BitVM3: Efficient Bitcoin Bridges via Garbled Circuits

Robin Linus Woll*
Stanford University
Stanford, USA
ZeroSync
Zug, Switzerland
robin@zerosync.org

Ioannis Alexopoulos*
TU Wien
Vienna, Austria
ioannis.alexopoulos@tuwien.ac.at

Lukas Aumayr
University of Edinburgh
Edinburgh, United Kingdom
Common Prefix
Tallinn, Estonia
laumayr@ed.ac.uk

Zeta Avarikioti
TU Wien
Vienna, Austria
Common Prefix
Tallinn, Estonia
georgia.avarikioti@tuwien.ac.at

Matteo Maffei
TU Wien
Vienna, Austria
matteo.maffei@tuwien.ac.at

David Tse
Stanford University
Stanford, USA
Byzantine Research
Palo Alto, USA
dntse@stanford.edu

Abstract

Bitcoin bridges, protocols that lock BTC on Bitcoin and represent it on a secondary system, underpin much of Bitcoin's application layer, yet remain poorly secured. Deployed bridges rely on federated custody with honest-majority assumptions, while BitVM2, the state of the art in trust-minimized bridging, incurs worst-case dispute costs of ~$16,000, requiring large operator bonds and deposits that restrict participation to well-capitalized parties.

We present BitVM3-bridge, a trust-minimized bridge from Bitcoin to (i) chains with finality certificates, such as Ethereum, and (ii) Bitcoin rollups. The bridge is powered by BitVM3-core, a modular abstraction for permissionless off-chain computation on Bitcoin using garbled circuits (GCs). A challenger evaluates a garbled circuit entirely off-chain and obtains a fraud-proof witness if and only if the operator's claim is incorrect. This reduces total on-chain costs to approximately $5, with the challenge transaction itself costing $0.20, a more than 3000× reduction that enables smaller bonds, broader operator participation, and smaller deposit sizes.

We further formalize BitVM3-core as a sound and complete on-chain proof system under standard cryptographic assumptions. Where prior GC-based proposals give informal or construction-specific security arguments, our framework captures existing constructions within a uniform model based on axiomatized security and functional assumptions. Finally, we introduce an on-chain Bitcoin light client secure in the variable-difficulty setting, enabling permissionless chain introspection and thereby the rollup variant of BitVM3-bridge.

1 Introduction

Bitcoin [37] remains the dominant cryptocurrency by market capitalization, yet its base layer offers only limited programmability. To extend Bitcoin with expressive execution capabilities, the ecosystem relies on *bridges* to Layer-1 chains with native programmability, such as Ethereum, as well as on Bitcoin rollups (e.g., EVM-based Layer-2 systems that utilize Bitcoin for data availability [7]). Therefore, bridges become a critical infrastructure component for enabling expressive (e.g., DeFi) applications on Bitcoin: they lock BTC on Bitcoin and represent it on a secondary system as a wrapped asset. As a result, the security of these bridges largely determines the security of Bitcoin's broader application layer.

On expressive chains, trustless bridge designs are possible via *light-client protocols* that verify source-chain consensus directly in a smart contract [3, 40, 43]. Bitcoin's limited scripting, however, precludes this approach: neither a SNARK verifier nor a consensus-signature checker fits within the script budget, and deploying one would require consensus-level changes. The challenge for Bitcoin bridges is therefore to achieve the security of light-client verification without modifying the base protocol.

Without on-chain verification, existing Bitcoin bridges fall back on trust. The dominant design is *federated custody*, in which a group of signers collectively controls the locked BTC. Wrapped Bitcoin (WBTC), with over $10B in total value locked at its peak, is the canonical example. These systems assume an honest majority among custodians, and if enough of them collude or are compromised, user funds can be frozen or stolen. The empirical record bears out the cost of this assumption: bridge exploits have produced more than $2.8B in cumulative losses since 2022 [16], including the Ronin validator-key compromise ($625M, 2022), the Wormhole signature-verification bug ($320M, 2022), and the recent exploit on the Kelp DAO bridge of LayerZero ($292M, 2026) [15].

BitVM-style constructions [7, 35] represented a breakthrough by reducing the trust assumption to *existential honesty*: the safety of locked funds requires only a single honest signer who deletes their key during setup, together with a single honest challenger who monitors the chain during execution. This paradigm has already reshaped the industry landscape, with projects such as Citrea[1], Alpen Labs[2], Babylon[3], and BOB[4] building on BitVM-based verification. In these designs, a set of permissioned operators facilitates the transfer of wrapped assets back to Bitcoin through peg-outs by fronting funds to the user and then reclaiming the locked deposit through an on-chain process that proves, via a light-client proof, that the corresponding burn occurred on the secondary system. This proving phase is optimistic: the operator's claim is accepted unless a challenger submits a counter-proof during a dispute window.

*Both contributed equally and are considered to be co-first authors.

[1] https://citrea.xyz
[2] https://www.alpenlabs.io
[3] https://babylonlabs.io
[4] https://www.gobob.xyz

BitVM2 [35], the state of the art in *permissionless* verification, suffers from two fundamental limitations. First, the cost and complexity of the bridge design are prohibitive: the transaction graph is intricate, and worst-case disputes require multi-megabyte on-chain transactions costing on the order of $16,000 in fees on mainnet [35], forcing operators to post multi-BTC bonds and restricting participation to well-capitalized parties. Second, the Bitcoin rollup bridge presented in BitVM2 assumes static mining difficulty and is therefore not deployable on Bitcoin mainnet[5], where difficulty is recalculated every 2016 blocks [21].

Table 1: Asymptotic comparison of BitVM variants, where C denotes the circuit size and P the proof size.

Protocol	Rounds	On-chain cost	Storage
BitVM1 [2]	$O(\log C)$	$O(\log C + P)$	$O(1)$
BitVM2 [35]	4	$O(P + C)$	$O(1)$
BitVM3 (this work)	2	$O(P)$	$O(C)$

In this work, we address exactly these two gaps. First, we leverage recent developments in garbled circuits [18] to design a simplified bridge architecture powered by BitVM3-core, a verification engine that replaces BitVM2's dispute mechanism with a garbled-circuit-based design. The key insight is that the challenger need not reproduce even a portion of the computation on-chain. Instead, the operator provides a garbled verifier circuit that the challenger evaluates entirely off-chain, and if and only if the claimed computation is incorrect, this evaluation yields a secret that prevents the operator from accessing the funds. This results in a transaction graph that is substantially simpler and an on-chain dispute cost reduction of approximately 3000× (Section 8): the Assert transaction costs roughly $5 and the Disprove fraud proof less than $0.20. The cost reduction has cascading practical consequences: smaller disputes imply smaller bonds, which admit a broader operator set and make smaller deposit denominations viable. Second, we introduce the first introspection protocol for Bitcoin: an on-chain light client that remains secure in the variable-difficulty setting, enabling permissionless chain reasoning on Bitcoin mainnet. With these two components in place, we present BitVM3-bridge, the first trust-minimized bridge architecture that is ready to deploy at scale, connecting Bitcoin to both chains with finality certificates (e.g., Ethereum) and Bitcoin rollups.

