Bridging Bitcoin to Second Layers via BITVM2

***Abstract*—A holy grail in blockchain infrastructure is a trustless bridge between Bitcoin and its second layers or other chains. We make progress toward this vision by introducing the first light-client based Bitcoin bridge. At its core lies BITVM2-CORE, a novel paradigm that enables arbitrary program execution on Bitcoin, combining Turing-complete expressiveness with the security of Bitcoin consensus. BITVM2-BRIDGE advances prior approaches by reducing the safety assumption from an honest majority (t-of-n) to existential honesty (1-of-n) during setup. Liveness is guaranteed with only one rational operator, and any user can act as a challenger, enabling permissionless verification. A production-level implementation of BITVM2 has been developed and a full challenge verification has been executed on the Bitcoin mainnet.**

1. Introduction

Bitcoin, the largest cryptocurrency by market cap, has two major drawbacks: (i) a *low throughput* of about 5-7 transactions per second, limiting its scalability; and (ii) a *limited scripting language* (Bitcoin Script). While the latter significantly reduces the attack surface, it also restricts expressiveness, making Bitcoin ill-suited for decentralized applications such as DeFi. As a result, these applications typically run on blockchains with smart contract capabilities like Ethereum.

Limitations of the base blockchain (L1) can often be mitigated by adding second layers (L2s). For example, Bitcoin scalability has been addressed by L2 protocols such as payment channels [28]. Payment channels enable the Bitcoin asset to be transacted off the Bitcoin chain and settled back on Bitcoin in a completely trustless manner. However, payment channels serve to scale one application: payments, Bitcoin's original application. To enable Bitcoin to support a broad range of other applications such as DeFi, Bitcoin needs L2s with general-purpose capabilities. To that end, the holy grail is a *bridge* that allows Bitcoin assets to be trustlessly transferred back and forth between such an L2 and the Bitcoin chain, where a wrapped version of the Bitcoin asset is used on the L2.

Trustless bridges between other blockchains are typically realized through *light clients* on the destination chain, which check if a certain transaction happened in the source chain before releasing tokens in the destination one. A good example is the Inter-blockchain Communication protocol (IBC) used extensively in the Cosmos ecosystem. Such light clients are typically implemented as smart contracts on the chains to be bridged. Since Bitcoin does not support smart contracts, existing Bitcoin bridges typically rely on multi- or threshold signature schemes, where a group of t-of-n signers is entrusted with safekeeping BTC. If these signers collude or are bribed, user deposits are lost.

Honesty Requirement	BitVM2 bridge	MultiSig bridge
Peg-in liveness	n-of-n signers 1-of-m operators	majority
Peg-in safety	1-of-n signers	majority
Peg-out liveness	1-of-m operators	majority
Peg-out safety	1-of-n signers 1 challenger	majority
Operator safety	1-of-n signers 1 challenger	N/A

Figure 1. Honest requirements of the BITVM2-BRIDGE as compared to a multi-signature bridge. Peg-in refers to the transfer of bitcoins from the main chain to the L2. Peg-out refers to the transfer back to the main chain. Precise definitions of security properties are found in Section 2.6.3. Security analysis is given in Section 6.

In this paper, we present the first light-client based Bitcoin bridge, the BITVM2-BRIDGE. The BITVM2-BRIDGE is run by n permissioned signers, m permissioned operators and an arbitrary number of *permissionless* challengers which watch over the operators. The honesty requirement of the BITVM2-BRIDGE is shown in Fig. 1 and is compared to that of a standard multi-signature bridge. The main security improvement over a multi-signature bridge is in the peg-out direction, where peg-out liveness and safety requires only existential honesty in the signers and the operators and challengers, as opposed to majority honesty in the case of a multi-signature bridge.

The signers act as an emulation of a *covenant*, a set of pre-signed transactions forming a contract on how and to whom the locked pegged-in funds can be released in a peg-out. Unlike in a multi-signature bridge, the signers only need to be present at peg-in, and once the transactions are pre-signed they can delete their keys. The peg-out process is achieved in a trust-minimizing way through the light client. A user first burns the

wrapped asset on the L2. Then one of the operators fronts the bitcoins to that user, and using a proof of burn, the operator claims the locked funds that were pegged in. The on-chain light client is used to verify the correctness of the proof.

The core of the light client is BITVM2-CORE, the first permissionless protocol to compute arbitrary functions in Bitcoin, which we consider of independent interest. In particular, BITVM2-CORE allows an operator to assert that a Bitcoin Script program g executed on input x returns $y = g(x)$. The operator stakes a deposit of d BTC, where $d \geqslant 0$, and posts a commitment to this claim on-chain. If the claim is correct, they reclaim the deposit. Otherwise, any party can challenge the claim by executing a portion of the program on-chain and disproving the claim, preventing the operator from reclaiming their deposit. This is achieved by encoding a SNARK verifier in a sequence of Bitcoin transactions.

A production-level reference implementation of BITVM2-CORE has been developed. An experiment on Bitcoin mainnet shows that a worst-case ("unhappy-path") dispute settles in under 8 hours and less than 0.15 BTC in fees, while 90 % of Disprove scripts remain below 3 MB. A careful incentive design ensures that the unhappy-path is never triggered by rational parties, since operators would lose the collateral in case of misbehavior and challengers have to pay a fee in order to trigger a dispute. The happy-path, which is the one normally executed, involves transactions that remain below 16.75 vkB, which is around 50k sat (roughly $ 53). We discuss how to further reduce this cost to 534 vbytes, or around 1.6k sats (around $ 1.66) in Section 8.9 This confirms BitVM2's practicality in real-world scenarios.

Related Work. A number of works approach the problem of bridging assets between different chains with minimal trust. Similarly to our work, zkBridge [33] leverages zkSNARKs to prove to one chain the state of another chain, thus achieving trustless bridging. Unfortunately, verifying zkSNARKs directly on Bitcoin is impossible due to its maximum block size, so zkBridge is incompatible with Bitcoin.

In parallel, several approaches to achieving arbitrary computation on Bitcoin have been proposed: BitVM [6], the predecessor of the current work, encodes arbitrary computation in a challenge-based protocol which puts needs 3 transactions on chain in the optimistic case. In the case of dispute, emphbisection game requires 79 sequential on-chain transactions, for a cost of approximately 0.007 BTC. In comparison, BITVM2-CORE requires less transactions, albeit larger. The main difference, however, is that BitVM is applicable in a permissioned, two-party setting, where prover and verifier are fixed in advance and the verifier has to be online to punish the prover in case of misbehavior: this hinders a number of use cases, such as the bridge here proposed.

Clementine [8] builds on the BitVM core described in the present work to provide a different efficiency-trust tradeoff than our bridge. In particular, Clementine assumes existential honesty among a fixed set of watchtowers, an assumption not needed in our work.

BitVMX [23] achieves a lower cost in terms of transactions compared to BitVM in exchange for requiring a permissioned set of operators. TOOP [12] extends BitVMX by moving funds to an address controlled by all active operators at the time of peg-out and having them send their private keys to the peg-out party. Unfortunately its practicality is questionable as it needs off-chain communication exponential in the number of operators in the worst case.

ColliderVM [20] focuses on providing stateful computation on Bitcoin. It improves on BitVM by (i) permitting permissionless operators and by (ii) replacing the bisection game with executing stateful computation directly on-chain. Its main limitations are (i) the high energy requirements needed to find hash collisions and (ii) its need to trust a permissioned set of parties both for liveness and for safety as in BitVM.

The BITVM2-BRIDGE relies on a group of signers to emulate *covenants*: constraints on when UTXOs are spendable that are inherited by descendant UTXOs. True covenants can remove our reliance on existentially honest signers. Unfortunately, covenants are not available in today's Bitcoin Script. As exemplified by [9], the few additional opcodes available in the Bitcoin fork BSV are enough to express covenants. Furthermore, PIPE [21] describes a way to simulate covenants on Bitcoin without a consensus change.

2. Model and Protocol Overview

In this section, we formalize the model and assumptions underpinning our protocol suite and provide a high-level overview of the BITVM2-BRIDGE. Our model captures the behavior and guarantees of two ledgers, the interacting parties (clients, operators, signer committee, and challengers), and the underlying cryptographic primitives.

While our main application is a trust-minimizing Bitcoin bridge, two foundational components developed in this work are of independent interest: (1) the BITVM2-CORE protocol, which enables on-chain verification of arbitrary computations on Bitcoin, and (2) an on-chain Bitcoin light client built atop BITVM2-CORE. Their correctness and security properties form essential building blocks and are proven separately in

our analysis as a prerequisite to establishing the security of the bridge.

Notation. We use ϵ and $[\,]$ to mean the empty string and empty sequence. By $x \,\|\, y$, we mean the string concatenation of x and y encoded in a way that x and y can be unambiguously retrieved. We denote by $|\mathcal{C}|$ the length of the sequence $\mathcal{C}$; by $\mathcal{C}[i]$ the i^{th} (zero-based) element of the sequence, and by $\mathcal{C}[-i]$ the i^{th} element from the end. We use $\mathcal{C}[i{:}j]$ to mean the subarray of $\mathcal{C}$ from the i^{th} element (inclusive) to the j^{th} element (exclusive). Omitting i takes the sequence to the beginning, and omitting j takes the sequence to the end. Given two sequences $\mathcal{C}_1, \mathcal{C}_2$, we use $\mathcal{C}_1 \preceq \mathcal{C}_2$ to mean that $\mathcal{C}_1$ is a prefix of $\mathcal{C}_2$. Given a set A, we denote its power set by $\wp(A)$.

2.1. Protocol Execution Model

Let Π be a protocol that runs between n parties belonging to the set $\mathcal{P} := \{P_1, \ldots, P_n\}$[1], in the presence of an adversary $\mathcal{A}$, a special party that can corrupt up to t other parties. Corrupting a party P_i means learning their internal state and taking full control over them from the moment of corruption until the end of protocol execution. The execution of protocol Π is driven by a special program $\mathcal{Z}$ called *environment* that spawns parties $P_1, \ldots, P_n$ and the adversary $\mathcal{A}$. All of the aforementioned programs can be thought of as interactive Turing machines (ITM). We denote as ITM instance (ITI) the actual execution of an ITM running a certain program. We use the notation $\{VIEW^{P_i,r,t,n}_{\Pi,\mathcal{A},\mathcal{Z}}(z)\}_{z\in\{0,1\}*}$ to denote the view of party P_i at round r of the execution of protocol Π run by environment $\mathcal{Z}$ with respect to an adversary $\mathcal{A}$.

2.2. Ledger and Network Model

We consider two ledgers, denoted $\mathcal{L}_A$ and $\mathcal{L}_B$, operated by respective ledger protocols $\Pi_{\mathcal{L}_A}$ and $\Pi_{\mathcal{L}_B}$. In our main application, $\mathcal{L}_A$ corresponds to Bitcoin, and $\mathcal{L}_B$ to a second-layer or alternative chain. Each ledger protocol maintains a sequence of confirmed transactions visible to participants known as full nodes. We model the ledger as a function ${}^{r}\mathcal{L}^{P}_{X}$, denoting the view of party P of ledger $\mathcal{L}_X$ at round r.

Definition 2.1 (Safety)**.** A distributed ledger protocol is *safe* if: (i) For any honest party P and any rounds $r_1 \leqslant r_2$, it holds that ${}^{r_1}\mathcal{L}^{P} \preceq {}^{r_2}\mathcal{L}^{P}$ (*Self-Consistency*); and (ii) For any honest parties P_1, P_2 and any round r, it holds that either ${}^{r}\mathcal{L}^{P_1} \preceq {}^{r}\mathcal{L}^{P_2}$ or ${}^{r}\mathcal{L}^{P_2} \preceq {}^{r}\mathcal{L}^{P_1}$ (*View-Consistency*).

1. The number n of parties that participate in the execution of protocol Π is fixed, but possibly unknown.

Definition 2.2 (Liveness)**.** A distributed ledger protocol execution, in a synchronous network, is *live*(u) if any transaction written to any honest party ledger at round r, appears in the ledger of all honest parties by round $r + u$, denoted as ${}^{r+u}\mathcal{L}^{\cap}$.

We assume a global round-based clock, shared across all parties and ledgers. Rounds abstract time into discrete steps. We assume a synchronous network, i.e., all messages broadcast to the network will be delivered to all parties within a known time bound Δ. Ledgers and blockchains are defined as in the Bitcoin backbone model [13], which in particular operates in the q-bounded setting, used in the formal light client analysis (cf. Appendix E).

2.3. UTXO Model Overview

We consider blockchains based on the *unspent transaction output* (UTXO) model, such as Bitcoin. In this model, coins exist in outputs of transactions and are spendable by satisfying predefined spending conditions expressed in a stack-based scripting language. Each output holds a coin value and a *locking script*, and can be spent by providing a valid *witness* that satisfies the script. Transactions consume existing outputs as inputs and create new outputs, preserving value.

We rely on standard spending conditions used in Bitcoin, such as signature locks ($\mathsf{CheckSig}$), multisignature locks ($\mathsf{CheckMultiSig}$), time-based conditions ($\mathsf{timelocks}$), and Taproot trees that enable complex, private policies (denoted $\langle \cdots \rangle$). For further details and additional primitives (e.g., SIGHASH flags), we refer to Section A.4.

To illustrate the notation used throughout this work for transactions and their associated spending conditions, consider the following example. Let tx be a transaction that consumes outputs from two previous transactions, $\mathsf{tx1}$ and an arbitrary transaction $*$, and creates two new outputs:

$$
\begin{aligned}
&\mathsf{tx} := \\
&\Big(\mathsf{in} = [(\mathsf{tx1}, 0, \mathsf{CheckSig}_{\mathsf{pk}_A} \wedge \mathsf{RelTimelock}(10)),\ *], \\
&\quad \mathsf{wit} = [\sigma_A,\ *], \\
&\quad \mathsf{out} = [(4,\ \mathsf{AbsTimelock}(500)),\ (2,\ \mathsf{CheckSig}_{\mathsf{pk}_B})]\Big)
\end{aligned}
$$

Here, the transaction tx spends (i) the first output of $\mathsf{tx1}$ which requires a signature from user A and is locked under a relative timelock of 10 blocks, and (ii) an additional output denoted by $*$. We use $*$ to denote valid inputs and witnesses that carry coins needed for the protocol but have logic irrelevant to the protocol. The witness for the first input includes σ_A, authorizing the spend. The transaction creates two outputs: one

holding 4 BTC, locked until block height 500, and another holding 2 BTC, spendable by user B.

2.4. Parties, System, and Threat Model

We consider a bridge protocol that enables the movement of bitcoins, denoted BTC, between the Bitcoin blockchain ($\mathcal{L}_A$) and a secondary ledger ($\mathcal{L}_B$), where wrapped coins are denoted wBTC. The protocol supports two primary operations:

Peg-in : A user (Alice) locks u units of BTC on $\mathcal{L}_A$. These are then minted as wrapped BTC (u wBTC) on $\mathcal{L}_B$.

Peg-out : A user (Bob) initiates a Peg-out by burning u wBTC on $\mathcal{L}_B$ and claiming an equivalent amount of BTC on $\mathcal{L}_A$. The burned u wBTC originated from Alice's Peg-in, possibly through a sequence of transfers.

Besides the *users* of the bridge that perform the Peg-in and Peg-out operations (i.e., Alice and Bob) the bridge protocol involves the following roles:

Signer Committee ($S_1, \ldots, S_n$) : A set of n signers is responsible for setting up the bridge by generating ephemeral keys used to authorize minting and to define valid burn conditions for wBTC. We assume the committee is honest during setup and that at least *one signer honestly deletes their key* during setup.

Operators ($O_1, \ldots, O_m$) : Operators execute the bridge logic by monitoring both ledgers and facilitating Peg-outs. Upon detecting a valid burn of wBTC, an operator pays the user on $\mathcal{L}_A$ using its own funds, and is later reimbursed by claiming the pegged-in BTC. We assume that *at least one operator is honest* during execution; the rest may be Byzantine.

Challengers : Challengers monitor $\mathcal{L}_A$ and ensure the security of Peg-out operations by disputing invalid claims by (malicious) Operators. The role is *permissionless*; any party may act as a challenger, including signers, operators, and users. At least *one honest challenger is assumed to exist at all times*, while everyone else can be Byzantine.

Our construction is *trust-minimizing*: it relies only on *existential honesty* for signers, operators, and challengers. Notably, the challenger role is *fully permissionless*, allowing any party to participate in dispute resolution. This stands in contrast to prior bridge protocols, which typically require an *honest majority* assumption and often restrict the challenger role to a permissioned set. We further note that on platforms supporting covenants, both signers and operators become redundant, further reducing trust assumptions.

All participants are computationally bounded. We assume cryptographically secure communication channels, hash functions, existentially unforgeable digital signature schemes, and succinct non-interactive arguments (SNARGs) [2]. Specifically, we rely on EUF-CMA secure signature schemes, such as Schnorr and Lamport signatures, as well as SNARGs with perfect completeness and computational soundness (see Appendix A for their definition).

2.5. BitVM2-bridge Protocol Overview

BitVM2-bridge implements a trust-minimizing bridge protocol that enables secure coin transfer between Bitcoin and another ledger (e.g., a Layer 2 protocol) through an optimistic design. The protocol allows arbitrary off-chain computations to be validated on Bitcoin via fraud proofs. At its core is a mechanism called BitVM2-core, which expresses and enforces computation using Bitcoin Script.

Setup. The protocol begins when a user, Alice, initiates a peg-in by creating (but not yet posting) a transaction, PegIn, that locks BTC on Bitcoin (ledger $\mathcal{L}_A$). Alice sends the peg-in request to the signer committee, which in response, generates ephemeral keys and collaborates with a set of designated operators to prepare *presigned* Bitcoin transactions. These transactions are designed to enforce correct behavior at Peg-out. Once setup completes and Alice receives the necessary presigned transactions, she posts PegIn on $\mathcal{L}_A$. Subsequently, she is credited the equivalent amount of wrapped BTC (denoted wBTC) on an external ledger ($\mathcal{L}_B$). This step is done trustlessly with the use of a light client [7] or atomic swap protocol [18], [30].

Transfer and Payment. Once minted, the wrapped BTC can be transferred across users on $\mathcal{L}_B$ as regular coins. These coins may change hands arbitrarily[3], and any user holding wBTC can later initiate a peg-out back to Bitcoin.

Peg-out. Suppose Bob becomes the owner of the wrapped BTC and decides to bridge the funds back to $\mathcal{L}_A$. To do this, Bob burns the wrapped BTC on $\mathcal{L}_B$ and requests a peg-out. An honest operator, observing the burn, transfers an equivalent amount of BTC to Bob on $\mathcal{L}_A$ using their own liquidity. This concludes Bob's role in the protocol: he has been paid back on Bitcoin.

2. Since we make implemented a SNARK verifier in Bitcoin script, sometimes we will refer to SNARKs. However, note that wrt to security, we are only interested in the weaker security notion of SNARGs.

3. Before accepting the coin on $\mathcal{L}_B$, the recipient should check that $\mathcal{L}_A$ contains a corresponding PegIn transaction.

Operator Reimbursement and Dispute Resolution. The operator must now reclaim the BTC they provided to Bob. This is done by publishing a presigned Claim transaction on $\mathcal{L}_A$, which triggers a dispute window. If no one challenges the Claim during the dispute window, the operator thereafter claims the PegIn coins. However, if the Claim is fraudulent, i.e., if Bob never burned the wrapped BTC or the operator failed to pay him, an honest challenger will submit a dispute (Challenge transaction) to prevent the reimbursement.

To justify the claim, the operator may then publish a presigned Assert transaction, committing to an argument that (i) a valid burn occurred on $\mathcal{L}_B$ and (ii) a corresponding payment by the operator occurred on $\mathcal{L}_A$. The Assert triggers a second dispute period. If the commitment made in Assert is incorrect, an honest challenger publishes a Disprove transaction during the dispute period. A valid Disprove will prevent the operator from taking the PegIn coins.

If no successful Dispute is published on-chain before the deadline, the Assert is assumed correct and the PegIn coins are awarded to the operator. If the challenge is malicious, they cannot construct a valid Disprove and thereby the operator will reclaim the PegIn coins.

