \neq (insType_\theta, addrA_\theta, addrB_\theta, addrC_\theta)$.

V spends the `CommitInstruction` output by unlocking the script $\mathsf{ClScriptInstr_j}$ (cf. Algorithm 14) and publishing the `DisproveProgram` transaction (cf. Eq. (13)). A script $\mathsf{ClScriptInstr}$ exists for each of the ℓ instructions in the program Π.

Algorithm 14 The script $\mathsf{ClScriptInstr_j}$, for $\mathsf{j} \in \{1, ..., \ell\}$. In the script $\mathsf{ClScriptInstr_j}$, 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 $\mathsf{ClScriptInstr_j}$.

function $\mathsf{ClScriptInstr_j}(\sigma_{PV}, pc_\theta, c_{pc_\theta}, insType_\theta, c_{insType_\theta}, addrA_\theta, c_{addrA_\theta}, addrB_\theta, c_{addrB_\theta}, addrC_\theta, c_{addrC_\theta})$
  $\mathsf{CheckMSigVerify}_{\mathsf{pk}_{PV}}(\sigma_{PV})$;
  $\mathsf{CheckCommVerify}_{\mathsf{pk}_{pc_\theta}}(pc_\theta, c_{pc_\theta})$;
  $\mathsf{CheckCommVerify}_{\mathsf{pk}_{insType_\theta}}(insType_\theta, c_{insType_\theta})$;
  $\mathsf{CheckCommVerify}_{\mathsf{pk}_{addrA_\theta}}(addrA_\theta, c_{addrA_\theta})$;
  $\mathsf{CheckCommVerify}_{\mathsf{pk}_{addrB_\theta}}(addrB_\theta, c_{addrB_\theta})$;
  $\mathsf{CheckCommVerify}_{\mathsf{pk}_{addrC_\theta}}(addrC_\theta, c_{addrC_\theta})$;
  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.

`DisproveProgram` :=

$$\Big(in = [(\texttt{CommitInstruction}, 0, \mathsf{ClScriptInstr}_{\mathsf{pc}_\theta})],\; wit = [(\sigma_{PV}, pc_\theta, c_{pc_\theta}, insType_\theta, c_{insType_\theta}, addrA_\theta, c_{addrA_\theta}, addrB_\theta, c_{addrB_\theta}, addrC_\theta, c_{addrC_\theta})],\; out = [(d\text{₿}; \mathsf{CheckSig}_{\mathsf{pk}_V})]\Big). \tag{13}$$

A.1.3. Challenge Read. V starts the challenge by publishing the `ChallengeRead` transaction (cf. Eq. (14)), spending the `CommitInstruction` output[12].

$$\texttt{ChallengeRead} := \Big(in = [(\texttt{CommitInstruction}, 0, \mathsf{CheckMSig}_{\mathsf{pk}_{PV}})], \quad wit = [(\sigma_{PV})], \quad out = [(d\text{₿}; \langle \mathsf{ReadChallScript}_1, \mathsf{TL}(\Delta) \wedge \mathsf{CheckSig}_{\mathsf{pk}_V} \rangle)]\Big). \tag{14}$$

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

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

The parties engage in the read bisection game (cf. [31, Section B.2]) . The game is played over the sequence $\mathcal{P}_R := (MR_{\mathcal{N}}, \ldots, M_{\mathcal{N}}[addrA_\theta])$, namely, a path from the root to one of the leaves in $MerkleTree_{M_{\mathcal{N}}}$, i.e., the Merkle tree of the memory at step $\mathcal{N}$. P responds by publishing the $\texttt{ReadResponse}_1$ transaction (cf. Eq. (15)), committing to $\mathsf{Node}_{\mathsf{d}_4} := \mathcal{P}_R[\mathsf{d}_4]$ in the witness, where $\mathsf{d}_4 = 1 \cdot 2^4$.

$$\texttt{ReadResponse}_1 := \Big(in = [(\texttt{ChallengeRead}, 0, \mathsf{ReadChallScript}_1)], \quad wit = [(\sigma_{PV}, \mathsf{Node}_{\mathsf{d}_4}, c_{\mathsf{Node}_{\mathsf{d}_4}})], \quad out = [(d\text{₿}; \langle \mathsf{ReadRespScript}_1, \mathsf{TL}(\Delta) \wedge \mathsf{CheckSig}_{\mathsf{pk}_P} \rangle)]\Big). \tag{15}$$

$\mathsf{ReadRespScript_i}$ 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. (16)), 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.