Contributions. This paper makes the following contributions:

(1) **BitVM3-bridge.** We design BitVM3-bridge, an end-to-end bridge protocol that securely connects Bitcoin to (i) side systems with finality certificates and expressive scripting languages and (ii) side systems that use Bitcoin as their L1. The protocol is analyzed formally and is the first, to the best of our knowledge, to support a permissionless bridge for Bitcoin rollups in the variable-difficulty setting.
(2) **BitVM3-core.** We formalize BitVM3-core, a modular abstraction for permissionless off-chain computation on Bitcoin based on garbled circuits (GCs), and prove that it constitutes a sound and complete on-chain proof system. Prior GC-based proposals typically provide either informal security arguments or construction-specific formalizations. In contrast, our framework captures existing constructions within a uniform model, offering a generic treatment based on axiomatized security and functional assumptions.
(3) **Bitcoin light client for the variable-difficulty setting.** We construct an on-chain Bitcoin light client that remains secure under variable mining difficulty and is therefore suitable for deployment on Bitcoin mainnet. This is the key component enabling the rollup variant of BitVM3-bridge.
(4) **Implementation and evaluation.** We implement BitVM3-bridge and evaluate on-chain and off-chain costs using the BitVM Alliance circuit [9]. The Assert transaction is ∼2.4 kvB (∼$5) and the Disprove transaction is ∼93 vB (<$0.20), a ∼3000× reduction over BitVM2's worst-case dispute.

[5] Concurrently deployed bridges such as Clementine [7] address variable difficulty but rely on a permissioned set of challengers, forgoing the fully permissionless dispute model that BitVM2 introduced.

2 Model and Protocol Overview

In this section, we describe the model and assumptions underpinning our protocols and provide a high-level overview of BitVM3-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.

2.1 Ledger and Network Model

We assume parties communicate over secure and authenticated channels, and that messages are delivered within a known bounded delay (synchrony). We consider two ledgers, denoted $\mathcal{L}_A$ and $\mathcal{L}_B$. In our main application, $\mathcal{L}_A$ corresponds to Bitcoin, and $\mathcal{L}_B$ to a second-layer or alternative chain. We assume both ledgers have the standard properties (informally): (i) *safety*, i.e., once a transaction is confirmed, it is immutable, and (ii) *liveness*, i.e., each valid transaction is included in the ledger within a known time bound [20].

2.2 UTXO Model

Bitcoin uses a UTXO model, in which coins are represented as *unspent transaction outputs* (UTXOs). A transaction body [tx] consists of *inputs*, which consume existing UTXOs, and *outputs*, which create new ones. Each output specifies a value and a locking script. An input takes the form $(\mathsf{txid}, k, \varphi)$, identifying output k of the transaction with identifier $\mathsf{txid} := \mathcal{H}([\mathsf{tx}])$, locked under condition φ. A complete transaction comprises its body together with a *witness* that provides the data required to satisfy the locking scripts of the consumed inputs. The sum of output values cannot exceed the sum of input values while the difference is paid as a transaction fee.

Bitcoin Script supports standard conditions including signature ($\mathsf{CheckSig}_{\mathsf{pk}_A}$) and multisignature checks ($\mathsf{CheckMultiSig}_{\mathsf{pk}_{A,B}}$), absolute and relative timelocks ($\mathsf{AbsTimelock}(t)$, $\mathsf{RelTimelock}(t)$), and hash-preimage locks ($\mathsf{Hashlock}(h)$). Taproot allows a UTXO to commit to alternative spending conditions arranged in a Merkle tree [42]. To spend a Taproot output, the spender reveals only the executed script leaf, its witness, and a Merkle proof of inclusion. We denote a Taproot output by $\langle \cdot \rangle$. Table 2 illustrates this notation.

Table 2: A transaction in our notation. The input spends output 0 of txid_1, locked under $\mathrm{CheckSig}_{\mathrm{pk}_A}$, using A's signature σ_A. The output is a Taproot UTXO with two spending paths: A can spend it alone by revealing a preimage of h, or A and B can co-sign after a relative delay of 10 blocks.

	Example Transaction
Inputs	(0) $(\mathrm{txid}_1, 0, \mathrm{CheckSig}_{\mathrm{pk}_A})$
Outputs	(0) $\big(5\ \mathrm{BTC},\ \langle\ \mathrm{CheckSig}_{\mathrm{pk}_A} \wedge \mathrm{Hashlock}(h)\ ,\ \mathrm{RelTimelock}(10) \wedge \mathrm{CheckMultiSig}_{\mathrm{pk}_{A,B}}\ \rangle\big)$
Witness	(0) σ_A

2.3 Parties, System, and Threat Model

We consider a bridge protocol that transfers bitcoins, denoted BTC, between the Bitcoin blockchain ($\mathcal{L}_A$) and a secondary ledger ($\mathcal{L}_B$), where the bridged asset on $\mathcal{L}_B$ is represented as wrapped BTC, denoted wBTC. The protocol supports two operations. In a *peg-in*, a user (Alice) locks u units of BTC on $\mathcal{L}_A$, after which u wBTC is issued on $\mathcal{L}_B$. In a *peg-out*, a user (Bob) redeems wrapped coins by burning u wBTC on $\mathcal{L}_B$ and receiving the corresponding BTC on $\mathcal{L}_A$. The redeemed wrapped coins may have originated from an earlier peg-in and subsequently changed hands on $\mathcal{L}_B$.

In addition to the users who perform peg-ins and peg-outs, the protocol involves three classes of parties. We describe each party's role and trust assumptions abstractly here. The cryptographic primitives used to realize these roles, together with their underlying assumptions, are introduced in Section 3.

- **Signer committee ($S_1, \ldots, S_n$).** The signer committee initializes the bridge during a one-time setup phase, fixing the conditions under which wBTC can be minted on $\mathcal{L}_B$ and the corresponding BTC released on $\mathcal{L}_A$. We assume the committee behaves honestly during setup, and that at least one signer honestly deletes its setup-time secrets afterwards.
- **Operators ($O_1, \ldots, O_m$).** Operators monitor both ledgers and execute peg-outs. Upon observing a valid burn of wBTC on $\mathcal{L}_B$, an operator fronts BTC to the user on $\mathcal{L}_A$ using its own liquidity, and is later reimbursed by claiming the corresponding pegged-in BTC. We assume that at least one operator is honest during execution while the remaining operators may be Byzantine.
- **Challengers.** Challengers monitor both $\mathcal{L}_A$ and $\mathcal{L}_B$ and dispute invalid reimbursement claims (i.e., claims that do not correspond to a burn transaction on $\mathcal{L}_B$). This role is permissionless: any party, including a signer, operator, or user, may act as a challenger. We assume that at least one honest challenger is online at all times. All others may be Byzantine.