Implementing Assert in Bitcoin Script. At the heart of our construction lies an interactive protocol, termed BITVM2-CORE, which is realized using Bitcoin Script. BITVM2-CORE enables honest operators to reclaim their funds (via Assert), while preventing malicious operators from withdrawing funds from the bridge (via Disprove). *But how can* Assert *express such a complex statement on Bitcoin Script?*

The key idea is to use Assert to commit to the output of a SNARK verifier attesting that the burn and payment indeed occurred. The core technical challenge lies in expressing the SNARK verifier logic on Bitcoin Script, which is inherently constrained. To overcome this, we observe that Bitcoin Script, via opcodes such as OP_ADD, can express arbitrary total functions, i.e., functions that terminate for all inputs. Based on this observation, we implement the verifier logic as a SNARK verifier circuit and compile it to Bitcoin Script. Since a full SNARK verifier typically exceeds Bitcoin's block size (e.g., 900MB for Groth16), we decompose it into sequential chunks, each small enough to fit in a Bitcoin transaction (max 4MB). The Assert transaction commits to each chunk's code and output. During a dispute, a Disprove transaction reveals the true output of a specific chunk by re-executing the chunk on-chain and comparing it against the operator's claim. This enables a full on-chain fraud proof mechanism without any changes to Bitcoin. To ensure that Assert encodes the desired program, it is presigned by the signer committee for each operator during setup.

A high-level illustration of the BITVM2-CORE protocol architecture and challenge flow is shown in Fig. 2.

Verifying transaction inclusion in $\mathcal{L}_B$. In the current setting, BITVM2-CORE must verify two key events: the inclusion of the peg-out transaction on Bitcoin (where the operator pays Bob), and the inclusion of Bob's burn transaction on $\mathcal{L}_B$. To simplify presentation, we assume that $\mathcal{L}_A$ (Bitcoin) includes state commitments of $\mathcal{L}_B$. This reduces the task of verifying transaction inclusion on $\mathcal{L}_B$ to verifying a Bitcoin transaction against a known state commitment; thereby enabling both verifications to be handled uniformly within Bitcoin.

To realize this, we present, to our knowledge, the first *on-chain light client for Bitcoin*, built using BITVM2-CORE. In the remainder of this section, we introduce the construction in stages: we begin by formalizing the protocol goals of BITVM2-CORE, then outline the requirements for the light client protocol, and finally define the desiderata for the full BITVM2-BRIDGE system.

2.6. Definitions and Properties

In the following, we expose the formal protocol goals for the BITVM2-CORE, the on-chain light client for Bitcoin, and the BITVM2-BRIDGE.

2.6.1. BITVM2-CORE security properties.

BITVM2-CORE is a protocol run between a prover P (in BITVM2-BRIDGE, the operator) and a set of verifiers V (in BITVM2-BRIDGE, the challengers), which participate in a distributed ledger protocol $\Pi_{\mathcal{L}}$ as full nodes. The protocol enables P to prove on-chain the outcome of a known function f.

The prover is entitled to publish a specific transaction conditioned on successful verification of the computation's result. For example, P may lock a deposit to qualify as a service provider for the computation of f in exchange for a service fee. In such a case, P can only reclaim their funds upon V verifying the result of the computation f.

At a high level, BITVM2-CORE satisfies two key properties: *completeness*, which means that an honest prover that knows the input of the correct execution (i.e., a valid witness) can always publish a prespecified transaction on-chain (e.g., the reclaim of the deposit), and *soundness*, that is, a malicious prover without a valid witness cannot publish such transaction.

Definition 2.3 (On-chain State Verification Protocol)**.** A BITVM2-CORE proof system is a protocol between a prover P and a set of verifiers V, where $P, V \subseteq \mathcal{P}_f$ are full nodes of a ledger protocol $\Pi_{\mathcal{L}}(\cdot)$. BITVM2-CORE

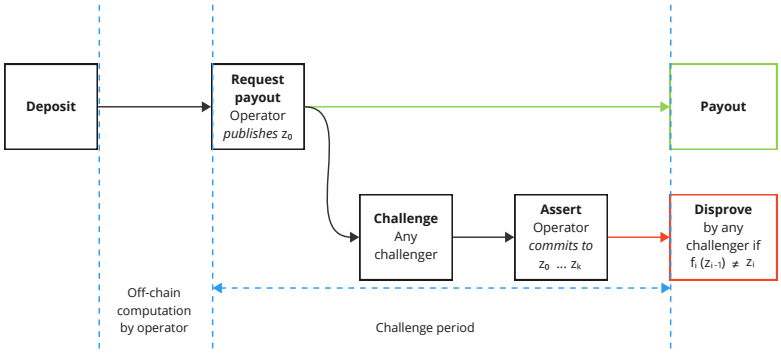

Figure 2. Simplified overview of the BITVM2-CORE transaction flow. The SNARK verifier program $f(x) = y$ is split into $f_1, f_2, ..., f_k$ with intermediary results $z_0, ..., z_k$. A challenged operator must reveal the intermediary states in the Assert transaction. This allows a challenger to disprove a false claim of an operator by executing the disputed sub-program f_i in a Bitcoin transaction.

consists of a tuple of procedures $(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$ defined as follows:

- $\mathsf{Setup}(1^\lambda, \hat{\mathcal{R}}_f) \rightarrow (\mathcal{T}, \mathcal{S})$: Given a security parameter 1^λ and an NP-relation $\hat{\mathcal{R}}_f$ defined over f[4], output two sets $\mathcal{T}$ and $\mathcal{S} \subseteq \mathcal{T}$. Informally, $\mathcal{T}$ encodes the possible execution paths restricted at setup, while $\mathcal{S}$ encodes the payout transactions, in which the prover claims the deposit. For an honest prover, at least one transaction of the set $\mathcal{S}$ will appear.
- $\mathsf{Prove}(\hat{\mathcal{R}}_f, Y, w) \rightarrow \pi$: Given a statement Y and a witness w, output a proof π that $(Y, w) \in \hat{\mathcal{R}}_f$.[5]
- $\mathsf{Verify}(\hat{\mathcal{R}}_f, Y, \pi) \rightarrow \{\mathsf{True}, \mathsf{False}\}$: Given the proof π for the statement Y, output True if π is valid, otherwise output False.

An on-chain state verification protocol is called *robust* if it satisfies the properties of completeness and soundness stated below. Completeness expresses that an honest operator will be able to reclaim its deposit. In particular, assume that the setup is done correctly and an honest prover P knows a witness w s.t. $(Y, w) \in \hat{\mathcal{R}}_f$ at a round r. Then, with overwhelming probability, at least one transaction where the prover reclaims the deposit (included in $\mathcal{S}$) will appear on the ledger ${}^{r'}\mathcal{L}^{\cap}$ where $r' \geqslant r + u_c$. On the other hand, soundness guarantees that a malicious prover P that does not know a valid witness w s.t. $(Y, w) \in \hat{\mathcal{R}}_f$, cannot reclaim the deposit. To this end, none of the transactions in $\mathcal{S}$ will appear on the ledger ${}^{r}\mathcal{L}^{\cap}$ for any round r, except with a negligible probability.

4. The relation induced by the function f will be defined when instantiating a protocol that implements the BITVM2-CORE interface.

5. The proof in our system is essentially the witness itself, but for capturing more generic cases where, for example, the proof hides the witness, we keep the definition more general.

Definition 2.4 (Completeness)**.** Parameterized by u_c, a BITVM2-CORE proof system is *complete* if, for all f and $\hat{\mathcal{R}}_f$, for all $Y \in \mathcal{L}_{\hat{\mathcal{R}}_f}$, for an honest prover P, and any execution of the ledger protocol $\Pi_{\mathcal{L}}$ the following holds. For any round r after the Setup, and every round $r' \geqslant r + u_c$:

$$\Pr[\exists\, tx \in \mathcal{S} : tx \in {}^{r'}\mathcal{L}^{\cap} \mid \mathsf{Cond1}] \geqslant 1 - \mathsf{negl}(\lambda) \quad (1)$$

where Cond1 denotes:

$$(\mathcal{T}, \mathcal{S}) \leftarrow \mathsf{Setup}(1^\lambda, \hat{\mathcal{R}}_f);$$
$$w \in \{VIEW^{P,r,t,n}_{\Pi_{\mathcal{L}},\mathcal{A},\mathcal{Z}}(z)\}_{z\in\{0,1\}^*} \wedge$$
$$\mathsf{Verify}(\hat{\mathcal{R}}_f, Y, \mathsf{Prove}(\hat{\mathcal{R}}_f, Y, w)) = \mathsf{True}$$

Definition 2.5 (Soundness)**.** A BITVM2-CORE proof system is *sound* if, for all PPT $\mathcal{A}$ in the q-bounded setting, for any execution of the ledger protocol $\Pi_{\mathcal{L}}$, for all f and $\hat{\mathcal{R}}_f$, and for all $Y \in \mathcal{L}_{\hat{\mathcal{R}}_f}$, it holds that for any round r after the Setup:

$$\Pr[\exists\, tx \in \mathcal{S} : tx \in {}^{r}\mathcal{L}^{\cap} \mid \mathsf{Cond2}] \leqslant \mathsf{negl}(\lambda) \quad (2)$$

where Cond2 denotes:

$$(\mathcal{T}, \mathcal{S}) \leftarrow \mathsf{Setup}(1^\lambda, \hat{\mathcal{R}}_f);$$
$$w \notin \{VIEW^{P,r,t,n}_{\Pi_{\mathcal{L}},\mathcal{A},\mathcal{Z}}(z)\}_{z\in\{0,1\}^*} \wedge$$
$$\mathsf{Verify}(\hat{\mathcal{R}}_f, Y, \mathsf{Prove}(\hat{\mathcal{R}}_f, Y, w)) = \mathsf{True}$$

2.6.2. Light client security properties. We now present the security definition of the light client protocol based on BITVM2-CORE. Intuitively, the light client is secure if, upon termination, it outputs a block that is both guaranteed to appear in the future chains of all honest parties (safety) and recent enough to capture progress in the honest network (liveness).

Definition 2.6 ((u,k)-Admissible Block at r)**.** Consider $u, k \in \mathbb{N}$ (liveness and safety parameter, respectively, of a ledger protocol). A block B that, at round r, fulfils the following properties is an *admissible block at* r:

Safety : $\mathsf{B} \in \mathcal{C}^{\bigcup}_{r+u}[:-k]$
Liveness : $\mathsf{B} \notin \mathcal{C}^{\bigcap}_{r}[:-k]$

Definition 2.7 (Chain Client Security)**.** An interactive Prover-Verifier protocol $\Pi(\mathcal{P}, V)$ is *secure* if, when the protocol terminates at r^*, V outputs a block B that is *admissible* at $r \leqslant r^*$.

2.6.3. BITVM2-BRIDGE security properties. We formalize a bridge protocol that operates atop two ledger protocols, $\Pi_{\mathcal{L}_A}$ and $\Pi_{\mathcal{L}_B}$. We denote an execution instance of this protocol as I. A bridge conditions the execution of one or more transactions on $\mathcal{L}_B$ upon the inclusion of one or more transactions on $\mathcal{L}_A$, which we refer to as *cross-chain request*. We assume a global clock governs both $\Pi_{\mathcal{L}_A}$ and $\Pi_{\mathcal{L}_B}$ [6].

To consistently track assets across chains, we associate each Peg-In event with a unique identifier $\mathtt{id}$. This identifier links all transactions, across both ledgers, that originate from the same Peg-In, allowing the bridge to reason about asset continuity and cross-chain state. Intuitively, $\mathtt{id}$ serves as a reference to the coins created on $\mathcal{L}_B$ as a result of locking assets on $\mathcal{L}_A$. We now formalize the bridge using $\mathtt{id}$ to identify related transactions across ledgers, and a relation $R(\mathtt{id})$ to specify valid cross-chain dependencies. Recall that $\wp(S)$ denotes the power set of set S.

Definition 2.8 (Unidirectional Bridge)**.** Let $X \in \{A, B\}$ and $V_{\mathcal{L}_{X,\mathtt{id}}}$ be the set of syntactically valid transactions w.r.t. assets with identifier $\mathtt{id}$ in $\Pi_{\mathcal{L}_X}$. Parameterized by a binary relation $R(\mathtt{id}) \subseteq \wp(V_{\mathcal{L}_{A,\mathtt{id}}}) \times \wp(V_{\mathcal{L}_{B,\mathtt{id}}})$, a *bridge* $\Lambda_{R(\mathtt{id})}(\Pi_{\mathcal{L}_A}, \Pi_{\mathcal{L}_B})$ is a protocol that relays requests (e.g., coin transfer) issued on a source ledger $\mathcal{L}_A$ to a destination ledger $\mathcal{L}_B$.

A bridge protocol is *robust* if it satisfies both *bridge liveness* and *bridge safety* in every execution I. Bridge liveness ensures that if a transaction from a designated set is included on the source ledger, a corresponding transaction will eventually appear on the destination ledger – guaranteeing that honest parties can successfully bridge their funds. Bridge safety requires that transactions on the destination ledger appear only if a corresponding transaction exists on the source, possibly with delay, preventing coins from being minted without a valid deposit and preserving the total supply.

6. Although block production rates may differ, we assume full nodes in both protocols can compare transaction inclusion times against the global clock.

Definition 2.9 (Bridge Liveness)**.** A bridge protocol $\Lambda_{R(\mathtt{id})}(\Pi_{\mathcal{L}_A}, \Pi_{\mathcal{L}_B})$ is *live* with *liveness* $u_\ell \in \mathbb{N}$ during I if, for all honest parties P_1 of $\mathcal{L}_A$ and P_2 of $\mathcal{L}_B$ and for all rounds $r_1, r_2 \in I$ with $r_2 \geqslant r_1 + u_\ell$ we have that, for all $\mathcal{A}, \mathcal{B}$ such that $(\mathcal{A}, \mathcal{B}) \in R(\mathtt{id})$, if $\mathtt{tx} \in \mathcal{A}$ is in the ledger ${}^{r_1}\mathcal{L}_A^{P_1}$, then there is $\mathtt{tx}' \in \mathcal{B}$ such that $\mathtt{tx}'$ is in the ledger ${}^{r_2}\mathcal{L}_B^{P_2}$ with overwhelming probability.

Definition 2.10 (Bridge Safety)**.** A bridge protocol $\Lambda_{R(\mathtt{id})}(\Pi_{\mathcal{L}_A}, \Pi_{\mathcal{L}_B})$ is *safe* with *safety* $u_s \in \mathbb{N}$ during I if, for all honest parties P_1 of $\mathcal{L}_A$ and P_2 of $\mathcal{L}_B$ and for all rounds $r_1, r_2 \in I$ with $r_2 \leqslant r_1 + u_s$ we have that, for all $\mathcal{A}, \mathcal{B}$ such that $(\mathcal{A}, \mathcal{B}) \in R(\mathtt{id})$, if $\mathtt{tx}' \in \mathcal{B}$ is in the ledger ${}^{r_2}\mathcal{L}_B^{P_2}$, then there is $\mathtt{tx} \in \mathcal{A}$ such that $\mathtt{tx}$ is in the ledger ${}^{r_1}\mathcal{L}_A^{P_1}$ with overwhelming probability.

We call bridge Λ $(u_s, u_l)-$**robust** during I, with $u_s, u_l \in \mathbb{N}$, if it is safe with safety u_s during I and live with liveness u_ℓ during I. We say that a bridge Λ is $(u_s, u_l)-$robust if it is $(u_s, u_l)-$robust for every execution I with overwhelming probability.

We now formalize the notion of a wrapped asset bridge, which supports bidirectional asset transfers between two ledgers.

Definition 2.11 (Wrapped Asset Bridge)**.** A *wrapped asset bridge* between ledgers $\mathcal{L}_A$ and $\mathcal{L}_B$ is defined by two unilateral bridge instances, $\Lambda^{\text{PegIn}}_{R_1}(\Pi_{\mathcal{L}_A}, \Pi_{\mathcal{L}_B})$ and $\Lambda^{\text{PegOut}}_{R_2}(\Pi_{\mathcal{L}_B}, \Pi_{\mathcal{L}_A})$, both associated with a common asset identifier $\mathtt{id}$. Each instance is parameterized by a relation, $R_1(\mathtt{id}) \subseteq \wp(V_{\mathcal{L}_A}) \times \wp(V_{\mathcal{L}_B})$, $R_2(\mathtt{id}) \subseteq \wp(V_{\mathcal{L}_A}) \times \wp(V_{\mathcal{L}_B})$, where $V_{\mathcal{L}_A}$ and $V_{\mathcal{L}_B}$ denote the sets of valid ledger states or transactions.

The relation $R_1(\mathtt{id})$ captures *PegIn* behavior, where locking assets on $\mathcal{L}_A$ implies minting on $\mathcal{L}_B$. Similarly, $R_2(\mathtt{id})$ captures *PegOut* behavior, where burning assets on $\mathcal{L}_B$ implies unlocking on $\mathcal{L}_A$. We further define *Bridge Operator Safety* (Definition F.1) to guarantee that honest operators can always recover their funds, even against Byzantine adversaries.

3. BITVM2-CORE: General Function Verification on Bitcoin

Bitcoin's scripting language is intentionally limited in expressiveness, lacking support for loops, recursion, or general-purpose computation. Nevertheless, any total function can be encoded as a Bitcoin Script program. Yet, in practice the 4MB maximum block size and the 1000 element stack limit[7] prevent encoding large programs, like SNARK verifiers, directly on-chain.

7. https://tinyurl.com/mt7ex7xh

In this section, we present BITVM2-CORE, a general function verification mechanism that enables any total function to be executed off-chain and verified on Bitcoin through a dispute resolution protocol. This mechanism forms the foundation of our bridge protocol but also stands as a general-purpose construction of independent interest.

The goal of BITVM2-CORE is to allow an operator O to assert that a Bitcoin Script program g executed on input x returns $y = g(x)$. The operator stakes a deposit of d BTC, where $d \geqslant 0$, and posts a commitment to this claim on-chain. If the claim is correct, they reclaim the deposit. Otherwise, any party can challenge the claim by executing a portion of the program on-chain and disproving the claim, preventing the operator from reclaiming their deposit.

We illustrate the design rationale through progressively refined strawman constructions. As a running example, we consider a complex function g that takes as input a blockchain $\mathcal{C}_Z$ from genesis till a desired height h and returns whether $\mathcal{C}_Z$ is valid – verifying expensive cryptographic procedures such as transaction validity and consensus rules.

(S1) Direct Function Verification

In our initial strawman, during setup, O posts an Assert transaction that consumes the output of a deposit and encodes in Bitcoin Script the entire program g. We stress that at setup O may not yet know a pair x, y such that $g(x) = y$, e.g., $\mathcal{C}_Z$ is not yet of height h.

When O learns x, y (e.g., $\mathcal{C}_Z$ reaches height h), they publish a Payout transaction that spends Assert by providing x, y and the signature of the operator σ_O. The equality $y = g(x)$ is verified directly on-chain and the deposit coins are moved to the output of Payout. If $y \neq g(x)$, the Payout transaction is invalid, hence O cannot reclaim their deposit.

While conceptually straightforward, this approach encounters a fundamental issue: both the script g (e.g., blockchain validation rules) and the input x (e.g., a chain $\mathcal{C}_Z$) far exceed Bitcoin's block size and stack limits by orders of magnitude. As a result, neither the Assert nor the Payout transactions can be posted on-chain, as both the output script of Assert, g, and the witness of Payout, x, exceed Bitcoin's limits.

(S2) Function Verification in a SNARK

We first address the large witness size of Payout by compiling g into a SNARK. The output of Assert now encodes a SNARK verifier for g, making the witness z provided in Payout succinct. In our example, the operator can spend Payout by publishing a succinct proof attesting to the existence of a chain $\mathcal{C}_Z$ of height h. However, the previous challenge persists: *the SNARK verifier itself is too large* to fit within Bitcoin's block size constraints, and thus cannot be embedded in Assert.