$$\texttt{ReadChallenge}_1 := \Big(in = [(\texttt{ReadResponse}_1, 0, \mathsf{ReadRespScript}_1)], \quad wit = [(\sigma_{PV}, b'_4, c_{b'_4})], \quad out = [(d\text{₿}; \langle \mathsf{ReadChallScript}_2, \mathsf{TL}(\Delta) \wedge \mathsf{CheckSig}_{\mathsf{pk}_V} \rangle))]\Big); \tag{16}$$

P and V continue playing the read bisection game by publishing transactions $\texttt{ReadResponse}_\texttt{i}$ (cf. Eq. (17) and $\texttt{ReadChallenge}_\texttt{j}$ (cf. Eq. (18)), respectively, with $i = 2, \ldots, 5$ and $j = 1, \ldots, 4$.

$$\texttt{ReadResponse}_\texttt{i} := \Big(in = [(\texttt{ReadChallenge}_{\texttt{i}-1}, 0, \mathsf{ReadChallScript_i})], \quad wit = [(\sigma_{PV}, \mathsf{Node}_{\mathsf{d}_{5-i}}, c_{\mathsf{Node}_{\mathsf{d}_{5-i}}})], \quad out = [(d\text{₿}; \langle \mathsf{ReadRespScript_i}, \mathsf{TL}(\Delta) \wedge \mathsf{CheckSig}_{\mathsf{pk}_P} \rangle)]\Big); \tag{17}$$

$$\texttt{ReadChallenge}_\texttt{j} := \Big(in = [(\texttt{ReadResponse}_\texttt{j}, 0, \mathsf{ReadRespScript_j})], \quad wit = [(\sigma_{PV}, b'_{5-j}, c_{b'_{5-j}})], \quad out = [(d\text{₿}; \langle \mathsf{ReadChallScript}_{\mathsf{j}+1}, \mathsf{TL}(\Delta) \wedge \mathsf{CheckSig}_{\mathsf{pk}_V} \rangle)]\Big); \tag{18}$$

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. (19)). 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 $\mathcal{N}_{Mer} = \sum_{k=0}^{4} b'_k \cdot 2^k$.

$$\texttt{ReadChallenge}_5 := \Big(in = [(\texttt{ReadResponse}_5, 0, \mathsf{ReadRespScript}_5)], \quad wit = [(\sigma_{PV}, b'_0, c_{b'_0})], \quad out = [(d\text{₿}; \langle \mathsf{HashReadScript}_1, \ldots, \mathsf{HashReadScript}_{20}, \mathsf{RootReadScript}_1, \ldots, \mathsf{RootReadScript}_{32}, \mathsf{ValueAScript}, \mathsf{TL}(\Delta) \wedge \mathsf{CheckSig}_{\mathsf{pk}_V} \rangle)]\Big). \tag{19}$$

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. (20)) 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[13]. 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 15. 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.

$$\texttt{CommitRead1} := \Big(in = [(\texttt{ReadChallenge}_5, 0, \mathsf{HashReadScript_i})], \quad wit = [(\sigma_{PV}, \mathcal{N}_{Mer}, c_{\mathcal{N}_{Mer}}, v_{pos}, c_{v_{pos}}, c_{addrA_\theta}, \mathsf{Nsib}, \mathsf{Npar}, c_{\mathsf{Npar}}, \mathsf{Nchild}, c_{\mathsf{Nchild}})], \quad out = [(d\text{₿}; \langle \mathsf{CommitRead1Script}, \mathsf{TL}(\Delta) \wedge \mathsf{CheckSig}_{\mathsf{pk}_P} \rangle)]\Big). \tag{20}$$

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

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

Algorithm 15 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.

V can punish P if they equivocate either on Npar, Nchild, v_{pos} by publishing `PunishRead1` (cf. Eq. (21)), which requires to unlock the CommitRead1Script (cf. Algorithm 16) script.

Algorithm 16 The script 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.

$$\begin{aligned}&\texttt{PunishRead1} := \\ &\Big(in = [(\texttt{CommitRead1}, 0, \mathsf{CommitRead1Script})], \\ &\quad wit = [(\sigma_{PV}, c_0, c_1)], out = [(d฿; \mathsf{CheckSig}_{\mathsf{pk}_V}]\Big).\end{aligned} \tag{21}$$

(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 ValueAScript. 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.

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 17.