Cryptographic assumptions. We assume all participants are probabilistic polynomial-time algorithms. Hash functions are modeled as random oracles, digital signatures are EUF-CMA secure, and the cryptographic building blocks introduced in Section 3 satisfy their stated correctness and security properties.

2.4 BitVM3-bridge Protocol Overview

BitVM3-bridge is a trust-minimized bridge between Bitcoin ($\mathcal{L}_A$) and a secondary ledger ($\mathcal{L}_B$), such as Ethereum or a Bitcoin rollup. It represents BTC locked on $\mathcal{L}_A$ as wrapped BTC (wBTC) on $\mathcal{L}_B$, redeemable at a 1:1 ratio. The protocol is powered by BitVM3-core (Section 4), our verification engine that lets an operator commit to a proof on Bitcoin and lets any challenger refute an invalid proof off-chain. We describe the protocol at a high level here (Figure 1), while the two instantiations are given in Sections 6.2 and 6.3.

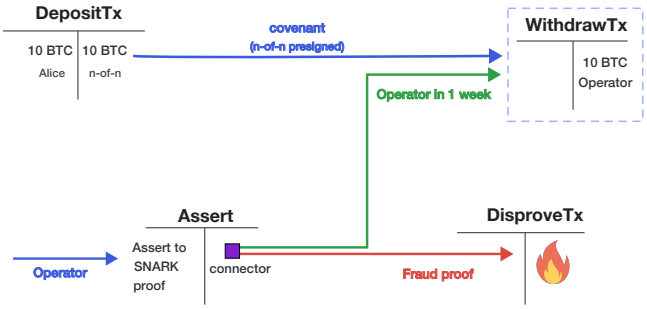

Figure 1: Minimum viable BitVM3-bridge design.

Peg-in. A user locks u BTC on $\mathcal{L}_A$ via a Deposit transaction whose spending conditions are fixed during a one-time setup by the signer committee. Deposit can be unlocked only by an operator that proves a corresponding peg-out occurred. Operators wishing to service the corresponding peg-out run the setup of BitVM3-core (Section 4) jointly with the committee to produce the artifacts that challengers will later use to adjudicate disputes. Once Deposit is confirmed on-chain, the corresponding u wBTC is issued on $\mathcal{L}_B$.

Transfer. After issuance, wBTC circulates freely on $\mathcal{L}_B$ as a standard asset. Any current holder may later initiate a peg-out.

Peg-out. To redeem u wBTC, a user receives $u - f_O$ BTC on $\mathcal{L}_A$, where f_O is the operator fee. The Bitcoin-side payout is fronted by an operator from its own liquidity. The exit mechanism depends on the instantiation: in the Ethereum variant, the user exits through swap markets on Ethereum, while in the rollup variant the user publishes an incomplete PegOut on the rollup, which an operator later completes and broadcasts on $\mathcal{L}_A$.

Operator reimbursement and dispute resolution. After fronting the user's payout, the operator reclaims the corresponding BTC from the bridge reserve via BitVM3-core. The reimbursement flow has three steps:

(1) **Assert.** The operator publishes Assert, committing to a short proof (e.g., SNARK) that both (i) the burn occurred on $\mathcal{L}_B$ and (ii) the corresponding Bitcoin-side payout confirmed on $\mathcal{L}_A$. Assert's witness exposes this proof in a form that any challenger can check off-chain using BitVM3-core.

(2) **Dispute.** Any challenger evaluates the operator's claim off-chain via BitVM3-core. If the proof is invalid, the evaluation yields a secret that the challenger uses to publish Disprove and spend the output of Assert, blocking the operator from reclaiming the reserve.

(3) **Withdraw.** If no valid Disprove is confirmed on-chain before the timelock expires, the operator completes Withdraw using the unspent output of Assert, reclaiming the locked BTC and netting the fee f_O.

The exact predicate proved inside Assert depends on the bridge instantiation: the Ethereum variant verifies an Ethereum finality certificate, whereas the Bitcoin-rollup variant verifies inclusion relative to the Bitcoin light client introduced in Section 5. A successful dispute consumes only the Assert output, not the reserve, so a fraudulent claim does not prevent a later honest operator from being reimbursed. Disputes settle in two on-chain rounds, with on-chain cost dominated by the size of the operator's proof.

2.5 Protocol Goals

We now present the (informal) security goals of BitVM3-core, the Bitcoin light client, and BitVM3-bridge.

BitVM3-core. BitVM3-core is a protocol between a prover P and a permissionless set of verifiers $\mathcal{V}$, executed over the Bitcoin ledger. Its purpose is to ensure that a connector output of a transaction `tx` (e.g., Assert) can be claimed by P after a timeout only when P has included a valid proof for the underlying statement in the witness of `tx`. We require two properties: (i) *completeness* states that an honest prover with a valid witness can always produce the prescribed assertion transaction and recover the connector output after the timeout and (ii) *soundness* states that a malicious prover without a valid witness cannot successfully do so: any such assertion can be refuted by an honest verifier, which obtains a valid dispute witness.

Light client. The Bitcoin light client is secure if it outputs a block that is both guaranteed to appear in the future chains of all honest parties (*safety*) and recent enough to reflect progress of the honest network (*liveness*). We call such a block *admissible*.

BitVM3-bridge. For each cross-chain transfer, identified by a unique identifier `id`, we require the following properties:

- *Bridge Liveness.* If a valid transfer is initiated on the source ledger, then the corresponding transfer is confirmed on the destination ledger within a bounded time.
- *Bridge Safety.* If a transfer appears on the destination ledger, then the corresponding transfer must already have been confirmed on the source ledger.
- *Bridge Operator Safety.* If an honest operator fronts funds to service a valid peg-out, then the operator can recover the corresponding amount from the bridge reserve.

Bridge liveness ensures that honest users who initiate a valid transfer on one ledger eventually receive the corresponding asset on the other. Bridge safety, in turn, ensures that assets cannot be created without a corresponding event on the source ledger, thereby preserving 1:1 supply across the two systems. Together, these two properties imply that the bridge is *robust*.