(S3) Splitting the Computation

To overcome Bitcoin's block size limitations, we split the SNARK verifier f into a sequence of smaller sub-programs $f_1, \ldots, f_k$, each of which *can* be executed within a Bitcoin transaction. Our goal is to design a mechanism such that, in the event of operator misbehavior, only a single sub-program must be verified on-chain. To achieve this, we introduce a fraud-proof-based protocol with the following objectives: (i) ensure that an honest operator is able to reclaim their deposit if they compute and provide all intermediate results $z_1, \ldots, z_k$ such that each sub-computation $z_i = f_i(z_{i-1})$ is correct for the input $z_0 = x$, and (ii) allow any party to challenge and disprove a faulty intermediate result z_i if the operator is malicious.

To achieve this, the operator must provide all intermediate states of the computation $z_0, z_1, \ldots, z_k$, where $z_k = f(z_0)$, when posting the Assert. Without this, a challenger has no basis for disproving the computation, as they do not know the intermediate states z_i. We therefore modify the deposit and Assert transactions as follows: The locking script of the deposit's output requires the z_i in the witness of Assert. The output of Assert uses a Taproot structure with $k + 1$ Tapleaves:

Happy path (Payout). The first leaf allows the operator to reclaim their deposit after a predefined delay Δ_A by publishing a Payout transaction containing their signature. This path can only be used if no challenge is submitted during the delay.

Unhappy path (Disprove). The remaining k leaves allow anyone to challenge the correctness of a sub-program f_i. Each Tapleaf contains a Bitcoin script that checks whether the claimed output z_i was correctly derived from input z_{i-1} via f_i. To dispute the claim, a challenger can publish a Disprove transaction with the relevant values and demonstrate that $z_i \neq f_i(z_{i-1})$, proving the operator lied.

If a valid challenge is submitted within the window Δ_A, the operator loses the deposit. An overview of the resulting protocol is shown in Fig. 3.

However, now Assert can only be published at runtime, as it requires the intermediate results $z_1, \ldots, z_k$ of the computation which are not known at setup. As a result, *the operator is not bound to the function f during setup*, allowing a malicious operator to cheat and reclaim the deposit by using a different function f.

(S4) Signer Committee

To enforce that f is fixed at setup, we employ a signer committee of n signers $S_1, \ldots, S_n$. The signers

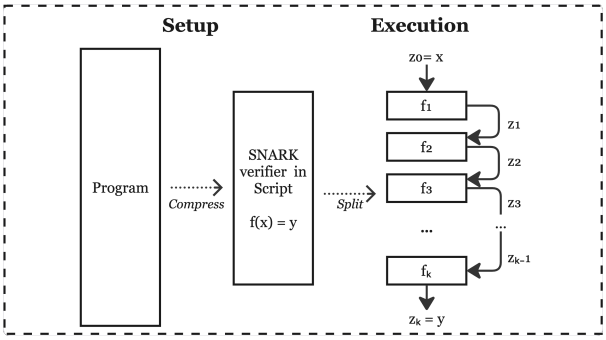

Figure 3. A SNARK verifier, representing an arbitrary program, is compiled to Bitcoin Script and split into sub-programs $f_1, \ldots, f_k$, each small enough to run in a Bitcoin block.

are assumed to be honest during setup to ensure it completes, but they cannot steal the operator's funds.

During setup, the operator O contacts the signer committee to authorize the validity of the Assert transaction, thereby binding themselves to the execution of f: Each signer generates a fresh key pair and uses it to pre-sign both Assert and Payout. The signers then delete their keys. The resulting UTXOs are locked with an n-of-n multi-signature condition $\mathsf{CheckMultiSig}_{\mathsf{C}}$ as well as the signature of the operator. We refer to the signer committee mechanism as CheckCovenant.

As long as at least one signer is honest and deletes their key after signing, O cannot deviate from the intended protocol. Once Assert is posted, O can only reclaim their deposit by following the prescribed execution and providing correct results. We highlight that, unlike similar constructions relying on honest majority committees, our approach requires only *existential honesty* among signers – a much weaker assumption.

This construction ensures incorrect claims can always be challenged. However, it *does not prevent dishonest challengers from falsely disputing honest executions*, since the script does not enforce that the z_i^c used to challenge (in Disprove) match the z_i that the operator published in Assert. We address this next.

(S5) Lamport Signatures

To prevent malicious challengers from forging arbitrary intermediate states in a Disprove transaction, we require the operator O to publish both the intermediate states z_i and their signatures σ_i in the Assert transaction. Each Disprove must include z_{i-1}, z_i and their valid signatures σ_{i-1}, σ_i in its witness, proving they were signed by O. While Bitcoin does not natively support verification of signatures over arbitrary messages, we can manually implement verification of Lamport signatures in Bitcoin Script using the native opcodes for hashing. We now present the full protocol in four phases, illustrated in Fig. 4:

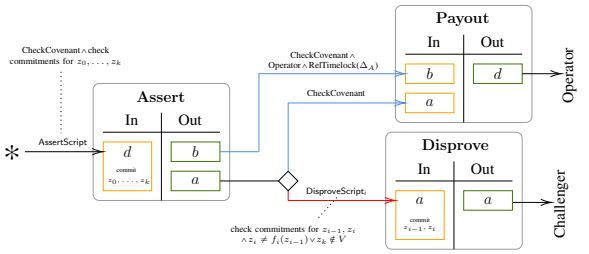

Figure 4. Illustration of the transactions Assert, Disprove, and Payout. The input of Assert can be any UTXO ($*$), but its spending script must be a Covenant including commitments to $z_0, \ldots, z_k$. For readability, we use: gray rounded rectangles for transactions, green for BTC in outputs, and orange for BTC spent. Blue arrows indicate operator actions, red arrows indicate other participants. Arrow labels show spending conditions. Gray dashed boxes highlight the parts of transactions hashed and pre-signed under non-SIGHASH_ALL flags.

Phase 1: Setup. For each z_i, the operator O generates a Lamport keypair (sk_{z_i}, pk_{z_i}) and embeds the public keys in the Assert and Payout transactions. These transactions are then sent to the signer committee, which presigns them to enforce the desired execution. The presigned transactions are returned to O and can be posted later. This setup phase is fully off-chain and is formalized in Eqs. (3) and (4) and illustrated in Fig. 4.

Phase 2: Off-chain Execution. The operator O executes the sub-programs $f_1, \ldots, f_k$ off-chain with input $z = z_0$ and produces the intermediary states $z_1, \ldots, z_{k-1}$ as well as the final state z_k. This output z_k should be $\{\mathsf{True}\}$.

Phase 3: Commit and Challenge. The operator publishes the Assert transaction (specified in Eq. (3)) on-chain, claiming to have correctly executed the program by providing the full trace $z_0, \ldots, z_k$. This transaction spends the deposit's output (indicated by $*$) holding d BTC, which must satisfy three conditions: (i) it must be spendable only via the Assert transaction (via connector outputs), (ii) (iii) it must provide valid Lamport signatures $c_{z_0}, \ldots, c_{z_k}$ verifiable against the public keys $pk_{z_0}, \ldots, pk_{z_k}$. This is expressed via AssertScript.

$$\begin{aligned}
&\mathsf{AssertScript} := \mathsf{CheckCovenant} \wedge \\
&\mathsf{CheckLampComm}_{\mathsf{pk}_{z_0}} \wedge \cdots \wedge \mathsf{CheckLampComm}_{\mathsf{pk}_{z_k}};
\end{aligned}$$

$$\begin{aligned}
\mathsf{Assert} := \Big(& in = [(*, *, \mathsf{AssertScript})], \\
& wit = [(*, \mathsf{Covenant}, z_0, \ldots, z_k, c_{z_0}, \ldots, c_{z_k})], \\
& out = [(b \text{ BTC}, \langle \mathsf{RelTimelock}(\Delta_A) \wedge \\
& \mathsf{CheckCovenant} \wedge \mathsf{CheckSig}_{\mathsf{pk}_O} \rangle), \\
& (a \text{ BTC}, \langle \mathsf{CheckCovenant}, \\
& \mathsf{DisproveScript}_1, \ldots, \mathsf{DisproveScript}_k, \rangle)] \Big),
\end{aligned} \tag{3}$$

Note that a denotes a small payout for the challenger that successfully disproves Assert, while $b = d - a$. More details follow in Phase 4.

Algorithm 1 The $\mathsf{DisproveScript_i}$ for each Tapleaf $\mathsf{i} \in \{1, \ldots, k\}$. The algorithm takes as input the committee's multisignature σ_{C}, the intermediate states z_{i-1}, z_i of the program f, and the Lamport commitments $c_{z_{i-1}}, c_{z_i}$.

function $\mathsf{DisproveScript_i}(\sigma_{\mathsf{C}},\ z_i,\ z_{i-1},\ c_{z_i},\ c_{z_{i-1}})$
  $\mathsf{CheckCovenant}()$
  $\mathsf{CheckLampCommVerify}_{pk_{z_{i-1}}}(z_{i-1}, c_{z_{i-1}})$
  Both $pk_{z_{i-1}}$ and pk_{z_i} are hard-coded in the script
  $\mathsf{CheckLampCommVerify}_{pk_{z_i}}(z_i, c_{z_i})$
  return $z_i \neq f_i(z_{i-1}) \vee (i = k \wedge z_k \notin V)$
  f_i, V and k are hard-coded in the script
end function

Phase 4: Resolution. If no one publishes a Disprove transaction within the challenge period Δ_A, the operator can redeem their deposit by publishing the Payout:

$$\begin{aligned}&\mathsf{Payout} := \\ &\Big(in = [(\mathsf{Assert}, 0, \langle \mathsf{RelTimelock}(\Delta_A) \wedge \\ &\quad \mathsf{CheckCovenant} \wedge \mathsf{CheckSig}_{\mathsf{pk}_O} \rangle), \\ &\quad (\mathsf{Assert}, 1, \langle \mathsf{CheckCovenant} \rangle)], \\ &\quad wit = [(\mathsf{Covenant}, \sigma_O), (\mathsf{Covenant})], \\ &\quad out = [(d\ \text{BTC}, \mathsf{CheckSig}_{\mathsf{pk}_O})]\Big),\end{aligned} \tag{4}$$

To dispute a false claim (i.e., some $z_i \neq f_i(z_{i-1})$), any observer can submit a Disprove transaction, which spends the second output of Assert. It satisfies one of the last k Tapleaf scripts, which are tied to subprograms f_i. These scripts verify that the committed intermediate state is incorrect using the public commitments from Assert.

$$\begin{aligned}&\mathsf{Disprove} := \\ &\Big(in = [(\mathsf{Assert}, 1, \langle \mathsf{DisproveScript_i} \rangle)], \\ &\quad wit = [(\mathsf{Covenant}, z_{i-1}, z_i, c_{z_{i-1}}, c_{z_i})], \\ &\quad out = [(a\ \text{BTC}, \mathsf{True})]\Big),\end{aligned} \tag{5}$$

The output determines the fate of the operator's deposit d BTC: a portion b BTC is burned (i.e., cannot be reclaimed), and the remainder $a = d - b$ may be claimed by the challenger. This is enforced using the `SIGHASH_SINGLE` flag [31], which commits to specific input-output pairs, while allowing more outputs to be added without a signature (e.g., bounty payouts, cover transaction fees). For simplicity, we denote this in the above definition as $(a\ \text{BTC}, \mathsf{True})$. We provide the script logic in Algorithm 1.

Optimistic BitVM2-core

To further reduce on-chain costs, we introduce an optimistic mode where the operator initially posts a lightweight Claim, asserting knowledge of a valid input-output pair (z, o) for some $f(z) = o$, without revealing the full computation. If no challenger disputes the claim by posting a Challenge transaction within delay Δ_B, the operator withdraws via PayoutOptimistic, bypassing full verification.

If challenged, the operator must fall back to the previous protocol, publishing Assert, allowing on-chain dispute resolution. Thus, optimistic verification substantially lowers on-chain footprint in typical honest scenarios while preserving security. The complete optimistic verification workflow is depicted in Fig. 5. Here, we only highlight the differences to (S5).

Phase 1: Setup. The signers presign PayoutOptimistic, enforcing the optimistic execution path.

Phase 3: Commit and Challenge. To initiate verification, the operator posts a Claim transaction (see Eq. (6)), asserting they know a pair z, o such that $f(z) = o$. They can reveal z either on-chain or off-chain (e.g., via a data availability layer) to enable external verification. With AssertScript defined in (S5), let

$$\begin{aligned}&\mathsf{Claim} := \\ &\Big(in = [*], \\ &\quad wit = [(\sigma_O \wedge *), z], \\ &\quad out = [(d\ \text{BTC}, \langle \mathsf{RelTimelock}(\Delta_B) \wedge \\ &\quad \mathsf{CheckCovenant}, \mathsf{AssertScript} \rangle), \\ &\quad (0\ \text{BTC}, \mathsf{CheckSig}_{\mathsf{pk}_O})]\Big),\end{aligned} \tag{6}$$

During the challenge period Δ_B, any party may dispute Claim by posting a Challenge transaction (specified in Eq. (7)), forcing the operator to respond with a corresponding Assert transaction, which can then be contested via a Disprove. The second output of Claim serves as a *connector output* (for definition of connector outputs, see Section A.4): it must remain unspent for the operator to use PayoutOptimistic and reclaim their deposit unchallenged.

$$\begin{aligned}&\mathsf{Challenge} := \\ &\Big(in = [(\mathsf{Claim}, 1, \mathsf{CheckSig}_{\mathsf{pk}_O}), (*)], \\ &\quad wit = [(\sigma_O), (*)], \\ &\quad out = [(c\ \text{BTC}, \mathsf{CheckSig}_{\mathsf{pk}_O})]\Big),\end{aligned} \tag{7}$$

Phase 4: Resolution. If unchallenged after Δ_B, the operator posts PayoutOptimistic (specified in Eq. (8)) to claim the funds and disable the dispute logic.

$$\begin{aligned}&\mathsf{PayoutOptimistic} := \\ &\Big(in = [(\mathsf{Claim}, 0, \langle \mathsf{RelTimelock}(\Delta_B) \wedge \\ &\quad \mathsf{CheckCovenant} \rangle), \\ &\quad (\mathsf{Claim}, 1, \mathsf{CheckSig}_{\mathsf{pk}_O})], \\ &\quad wit = [(\mathsf{Covenant}), (\sigma_O)], \\ &\quad out = [(d\ \text{BTC}, \mathsf{CheckSig}_{\mathsf{pk}_O})]\Big)\end{aligned} \tag{8}$$

We prove BitVM2-core satisfies both completeness

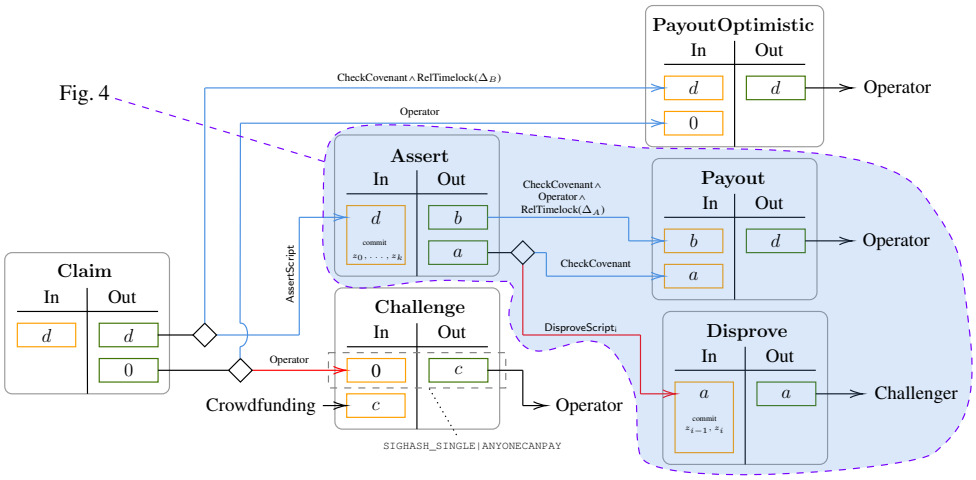

Figure 5. Illustration of the optimistic function verifier.

and soundness (Definitions 2.4 and 2.5) assuming existential honesty of signers, operators, and challengers in Appendix D.

Reward and Collateral. To deter a malicious challenger from griefing an honest operator by repeatedly triggering costly Assert transactions, the protocol requires the challenger to post collateral c BTC that covers the operator's on-chain costs. This collateral can be crowdfunded using the SIGHASH_SINGLE|ANYONECANPAY flag, allowing multiple users to jointly contribute inputs to the Challenge transaction. See Section 8.1 for details.

4. On-chain Bitcoin Light Client

In this section, we construct an on-chain Bitcoin light client for static difficulty using BitVM2-core. This serves both as a standalone primitive and as a building block in our bridge. The goal is to commit on-chain to the tip of a chain $\mathcal{C}$ via a Lamport signature, ensuring that $\mathcal{C}$ includes a block B that is both *safe* and *live* (Definition E.5), meaning it belongs to the stable chain adopted by honest nodes and is sufficiently recent.

To achieve this, we instantiate BitVM2-core with a relation $R := \{(\phi, w) : \Pi_{lc}(\phi, w) = \mathsf{True}\}$, where Π_{lc} is defined in Algorithm 2.

At runtime, the prover (operator) posts an Assert transaction, signing states $z_0, \ldots, z_k$ via Lamport keys, where $z_0 := (\phi, \pi)$ and ϕ consists of a chain tip $\mathcal{C}[-1]$, height $h = |\mathcal{C}|$, and a SNARG proof π that certifies a valid PoW chain of length h from B_0 to $\mathcal{C}[-1]$. The admissible block is set to $\mathsf{B} := \mathcal{C}[-m-k]$.

To guarantee liveness, we enforce a relative timelock $\mathsf{AbsTimelock}(h + 3\Delta)$ on a connector output in Assert, ensuring that h is recent. If an operator posts an outdated height, challengers can disprove it before payout. The only change from the standard protocol is this added spending condition (see Fig. 6).

$$
\begin{aligned}
&\mathsf{Assert} := \\
&\Big(in = [(\mathsf{Claim}, 0, \mathsf{AssertScript})], \\
&\quad wit = [(*, \mathsf{Covenant}, z_0, \ldots, z_k, c_{z_0}, \ldots, c_{z_k})], \\
&\quad out = [(b \text{ BTC}, \mathsf{CheckCovenant}), \\
&\quad (a \text{ BTC}, \langle \mathsf{CheckCovenant} \wedge \mathsf{RelTimelock}(\Delta), \\
&\quad \mathsf{DisproveScript}_1, \ldots, \mathsf{DisproveScript}_k, \\
&\quad \mathsf{CheckLampComm}_{\mathsf{pk}_h} \wedge \mathsf{AbsTimelock}(h + 3\Delta)\rangle)]\Big)
\end{aligned}
\tag{9}
$$

Let Δ denote the block confirmation window. Setting the timelock to 3Δ ensures that an honest operator has sufficient time to post Assert, wait for confirmation, and publish Payout. Intuitively, this construction guarantees that $\mathsf{B} := \mathcal{C}[-m-k]$ is *live*, since it lies within $m + k$ blocks from the tip, and *safe*, as the adversary cannot outpace honest miners. We formalize this in Theorem 6.3 in Section 6.2. A formal proof in Bitcoin backbone [13] is in Appendix E.

5. The BitVM2-bridge Design

A core application of BitVM2-core is enabling trust-minimized bridges between Bitcoin ($\mathcal{L}_A$) and other blockchains or Layer 2 networks ($\mathcal{L}_B$), collectively called *sidesystems*. As outlined earlier, the goal of our bridge is to securely represent Bitcoin on the sidesystem as a wrapped asset (wBTC), redeemable at a 1:1 ratio for BTC.

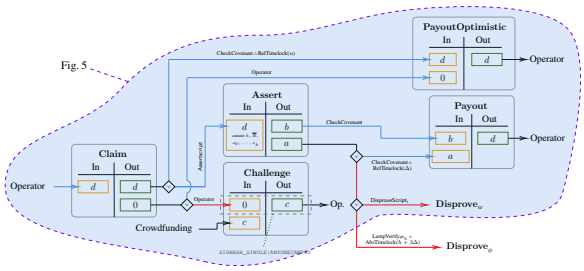

Figure 6. Illustration of the on-chain Bitcoin light client. Its goal is to verify to Bitcoin that a block B is in the ledger of Bitcoin. It does so by verifying that 1) the B is in a valid PoW chain $\mathcal{C}$, 2) the chain $\mathcal{C}$ is a prefix of the stable chain adopted by honest Bitcoin miners. The former is achieved using BITVM2-CORE (Figure 5 and the latter is achieved by verifying that the height h of $\mathcal{C}$ is close to the length of the longest Bitcoin chain. This latter verification is piggy-backed onto BITVM2-CORE by adding a Lamport signed height h in Assert, and adding a Disprove path to allow challenge if the height is not close to the height of the longest chain.