Algorithm 17 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. (22)).

$$\texttt{CommitRead3} := \Big(in = [(\texttt{ReadChallenge}_5, 0, \mathsf{ReadRootScript_i})], \\ wit = [(\sigma_{PV}, \mathcal{N}_{Mer}, c_{\mathcal{N}_{Mer}}, \mathcal{N}, c_{\mathcal{N}}, \mathsf{Node_{d_0}}, c_{\mathsf{Node_{d_0}}})], \\ out = [(d\text{B}; \langle \mathsf{CommitRead3Script}, \mathsf{TL}(\Delta) \wedge \mathsf{CheckSig}_{\mathsf{pk}_P} \rangle)]\Big). \tag{22}$$

V can punish P if they equivocate on $\mathsf{Node_{d_0}}$, MR_i or $addrA_\theta$ by publishing PunishRead3 (cf. Eq. (23)), which unlocks CommitRead3Script, analogous to CommitRead1Script but with pk_{MR_i}, $\mathsf{pk}_{\mathsf{Node_{d_0}}}$ instead of $\mathsf{pk_{Npar}}$, $\mathsf{pk_{Nchild}}$.

$$\texttt{PunishRead3} := \Big(in = [(\texttt{CommitRead3}, 0, \mathsf{CommitRead3Script})], \\ wit = [(\sigma_{PV}, c_0, c_1)], out = [(d\text{B}; \mathsf{CheckSig}_{\mathsf{pk}_V}]\Big). \tag{23}$$

A.1.4. 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[14]. As a result, the memory root $MR_{\mathcal{N}'}$ is incorrect.

The parties engage in the write bisection game (cf. [31, Section B.3]) over the sequences $\mathcal{P}_W := (MR_{\mathcal{N}}, \dots, M_{\mathcal{N}}[addrC_\theta])$ and $\mathcal{P}'_W := (MR_{\mathcal{N}'}, \dots, M'_{\mathcal{N}}[addrC_\theta])$, 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 $\{(\mathsf{Node_{d_4}}, \mathsf{Node'_{d_4}}), \dots, (\mathsf{Node_{d_0}}, \mathsf{Node'_{d_0}})\}$, where $\mathsf{Node_{d_4}}, \dots, \mathsf{Node_{d_0}} \in \mathcal{P}_W$ and $\mathsf{Node'_{d_4}}, \dots, \mathsf{Node'_{d_0}} \in \mathcal{P}'_W$. Analogous to the *Challenge Read* branch, V commits bit by bit to an integer $\mathcal{N}_{Mer} = \sum_{k=0}^{4} b'_k \cdot 2^k$, which conditions how P can unlock $\texttt{WriteChallenge}_5$. 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 $\mathsf{HashWriteScript_1}$, ..., $\mathsf{HashWriteScript_{20}}$ via publishing the CommitWrite1 transaction (cf. Eq. (24)). Each script $\mathsf{HashWriteScript_i}$ is identical to script $\mathsf{HashReadScript_i}$, for $\mathsf{i} = 1, \dots, 20$, except that it also verifies the commitments of the parent and child nodes on the path $\mathcal{P}'_W$ (their public key are hard-coded in the script accordingly) and it checks whether they really are a parent-child pair. We present script $\mathsf{HashWriteScript_1}$ in the extended version of this paper [31].

$$\texttt{CommitWrite1} := \Big(in = [(\texttt{WriteChallenge}_5, 0, \mathsf{HashWriteScript_i})], \\ wit = [(\sigma_{PV}, \mathcal{N}_{Mer}, c_{\mathcal{N}_{Mer}}, v_{pos}, c_{v_{pos}}, c_{addrC_\theta}, \mathsf{Nsib}, \\ \mathsf{Npar}, c_{\mathsf{Npar}}, \mathsf{Nchild}, c_{\mathsf{Nchild}}, \mathsf{Npar}', c_{\mathsf{Npar}'}, \mathsf{Nchild}', \\ c_{\mathsf{Nchild}'})], \\ out = [(d\text{B}; \langle \mathsf{CommitWrite1Script}, \mathsf{TL}(\Delta) \wedge \mathsf{CheckSig}_{\mathsf{pk}_P} \rangle)]\Big); \tag{24}$$

The script CommitWrite1Script is identical to CommitRead1Script (cf. Algorithm 16) 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 $\mathsf{Node_{d_0}}$ and $valC_\theta$ or between $\mathsf{Node'_{d_0}}$ and $valC_\theta$. This case is analogous to the "commit value A" case of the *Challenge Read* branch. For ValueCScript, the difference with $\mathsf{HashWriteScript_i}$, is that: (i) CountZero = 0; (ii) the parent nodes are $\mathsf{Node_{d_0}}$ and $\mathsf{Node'_{d_0}}$, and (iii) the child node is $valC_\theta$.

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

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

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