Our setting is a bidirectional wrapped-asset bridge. A peg-in locks BTC on $\mathcal{L}_A$ and issues the corresponding wrapped asset on $\mathcal{L}_B$, while a peg-out burns the wrapped asset on $\mathcal{L}_B$ and releases BTC on $\mathcal{L}_A$. *Robustness* guarantees that assets can move safely between the two ledgers without violating supply consistency.

Finally, because peg-outs are executed by operators who front their own liquidity to users, practical liveness also requires *Bridge Operator Safety*. This property strengthens peg-out liveness by guaranteeing that honest operators do not incur losses when servicing valid exits, even in the presence of Byzantine adversaries. It is therefore essential not only for correctness, but also for the practical deployability of the bridge.

3 Building Blocks

This section introduces the five cryptographic primitives used throughout the paper. We give the interface and intuition needed to follow the construction here, and defer full algorithmic and security definitions to Section B.

Succinct Non-Interactive Arguments of Knowledge. We use a standard preprocessing SNARK to let an operator prove peg-out validity in a small, constant-size proof. The scheme is defined for NP relations $\mathcal{R} \leftarrow \hat{\mathcal{R}}(\lambda)$, with algorithms Setup, Prove, and Vrfy. On input a relation $\mathcal{R}$, Setup outputs a common reference string crs. On input $(\mathcal{R}, \mathsf{crs}, \phi, w)$ with $(\phi, w) \in \mathcal{R}$, Prove outputs a proof π, and $\mathsf{Vrfy}(\mathcal{R}, \mathsf{crs}, \phi, \pi)$ checks whether π is a valid proof for statement ϕ. We assume *perfect completeness* and *knowledge soundness*, and instantiate the protocol with Groth16 [27]. We do not use the zero-knowledge property.

Garbling Schemes. We use a privacy-free garbling scheme to convert the SNARK verifier into a circuit that challengers evaluate off-chain, off-loading verification work that Bitcoin Script cannot perform on-chain. The scheme $\mathcal{G} = (\mathsf{Gb}, \mathsf{En}, \mathsf{Ev}, \mathsf{De})$ operates over Boolean circuits [8]. On input a function f, the garbler $\mathsf{Gb}(1^\kappa, f)$ outputs a garbled circuit F, encoding information e, and decoding information d. The encoder $\mathsf{En}(e, x)$ maps a plaintext input x to garbled labels L_x, the evaluator $\mathsf{Ev}(F, L_x)$ produces garbled output labels L_y, and the decoder $\mathsf{De}(d, L_y)$ recovers the plaintext output.

We require two standard properties. *Correctness* states that, for every input x, decoding an honestly evaluated garbling yields $f(x)$. *Authenticity* states that no efficient adversary, given F and a valid encoded input, can produce a different garbled output that decodes to a valid plaintext value except with negligible probability.

Garbled Encoding Extractable Signatures. A garbled encoding extractable signature scheme (GS) binds the operator's on-chain commitment to the input of the garbled circuit, so any challenger can recover the encoded proof from the operator's witness. Concretely, GS is a signature scheme tied to the encoding information of a specific garbled circuit. Its key property, *extractability*, is that any valid signature on a message x deterministically reveals the garbled-circuit encoding $\mathsf{En}(e, x)$. Beyond standard *correctness* and *unforgeability*, a secure GS lets a verifier recover the encoded input on which to evaluate the garbled circuit.

A Lamport one-time signature over the input labels of a single garbled circuit is an immediate instantiation of this notion. In this case, the public key consists of the hashes of the two labels corresponding to each input bit, and a signature on x reveals exactly one label per input position. Consequently, GS.Extract returns precisely the garbled encoding $\mathsf{En}(e, x)$.

For consistency with Bitcoin Script, we refer to the on-chain verification predicate as $\mathsf{CheckGS}_{\mathsf{pk}}(\sigma)$. The full algorithmic interface and security definitions appear in Section B.4.

Non-Equivocation Signatures. One-time signature schemes such as Lamport allow us to bind a party to a single value across multiple

transactions. By hardcoding the same public key pk into the locking scripts of two or more transactions, the signer can sign at most one consistent value without revealing conflicting signatures: signing two distinct messages under pk necessarily exposes both preimages for at least one bit position. We denote the on-chain check that verifies a single Lamport signature on message m under public key pk as $\mathsf{LampVerify}_{\mathsf{pk}}(m)$, and reserve $\mathsf{PunishEquiv}_{\mathsf{pk}}(\sigma_1, \sigma_2)$ for the punishment condition unlocked by two conflicting signatures. The light client (Section 5), and therefore the rollup bridge (Section 6.3), uses this to bind operators to a single block hash per checkpoint.

Covenant Emulation. BitVM3-bridge requires restricting how UTXOs may be spent in the future beyond what Bitcoin's standard scripts express. For instance, in the protocol flow of Figure 1, the security of the bridge requires that Withdraw can only spend the bridge reserve (created by Deposit) when accompanied by a spend of a valid Assert. Without this binding, an operator could claim the reserve without ever proving the corresponding peg-out. Bitcoin *covenants* [28, 36, 38] would express such constraints directly, but are not currently deployed.

We emulate covenant semantics in two steps: (i) the locking script of Deposit requires an n-of-n multisignature from the signer committee, and (ii) at setup, the committee signs only the transactions that respect the intended graph (e.g., Withdraw, which must consume a specific valid Assert) under `SIGHASH_ALL`. Because `SIGHASH_ALL` commits each signature to the transaction's exact inputs and outputs, substituting a different Assert would invalidate the signature. We abstract the on-chain check verifying these committee signatures as a script predicate CheckCovenant, used in transaction templates throughout the paper.

4 BitVM3-core

As a stepping stone toward the full bridge construction, we isolate its key verification primitive. As illustrated in Figure 3, an operator must first post an Assert transaction containing a proof of eligibility before withdrawing bridge funds. Since this proof cannot be verified directly on Bitcoin, the primitive must enforce the claim optimistically: if the proof is valid, the operator can proceed to Withdraw after a timelock. If it is invalid, any challenger can refute the claim and prevent the withdrawal during the dispute window.

Prior approaches to optimistic verification on Bitcoin, such as BitVM1's interactive bisection and BitVM2's chunked one-shot dispute, enforce this by re-executing part of the proof verifier on-chain. This incurs costs in script size, state-authentication overhead, and round complexity (Table 1). Our key observation is that on-chain re-execution is unnecessary: the verifier need only be evaluable off-chain in a way whose outcome is enforceable on-chain through a compact witness. Garbling schemes provide exactly this primitive.