Algorithm 2 Chainstate proof Π_{lc} defining the SNARG relation R. $\phi = (\overline{B}, h)$ and $w := \mathcal{C}$ is a list of blocks with $|w| > m + k$. The algorithm checks that w is a valid PoW chain from the genesis B_0 to $\overline{B}$ at height h.

function $\Pi_{lc}(\phi,\ w)$
  for $i \in \{0, \ldots, |w| - 2\}$ **do**
    if $w[i+1].parent \neq \mathcal{H}(w[i]) \vee \mathcal{H}(w[i+1]) \geqslant T$ **then**
      return False
    end if
  end for
  return $B_0 = \mathcal{H}(w[0]) \wedge \phi.\overline{B} = \mathcal{H}(w[-1]) \wedge \phi.h = |w| - 1$ *$w[-m-k]$ is admissible*
end function

Bridges typically require trust assumptions, with current Bitcoin bridges relying heavily on multisignatures and honest majority assumptions [34]. While ideally, peg-in/out transactions could rely on native blockchain light clients, Bitcoin's limited scripting capability and lack of covenants restrict direct implementation: since the receiver (Bob) of the peg-out is not known at the time of the peg-in, it is impossible to enforce that only Bob can unlock the v BTC.

Our design addresses this limitation by introducing a known operator set that fronts funds during peg-out, and then reclaims their funds by leveraging the Bitcoin light client introduced in Section 4. This setup fixes operators at setup, bypassing the unknown recipient issue.

For simplicity, we assume the sidesystem operates as a rollup, using Bitcoin as its underlying consensus layer. Specifically, the rollup state transitions are verifiable via data commitments published on Bitcoin. Formally, we define a modified relation for the SNARG within the Bitcoin light client, illustrated in Algorithm 3. This modification ensures the PegOut transaction is indeed reflected in the sidesystem state.

Algorithm 3 The chainstate proof Π, which defines the relation R over which we define our SNARGs.

function $\Pi(\phi,\ w)$
  if $\mathsf{PegOut} \notin w[:-m-k] \vee \mathsf{PegOut} \notin \mathsf{Burn} \vee \mathsf{Burn} \notin \mathsf{SidesystemState}(w[:-m-k])$ **then**
    return False
  end if
  return $\Pi_{lc}(\phi, w)$ *cf. Algorithm 2*
end function

Peg-In. Alice sends a peg-in request message to the signing committee, whose members create ephemeral keys and reply with the corresponding public key. With these keys, she constructs (but does not broadcast) the transaction PegIn, locking BTC on $\mathcal{L}_A$.

$$\begin{aligned}\mathsf{PegIn} := \Big(&in = [(*, *, \mathsf{CheckSig}_{\mathsf{pk}_A})],\\ &wit = [(\sigma_A)],\\ &out = [(v\ \mathrm{BTC}, \mathsf{CheckCovenant})]\Big),\end{aligned} \tag{10}$$

Alice shares PegIn with all operators who execute the setup of the Bitcoin light client (cf. Section 4), including PegIn as an additional input. Specifically, both the Payout and PayoutOptimistic transactions now require an extra input, spending the output of PegIn. This modified transaction graph is depicted in Fig. 7. Once operators complete this step, Alice publishes PegIn on Bitcoin, which results in minting equivalent wrapped BTC (v wBTC) on $\mathcal{L}_B$. This trustless minting can be handled via light clients on $\mathcal{L}_B$ [7] or atomic swaps [18], [30] and is thus omitted here.

Peg-Out. Wrapped BTC circulate freely on $\mathcal{L}_B$ until a user, Bob, decides to peg out. Bob constructs a partially signed PegOut transaction that takes as input a fixed, arbitrary zero-value output and an unspecified second input $*$, sending $v - f_O$ coins to himself. He signs the first input and output using `SIGHASH_SINGLE|ANYONECANPAY`. Bob then posts Burn on $\mathcal{L}_B$, burning his v wBTC and embeds the presigned PegOut (e.g., via an `OP_RETURN` output), denoted $\mathsf{PegOut} \in \mathsf{Burn}$ in Algorithm 3.

$$\begin{aligned}&\mathsf{PegOut} :=\\ &\Big(in = [(*, *, \mathsf{CheckSig}_{\mathsf{pk}_B}), *],\\ &\ wit = [(\sigma_B), (*)],\\ &\ out = [(v - f_O\ \mathrm{BTC}, \mathsf{CheckSig}_{\mathsf{pk}_B})]\Big),\end{aligned} \tag{11}$$

Operators monitoring $\mathcal{L}_B$ extract the PegOut, complete it by adding the second input using their own funds,

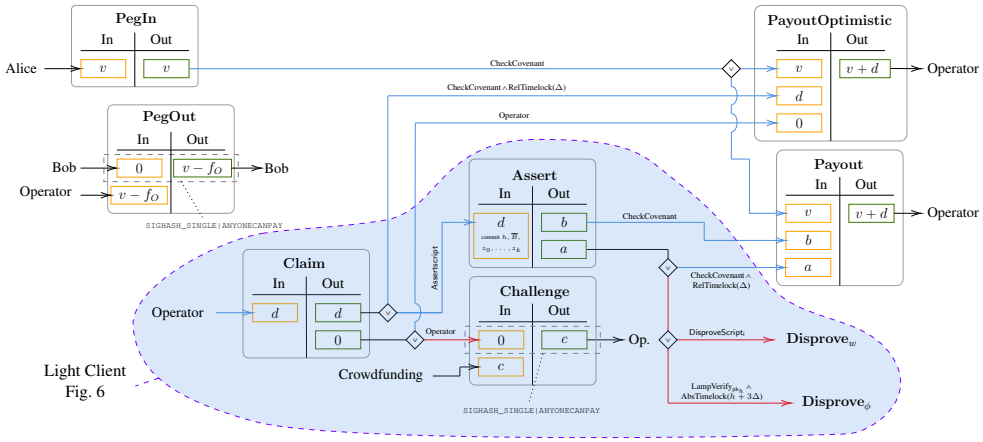

Figure 7. The BITVM2-BRIDGE construction.

and post it to $\mathcal{L}_A$. Since the first input is uniquely spendable, only one operator can succeed. Operators compete, since successful completion yields a fee f_O from the transaction. The operator whose PegOut is included on $\mathcal{L}_A$ proceeds by executing the Bitcoin light client protocol, i.e., post Claim to recover their funds from the bridge. Due to our design (see Section 6 and Appendix D), an honest operator always manages to claim the locked BTC (v BTC), recovering their fronted $v - f_O$ BTC and netting a profit of f_O.

In summary, this construction achieves, for the first time, trust-minimized bridging between Bitcoin and sidesystems, leveraging the security guarantees of BITVM2-CORE and the proposed Bitcoin light client, overcoming inherent scripting limitations in Bitcoin.

6. Analysis Overview

We outline our security analysis, detailed in Appendix D.

6.1. BITVM2-CORE

First, we show that BITVM2-CORE satisfies *soundness*, i.e., no malicious operator without a valid witness can reclaim the deposit, and *completeness*, i.e., an honest operator can always reclaim their deposit.

Theorem 6.1. *If the SNARG is sound, the ledger is safe and live, and a set of signers and a set of verifiers with existential honesty,* BITVM2-CORE *satisfies soundness (Definition 2.5).*

Proof sketch. If a payout is executed without a valid witness, then either (i) PayoutOptimistic appears without a preceding Challenge, or (ii) Payout appears after a Challenge and Assert, but Disprove does not. In case (i), either the SNARG verification falsely accepted a proof (violating SNARG soundness) or the Challenge transaction was not included due to a liveness or safety violation of the ledger (i.e., Challenge was not included or it was included in the ledger but was later removed). In case (ii), either Disprove was not included because of liveness failure,or was initially included but was later forked out All these cases contradict the soundness of the SNARG or the security of the ledger and thus occur with negligible probability. □

Theorem 6.2. *Assuming the SNARG is complete, the ledger is safe and live, and a set of signers with existential honesty,* BITVM2-CORE *satisfies completeness (Definition 2.4).*

Proof sketch. If the honest prover knows a valid witness but no payout appears on-chain, then either (i) Claim or Assert fails to be included in the ledger despite being submitted within their allowed time windows (violating ledger liveness) or the transactions were included and later removed (violating ledger safety), or (ii) Disprove is included even though the SNARG proof was valid (violating SNARG completeness). Alternatively, PayoutOptimistic or Payout fail to appear despite being submitted timely by the honest prover, which again contradicts ledger liveness or safety. In all cases, one of the underlying assumptions fails, resulting in a violation that occurs with negligible probability. □

6.2. On-chain Bitcoin Light Client

Theorem 6.3 (Light Client Security)**.** *For any $m > (\frac{2}{\delta} - 1)x$, our light client protocol accepts a (u, k)-admissible* B*, where k is the common prefix parameter,*

and u the time it takes for the chain of any honest party to grow by more than $m + x$ blocks.

Proof sketch. Safety holds by contradiction: suppose a block $\hat{B}$ is accepted as admissible at round r, but is not in the view of any honest party at that round. Then the adversary must have produced a chain containing $\hat{B}$ that is at least m blocks longer than the honest chain prefix and still consistent with k blocks of honest history. By the patience lemma and typical execution properties of the Bitcoin backbone, this requires $m \leqslant (\frac{2}{\delta} - 1)x$, contradicting the assumption. Liveness follows directly from the definition of u as the number of rounds it takes for the honest chain to grow by more than $m + x$ blocks, ensuring that $\mathsf{B} = \mathcal{C}[-m-k]$ was not yet visible at round r, but is part of the chain at round $r + u$.

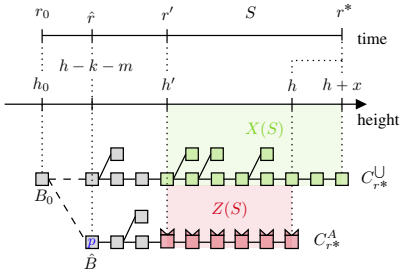

Figure 8. Proof illustration

6.3. BITVM2-BRIDGE

Theorem 6.4 (PegIn Safety)**.** *The PegIn bridge $\Lambda^{PegIn}_{R_1}$ is safe under the assumption that at least one signer is honest during setup and that $\Pi_{\mathcal{L}_A}$ satisfies ledger safety.*

Proof sketch. If a mint appears on $\mathcal{L}_B$ without a corresponding PegIn on $\mathcal{L}_A$, then either the committee wrongly signed a mint without proof-of-lock (contradicting the existence of an honest signer) or a valid PegIn was removed from the ledger (violating $\mathcal{L}_A$ safety). Thus, safety holds except with negligible probability.

Theorem 6.5 (PegIn Liveness)**.** *The PegIn bridge $\Lambda^{PegIn}_{R_1}$ is live assuming (i) at least one honest operator exists during setup, (ii) the all signers are honest during setup, and (iii) both $\Pi_{\mathcal{L}_A}$ and $\Pi_{\mathcal{L}_B}$ are live.*

Proof sketch. If a PegIn on $\mathcal{L}_A$ does not result in a mint on $\mathcal{L}_B$, then either there was no honest operator to initiate the mint (violating (i)), or the mint was not included despite being valid (violating $\mathcal{L}_B$ liveness), or the committee failed to sign a valid mint (violating (ii)). Thus, liveness holds except with negligible probability.

Theorem 6.6 (PegOut Safety)**.** *The PegOut bridge $\Lambda^{PegOut}_{R_2}$ satisfies safety with parameter u_s assuming at least one honest signer post-setup, and $\mathcal{L}_B$ satisfies safety.*

Proof sketch. If a PegOut appears on $\mathcal{L}_A$ without a corresponding Burn on $\mathcal{L}_B$, either (i) the committee misbehaved in setup, contradicting setup honesty, or (ii) the adversary dynamically corrupted all the signers and thus can presign arbitrary transactions, which cannot occur if one signer remains honest post-setup; or (iii) the Burn was included and removed (violating $\mathcal{L}_B$ safety). All cases occur with negligible probability.

Theorem 6.7 (PegOut Liveness)**.** *The PegOut bridge $\Lambda^{PegOut}_{R_2}$ satisfies liveness with parameter $u = u_\ell$ assuming all signers are honest during setup while at least one remains honest post-setup, $\mathcal{L}_A$ is safe and live, and an honest operator monitors $\mathcal{L}_B$.*

Proof sketch. Suppose Burn is posted on $\mathcal{L}_B$ at round r, but no corresponding PegOut transaction appears on $\mathcal{L}_A$ by round $r + u$. This can occur only if: (i) the signer committee failed to construct valid PegOut transactions during setup, violating setup honesty; (ii) an honest operator submitted a valid PegOut transaction but it was not included within u rounds, violating $\mathcal{L}_A$'s liveness; or (iii) PegOut was included but was later removed, violating $\mathcal{L}_A$'s safety. Hence, PegOut liveness holds except with negligible probability.

Definition 6.1 (Bridge Operator Safety)**.** A bridge protocol $\Lambda^{\text{PegIn}}_{R_1}, \Lambda^{\text{PegOut}}_{R_2}$ is said to be *budget balance complete* with *recovery parameter* $u_\Lambda \in \mathbb{N}$ during the execution I if, for any honest operator p, the following holds: If there is a transaction $\mathtt{tx} \in \mathsf{PegOut}(\mathtt{id})$ such that $(\mathtt{tx}, p)$ is included in ${}^{r_1}\mathcal{L}^{P_1}_A$ for all honest parties P_1 at some round r_1, then there must exist a transaction $\mathtt{tx}' \in \{\mathsf{Payout}, \mathsf{PayoutOptimistic}\}$ such that for every round $r_2 \geqslant r_1 + u_\Lambda$, $(\mathtt{tx}', p) \in {}^{r_2}\mathcal{L}^{P_1}_A$ for all honest P_1.

Theorem 6.8 (Bridge Operator Safety)**.** *The bridge satisfies operator safety with recovery parameter u_Λ, assuming the signer committee is honest during setup and one member remains honest after, an honest challenger exist, both ledgers are live and safe, the light client is secure, and* BITVM2-CORE *is complete and sound.*

Proof sketch. Suppose, for contradiction, that an honest operator p_i posts a PegOut transaction on $\mathcal{L}_A$ at round r_1, but no corresponding payout $\mathsf{tx}' \in \{\mathsf{Payout}, \mathsf{PayoutOptimistic}\}$ appears within u_Λ rounds. This can happen in two ways. First, another operator $p_j \neq p_i$ posts a payout tx' using the same PegIn output. This contradicts the soundness of BITVM2-CORE if p_j succeeds without a valid witness or light client proof,

or contradicts light client safety if p_j fabricated a proof using a block that does not contain (PegOut, p_j), either or the committee misbehaved in setup (contradicting that signers are honest during setup) or the adversary can presign arbitrary transaction because they dynamically corrupted all signers (contradicting that one signer remains honest post-setup). Second, no payout is posted at all. This contradicts setup honesty if transactions were not correctly presigned, or light client liveness if the operator cannot derive a valid witness after the block becomes deep, or completeness of BITVM2-CORE if the operator has a valid witness but cannot succeed. In all cases, assumptions are violated with negligible probability, so operator safety holds.

7. Evaluation

We provide a production-level reference implementation, which is the result of a large open-source effort, of which we are among the core contributors. The anonymized code repository of our `BitVM implementation` can be found here [1]. Additionally, we successfully submitted the BitVM transactions on the Bitcoin *mainnet*.

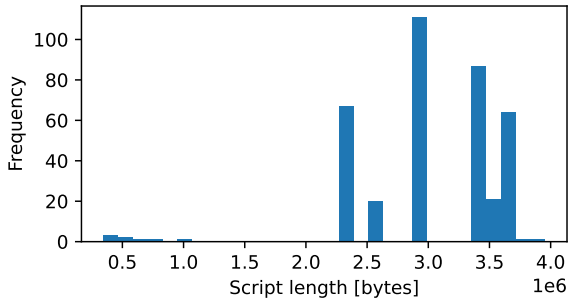

Figure 9. Histogram of script lengths in our implementation

The 379 Disprove scripts span roughly 500 kB to 3.99 MB, with dense peaks near $\approx$ 3 MB (precompute/hash phases), and a smaller tail above 3.5 MB for the largest composite steps. Most scripts therefore sit in the 2–3 MB band, with only a handful below half a megabyte or beyond 3.5 MB.

Based on historical fee data, the fee required to ensure inclusion of a transaction on Bitcoin mainnet within a two-week window has rarely exceeded $\sim$50 sat/vB. At that price, a $\sim$4 MB disprove entails roughly 2 BTC of collateral per operator. Should fees spike higher, the transaction can still be confirmed through cooperative child-pays-for-parent (CPFP) bumping.

On June 3, 2025, we validated the full unhappy path on Bitcoin mainnet: starting with the Claim [4], followed by the Challenge [3], then the Assert [2], and

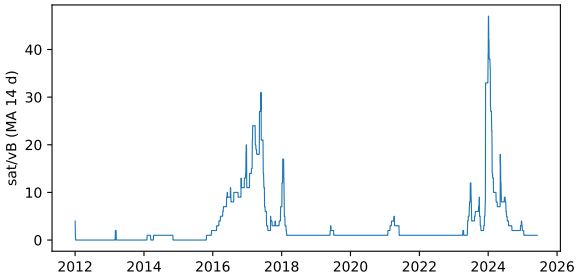

Figure 10. Fee-rate that gives a $\geqslant 95\%$ chance of confirmation within 14 days (mempool.space data).

finally the (largest) Disprove [5] transaction. The sequence confirmed in 42 blocks (7h 36min), demonstrating that a challenge period is viable even under mainnet congestion. The total cost was 14.9M sat ($\approx \$16,000$ as of June 2025), with Assert and Disprove accounting for 3.89M and 10.99M sat, respectively. This path is unlikely in practice, as honest operators avoid losing collateral and challengers must pay these fees in case their challenge fails (cf. Section 3). Rather, this amount reflects the collateral required per peg-in in BITVM2-BRIDGE. Reducing the size of the Assert and Disprove transactions remains an active area of research.

The happy path, which is the one normally executed, involves instead only Pegout, Claim, and PayoutOptimistic that remain below 16.75 vkB, which is around 50k sat (roughly $ 53). We discuss how to further reduce this cost to 534 vbytes, or around 1.6k sats (around $ 1.66) in Section 8.9.

8. Practicality, Limitations, and Extensions

In this section, we outline practical considerations, limitations, design trade-offs, and potential extensions to BITVM2-BRIDGE.

8.1. Disincentivizing malicious challenges

Currently, any user can always challenge the Claim of an operator, forcing them to reveal the Assert transaction, which results in increased on-chain costs even for an honest operator. We disincentivize such griefing attacks against the operator by requiring the challenger to put up some *collateral* c BTC. This collateral c BTC should be enough to cover the transaction fees of Assert. **Crowdfunding Collateral for Challenges.** While the collateral protects operators, it may deter challengers from triggering the challenge, should it be too expensive for individual users. We mitigate this by crowdfunding the challenge collateral: The operator presigns the first input and output of

the Challenge transaction with the SIGHASH flag [31] `SIGHASH_SINGLE|ANYONECANPAY`, thus allowing challengers to add more inputs to Challenge. Consequently, instead of a single challenger covering the upfront costs for Challenge, the collateral can be shared among users, amortizing costs. Each participating challenger simply adds their input and sends the transaction along until enough inputs are provided to cover c BTC.

8.2. Practical Limitations

Our design uses clever tricks and workarounds to enable optimistic verification within existing Bitcoin Script, avoiding the need for a fork.

- **Covenant emulation via a Signer committee**. One of the limitations of BITVM2-CORE stems from the need to emulate covenants using a signer committee. Since we only use n-of-n multisignatures, i.e., without threshold signing, the signer set can be scaled far beyond the "vanilla" Bitcoin multisignature to 100+ signers using protocols like MuSig [25], [26], [27], at the cost of increased off-chain communication and coordination effort. While the implementation of this setup is beyond the scope of this paper, we note that such setups have been successfully executed at a much larger scale, for example, Ethereum's KZG trusted setup ceremony[8] which involved over 140,000 participants, executed via a simple website.
- **Large, non-standard transactions**. As of this writing, the Assert and Disprove transactions in the BITVM2-CORE implementation exceed the 400kB "standardness" rule, i.e., these transactions will not be relayed by non-modified Bitcoin full nodes. As a result, we cannot use the Bitcoin P2P network to deliver these transactions to miners and must establish other communication channels, ideally as direct as possible to enable timely inclusion in the Bitcoin blockchain. Currently, some mining pools offer public services to include valid but non-standard transactions.[9] We discuss how we could reduce the transaction sizes in Section 8.5.
- **Fixed deposit amounts**. BITVM2-CORE supports only fixed deposit sizes, limiting BITVM2-BRIDGE flexibility—users must gather the exact wrapped BTC amount to peg out. A simple workaround is deploying multiple BITVM2-CORE instances with varying sizes (e.g., 0.5-100 BTC). In practice, we expect professional users to handle peg-ins/outs as a service (Section 8.6).
- **Operators must front BTC during peg-out.** By design, operators must front the BTC to the withdrawing user from their own balance. The benefit of this design is that users do not have to wait for the challenge period to pass and receive their BTC right away. The drawback is that this introduces a capital cost for operators, who must have sufficient liquidity in BTC during peg-out. This cost must be accounted for in the operator fee f_O, or additional incentives paid by the sidesystem, e.g., sharing sidesystem transaction fees. Future additions to Bitcoin's consensus could allow for a design in which the identity of the spender of the PegOut transaction does not need to be determined at the time of setup, removing the need for operators.
- **Sidesystem light client complexity**. In this paper, we assumed that the sidesystem is a Bitcoin rollup, i.e., uses the Bitcoin blockchain as its consensus by posting data commitments and verifying state transitions. BITVM2-CORE can theoretically be used to bridge BTC to blockchains with consensus protocols completely independent from Bitcoin. This would require implementing a light client of the remote chain in the SNARK verifier.

8.3. Winternitz instead of Lamport Signatures

Winternitz signatures [10] can replace Lamport signatures. This reduces signature size by about 50%.

8.4. Other Bitcoin Light Client Approaches

8.4.1. Light Client with Periodic Commitments. In an alternative light client design, operators regularly commit on-chain to the latest Bitcoin block they are aware of using Lamport/Winternitz signatures, which allow challenging invalid commitments. However, this method is impractical due to the substantial communication overhead and the associated on-chain costs.

8.4.2. Using Optimistic On-chain Commitments. A slight improvement over the design of Section 8.4.1 is making the commitments optimistic: instead of committing periodically to the latest Bitcoin block on-chain, the operators put their commitments off-chain, e.g., on a bulletin board or another public blockchain with cheap data storage. In case of missing or wrong commitments, challengers can challenge the operator, forcing them to put the commitment on-chain, which in turn can be challenged using a protocol similar to Section 3.

However, in the worst-case scenario, a malicious challenger can force operators to commit all data on-chain by issuing continuous challenges. While a possible mitigation could be to require the challengers to

8. https://github.com/ethereum/kzg-ceremony?tab=readme-ov-file
9. See e.g. https://slipstream.mara.com/.

cover or split the operator's on-chain costs, the additional capital requirements imposed on the challengers render such a scheme impractical.

8.5. Tuning Assert and Disprove Sizes

So far, we assumed that there is only a single Assert transaction in which the operator must commit to the intermediary states of the program execution. Given the Bitcoin block size limit, we estimate that we can split a program into at most 960 sub-programs of 4MB each[10] using a single 4MB Assert transaction. In theory, it is possible to use n Assert transactions in parallel which would allow us to (i) reduce the size of each Assert transaction, and/or (ii) reduce the size of the sub-programs and hence the size of the Disprove transaction. Thereby, in the Claim transaction the operator must commit to the state of the computation at the end each of the n sub-program groups as split across Assert transactions. A notable drawback of this approach is that the required number of pre-signed transactions (when emulating covenants via a signer committee) increases exponentially. Specifically, we must pre-sign 2^n Payout transactions to ensure an honest operator can reclaim funds, regardless of which of the n Assert transactions a malicious challenger forces them to reveal.

8.6. Fast and Flexible Bridging via Atomic Swaps

Considering the fixed deposit sizes, as well as the need to run additional software to initiate a BITVM2-CORE deposit, we expect that peg-ins and peg-outs in BITVM2-BRIDGE will be performed by professional users as a service, potentially by operators themselves acting as market makers. In turn, we anticipate that the majority of users enter and exit the sidesystem using cross-chain swaps such as [19]. Specifically, in our BITVM2-BRIDGE protocol, Alice would not only peg-in but also handle the peg-outs. To this end, Alice maintains balances in both BTC on Bitcoin and wBTC on the sidesystem, offering Bob to swap in and out against a fee, and re-balancing as needed by performing peg-in and peg-out operations. We observe that this model has been implemented at scale on Ethereum and Ethereum L2s via the so-called "liquidity bridges"[11].

8.7. Rotating Operators

In our design we assume that each BITVM2-BRIDGE instance has a pre-defined operator set. While it is possible to add and remove operators during runtime, the implementation of a such mechanism depends on the Covenant model. In our case, where we emulate covenants via a signer committee, the m signers cannot add or remove operators later on, as this would contradict our protocol and model that assumes at least one honest signer that deletes their key. Even if such modifications were possible, for example, by employing key-evolving cryptography, our signer set is designed to be large for safety reasons. As a result, the coordination overhead may be significant and the failure rate (e.g. a signer being offline) high, possibly making ad-hoc operator rotation impractical for an existing BITVM2-BRIDGE instance. Instead, we can introduce a pre-defined, periodic rotation schedule embedded in the BITVM2-BRIDGE ruleset, i.e., the SNARG verifier. Under this model, as long as there exists one honest operator per BITVM2-BRIDGE instance, we could rotate operators for existing instances by executing a peg-out transaction that at the same time acts as a peg-in transaction for a new BITVM2-BRIDGE instance with a different operator set. A similar rotation design is implemented by tBTCv2[12].

The design of mechanisms for selecting operators for a BITVM2-BRIDGE instance is outside the scope of this paper. A design implemented across Proof-of-Stake systems, for example, is random sampling based staked tokens, e.g. BTC [24] [11]. Similarly, one could sample operators based on Proof-of-Work performed over a certain period, e.g., in sidesystems merge-mined with Bitcoin.

8.8. Consensus Changes Enhancing BITVM2

A range of proposed consensus changes would enable improved bridge designs. Most notably, big integer arithmetic and covenants.

Having big integer arithmetic, as proposed by the *Great Script Restoration* [29], would substantially reduce the SNARK verifier's script size from gigabytes down to megabytes, which could potentially allow for SNARK verification in a single transaction, drastically simplifying the overall design.

Any covenants proposal that allows to commit to transaction inputs would remove the need for the signer committee and guarantee unconditional safety of deposits. Potential proposals include introspection opcodes (`OP_INSPECTINPUTOUTPOINT`) or `TXHASH`. Introspection opcodes are the most efficient approach in terms of transaction fees.

10. https://bitvm.org/snark

11. See for example https://uniswap.org/whitepaper-uniswapx.pdf and https://docs.across.to/introduction/what-is-across.

12. https://github.com/keep-network/tbtc-v2/tree/main/docs

8.9. Reducing Cost of Happy Path

We can reduce the cost of the happy path, by moving the AssertScript from Claim to a transaction that sits between Claim and Assert, thus making Claim significantly smaller.

References

[1] BitVM reference implementation: Anonymized code repository. https://anonymous.4open.science/r/BitVM-C025.

[2] Bitcoin transaction 32129c9...08711f. https://mempool.space/tx/32129c9ba93f5cf69fe135c3108f869b80b3370dbbcaf22ca33ed309cb08711f, 2025. Accessed 5 Jun 2025.

[3] Bitcoin transaction 49f3425...5d80581. https://mempool.space/tx/49f3425b595304c4d79d0b890837c21c76c518df87ef403f99faeaa6a5d80581, 2025. Accessed 5 Jun 2025.

[4] Bitcoin transaction 6b096f3...d4d147a. https://mempool.space/tx/6b096f35814cf26c3df031b22d91dfda56d25356f00c07d89cdebff71d4d147a, 2025. Accessed 5 Jun 2025.

[5] Bitcoin transaction 8ecfef...510249. https://mempool.space/tx/8ecfefd438a229c7cea10f6973f49d8bbe3a620fd557f7f2cb3658ac59510249, 2025. Accessed 5 Jun 2025.

[6] Lukas Aumayr, Zeta Avarikioti, Robin Linus, Matteo Maffei, Andrea Pelosi, Christos Stefo, and Alexei Zamyatin. BitVM: Quasi-turing complete computation on bitcoin. Cryptology ePrint Archive, Paper 2024/1995, 2024.

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

[8] Ekrem Bal, Lukas Aumayr, Atacan İyidoğan, Giulia Scaffino, Hakan Karakuş, Cengiz Eray Aslan, and Orfeas Stefanos Thyfronitis Litos. Clementine: A collateral-efficient, trust-minimized, and scalable bitcoin bridge. https://citrea.xyz/clementine_whitepaper.pdf, 2025.

[9] Federico Barbacovi and Enrique Larraia. Enforcing arbitrary constraints on bitcoin transactions. Cryptology ePrint Archive, Paper 2025/912, 2025.

[10] Dan Boneh and Victor Shoup. A graduate course in applied cryptography. *Draft 0.6*, 2023.

[11] Xinshu Dong, Orfeas Stefanos Thyfronitis Litos, Ertem Nusret Tas, David Tse, Robin Linus Woll, Lei Yang, and Mingchao Yu. Remote staking with economic safety. *arXiv preprint arXiv:2408.01896*, 2024.

[12] Ariel Futoransky, Fadi Barbara, Ramses Fernandez, Gabriel Larotonda, and Sergio Demian Lerner. TOOP: A transfer of ownership protocol over bitcoin. Cryptology ePrint Archive, Paper 2025/964, 2025.

[13] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015*, pages 281–310, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[14] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Advances in Cryptology-EUROCRYPT 2015*, pages 281–310. Springer, 2015.

[15] Craig Gentry and Daniel Wichs. Separating succinct non-interactive arguments from all falsifiable assumptions. In *Proceedings of the Forty-Third Annual ACM Symposium on Theory of Computing*, STOC '11, page 99–108, New York, NY, USA, 2011. Association for Computing Machinery.

[16] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281–28, 04 1988. Copyright - Copyright] © 1988 Society for Industrial and Applied Mathematics; Last updated - 2023-12-04.

[17] Jens Groth. On the size of pairing-based non-interactive arguments. In *Advances in Cryptology–EUROCRYPT*, pages 305–326. Springer, 2016.

[18] Maurice Herlihy. Atomic cross-chain swaps. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, PODC '18, page 245–254, New York, NY, USA, 2018. Association for Computing Machinery.

[19] Maurice Herlihy. Atomic cross-chain swaps. arXiv:1801.09515, 2018. Accessed:2018-01-31.

[20] Victor I. Kolobov, Avihu M. Levy, and Moni Naor. ColliderVM: Stateful computation on bitcoin without fraud proofs. Cryptology ePrint Archive, Paper 2025/591, 2025.

[21] Misha Komarov. Bitcoin pipes: Covenants on bitcoin without soft fork. https://delvingbitcoin.org/t/bitcoin-pipes-covenants-on-bitcoin-without-soft-fork/1195, 2024.

[22] Leslie Lamport. Constructing digital signatures from a one way function. Technical Report CSL-98, October 1979.

[23] Sergio Demian Lerner, Ramon Amela, Shreemoy Mishra, Martin Jonas, and Javier Álvarez Cid-Fuentes. Bitvmx: A cpu for universal computation on bitcoin, 2024.

[24] Robin Linus. Stakechain: A bitcoin-backed proof-of-stake. In *International Conference on Financial Cryptography and Data Security*, pages 3–14. Springer, 2022.

[25] Gregory Maxwell, Andrew Poelstra, Yannick Seurin, and Pieter Wuille. Simple schnorr multi-signatures with applications to bitcoin. *Designs, Codes and Cryptography*, 87(9):2139–2164, 2019.

[26] Jonas Nick, Tim Ruffing, and Yannick Seurin. Musig2: Simple two-round schnorr multi-signatures. In *Annual International Cryptology Conference*, pages 189–221. Springer, 2021.

[27] Jonas Nick, Tim Ruffing, Yannick Seurin, and Pieter Wuille. Musig-dn: Schnorr multi-signatures with verifiably deterministic nonces. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1717–1731, 2020.

[28] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network. Whitepaper, 2015.

[29] Rusty Russell. The great script restoration, jan 2024. https://github.com/bitcoin/bips/blob/c2f268e83031b9b67e798c5c72a1171bfc463d1f/bip-unknown-var-budget-script.mediawiki.

[30] Sri AravindaKrishnan Thyagarajan, Giulio Malavolta, and Pedro Moreno-Sanchez. Universal atomic swaps: Secure exchange of coins across all blockchains. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1299–1316, 2022.

[31] Bitcoin Wiki. Contract: Sighash flags, sep 2023.

[32] Pieter Wuille, Jonas Nick, and Anthony Towns. Bip 0341, taproot: Segwit version 1 spending rules, jan 2020.

[33] Tiancheng Xie, Jiaheng Zhang, Zerui Cheng, Fan Zhang, Yupeng Zhang, Yongzheng Jia, Dan Boneh, and Dawn Song. zkbridge: Trustless cross-chain bridges made practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 3003–3017, New York, NY, USA, 2022. Association for Computing Machinery.

[34] Alexei Zamyatin, Mustafa Al-Bassam, Dionysis Zindros, Eleftherios Kokoris-Kogias, Pedro Moreno-Sanchez, Aggelos Kiayias, and William J Knottenbelt. Sok: Communication across distributed ledgers, 2019.

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

Appendix A. Background and Notation

A.1. Digital signatures

A digital signature scheme Σ is a tuple of three algorithms: KeyGen, Sign, and Vrfy.

- $(pk, sk) \leftarrow \Sigma.\mathsf{KeyGen}(1^\lambda)$ is a probabilistic, polynomial time (PPT) algorithm that takes a security parameter λ as input and returns a key pair, consisting of a secret (or private) sk and a public key pk.
- $\sigma \leftarrow \Sigma.\mathsf{Sign}(sk, m)$ is a PPT algorithm that takes as input a secret key sk and a message $m \in \{0,1\}^*$, and outputs an authentication tag, or *signature*, σ.
- $\{\mathsf{True}, \mathsf{False}\} \leftarrow \Sigma.\mathsf{Vrfy}(pk, \sigma, m)$ is a deterministic, polynomial time (DPT) algorithm that takes as input a public key pk, a signature σ and a message $m \in \{0,1\}^*$, outputs True iff σ is a valid signature for m generated by the secret key sk, corresponding to pk, i.e., (pk, sk) is a key pair generated by $\Sigma.\mathsf{KeyGen}$. For consistency with Bitcoin Script, we will refer to this as $\mathsf{CheckSig}_{\mathsf{pk}}(\sigma)$, the message being the transaction.

In this work, we make use of signature schemes that are EUF-CMA secure [16].

A.2. Succinct non-interactive arguments (SNARGs)

For this definition, we closely follow, e.g., [15], [17]. Let $\mathcal{R} \leftarrow \hat{\mathcal{R}}(\lambda)$ be a relation generator that takes as input a security parameter λ, and returns a polynomial time decidable binary relation $\mathcal{R}$. We denote ϕ as the statement and w as the witness for the pairs $(\phi, w) \in \mathcal{R}$. We define an efficient publicly verifiable non-interactive argument for $\hat{\mathcal{R}}$ as a tuple of three PPT algorithms: Setup, Prove, and Vrfy.

- $crs \leftarrow \mathsf{SNARG.Setup}(\mathcal{R})$ takes as input a relation $\mathcal{R}$, and outputs a common reference string crs.
- $\pi \leftarrow \mathsf{SNARG.Prove}(\mathcal{R}, crs, \phi, w)$ takes as input a common reference string crs along with $(\phi, w) \in \mathcal{R}$, and returns an argument π.
- $\{\mathsf{True}, \mathsf{False}\} \leftarrow \mathsf{SNARG.Vrfy}(\mathcal{R}, crs, \phi, \pi)$ takes as input a common reference string crs, a statement ϕ, and an argument π and returns True or False, (informally) depending on whether or not π is a valid argument.

$(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Vrfy})$ is a non-interactive argument for $\hat{\mathcal{R}}$ if it has *perfect completeness* and *computational soundness*, as defined in [15], [17]. On a high level, the former means that given any true statement ϕ, an honest prover can convince an honest verifier with overwhelming probability; the latter means that it is not possible to prove a false statement, i.e., convince the verifier if no witness exists, also with overwhelming probability. Finally, a non-interactive argument, where the verifier runs in polynomial time in $\lambda + |\phi|$ and the proof size is polynomial in λ, is denoted as (preprocessing) succinct non-interactive argument (SNARG). Note that while we later use an implementation of [17], we are not interested nor do we utilize the zero-knowledge property or the stronger notion of succinct non-interactive arguments of knowledge (SNARKs).

A.3. 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* [22] Lamp consists of a triple of algorithms $(\mathsf{KeyGen}, \mathsf{Sig}, \mathsf{Vrfy})$, where:

- $(pk_\mathcal{M}, sk_\mathcal{M}) \leftarrow \mathsf{Lamp.KeyGen}(\ell)$ (Algorithm 4), is a 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 a ℓ-bit message. For readability, $\mathcal{M} = \{0,1\}^\ell$ is an alias for the ℓ-bit message space.
- $c_m \leftarrow \mathsf{Lamp.Sig}_{sk_\mathcal{M}}(m)$ (Algorithm 5), is a 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.
- $\{\mathsf{True}, \mathsf{False}\} \leftarrow \mathsf{Lamp.Vrfy}_{pk_\mathcal{M}}(m, c_m)$ (Algorithm 6), is a DPT algorithm parameterized by a public key $pk_\mathcal{M}$ that takes as input a message m, a signature c_m, and outputs True iff c_m is a valid signature for m generated by the secret key $sk_\mathcal{M}$, corresponding to $pk_\mathcal{M}$, i.e., $(pk_\mathcal{M}, sk_\mathcal{M})$ is a key pair generated by $\mathsf{Lamp.KeyGen}$.

Lamport signatures are secure one-time signatures. We write $sk_\mathcal{M}$ and $pk_\mathcal{M}$ to denote the secret key and the

Algorithm 4 The key generation algorithm Lamp.KeyGen for a ℓ-bit messages space, which we shall call $\mathcal{M}$. Throughout these algorithms, we use matrix notation, i.e., for a given two-dimensional matrix a, $a[i,j]$ refers to the element at row i and column j of it.

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

Algorithm 5 The Lamport signature algorithm Lamp.Sig, parameterized over a secret key $sk_{\mathcal{M}}$ for a ℓ-bit sized message space $\mathcal{M}$.

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

Algorithm 6 The Lamport verification algorithm Lamp.Vrfy, parameterized over a public key $pk_{\mathcal{M}}$ for a ℓ-bit message space $\mathcal{M}$.

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

public key associated with the message space $\mathcal{M}$. This key pair can be used to sign any message in $\mathcal{M}$, but once signature c_m is created, the key pair is committed to one specific message $m \in \mathcal{M}$. In other words, as long as only one message $m \in \mathcal{M}$ is signed with a single key pair, no polynomially bounded adversary will be able to forge a signature over another message $m' \neq m$ for that key pair with non-negligible probability.