4.1 Protocol Flow

BitVM3-core is a protocol between a *prover* P and a permissionless set of verifiers $\mathcal{V}$, called *challengers*. All parties are full nodes of a live and safe ledger $\mathcal{L}$. The protocol uses the SNARK, garbling, and garbled-encoding extractable signature schemes from Section 3, and assumes all parties have access to a deterministic compiler from algorithm descriptions to circuits.

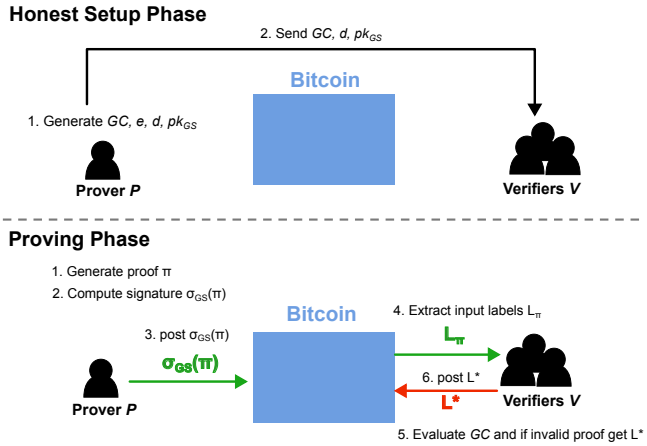

Figure 2: BitVM3-core protocol flow assuming the prover behaves honestly during setup.

The protocol proceeds in two phases, *Setup* and *Proving* (Figure 2). In this section we assume that P behaves honestly during setup but may deviate arbitrarily during the proving phase. We later relax this assumption and describe how to realize the setup phase in a way that tolerates a malicious P.

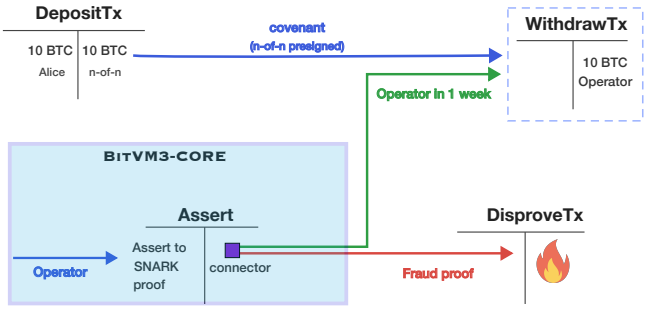

Figure 3: BitVM3-core enforces that the output of Assert **remains unspent only if the operator provides a correct proof.**

Setup. The prover runs the garbler algorithm of the garbling scheme $\mathcal{G}$ on the circuit representing the SNARK verifier for relation $\mathcal{R}$, with the proof π as a free input. The statement ϕ may be hardcoded in the circuit if known at setup time, or supplied as an additional input otherwise. This produces a garbled circuit GC, encoding information e, and decoding information d. From e, the prover derives a key pair $(\mathsf{pk}_{\mathrm{GS}}, \mathsf{sk}_{GS})$ for the garbled encoding extractable signature scheme, and constructs Assert so that spending it requires a valid signature under $\mathsf{CheckGS}_{\mathsf{pk}_{\mathrm{GS}}}$. Let L^* be the output label of the circuit that decodes to False. The output of Assert is a Taproot output with two spending conditions:

(1) *Disprove path:* a hash lock on $\mathcal{H}(L^*)$
(2) *Timeout path:* a relative timelock t, after which the prover may spend the output.

We show the general structure of Assert in Table 3. Finally, the prover shares the garbled circuit GC, the decoding information d,

$\mathsf{pk}_{\mathrm{GS}}$, and the Assert transaction body with the challengers or makes them publicly available, e.g., via a bulletin board.

Table 3: The Assert **transaction template. Parameters marked $*$ are not crucial to the logic of BitVM3-core.**

Assert **Transaction**	
Inputs	(0) $(*,\ *,\ \mathsf{CheckGS}_{\mathsf{pk}_{\mathrm{GS}}})$
Outputs	(0) $\big(*\ \text{BTC},\ \langle \mathsf{Hashlock}(\mathcal{H}(L^*)),\ \mathsf{RelTimelock}(t) \wedge * \rangle\big)$
Witness	(0) $*,\ \sigma_{\mathrm{GS}}$

Proving. The proving phase begins when the prover posts Assert on-chain with a signature σ_{GS} as its witness. By the extractability property of GS, any challenger can recover from this witness a garbled encoding of the claimed proof π, and then evaluate the garbled circuit on it.

Two cases arise depending on whether P is honest or not:

- *Invalid proof.* If the garbled evaluation decodes to False (using decoding information d), then by correctness of the garbling scheme the underlying SNARK verifier, which satisfies knowledge soundness, also rejects π. A challenger can therefore spend the challenge path by publishing Disprove (Table 4) with the false-output label as its witness. This case underpins BitVM3-core *soundness* (Theorem 2).
- *Valid proof.* The garbled evaluation decodes to True by completeness of the SNARK and correctness of the garbling scheme. Then, by authenticity of the garbling scheme, no challenger can produce L^* except with negligible probability. The connector output remains unspent, and once the timelock t expires the prover can claim it via the timeout path. This case underpins BitVM3-core *completeness* (Theorem 1).

Table 4: The Disprove **transaction template. L^* is the output label of the circuit that decodes to** False**.**

Disprove **Transaction**	
Inputs	(0) $\big(\mathsf{Assert},\ 0,\ \mathsf{Hashlock}(\mathcal{H}(L^*))\big)$
Outputs	(0) $(*\ \text{BTC},\ \mathsf{True})$
Witness	(0) L^*

We give a full description of the BitVM3-core protocol in Appendix B.5, Figure 7.

4.2 Setup Verification

The security guarantees of BitVM3-core rely on the setup phase being executed correctly. In particular, three conditions must hold: (i) the garbled circuit GC was generated honestly for the correct function (the SNARK verifier), (ii) the key pair $(\mathsf{pk}_{\mathrm{GS}}, \mathsf{sk}_{GS})$ was derived from the encoding information of that garbling, and (iii) the Assert transaction is well-formed, specifically, its output includes a hash lock on the false-output label, i.e., $\mathsf{Hashlock}(\mathcal{H}(L^*))$.

To enforce these conditions without trusting the prover, we employ standard techniques from the literature. One approach is *cut-and-choose*: the prover generates multiple circuits, a random subset is opened and checked for correctness, and the remaining circuits are used in the protocol[6]. Alternatively, the prover can produce a SNARK or STARK attesting that the setup was performed correctly with respect to the agreed-upon function and parameters.