More concretely, when a party signs a message using a Lamport signature scheme, they reveal, for every bit $m[i]$ of the message, one of the two preimages $x[0,i]$ and $x[1,i]$, with $i = 0, \ldots, \ell - 1$. This means that the signer is claiming that $m[i]$ is either 0 or 1. Notice that committing to an ℓ-bit message m is just the same as making ℓ commitments to 1-bit messages (one for each bit of m).

For a formal definition of one-time security and the proof that Lamport signatures are one-time secure digital signature schemes (assuming the existence of one-way functions), refer to, e.g., [10]. Lamport signatures, and in particular Algorithm 6, is implementable in Bitcoin Script; [35] contains a sample implementation.

Since we leverage Lamport signatures to enable a party to commit to a bit (and thus to a message), from now on, we will refer to the Algorithm 5 as LampComm instead of Lamp.Sig and to the Algorithm 6 as CheckLampComm instead of Lamp.Vrfy.

A.4. Transactions in the UTXO model

We identify a user U on a ledger L by the key pair $(\mathsf{pk_U}, \mathsf{sk_U})$ of a signature scheme Σ, used to prove ownership over coins. We let $\sigma_{\mathsf{U}}(m)$ be the digital signature of U over a message $m \in \{0,1\}^*$. If it is clear what message is signed, we sometimes use σ_{U} as shorthand.

In the *unspent transaction output* (UTXO) model, each transaction output is associated with a coin value (in BTC). An output is defined as an attribute tuple $\mathsf{out} := (a \text{ BTC}, \mathsf{lockScript})$, i.e., it consists of an amount $\mathsf{out}.a \in \mathbb{R}_{\geqslant 0}$ of coins BTC and the condition(s) out.lockScript under which it can be spent. A transaction Tx maps a non-empty list of existing, unspent outputs, to a non-empty list of newly created Tx.outputs. To distinguish them, we refer to the former as Tx.inputs of the transaction. An input, $\mathsf{in} := (\mathsf{PrevTx}, \mathsf{outIndex}, \mathsf{lockScript})$, uniquely identifies one existing output by referencing a transaction PrevTx and an output index outIndex, and is repeating the output's spending condition lockScript for convenience. We will refer to the lists of inputs and outputs of a transaction as *transaction skeleton*.

Formally, we define a transaction as $\mathsf{Tx} := (\mathsf{inputs}, \mathsf{witnesses}, \mathsf{outputs})$, which besides the afore-

mentioned inputs $\mathsf{Tx.inputs} := [\mathsf{in}_1, \ldots, \mathsf{in}_n]$ and outputs $\mathsf{Tx.outputs} := [\mathsf{out}_1, \ldots, \mathsf{out}_m]$ contains the witness data, $\mathsf{Tx.witnesses} := [\mathsf{w}_1, \ldots, \mathsf{w}_n]$, which is the list of the tuples that fulfill the spending conditions of the inputs of the transaction, one witness for each input. The locking script, expressed in the ledger's scripting language, is executed with the corresponding witness as script input. If this execution returns False, the transaction is invalid. If it returns True, the spending condition is fulfilled.

For a transaction to be valid, all witnesses must fulfill the locking condition of their corresponding input; all of the transaction's inputs must be unspent; the sum of the value of the outputs must be smaller or equal to the sum of the value of the inputs. If it is smaller, the difference is given to the miners.

Transaction spending conditions. We are particularly interested in Bitcoin, which has a stack-based scripting language. We now describe a subset of spending conditions supported on Bitcoin that are used in this paper. Each of the following can be combined using logical operators $\wedge$ (and), $\vee$ (or) to create more complex spending conditions.

- **Signature locks.** An output locked with $\mathsf{CheckSig}_{\mathsf{pk_U}}$ can only be spent, if the spending transaction is signed with the secret key corresponding to the key pair $(\mathsf{sk_U}, \mathsf{pk_U})$.
- **Multisignature locks.** To fulfill a multisignature spending condition, a certain number k out of n signatures are required. For example, for users A and B, a 2-of-2 multisignature spending condition is denoted as $\mathsf{CheckMultiSig}_{\mathsf{pk}_{A,B}}$ and the respective signature as $\sigma_{A,B}$.
- **Timelocks** lock a transaction output until a specified time in the future (absolute timelock) or until a specific time after the transaction is included on-chain (relative timelock). We denote the former as $\mathsf{AbsTimelock}(\Delta)$, and the latter as $\mathsf{RelTimelock}(\Delta)$. In the following, we use timelocks in conjunction with other spending conditions. For instance, if the UTXO $\mathsf{Tx.out}_1$ has locking script $\mathsf{lockScript} := \mathsf{RelTimelock}(\Delta) \wedge \mathsf{CheckSig}_{\mathsf{pk_U}}$, the user U can spend the UTXO $\mathsf{Tx.out}_1$ after that a certain amount of time T has passed from the moment that $\mathsf{Tx.out}_1$ has been published on-chain.
- **Taproot Trees** [32], or Taptrees, make a UTXO spendable by satisfying one among multiple spending conditions. The spending conditions are (Tap)leaves of a Merkle tree. To spend a UTXO that has a Taptree as a locking script, a user needs to provide a witness for one of the leaves along with a Merkle inclusion proof of such leaf into the Taptree. In the following, we denote the Tapleaves of a Taptree locking script as $\langle \mathsf{leaf}_1, \ldots, \mathsf{leaf}_r \rangle$; when a user fulfills script leaf_i to unlock the j-th UTXO of the transaction Tx, we write the corresponding input as $(\mathsf{Tx}, j, \langle \mathsf{leaf}_i \rangle)$. Every time that a user spends a UTXO via a Tapleaf of a Taptree, we assume that the user has provided a valid Merkle inclusion proof for the Tapleaf.
- **Other conditions.** We denote with True a condition that is always fulfilled and with False a condition that can never be fulfilled. In the latter case, the coins can not be redeemed, and they are *burnt* instead.

Additionally, we use $*$ to denote a transaction input, witness, or output that can be anything (valid according to Bitcoin consensus rules), but is irrelevant to our protocol.

Combining spending conditions. When we need to express long spending conditions, we explicitly give their pseudocode, combining the spending conditions presented above with other standard language constructions that are expressible in Bitcoin Script, e.g., the *if-then-else* construction. Specifically, when in a long script LongScript we append the Verify keyword to one of its sub-spending conditions, say script, that returns either True or False, we aim to mimic the behavior of the Bitcoin `OP_VERIFY` opcode: if script returns True, pop True from the stack and go on with the rest of the script execution; if script returns False, mark the transaction as invalid (and thus fail to fulfill LongScript).

SIGHASH flags. SIGHASH flags specify which part of the transaction data is included in the hash that is signed as part of a signature lock. These flags are primarily used to help coordinate multiple users in creating and signing a transaction.[13]

- ALL: All inputs and outputs are signed. Transaction is only valid as is.
- NONE: All outputs are signed but no inputs. Any number inputs can be used to fund this transaction.
- SINGLE: All inputs but only one output are signed, i.e., other outputs can be added arbitrarily.

In addition, the ANYONECANPAY flag signs only one input and can be combined with the other flags to create more advanced constructions.

Connector outputs. *Connector Outputs*[14] is a technique to ensure that only one of a given set of Bitcoin

13. See https://en.bitcoin.it/wiki/Contract#SIGHASH_flags. A helpful visualization can be found here: https://tinyurl.com/mr2kshzd

14. For example, described in https://tinyurl.com/2p566ynp.

transactions is valid and can be included in the Bitcoin blockchain. Specifically, we create multiple transactions that require as input the same connector output. We achieve this by introducing an additional input (that references the connector output) to each transaction and set the `SIGHASH_ALL` flag, requiring all inputs to be present for the transaction to be valid. As soon as one of these transactions is included on-chain, spending the connector output, the other transactions become invalid. The connector output can thereby specify a variety of custom spending conditions.

Appendix B. The BITVM2-CORE protocol

Here we present the BITVM2-CORE protocol that implements the interface defined in Definition 2.3. In addition to the prover P and the set of verifiers $\{V_i\}_{i\leqslant n}$, a set of signers $SIG := \{Signer_i\}_{i\leqslant m}$ participate in the protocol. The prover, the verifiers, and the signers are full nodes of the ledger protocol $\Pi_{\mathcal{L}}(\cdot)$ (and, as such, have access to the methods READ and WRITE as defined in Appendix D).

The protocol internally uses a EUF-CMA secure signature scheme $\Sigma := (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Vrfy})$ as the ledger signature scheme. The protocol is parameterized over two positive integers k, ℓ. Participants are given the common reference string crs, result of the $\mathsf{SNARG.Setup}(\mathcal{R})$ procedure. Moreover, participants in the protocol have access to the following subroutines:

- $\mathsf{Chunk}(\Pi, k) \to (f_1, \ldots, f_k)$: breaks a Bitcoin script program Π of length l in k chunks, each of length l/k[15].
- $\mathsf{CreateTx}(TxName, [pk_1, \ldots, pk_t])$: builds the transaction skeleton of TxName as specified by the corresponding equation described in Section 3. Additionally, it takes the optional argument(s) $pk_1, \ldots, pk_t$ and hard-codes them in $TxName$'s locking script.
- $\mathsf{Verify}(\mathcal{S}, \mathcal{T}) \to b$: returns True if the transactions in the sets $\mathcal{S}$ and $\mathcal{T}$ are constructed as expected, i.e., as specified by the corresponding equations in Section 3. Returns False otherwise.

In the protocol, we use the notation $m \hookrightarrow P$ to mean that the active party is sending the message m to party P, and $m \hookleftarrow P$ to mean that the party P is receiving message m. Moreover, the notation $tx \overset{\tau}{\hookrightarrow} \mathcal{L}$ means that the active party calls WRITE$(\cdot)$ method of the ledger protocol with input tx at round τ. The notation $tx \overset{\tau}{\hookleftarrow} \mathcal{L}$, indicates the event of a successful READ(tx) operation for the transaction tx at round τ. For brevity, when a party attempts to write a transaction on the ledger, we explicitly mention only the portion of the transaction witness that is relevant for the protocol logic. For a complete description of every transaction witness, refer to Section 3.

15. if $k \nmid l$, the last chunk will be shorter.

B.1. $\mathsf{Setup}(1^\lambda, \mathcal{R}) \to (\mathcal{T}, \mathcal{S})$:

$\mathbf{Signer_i}$

1) $(pk_{sig_i}, sk_{sig_i}) \leftarrow \Sigma.\mathsf{KeyGen}(1^\lambda)$.
2) $pk_{sig_i} \hookrightarrow Prover$.

Upon $(\mathcal{S}, \mathcal{T}) \hookleftarrow Prover$:

1) If $\mathsf{Verify}(\mathcal{S}, \mathcal{T})$ is True, run $\sigma_1 \leftarrow \Sigma.\mathsf{Sign}(sk_{sig_i}, \mathsf{Payout})$, $\sigma_2 \leftarrow \Sigma.\mathsf{Sign}(sk_{sig_i}, \mathsf{PayoutOptimistic})$, $\sigma_3 \leftarrow \Sigma.\mathsf{Sign}(sk_{sig_i}, \mathsf{Claim})$.
2) $(\sigma_1, \sigma_2, \sigma_3) \hookrightarrow Prover$.
3) Delete secret key sk_{sig_i}.

Prover :

1) $(f_1, \ldots, f_k) \leftarrow \mathsf{Chunk}(\mathsf{SNARG.Vrfy}, k)$.
2) For each $i = 0, \ldots, k$, run $(sk_{z_i}, pk_{z_i}) \leftarrow \mathsf{Lamp.KeyGen}(\ell)$.

Wait until $pk_{sig_i} \hookleftarrow Signer_i$ for all $Signer_i \in SIG$.

1) Run $\mathsf{CreateTx}(\mathsf{Payout}, pk_{sig_1}, \ldots, pk_{sig_m})$, $\mathsf{CreateTx}(\mathsf{PayoutOptimistic}, pk_{sig_1}, \ldots, pk_{sig_m})$, $\mathsf{CreateTx}(\mathsf{Claim}, pk_{sig_1}, \ldots, pk_{sig_m}, pk_{z_0}, \ldots, pk_{z_k})$, $\mathsf{CreateTx}(\mathsf{Assert}, pk_{z_0}, \ldots, pk_{z_k})$, $\mathsf{CreateTx}(\mathsf{Challenge})$, $\mathsf{CreateTx}(\mathsf{Disprove})$.
2) Let $\mathcal{S} := \{\mathsf{Payout}, \mathsf{PayoutOptimistic}\}$ and $\mathcal{T} := \mathcal{S} \cup \{\mathsf{Claim}, \mathsf{Assert}, \mathsf{Challenge}, \mathsf{Disprove}\}$.
3) For each $i = 1, \ldots, m$, $(\mathcal{S}, \mathcal{T}) \hookrightarrow Signer_i$.

Wait until $(\sigma_1^i, \sigma_2^i, \sigma_3^i) \hookleftarrow Signer_i$ for all $Signer_i \in SIG$.

1) Let Payout' be the Payout transaction with the signatures $\sigma_1^1, \ldots, \sigma_1^m$ added as part of the transaction witness. Define $\mathsf{PayoutOptimistic}'$ and Claim' likewise. Let $\mathcal{S}' := \{\mathsf{Payout}', \mathsf{PayoutOptimistic}'\}$ and $\mathcal{T}' := \mathcal{S}' \cup \{\mathsf{Claim}', \mathsf{Assert}, \mathsf{Challenge}, \mathsf{Disprove}\}$.
2) Return $(\mathcal{S}', \mathcal{T}')$.

B.2. $\mathsf{Prove}(\mathcal{R}, Y, w) \to \pi$:

Prover :

1) Run $\mathsf{SNARG.Prove}(\mathcal{R}, crs, Y, w) \to \pi$.
2) Let $z_0 := (\mathcal{R}, crs, Y, \pi)$. Run $z_1 \leftarrow f_1(z_0), z_2 \leftarrow f_2(z_1), \ldots, z_k \leftarrow f_k(z_{k-1})$.
3) If z_k is True, let Claim'' be the Claim' transaction with z_0 added as part of the transaction witness. $\mathsf{Claim}'' \overset{\tau}{\hookrightarrow} \mathcal{L}$.
4) Output π.

B.3. $\mathsf{Verify}(\mathcal{R}, Y, w) \to \{\mathbf{True}, \mathbf{False}\}$**:**

$\underline{\mathbf{Challenger_1}}$:

Upon $\mathsf{Claim}'' \xleftarrow{\tau_1 \leqslant \tau + u} \mathcal{L}$

1) $(f_1, \ldots, f_k) \leftarrow \mathsf{Chunk}(\mathsf{SNARG.Vrfy}, k)$.
2) Read z_0 from the witness of the Claim'' transaction.
3) Run $z'_1 \leftarrow f_1(z_0), z'_2 \leftarrow f_2(z'_1), \ldots, z'_k \leftarrow f_k(z'_{k-1})$.
4) If $z'_k = \mathsf{False}$, $\mathsf{Challenge} \xhookrightarrow{\tau_1} \mathcal{L}$.

Upon $\mathsf{Assert}' \xleftarrow{\tau_3 \leqslant \tau + 3u} \mathcal{L}$:

1) Read $z_0, \ldots, z_k$ from the witness of the Assert' transaction.
2) If there is a z_i such that $z_i \neq f_i(z_{i-1})$, let $\mathsf{Disprove}'$ be the $\mathsf{Disprove}$ transaction with $z_{i-1}, z_i, c_{z_{i-1}}, c_{z_i}$ added as part of the transaction witness. $\mathsf{Disprove}' \xhookrightarrow{\tau_3} \mathcal{L}$. Output **False**.
3) If z_k is False, let $\mathsf{Disprove}'$ be the $\mathsf{Disprove}$ transaction with z_k, c_{z_k} added as part of the transaction witness. $\mathsf{Disprove}' \xhookrightarrow{\tau_3} \mathcal{L}$. Output **False**.

$\underline{\mathbf{Prover}}$:

After round $\tau + \Delta_B$ and $\mathsf{Challenge} \notin \mathcal{L}$:

1) $\mathsf{PayoutOptimistic} \xhookrightarrow{\tau_4 > \tau + \Delta_B} \mathcal{L}$.
2) Output **True**.

Upon $\mathsf{Challenge} \xleftarrow{\tau_2 \leqslant \tau + 2u} \mathcal{L}$:

1) Let Assert' be the Assert transaction with $z_0, c_{z_0}, \ldots, z_k, c_{z_k}$ added as part of the transaction witness. $\mathsf{Assert}' \xhookrightarrow{\tau_2} \mathcal{L}$.

After round $\tau_2 + \Delta_A$ and $\mathsf{Disprove} \notin \mathcal{L}$:

1) $\mathsf{Payout} \xhookrightarrow{\tau_5 > \tau_2 + \Delta_A} \mathcal{L}$.
2) Output **True**.

Appendix C. Covenant Emulation

In this section, we formally define covenant emulation and prove that the method described at a high level in Section 3, "Signer Committee" Subsection (and its algorithm, described in Appendix B) implements this interface. We first need some preliminary definitions.

Definition C.1 (Spending Tree)**.** Given a transaction skeleton $\mathsf{Tx} := ([\mathsf{in}_1, \ldots, \mathsf{in}_n], [\mathsf{out}_1, \ldots, \mathsf{out}_m])$, where $\mathsf{out}_i := (a_i \text{ BTC}, \mathsf{lockScript}_i)$ for all i, we define the *spending tree* of Tx as the tree T of transaction skeletons, where:

(i) Tx is the root of T,

(ii) each child node of Tx is a transaction that spends (one of) the output(s) locking script, for any possible a and $\mathsf{lockScript}$,

(iii) each child node is the root of a spending tree.

In a spending tree with root Tx, we call *spending path* between Tx and an intermediate node Tx' the nodes on the path from Tx to Tx'. By writing $\mathsf{Tx} < \mathsf{Tx}'$, we denote that there is a spending path between Tx and Tx', and by $\mathsf{Tx} \not< \mathsf{Tx}'$ we denote that no such path exists.

Definition C.2 (Covenant Emulation System)**.** Given a transaction skeleton Tx, a *covenant emulation system (CES)* for Tx is a protocol that, on input Tx and a security parameter λ, outputs a tree T' with root Tx, where T' that is a subtree of the spending tree T.

We refer to the tree output of a CES protocol as *covenant tree*. We denote the i-th leaf of the covenant tree T' as $T'.leaf(i)$. A CES is called *restrictive* if, given its output covenant tree, a polynomially bounded adversary cannot publish on-chain a transaction that is in the spending path of the root but not in the spending path of any leaf of the covenant tree, unless with negligible probability.

Definition C.3 (Restrictiveness)**.** A CES is *restrictive* if, for all PPT $\mathcal{A}$, for any execution of the ledger protocol $\Pi_{\mathcal{L}}$, it holds that for any round r after the CES protocol execution:

$$\Pr[\mathsf{Tx}' \in {}^{r}\mathcal{L}^{\cap} : \mathsf{Tx} < \mathsf{Tx}' \wedge \forall i, T'.leaf(i) \not< \mathsf{Tx}' \mid T' \leftarrow CES(\lambda, \mathsf{Tx})] \leqslant \mathsf{negl}(\lambda). \tag{12}$$

In the Setup procedure of the BITVM2-CORE protocol (cf. Algorithm B.1), we implement a Covenant Emulation System. The BITVM2-CORE prover collaborates with a set of signers to create a covenant tree to condition the spending of the Claim transaction. Specifically, the prover creates the transaction skeletons that are the nodes of the covenant tree with root Claim, and sends them to the signers. They receive the transactions and sign some of them only if the covenant tree is constructed as expected. The resulting covenant tree is shown in Figure Fig. 11.

Lemma C.1. *Assuming that the signature scheme Σ is EUF-CMA secure and that there is an honest signer, the procedure described in Algorithm B.1 is a restrictive Covenant Emulation System.*

Proof. Suppose, by contradiction, that there is a PPT adversary $\mathcal{A}$ that violates the restrictiveness property. Namely, there is a ledger protocol instance I and a round r such that, with non-negligible probability, $\mathcal{A}$ manages to publish on the ledger a transaction Tx' such that Tx' is in the spending path of the Claim transaction but not in the spending path of neither Challenge,

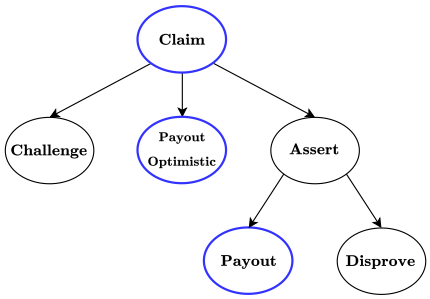

Figure 11. The covenant tree for the Claim transaction built by the Setup procedure of the BITVM2-CORE protocol. Transactions depicted with a blue border are signed by the signing committee during protocol execution.