5 Light Client

In the simplified static population difficulty setting [20], the amount of (honest) work over a given time interval is predictable, as the (honest) hash rate is assumed to be constant. This observation underlies the static on-chain light client of [35]: the operator commits (via a Lamport signature) to a recent block height, whose recency is checked via an absolute timelock on the block height (e.g., `OP_CLTV`[7] in Bitcoin). The operator is then forced to provide a valid chain of headers up to that height.

In the dynamic setting, this construction fails: an adversarial operator may instead provide a chain with the same block height but whose blocks satisfy an artificially reduced difficulty target, thereby requiring significantly less total work. This breaks the soundness of the construction. To overcome this, we need a way to read the difficulty from the chain in addition to the block height.

First attempt. Consider a strawman construction in which the operator commits (via a Lamport/Winternitz signature on-chain) to the epoch-boundary block hash $\mathcal{H}(B_i)$ and timestamp T_i for each epoch i, i.e., every 2016 blocks when Bitcoin adjusts its difficulty. These commitments are then used inside a SNARK proof (verified in BitVM3-core) to (i) reconstruct the difficulty adjustment rule across epochs and (ii) verify that every block satisfies the corresponding proof-of-work target.

This already improves over the static approach, but it remains insecure. An adversarial operator can manipulate timestamp commitments by setting T_i far into the future. Since difficulty adjustment depends on the observed time elapsed between epochs, this inflates the apparent duration and induces the maximum allowed decrease in difficulty, i.e., 4× in Bitcoin. As a result, the adversary can construct a chain whose blocks satisfy a much weaker target, and thus require substantially less work while still appearing valid under the reconstructed difficulty. Concretely, an adversary controlling roughly 20% of the hash rate could keep up with the honest miners.

This illustrates that timestamp commitments must be tied to the blockchain's consensus notion of time, rather than being freely chosen by the operator.

Solution overview. To address this issue, we bind timestamp commitments to consensus time using absolute timelocks. We illustrate the solution in Figure 4. Specifically, when the operator commits to $\mathcal{H}(B_i)$ and T_i, it must spend an output that is locked with an absolute timelock $\mathsf{AbsTimelock}_{1,i}$ set to $T_i - 2$ hours (implemented

[6] Because the set of verifiers is permissionless, the challenge randomness for the cut-and-choose must be drawn from a public randomness beacon or derived jointly from a committee of verifiers via a coin-tossing protocol.

[7] `OP_CLTV` prevents a transaction input from being spent until a specified absolute time or block height, depending on whether the argument passed is less than or greater than 500000000.

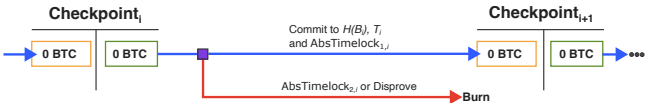

Figure 4: Light-client illustration. AbsTimelock$_{1,i}$ **is set to** $T_i - 2$ hours**, and** AbsTimelock$_{2,i}$ **is set to** $i \cdot 2016 + k + m_t$**.**

by OP_CLTV in Bitcoin Script). Note that the 2 hours account for the fact that Bitcoin blocks may be slightly ahead of the network time.

Crucially, this timelock is evaluated with respect to the consensus time (e.g., median-time-past in Bitcoin) of the blockchain. If the operator sets T_i too far in the future, then consensus time will not yet satisfy the timelock condition, and the operator is unable to advance the checkpoint. If the operator fails to advance before a predetermined liveness deadline, any challenger can instead spend the output via a *burn* path, which invalidates this and all future checkpoints. Thus, any checkpoint that successfully advances must use a timestamp consistent with consensus time, preventing adversarial manipulation of the difficulty up to some ϵ.

We next argue that this mechanism also preserves liveness. An adversary might attempt to prevent progress of an honest operator by manipulating timestamps so as to keep consensus time artificially low. However, since consensus time is derived from a sliding window of recent blocks (e.g., the median of the last 11 timestamps in Bitcoin), maintaining such a delay requires controlling a majority of timestamps within that window. Results on Bitcoin timestamp accuracy [22, Concentrated Chain Quality] imply that within a bounded number of blocks, m_t, the chain will contain a sufficiently large window of honest timestamps. This ensures that consensus time advances past $T_i - 2$ hours, allowing an honest operator to satisfy the timelock and progress.

The liveness deadline is set to allow for this stabilization period before challengers can intervene. In particular, the deadline AbsTimelock$_{2,i}$ takes the form $i \cdot 2016 + k + m_t$, where k is the common prefix parameter [20] after which blocks are considered finalized, and m_t captures the time required for timestamp convergence (Concentrated Chain Quality).

Finally, the operator provides a proof attesting to the validity of the header chain for epoch i, including correct difficulty recalculation. This proof is verified on-chain via a BitVM3-core instance capturing a corresponding SNARK verifier. If the proof is invalid, any challenger can immediately invalidate this and future checkpoints via a *Disprove* path. The formal protocol and transaction structure are given in Appendix D.1 and Figure 14.

6 BitVM3-bridge

BitVM3-bridge instantiates the bridge protocol for two settings: bridging Bitcoin to Ethereum (Section 6.2) and bridging Bitcoin to a Bitcoin rollup (Section 6.3). Both variants share the same setup phase, which we describe first.

6.1 Setup

The bridge is initialized once by an n-of-n committee of signers $S_1, \ldots, S_n$ (Section 2.3). Each signer samples a fresh key pair

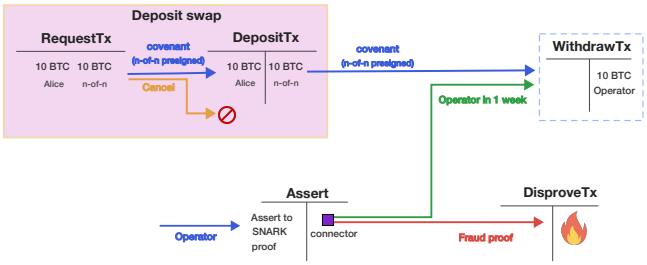

Figure 5: The deposit swap mechanism.

$(\mathsf{pk}_i, \mathsf{sk}_i)$, with the aggregate public key $\mathsf{pk}_S = (\mathsf{pk}_1, \ldots, \mathsf{pk}_n)$ serving as the joint authorization key in CheckCovenant (covenant emulation, Section 3). For each operator that wishes to service peg-outs, the committee performs two tasks:

(1) **Verifying BitVM3-core setup.** The committee runs the BitVM3-core setup procedure (Section 4) as a verifier, checking the correctness of the operator's garbled circuit and Assert, and publishing the circuit so that any challenger can later evaluate it (e.g., via a public bulletin board or torrent).

(2) **Pre-signing** Withdraw**.** The committee pre-signs the corresponding Withdraw transaction under SIGHASH_ALL. Since this signature commits to all inputs and outputs, it binds Withdraw to the specific Deposit being claimed and to the verified Assert from the previous step.

The secret keys $\mathsf{sk}_1, \ldots, \mathsf{sk}_n$ are *ephemeral*: each signer is expected to delete its key after signing. Bridge safety requires only that at least one signer does so honestly.

6.2 Ethereum Bridge

We first instantiate BitVM3-bridge for bridging Bitcoin to Ethereum, using Ethereum as a representative chain whose consensus exposes externally verifiable finality certificates. The same approach extends to other chains with succinctly verifiable finality certificates.

Finality certificates. Ethereum finality is governed by Casper FFG [13], but verifying signatures from the full validator set inside BitVM3-core is prohibitively expensive. We therefore use Ethereum's sync-committee protocol. A rotating committee of 512 validators signs beacon-chain headers using an aggregate BLS signature [10, 11]. Each beacon header contains a state root that commits to the beacon state[8], including the current and next committees.

During setup, the Ethereum variant fixes a *bootstrap state* consisting of a recent beacon header and its current sync committee. Starting from this anchor, the SNARK verified by BitVM3-core checks a sequence of committee handoffs: each committee must be authenticated by a beacon state signed by the previous committee. The circuit then verifies that the target Ethereum block containing the relevant bridge event, such as a wBTC burn, is finalized under the corresponding sync committee.

Peg-in. A user locks u BTC on Bitcoin into the bridge reserve via a deposit-swap protocol coordinated through the signer committee.

[8]The beacon state is Ethereum's consensus-layer state. It contains validator, epoch, and sync-committee data, and commits to the execution-layer state where bridge events such as wBTC burns occur.

Ethereum BitVM3-bridge

Parameters: Bitcoin $\mathcal{L}_A$, Ethereum $\mathcal{L}_B$, u, f_O, $\Delta > u_{\mathcal{L}_A} + u_{\mathcal{L}_B}$, and timeout t.
Predicate: $Q(\pi) :=$ SNARK.Vrfy$(\mathcal{R}, \text{crs}, \phi, \pi)$, where $\mathcal{R}$ is the relation in Algorithm 4 and $\phi = (C_{\text{anchor}}, \text{txid}, \text{sid})$.
Peg-in: setup and deposit swap
1: Alice samples high-entropy s_a, computes $\mathcal{H}(s_a)$, and broadcasts Request locking u BTC on $\mathcal{L}_A$.
2: Operator and signers run BitVM3-core setup: $(\text{sid}, S, \text{aux}) \leftarrow \text{Setup}(1^\kappa, Q)$.
3: Signers verify aux and confirm Request's inclusion.
4: Signers pre-sign Deposit spending Request, and Withdraw spending Deposit and the Assert connector.
5: Alice registers $\mathcal{H}(s_a)$ and the Deposit committee signature on $\mathcal{L}_B$, any party broadcasts Deposit on $\mathcal{L}_A$.
6: The contract starts the delay window Δ.
7: **if** no party relays s_a before Δ expires **then**
8: The contract mints u wBTC to Alice.
Transfer and user exit
9: Users trade wBTC freely on $\mathcal{L}_B$.
10: Operators provide liquidity through atomic swaps: acquire u wBTC and pay the client $(u - f_O)$ BTC on $\mathcal{L}_A$.
Peg-out: execution and reimbursement
11: The operator burns u wBTC on $\mathcal{L}_B$, referencing Deposit.txid and sid.
12: The operator generates a valid SNARK proof π satisfying Q for the burn's finality certificate.
13: Operator and challengers run BitVM3-core.Execute$(\text{sid}, \pi, \text{aux})$ as follows:
14: The operator broadcasts Assert, locking its bond and committing to the garbled proof labels.
15: Challengers evaluate the garbled circuit off-chain.
16: **if** $Q(\pi) = 0$ **then**
17: Challengers obtain L^* and broadcast Disprove before timeout t.
18: **if** no Disprove occurs before t expires **then**
19: The operator broadcasts the pre-signed Withdraw, consuming the Deposit reserve UTXO and Assert connector and reclaiming u BTC.

Figure 6: Compact pseudocode of the Ethereum instantiation of BitVM3-bridge using BitVM3-core. See Figure 15 in Appendix E for the full protocol description.

The user posts a Request transaction on Bitcoin with two spending paths: a deposit path spendable by the n-of-n signer committee, and a cancel path that lets the user reclaim the funds by revealing a secret s_a to prevent denial-of-service. The signers sign a Deposit transaction spending the deposit path, and the user registers the request on Ethereum along with the signature and $\mathcal{H}(s_a)$. Any party can therefore broadcast Deposit on Bitcoin and the Ethereum contract mints u wBTC to the user's address after a delay window Δ. During this window, the user may still cancel by revealing s_a on Bitcoin, which allows anyone to relay the secret to Ethereum and abort the mint. This construction avoids the need for a Bitcoin light client on the Ethereum side during deposits. We illustrate this mechanism in Figure 5 and show the transactions in Table 5.

Peg-out and operator reimbursement. Withdrawals from the bridge are permissioned: only registered operators may redeem BTC directly from the reserve. Regular users wishing to convert wBTC back to BTC therefore do so through swap markets on Ethereum: a user transfers u wBTC to an operator on Ethereum, and in return the operator pays the user $u - f_O$ BTC on Bitcoin, where f_O is the operator service fee. Since operators can later redeem the full u BTC from the reserve, they are naturally incentivized to act as liquidity providers in these markets.

Table 5: The Request **and** Deposit **transactions used in the deposit-swap peg-in mechanism. The** Request **output can either be canceled by revealing s_a or spent by the committee-authorized** Deposit **transaction.**

Request **Transaction**	
Inputs	(0) $(*, *, *)$
Outputs	(0) $(u$ BTC, $\langle \text{Hashlock}(s_a),\ \text{CheckCovenant} \rangle)$
Witness	(0) $*$

Deposit **Transaction**	
Inputs	(0) (Request, 0, CheckCovenant)
Outputs	(0) $(u$ BTC, CheckCovenant)
Witness	(0) Covenant

To redeem u BTC from the reserve, an operator burns u wBTC on Ethereum and proves the burn to Bitcoin via BitVM3-core (Figure 6). The SNARK proof π committed inside Assert verifies an Ethereum finality certificate rooted at the bootstrap state, culminating in a finalized Ethereum block that contains the operator's burn. The burn transaction references the specific Bitcoin deposit UTXO being claimed. The SNARK verifies this binding, preventing the operator from reusing a burn proof against a different deposit.