Disprove, Payout, or PayoutOptimistic. There are two ways this event can occur:

- The adversary generates a transaction skeleton Tx' such that, without loss of generality, Tx' spends Claim's output 0. The adversary sends it to the signers, and they all sign (i.e., add their signature to the witness of Tx'). Then, after the timelock Δ_B (cf. Eq. (6)), $\mathcal{A}$ publishes Tx' on the ledger. In this case, the existential honesty assumption is violated.
- The adversary creates Tx', that, without loss of generality, unlocks Claim's output 0 and forges valid signatures for one or more signer committee members, thus breaking the EUF-CMA security (cf. [16]) of the signature scheme Σ.

In both cases, an assumption is violated, leading to a contradiction. □

Appendix D. Proofs of BITVM2-CORE

In this section, we will prove that the BITVM2-CORE protocol presented in Appendix B satisfies completeness and soundness as defined in Definition 2.4 and Definition 2.5, respectively. For both security properties, we define a game between a challenger and the adversary (either the prover is the challenger and any verifier is the adversary, or vice versa). Then, we show a reduction from winning one game to breaking either one of the ledger protocol properties or the completeness or the soundness of the SNARG system that we use.

During a game execution, the challenger and the adversary may interact, potentially multiple times, with the robust ledger protocol $\Pi_{\mathcal{L}}(\cdot)$, i.e., a ledger protocol that guarantees safety (see Definition 2.1) and liveness (see Definition 2.2). The ledger protocol exposes two functionalities: READ, which checks whether a transaction is included in the ledger (i.e., it is confirmed by all honest parties), and WRITE, which attempts to include a transaction in the ledger (i.e., all honest parties will report the transaction). More specifically:

- READ(tx): when READ is called at time t_{cur}, it returns 1 if the transaction tx has been written to the ledger by that time. Otherwise it returns 0.
- WRITE(tx): when WRITE is invoked at time t_{cur}, it attempts write the transaction tx to the ledger at time t_{cur}.

Before we define the games, we need to lay out some preliminary steps. First, in Algorithm B.1, the Chunk subroutine is used. The Chunk algorithm breaks a Bitcoin script program into k chunks. In the following lemma, we prove that running these chunks sequentially is equivalent to running the original program.

Lemma D.1. *Let $f : X \to X$ be a function computed by a Bitcoin script program $\Pi := (i_1, \ldots, i_l)$. Let Π be partitioned into k consecutive segments of equal length: $\Pi = \Pi_1 || \Pi_2 || \ldots || \Pi_k$, where $||$ denotes the concatenation operator. Each $\Pi_j := (i_{(j-1)\frac{l}{k}+1}, \ldots, i_{j\frac{l}{k}})$ computes a function $f_j : X \to X$. Then for all $x \in X$,*

$$f(x) = f_k \circ f_{k-1} \circ \cdots \circ f_1(x),$$

where $\circ$ denotes the function composition operator.

Proof. Let us model the semantics of Bitcoin Script programs as functions over stack states. Let $\mathsf{Eval}_\Pi : Stack \to Stack$ denote the deterministic execution of program Π on the stack. Let $x \in X$ be an input encoded as an initial stack state $s_0 := \mathsf{push}(x)$, where push is the algorithm that pushes x on the (initially empty) stack. Let $\mathsf{Eval}_\Pi(s_0) = s_k$. Define $f(x) := \mathsf{decode}(s_k)$, where decode takes as input a stack state and returns an output that belongs to X. Define

$$s_1 := \mathsf{Eval}_{\Pi_1}(s_0), s_2 := \mathsf{Eval}_{\Pi_2}(s_1),$$
$$\ldots, s_k := \mathsf{Eval}_{\Pi_k}(s_{k-1}).$$

Since $\Pi = \Pi_1 || \Pi_2 || \ldots || \Pi_k$, we have that $\mathsf{Eval}_\Pi(s_0) = \mathsf{Eval}_{\Pi_k} \circ \cdots \circ \mathsf{Eval}_{\Pi_1}(s_0) = s_k$. Define each $f_j : X \to X$ as $f_j(x) := \mathsf{decode} \circ \mathsf{Eval}_{\Pi_j} \circ \mathsf{push}(x)$. It follows that

$$f(x) = \mathsf{decode}(s_k) = f_k \circ f_{k-1} \circ \cdots \circ f_1(x),$$

which concludes the proof. □

Moreover, we make a key observation that will be crucial in the following security analysis: the adversary cannot interact in an arbitrary way with the ledger protocol through its functionalities, but it is limited by a restrictive covenant emulation system. In our protocol, during the setup phase, the prover and a set of signers collaborate to create a transaction set whose

spending conditions induce a restrictive CES, as shown in Lemma C.1. The adversary cannot write a transaction that creates an output that is not in the covenant tree, except with a negligible probability of success.

Interaction with the ledger. The verifier has access to the ledger's READ and WRITE functionalities. The prover may request the verifier to perform read or write operations on the ledger. We allow the adversary to choose a round $t \in \{t_{cur}, ..., t_{cur}+u\}$ (where t_{cur} is the current round) in which it wants a specific transaction tx to appear on-chain.

Lemma D.2. *Assuming a distributed ledger $\Pi_{\mathcal{L}}$ that is safe (cf. Definition 2.1),* READ *functionality is* monotonic*; that is, for any transaction tx, if there exists a time t_1 such that $\mathsf{READ}(tx) = 1$ when invoked at time t_1, then $\mathsf{READ}(tx) = 1$ for all $t_2 \geqslant t_1$.*

Proof. Assume there exists an execution I in which there is a transaction tx and two rounds t_1 and t_2, such that $t_1 \leqslant t_2$, and the following hold: (i) READ(*tx*) $= 1$ at time t_1, and (ii) READ(*tx*) $= 0$ at time t_2.

Thus, there exists an honest node p such that the transaction tx is included in its local ledger view C_1 at time t_1, but not included in its ledger view C_2 at a later time t_2. This violates the self-consistency of honest parties' views and therefore the safety property of $\Pi_{\mathcal{L}}$. □

Lemma D.3. *Assuming a distributed ledger $\Pi_{\mathcal{L}}$ that is live(u) (cf. Definition 2.2), for every transaction tx that is witnessed by all honest parties in some round t, there exists a round $t' \in \{t, \ldots, t+u\}$ such that* WRITE*(tx) = 1.*

Proof. This follows directly from the liveness property of $\Pi_{\mathcal{L}}$, which guarantees that transactions broadcast to all honest parties are eventually included in the ledger within u rounds. □

D.1. Proofs of BITVM2-CORE

Theorem D.4. *If the SNARG is sound, the ledger is safe and live, and a set of signers and a set of verifiers with existential honesty,* BITVM2-CORE *satisfies soundness (Definition 2.5).*

Proof. Assuming an adversary $\mathcal{A}$ violating the soundness of the Core BitVM, we will construct an adversary $\mathcal{B}$ that, using $\mathcal{A}$, violates either the soundness of the SNARG, or the properties of the underlying distributed ledger protocol $\Pi_{\mathcal{L}}$. To this end, we desctibe the BITVM2-CORE execution as a game between a malicious prover and honest challenger in Algorithm 7. Assume an execution where $\mathcal{A}$ violates soundness. Namely, after a successful setup returning the set of

Algorithm 7 Soundness-Game: Run by the challenger against a PPT and q-bounded adversary $\mathcal{A}$. If the game outputs 1, the adversary wins.

$crs \leftarrow \mathsf{SNARG.Setup}(R)$
$(T, S) \leftarrow \text{Setup}(1^\lambda, \mathcal{R})$
Claim$(Y, \pi), t_1 \leftarrow \mathcal{A}(T)^{TIME()}$ ▷ $\mathcal{A}$ requests to write on-chain Claim(Y, π) at time t_1
if READ(Claim$(Y, \pi)) = 1$ in round $t_1 - 1$ **then**
  Return
end if
if WRITE(Claim$(Y, \pi)) = 1$ in round t_1 **then**
  if $\neg$ SNARG.Vrfy$(\mathcal{R}, crs, Y, \pi)$ **then**
    $t_C \leftarrow \mathcal{A}^{TIME(\leqslant t_1 + \Delta_B)}$
    WRITE(Challenge(Y, π)) in round t_C
  end if
end if
if READ(Challenge$(Y, \pi)) = 1$ in round t_C **then**
  Assert$(Y, \pi), t_2 \leftarrow A(T)$
  WRITE(Assert(Y, π)) in round t_2
end if
if READ(Assert$(Y, \pi) = 1$ in round t_2 **then**
  $t_D \leftarrow \mathcal{A}^{TIME(\leqslant t_2 + \Delta_A)}$
  WRITE(Disprove(Y, π)) in round t_D
end if

At time $t_1 + \Delta_B$:
  PayoutOptimistic(Y, π), $t \leftarrow A(S)^{TIME(> t_1 + \Delta_B)}$
  if (READ(Challenge$(Y, \pi)) = 0 \wedge$ Write(PayoutOptimistic$(Y)) = 1$) in round t **then**
    Output 1
  end if

At time $t_2 + \Delta_A$:
  Payout(Y, π), $t \leftarrow A(S)^{TIME(> t_2 + \Delta_A)}$
  if READ(Disprove$(Y, \pi)) = 1$ in round t **then**
    Output 0
  else if Write(Payout$(Y, \pi)) = 1$ in round t
    Output 1
  else Output 0
  end if

transactions $(\mathcal{T}, \mathcal{S})$ for some NP-relation $\hat{\mathcal{R}}$, there exists a statement $Y \in \mathcal{L}_{\hat{\mathcal{R}}}$ such that:

(i) There is no witness w in the view of $\mathcal{A}$ such that $(Y, w) \in \hat{\mathcal{R}}$.

(ii) A transaction from the set $\mathcal{S}$ is eventually included in the ledger L.

Now, we discuss all the possible ways this event can occur:

- *PayoutOptimistic is on-chain* (line 24): The condition in line 23 is satisfied only if `Challenge` is not committed on-chain. This may occur in one of the following cases:
  - *`WRITE` failure:* This occurs if `WRITE(Challenge)` in round t_c fails (line 10). Since this write was invoked at time t_1 with $t_c\{\} \leqslant t_1 + u \leqslant t_1 + \Delta_B$, and assuming liveness hold such a failure contradicts Lemma D.3.
  - *`READ` failure:* Here, we assume an execution where `READ` is not monotonic. That is, `WRITE(Challenge)` succeeds in round t_c (so `READ(Challenge)` is successful at time t_c), but `READ` fails at time $t > t_1 + \Delta_B$ (line 23). However, this contradicts Lemma D.2.
  - *SNARG soundness violation:* This case arises if the SNARG verification check in line 8 passes despite the absence of a valid witness, thereby violating the soundness of the SNARG system.
- *Payout is on-chain* (line 31): The condition in line 28 fails when `Disprove` is not committed on-chain. This can happen in the following cases:
  - *`WRITE` failure:* This occurs if `WRITE(Disprove)` fails in round t_D (line 19). Since it was invoked at time t_2 and $t_D \leqslant t_2 + u \leqslant t_2 + \Delta_A$, such a failure such a failure contradicts Lemma D.3.
  - *`READ` failure:* Again, we assume an execution where `READ` is non monotonic. Specifically, we assume that `WRITE(Disprove)` is successful in the round t_D (so `READ` is successful in round t_D), but `READ` fails in round $t > t_2 + \Delta_A$ (line 28) contradicting Lemma D.2.

□

Theorem D.5. *Assuming the SNARG is complete, the ledger is safe and live, and a set of signers with existential honesty,* BITVM2-CORE *satisfies completeness (Definition 2.4).*

Proof. Assume, for contradiction, the adversary $\mathcal{A}$ violates the completeness of the Core BitVM. We will construct an adversary $\mathcal{B}$ that, using $\mathcal{A}$, breaks either the completeness of the SNARG, or the properties of the underlying distributed ledger protocol $\Pi_{\mathcal{L}}$. To argue that, we present the BITVM2-CORE execution as a game between an honest prover and a malicious challenger in Algorithm 8.

Assume an execution in which $\mathcal{A}$ succeeds in violating completeness. That is, after a successful setup returning a set of transactions $(\mathcal{T}, \mathcal{S})$ for some NP-relation $\hat{\mathcal{R}}$, there exists a statement $Y \in \mathcal{L}_{\hat{\mathcal{R}}}$ such that:

(i) There exists a witness w in the prover's view such that $(Y, w) \in \hat{\mathcal{R}}$.

(ii) None of the transaction in the set $\mathcal{S}$ gets eventually included in the ledger L.

We now analyze the possible causes of this violation:

- *WRITE(Claim(Y, π)) failure* (line 13): This happens with a negligible probability in the security parameter λ.
- *Assert(Y, π)) is not on-chain in the end of the execution*: This can happen either if WRITE(Assert(Y, π)) fails in round t_2 (line 17), or if READ is not monotonic, i.e., READ(Assert(Y, π)) is successful in round t_2 but fails in the end of the execution. None of these scenarios can occur according to Lemmas D.2 and D.3.
- *READ(Disprove(Y, π)) is successful (line 34):* This occurs if the SNARG verification check in line 21 fails, despite the prover possessing a valid witness w such that $(Y, w) \in \hat{\mathcal{R}}$. In this case, we can construct an adversary $\mathcal{B}$ that breaks the completeness of the SNARG scheme.

□

Appendix E. Light Client Analysis

In this section, we first expose the necessary preliminaries from the Bitcoin backbone protocol analysis [14], then introduce the new definitions required for our analysis, and finally prove security for our on-chain light client protocol.

E.1. Formal Properties and Definitions

The properties of blockchain protocols defined in the backbone model are presented below. Such properties are defined as predicates over the random variable $\text{view}^{t,n}_{\Pi,A,Z}$ by quantifying over all possible adversaries A and environments Z that are polynomially bounded. Note that blockchain protocols typically satisfy properties with a small probability of error in a security parameter κ (or others). The probability space is determined by random queries to the random oracle functionality and by the private coins of all interactive Turing machine instances.

Algorithm 8 Completeness-Game: Run by the challenger against a PPT and q-bounded adversary $\mathcal{A}$. If the game outputs 1, the adversary wins.

Input: λ: security parameter, relation $\mathcal{R}$

$crs \leftarrow \mathsf{SNARG.Setup}(R)$
$(T, S) \leftarrow \mathsf{Setup}(1^\lambda, \mathcal{R})$
$\mathsf{Claim}(Y, \pi), t_1 \leftarrow \mathcal{A}(T)^{TIME()}$ ▷ $\mathcal{A}$ requests to write on-chain $\mathsf{Claim}(Y, \pi)$ at time t_1
if WRITE($\mathsf{Claim}(Y, \pi)$) $= 1$ in round t_1 **then**
  $t_C \leftarrow \mathcal{A}$ ▷ $\mathcal{A}$ requests to write on-chain $\mathsf{Challenge}(Y, \pi)$ at time t_C
  if $t_C \leqslant t_1 + \Delta_B$ **then**
    WRITE($\mathsf{Challenge}(Y, \pi)$) in round t_C
  else
    $t \leftarrow \mathcal{A}^{TIME(\leqslant t_1 + \Delta_B + 1)}$
    WRITE(PayoutOptimistic(Y, π)) in round t
  end if
else Output 1
end if
if READ($\mathsf{Challenge}(Y, \pi)$) $= 1$ in round t_C **then**
  $t_2 \leftarrow \mathcal{A}^{TIME(\leqslant t_1 + \Delta_C)}$
  WRITE($\mathsf{Assert}(Y, \pi)$) in round t_2
end if
if READ($\mathsf{Assert}(Y, \pi)$) $= 1$ in round t_2 **then**
  $Disprove(Y, \pi), t_D \leftarrow \mathcal{A}(T)^{TIME(\leqslant t_2 + \Delta_A)}$
  if $\neg$ $\mathsf{SNARG.Vrfy}(\mathcal{R}, crs, Y, \pi)$ **then**
    WRITE($\mathsf{Disprove}(Y, \pi)$) in round t_D
  else
    $t_3 \leftarrow \mathcal{A}(T)^{TIME(\leqslant t_2 + \Delta_A + 1)}$
    WRITE($\mathsf{Payout}(Y, \pi)$) in round t_3
  end if
end if

At time $t_1 + \Delta_B + 1$:
  if READ(PayoutOptimistic(Y, π)) $= 1$ **then**
    Output 0
  end if

At time $t_2 + \Delta_A$:
  if READ($\mathsf{Disprove}(Y, \pi)$) $= 1$ **then**
    Output 1
  end if

At time $t_2 + \Delta_A + 1$:
  if READ($\mathsf{Payout}(Y, \pi)$) $= 1$
    Output 0
  end if

The *q*-bounded setting. Parties belonging to $\mathcal{P}$, which we call consensus nodes, and the adversary $\mathcal{A}$ have access to an ITM, which we name *random oracle*, that works as follows. Let $H(\cdot) : \{0,1\}^* \to \{0,1\}^k$ be a hash function. The random oracle maintains a table of pairs $(x, y) \in \{0,1\}^* \times \{0,1\}^k$. When a party invokes the random oracle giving as input a string $x \in \{0,1\}^*$, the random oracle outputs the string y to the caller. If x has not been queried before, the random oracle returns a value y selected at random from $\{0,1\}^k$ and stores the value (x, y) in its table.

In each round, each party is allowed to query the random oracle subroutine at most q times. Consequently, in each round, the adversary is allowed to ask at most $t' \cdot q$ queries, where $t' \leqslant t$, the maximum number of parties that the adversary can corrupt during a protocol execution. In the following, we refer to the setting where the parties and the adversary are restricted in the number of queries per round as the *q-bounded synchronous setting*.

Definition E.1 (Common Prefix Property [14])**.** The common prefix property Q_{cp} with parameter $k \in \mathbb{N}$ states that for any pair of honest players P_1, P_2 adopting the chains C_1, C_2 at rounds $r_1 \leqslant r_2$ in $\text{view}^{t,n}_{\Pi,A,Z}$ respectively, it holds that $C_1^{\lceil k \rceil} \preceq C_2$.

Definition E.2 (Chain Quality Property [14])**.** The chain quality property Q_{cq} with parameters $\mu \in \mathbb{R}$ and $\ell \in \mathbb{N}$ states that for any honest party P with chain C in $\text{view}^{t,n}_{\Pi,A,Z}$, it holds that for any ℓ consecutive blocks of C, the ratio of honest blocks is at least μ.

Definition E.3 (Chain Growth Property [14])**.** The chain growth property Q_{cg} with parameters $\tau \in \mathbb{R}$ and $s \in \mathbb{N}$ states that for any honest party P that has a chain C in $\text{view}^{t,n}_{\Pi,A,Z}$, it holds that after any s consecutive rounds, it adopts a chain that is at least $\tau \cdot s$ blocks longer than C.

Closely following [14], we will call a query $q \in \mathbb{N}$ of a party successful if it returns a valid solution to the PoW. For each round i, $j \in [q]$, and $k \in [t]$, we define Boolean random variables X_i, Y_i, and Z_{ijk} as follows. If at round i an honest party obtains a PoW, then $X_i = 1$, otherwise $X_i = 0$. If at round i exactly one honest party obtains a PoW, then $Y_i = 1$, otherwise $Y_i = 0$. Regarding the adversary, if at round i, the j-th query of the k-th corrupted party is successful, then $Z_{ijk} = 1$, otherwise $Z_{ijk} = 0$. Define also $Z_i = \sum_{k=1}^{t} \sum_{j=1}^{q} Z_{ijk}$. For a set of rounds S, let $X(S) = \sum_{r \in S} X_r$ and similarly define $Y(S)$ and $Z(S)$. Further, if $X_i = 1$, we call i a successful round and if $Y_i = 1$, a uniquely successful round. We denote with f the probability that at least one honest party succeeds in finding a PoW in a round.

Definition E.4 (Typical Execution [14])**.** An execution is (ϵ, λ)-typical (or just typical), for $\epsilon \in (0,1)$ and integer $\lambda \geqslant 2/f$, if, for any set S of at least λ consecutive rounds, the following hold.

(a) $(1-\epsilon)\mathbb{E}[X(S)] < X(S) < (1+\epsilon)\mathbb{E}[X(S)]$ and $(1-\epsilon)\mathbb{E}[Y(S)] < Y(S)$.
(b) $Z(S) < \mathbb{E}[Z(S)] + \epsilon\mathbb{E}[X(S)]$.
(c) No insertions, no copies, and no predictions occurred.

Let n be the number of consensus nodes, out of which t are controlled by the adversary. Let Q be an upper bound on the number of computation or verification queries to the random oracle. Let L be the total number of rounds in the execution, and λ, κ security parameters. Finally, we denote with ν the min-entropy of the value that the miner attempts to insert in the chain.

Theorem E.1 (Theorem 4.5 in [14])**.** *An execution is not typical with probability less than*

$$\epsilon_{\mathsf{typ}} = 4L^2 e^{-\Omega(\epsilon^2 \lambda f)} + 3Q^2 2^{-\kappa} + [(n-t)L]^2 2^{-\nu}.$$

Lemma E.2 (Lemma 4.6 in [14])**.** *The following hold for any set S of at least λ consecutive rounds in a typical execution. For $S = \{i : r < i < s\}$ and $S' = \{i : r \leqslant i \leqslant s\}$, $Z(S') < Y(S)$.*

Lemma E.3 (Lemma 4.8 in [14], (aka Patience Lemma))**.** *In a typical execution, any $k \geqslant 2\lambda f$ consecutive blocks of a chain have been computed in more than $\frac{k}{2f}$ consecutive rounds.*

E.2. Light Client Security Definitions

Now, recall the definition of an admissible block [7].

Definition E.5 ((u,k)-Admissible Block at r)**.** Consider $u, k \in \mathbb{N}$. A block B that, at round r, fulfils the following properties is an *admissible block at r*:

- **Safety**: $\mathsf{B} \in \mathcal{C}^{\bigcup}_{r+u}[:-k]$
- **Liveness**: $\mathsf{B} \notin \mathcal{C}^{\bigcap}_{r}[:-k]$

Theorem E.4 (Light Client Security (restated))**.** *For any $m > (\frac{2}{\delta} - 1)x$, our light client protocol accepts a (u,k)-admissible B, where k is the common prefix parameter (Definition E.1), and u the time it takes for the chain of any honest party to grow by more than $m + x$ blocks.*

E.3. Safety proof

Notation. $C^{\bigcup}_r$ denotes the block tree that is the union of the chains in honest parties' views at round r.

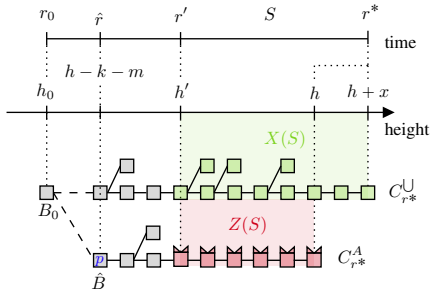

Figure 12. Proof illustration

k denotes the common prefix parameter. Let n be the number of parties, t of which are adversarial. $0 < \delta \leqslant 1$ denotes the honest advantage, i.e., $\frac{t}{n-t} \leqslant 1-\delta$.

Bad scenario. Compare with Fig. 12. For a given m, at some round r, there exists a valid chain C^A_r with some length h and a valid block tree $C^{\bigcup}_r$ with a maximum length of at most $h+x$, which share a common prefix, and there is a block $\hat{B}$ (which contains PegOut), such that $\hat{B} \in C^A_r[:-m-k]$, but $\hat{B} \notin C^{\bigcup}_r$. Denote r^* as the first round, for which this holds.

Proof. Towards a contradiction, suppose that $m > (\frac{2}{\delta} - 1)x$, but safety does not hold, i.e., a bad scenario occurs.

We denote as B_0 the latest block for which $B_0 \in C^{\bigcup}_r$ and $B_0 \in C^A_r[:-m]$. Let r_0 be the round in which B_0 was produced, and h_0 the height of B_0. We observe that C^A_r can contain honestly produced blocks up to a height at most $h_0 + k$, otherwise, the common prefix property (Definition E.1) is violated. We denote h' the height of the honestly produced block in C^A_r with the maximal height, where $h_0 \leqslant h' \leqslant h_0 + k$. Let r' be the round in which $C^A_r[h']$ is produced. Since $C^A_r[h']$ is an honestly produced block, we know that $C^{\bigcup}_{r'}$ has a maximum length of at least h' and that from this round, no honest party will extend $C^A_{r'}$. Let h_d denote $h_0 + k - h'$.

Let us look at the set of rounds $S := \{r', \ldots, r^*\}$. Because we suppose to be in the bad scenario, the number of successful honest rounds $X(S) \leqslant m+x+h_d$, the number of successful adversarial rounds $Z(S) \geqslant m + h_d$, otherwise the honest parties would have won.

Since S contains more than k consecutive blocks, we can apply the patience lemma (Lemma E.3), and thus typicality (Definition E.4) applies. We know that in a typical execution $Z(S) < (1-\frac{\delta}{2})X(S)$ holds [14].

$$m > (\frac{2}{\delta} - 1)x$$

$$\Rightarrow m > (\frac{2}{\delta} - 1)x - h_d$$

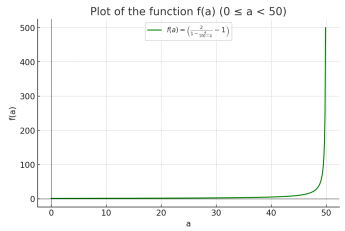

Figure 13. How much m needs to be larger than x as a function of adversarial hashrate a

$$(1-\frac{\delta}{2})x < \frac{\delta}{2}(m+h_d)$$

$$(1-\frac{\delta}{2})(x+m+h_d) < m+h_d$$

$$(1-\frac{\delta}{2})(x+m+h_d)+x < m+x+h_d$$

$$\Rightarrow (1-\frac{\delta}{2})X(S)+x < m+x+h_d$$

$$\Rightarrow Z(S)+x < m+x+h_d$$

$$Z(S) < m+h_d$$

This is a contradiction to $Z(S) \geqslant m+h_d$. □

As a sanity check, we evaluate some concrete numbers. Say $x = 100$ and $\delta = 0.18$ (this corresponds roughly to a 45% adversary, since $\frac{45}{55} \approx 1-0.18$).

$$m > (\frac{2}{1-\frac{45}{55}} - 1)\cdot 100$$

$$m > 1000$$

A comprehensive graph is shown in Fig. 13.

E.4. Liveness Proof

Proof. Liveness follows from the fact that u is defined as the time it takes for the honest chain to grow by at least $m+1$ blocks. Since $\mathsf{B} := \mathcal{C}[-m-k]$, and chain has grown by at least $m+1$ blocks between r and $r+u$, B was not part of $\mathcal{C}_r^{\cup}[:-k]$. □

Appendix F. Bridge Security Analysis

In this section, we will prove that our wrapped asset bridge protocol satisfies bridge safety and bridge liveness, both for PegIn and PegOut (see Definitions 2.9 to 2.11).

Along with bridge safety and bridge liveness, we include an additional desired property for our bridge. We define *Bridge Operator Safety* (Definition F.1), to ensure that operators will not lose money by correctly operating the bridge protocol (even against Byzantine adversaries). This property guarantees that whenever an honest operator fronts money to a client, they can later reclaim it.

To this end, we additionally use the notation $(\mathtt{tx}, p)$ to indicate that the transaction $\mathtt{tx}$ is related to operator p. Specifically, it is paid by p if $\mathtt{tx} \in \mathsf{PegOut}$, and it pays p if $\mathtt{tx} \in \{\mathsf{Payout}, \mathsf{PayoutOptimistic}\}$.

Definition F.1 (Bridge Operator Safety)**.** A bridge protocol $\Lambda_{R_1}^{\text{PegIn}}, \Lambda_{R_2}^{\text{PegOut}}$ is said to be *budget balance complete* with *recovery parameter* $u_\Lambda \in \mathbb{N}$ during the execution I if, for any honest operator p, the following holds:

If there is a transaction $\mathtt{tx} \in \mathsf{PegOut}(\mathtt{id})$ such that $(\mathtt{tx}, p)$ is included in ${}^{r_1}\mathcal{L}_A^{P_1}$ for all honest parties P_1 at some round r_1, then there must exist a transaction $\mathtt{tx}' \in \{\mathsf{Payout}, \mathsf{PayoutOptimistic}\}$ such that for every round $r_2 \geqslant r_1 + u_\Lambda$, $(\mathtt{tx}', p) \in {}^{r_2}\mathcal{L}_A^{P_1}$ for all honest P_1.

F.1. PegIn Bridge

The PegIn bridge $\Lambda_{R_1}^{\text{PegIn}}$ is parameterized by the relation $R_1(\mathtt{id}) = (\{\mathsf{PegIn}(\mathtt{id})\}, \{\mathsf{Mint}(\mathtt{id})\})$, which binds lock transactions on $\mathcal{L}_A$ to minting transactions on $\mathcal{L}_B$. As discussed in Section 5, the responsibility for ensuring the safety and liveness of $\Lambda_{R_1}^{\text{PegIn}}$ lies with the sidechain system.

F.2. PegOut Bridge

We now prove that the PegOut bridge $\Lambda_{R_2}^{\text{PegOut}}$ satisfies both *bridge safety* and *bridge liveness*. The bridge is parameterized by the relation

$$R_2(\mathtt{id}) = \begin{pmatrix} \{\mathsf{PegOut}(\mathtt{id})\}, \\ \{\mathsf{Burn}(\mathtt{id})\} \end{pmatrix},$$

which binds burn transactions on $\mathcal{L}_B$ to corresponding payout transactions on $\mathcal{L}_A$.

In addition, we show that the bridge satisfies *bridge operator safety*: when a PegOut is initiated by an honest operator, the operator will eventually be able to successfully claim their funds.

Theorem F.1 (PegOut Bridge Safety)**.** *The PegOut bridge $\Lambda_{R_2}^{\text{PegOut}}$ satisfies safety with parameter u_s assuming at least one honest signer post-setup, and $\mathcal{L}_B$ satisfies safety.*

Proof. Assume, for contradiction, that a transaction $\mathtt{tx}' \in \mathsf{Pegout}(\mathtt{id})$ is included in $\mathcal{L}_A$ at some round r_2, but no transaction $\mathtt{tx} \in \mathsf{Burn}(\mathtt{id})$ appears in $\mathcal{L}_B$ at round $r_1 \geqslant r_2 - u_s$.

The scenario can occur only for one of the following reasons:

- *Setup failure.* A setup failure occurs if the first input of $\mathtt{tx}' \in \mathsf{Pegout}(\mathtt{id})$, which is supposed to be fixed, exists without a corresponding $\mathsf{PegIn}(\mathtt{id})$ at the time the $\mathsf{Pegout}(\mathtt{id})$ transaction was posted on-chain. This implies that the signer committee incorrectly authorized a $\mathtt{tx}'$ transaction that violates this constraint. Such a failure contradicts either the assumption that committee members act honestly during the setup phase, or the assumption of existential honesty after setup, which ensures that at least one honest signer deletes their signing key and cannot be corrupted dynamically. If this assumption is violated, a fully corrupted committee could effectively re-run the setup phase and authorize arbitrary, invalid transactions.
- *Safety failure of $\mathcal{L}_B$.* A transaction $\mathtt{tx} \in \mathsf{Burn}(\mathtt{id})$ was included in $\mathcal{L}_B$ at some round $r < r_1$, and an operator p initiated the PegOut process, resulting in $(\mathtt{tx}', p)$ appearing in $\mathcal{L}_A$ at round r_2. However, $\mathtt{tx}$ does not persist in $\mathcal{L}_B$ for some round after r_1. Such a failure contradicts the *self-consistency* of honest parties' views and therefore violates the *safety* property of $\mathcal{L}_B$.

□

Theorem F.2 (PegOut Bridge Liveness)**.** *The PegOut bridge $\Lambda_{R_2}^{PegOut}$ satisfies liveness with parameter u_ℓ assuming all signers are honest during setup while at least one remains honest post-setup, $\mathcal{L}_A$ is safe and live, and an honest operator monitors $\mathcal{L}_B$.*

Proof. Let $\mathsf{Burn}(\mathtt{id})$ be included in $\mathcal{L}_B$ at round r_1. By the bridge protocol, an honest operator observing this event is required to initiate the PegOut process at round r_1 by posting a transaction $\mathtt{tx}' \in \mathsf{Pegout}(\mathtt{id})$ to $\mathcal{L}_A$. This should result in some transaction $(\mathtt{tx}', p)$, for some operator p, appearing in $\mathcal{L}_A$ at round $r_1 + u$ where u is the liveness parameter of $\mathcal{L}_A$. This process can fail only in the following ways:

- *Setup failure.* The signer committee failed to construct the transactions as specified during the setup, making it infeasible for an honest operator to post a transaction $\mathtt{tx}' \in \mathsf{Pegout}(\mathtt{id})$. This violates the assumption that all committee members behave honestly during the setup phase.
- *Liveness failure of $\mathcal{L}_A$.* Any honest operator p that submits a transaction $(\mathtt{tx}', p)$ to $\mathcal{L}_A$ fails to have it included within u rounds. This contradicts the liveness property of $\mathcal{L}_A$.
- *Safety failure of $\mathcal{L}_A$.* A transaction $(\mathtt{tx}', p)$, for some operator p, was included in $\mathcal{L}_A$ at round $r_1 + u_\ell$. However, the transaction does not persist in $\mathcal{L}_A$ at some round after $r_1 + u_\ell$. Such a failure contradicts the *self-consistency* of honest parties' views and therefore violates the *safety* property of $\mathcal{L}_A$.

□

Theorem F.3 (Bridge Operator Safety)**.** *The bridge satisfies operator safety with recovery parameter u_Λ, assuming the signer committee is honest during setup and one member remains honest after, an honest challenger exist, both ledgers are live and safe, the light client is secure, and* BitVM2-core *is complete and sound.*

Proof. Assume, for contradiction, that the transaction (tx, p_i) with $tx \in \mathsf{PegOut}(\mathtt{id})$ appears in the ledger of all honest parties in $\mathcal{L}_A$ at round r_1, for some honest operator p_i. However, no transaction $(\mathtt{tx}', p_i)$, where $\mathtt{tx}' \in \mathsf{Payout}(\mathtt{id}), \mathsf{PayoutOptimistic}(\mathtt{id})$, appears in $\mathcal{L}_A$ at round $r_1 + u_\Lambda$.

At most one transaction $\mathtt{tx}' \in \{\mathsf{Payout}(\mathtt{id}), \mathsf{PayoutOptimistic}(\mathtt{id})\}$ can be published. This is because each such transaction has as input a transaction in $\mathsf{PegIn}(\mathtt{id})$ output, which can be spent only once in the UTXO model. Based on this observation, we distinguish the following cases:

Case 1: There exists a transaction $(\mathtt{tx}', p_j)$ in $\mathcal{L}_A$ at some round $r' < r_1 + u_\Lambda$, where $p_j \neq p_i$. The transaction $(\mathtt{tx}', p_j)$ can appear in $\mathcal{L}_A$ without a corresponding $(\mathtt{tx}, p_j)$ for the following reasons:

- *Signer committee violation.* The signer committee incorrectly authorized some transaction related to the BitVM2-core instance during setup—e.g., by pre-signing a transaction that enables a dishonest operator to extract funds from the bridge arbitrarily. This violates either the assumption that the committee behaves honestly during setup, or the assumption of existential honesty after setup which guarantees that at least one honest signer cannot be corrupted and deletes their signing key. If the latter assumption fails, a newly corrupted committee could effectively recreate the setup phase and authorize arbitrary, malicious transactions as if they had been honest from the beginning.
- Suppose p_j knows a pair (ϕ, w) such that $(\phi, w) \in R := \{(\phi, w) : \Pi(\phi, w) = \mathsf{True}\}$ (cf. Algorithm 3), even though no transaction $(\mathtt{tx}, p_j)$ appears in $\mathcal{L}_A$ at round r'. Then p_j must have produced a valid witness w for the statement ϕ, where $w[-m-k]$ includes $(\mathtt{tx}, p_j)$ and also satisfies the light client relation $R := \{(\phi, w) : \Pi_{lc}(\phi, w) = \mathsf{True}\}$. That means that $w[-m-k]$ is an admissible block. However, since this transaction does not appear in n $\mathcal{L}_A$ at round r', this contradicts the safety of

the light client (Definition E.5). By Section E.3, such a violation can occur only with negligible probability.

- *BitVM Core soundness failure.* Suppose none of the above scenarios occur, and thus p_j does not know any pair (ϕ, w) that satisfies the relation $R := \{(\phi, w) : \Pi(\phi, w) = \mathsf{True}\}$. Yet one of the transactions $(\mathtt{tx}', p_j)$ for $\mathtt{tx}' \in \{\mathsf{Payout}(\mathtt{id}), \mathsf{PayoutOptimistic}(\mathtt{id})\}$ is committed to $\mathcal{L}_A$. This violates the soundness of BitVM. Assuming a correct setup, at least one committee member is honest, and an honest challenger participates during execution, such a failure can occur only with negligible probability, as shown in Theorem D.4.

Case 2: No transaction $\mathtt{tx}'$ appears on chain for all rounds $r \leqslant r_1 + u_\Lambda$.

- *Setup failure.* The signer committee did not construct the transactions as specified by the Setup phase of the BITVM2-CORE instance, making it infeasible for an honest operator to reclaim its money. This violates the assumption that all committee members act honestly during the setup phase.
- *Light client liveness failure.* In this scenario, although $(\mathtt{tx}, p_i)$ appears in $\mathcal{L}_A$ at round r_1, p_i does not yet know a pair (ϕ, w) such that $(\phi, w) \in R := \{(\phi, w) : \Pi(\phi, w) = \mathsf{True}\}$ (see Algorithm 3). However, we argue that once the block B containing $(\mathtt{tx}, p_i)$ becomes $m + k$ blocks deep in the chain p_i can construct a valid pair (ϕ, w) as follows.
  In particular, p_i can provide the ledger $\mathcal{L}_A$ as the witness w, and demonstrate that $B = w[-m - k]$ is an admissible block. Since B includes the transaction $(\mathtt{tx}, p_i)$ and satisfies the light client relation $R := \{(\phi, w) : \Pi_{lc}(\phi, w) = \mathsf{True}\}$ with overwhelming probability by the light client liveness property (Definition E.5, Section E.4), the pair (ϕ, w) is valid.
- *BitVM Core completeness failure:* Suppose none of the previously considered scenarios apply, and thus p_i knows a pair (ϕ, w) satisfying the relation $R := (\phi, w) : \Pi(\phi, w) = \mathsf{True}$. Still, no transaction $\mathtt{tx}' \in \{\mathsf{Payout}(\mathtt{id}), \mathsf{PayoutOptimistic}(\mathtt{id})\}$ is committed on-chain within u_c rounds. This scenario contradicts the completeness of BitVM. However, under correct setup and assuming the operator is honest, such a failure can occur only with negligible probability, as guaranteed by Theorem D.5.

In both cases, we reach a contradiction, concluding that if $(\mathsf{PegOut}(\mathtt{id}), p_i)$ appears on $\mathcal{L}_A$, then a corresponding transaction $(\mathtt{tx}', p_i)$ with $\mathtt{tx}' \in \{\mathsf{Payout}(\mathtt{id}), \mathsf{PayoutOptimistic}(\mathtt{id})\}$ must appear on $\mathcal{L}_A$ within $u_\Lambda = u_{\mathcal{L}_A} + u_c$ rounds (it takes $u_{\mathcal{L}_A}$ rounds to submit `Claim` on-chain, and an additional u_c rounds for the BITVM2-CORE instance to complete once the `Claim` is posted.). □