By BitVM3-core's security properties (Theorems 1 and 2):

- If π is a valid proof, the Assert output remains unspent past the timelock and the operator claims the reserve via the Withdraw (Table 6) pre-signed at step 2 of Section 6.1, which by Section 3 is bound to this specific Deposit and Assert.
- Otherwise, any challenger who evaluates the circuit published at setup obtains a false-output label and posts Disprove, blocking the operator.

6.3 Bitcoin Rollup Bridge

The rollup variant of BitVM3-bridge differs from the Ethereum variant (Section 6.2) in two ways. First, user exits are non-interactive: because the Bitcoin light client of Section 5 is already required for operator reimbursement, no separate swap market between users and operators is needed. Second, operator reimbursement is verified relative to a checkpoint of the on-chain Bitcoin light client rather than an Ethereum finality certificate. The peg-in procedure is identical to that of Section 6.2 and we do not repeat it here.

Peg-out. A user holding u wBTC on the rollup constructs a PegOut transaction (Table 7) paying $u - f_O$ BTC to their Bitcoin address, where f_O is the operator fee. The transaction has two inputs: a fixed, zero-value connector output controlled by the user, and an

Table 6: The Withdraw **transaction used for operator reimbursement. It spends both the bridge reserve output from** Deposit **and the unspent connector output from** Assert**. Hence the operator can withdraw only if no valid** Disprove **consumed the** Assert **output during the challenge window.**

Withdraw **Transaction**	
Inputs	(0) (Deposit, 0, CheckCovenant) (1) $\left(\text{Assert}, 0, \text{RelTimelock}(t) \wedge \text{CheckSig}_{\text{pk}_O}\right)$
Outputs	(0) $(u \text{ BTC}, \text{CheckSig}_{\text{pk}_O})$
Witness	(0) Covenant (1) σ_O

unspecified funding input. The user signs only the first input and the output, fixing the recipient and amount while allowing any operator to complete the transaction with their own funds (e.g., via `SIGHASH_SINGLE|ANYONECANPAY`). The user then burns u wBTC on the rollup, embedding the pre-signed PegOut (e.g., via `OP_RETURN`, a data-carrying output) in the burn transaction.

An operator monitoring the rollup extracts the embedded PegOut, completes it, and broadcasts it on Bitcoin, earning the fee f_O. Because the user's connector output is unique, only one operator can complete a given PegOut, ruling out concurrency race conditions.

Table 7: The PegOut **transaction template.** B **signs the first input and output, while the second input is left unspecified.**

PegOut **Transaction**	
Inputs	(0) $(*, *, \text{CheckSig}_{\text{pk}_B})$ (1) $(*, *, *)$
Outputs	(0) $(u - f_O \text{ BTC}, \text{CheckSig}_{\text{pk}_B})$
Witness	(0) σ_B (1) $*$

Operator reimbursement. After publishing PegOut, the operator recovers funds from the bridge reserve via the bridge BitVM3-core instance established during setup. All peg-out claims for a given deposit by a given operator share the same garbled circuit, regardless of the checkpoint the claim is made against. What varies is the checkpoint hash, which is supplied as part of the public statement rather than fixed in the circuit: for a claim against checkpoint i, the circuit verifies, relative to $\mathcal{H}(B_i)$, that (i) Burn appears in the rollup commitment data stored on the Bitcoin blockchain and (ii) the corresponding PegOut is confirmed on Bitcoin. The remaining public inputs bind the proof to the particular peg-out instance.

Each checkpoint i and peg-out instance j has an associated transaction pair $(\text{Assert}_{i,j}, \text{Withdraw}_{i,j})$, where $\text{Assert}_{i,j}$ consumes output j of Checkpoint_i (cf. Table 10).

Binding the BitVM3-core input to the checkpoint. The bridge BitVM3-core circuit takes two inputs: the proof π and the public statement, which includes the checkpoint hash $\mathcal{H}(B_i)$. These inputs are authorized by separate witness components of $\text{Assert}_{i,j}$:

- *Proof.* Input (0) is locked under $\text{CheckGS}_{\text{pk}_{\text{GS}}}$, the operator's signature σ_{GS} reveals the garbled encoding of π.
- *Public statement.* Input (1) requires a Lamport signature $\sigma^A_{H,i}$ on $\mathcal{H}(B_i)$ under an Assert-specific key $\text{pk}^A_{H,i}$. This signature authorizes the checkpoint input used by the bridge circuit.

The hash supplied to the bridge circuit must match the hash that was committed at checkpoint time, when the light client (Section 5) verified $\mathcal{H}(B_i)$ to be admissible (i.e., recent and stable). $\text{Assert}_{i,j}$ enforces this consistency on-chain: input (1) also requires the Lamport signature $\sigma^C_{H,i}$ used by Checkpoint_i under $\text{pk}^C_{H,i}$, and the script checks equality of the two signed messages ($H_i = H_A$ in Table 8).

Two further on-chain mechanisms ensure that the joint input is well formed:

Equivocation punishment. The equality check above binds $\sigma^C_{H,i}$ and $\sigma^A_{H,i}$ within a single $\text{Assert}_{i,j}$, but does not by itself prevent the operator from signing different checkpoint hashes under $\text{pk}^C_{H,i}$ in different transactions. Since $\text{pk}^C_{H,i}$ is a one-time Lamport key, any two valid signatures under this key on distinct hashes form an equivocation witness (non-equivocation signatures, Section 3). Output 0 of $\text{Assert}_{i,j}$ therefore exposes a PunishEquiv_i branch that any challenger can spend to block $\text{Withdraw}_{i,j}$.

Dispute-window enforcement. The Checkpoint_i output consumed by $\text{Assert}_{i,j}$ also enforces a relative timelock Δ, ensuring that $\text{Assert}_{i,j}$ can be posted only after Checkpoint_i's dispute window closes.

Table 8: $\text{Assert}_{i,j}$ **for peg-out instance** j **under checkpoint** i**.**

$\text{Assert}_{i,j}$ **Transaction (Rollup Variant)**	
Inputs	(0) $(*, *, \text{CheckGS}_{\text{pk}_{\text{GS}}})$ (1) $(\text{Checkpoint}_i, j, \text{CheckSig}_{\text{pk}_O} \wedge \text{RelTimelock}(\Delta) \wedge \text{LampVerify}_{\text{pk}^C_{H,i}}(H_i) \wedge \text{LampVerify}_{\text{pk}^A_{H,i}}(H_A) \wedge H_i = H_A)$
Outputs	(0) $\left(* \text{ BTC}, \left\langle \text{Hashlock}(\mathcal{H}(L^*)), \text{PunishEquiv}_i, \text{RelTimelock}(t) \wedge \text{CheckSig}_{\text{pk}_O} \right\rangle\right)$
Witness	(0) $*, \sigma_{\text{GS}}$ (1) $\sigma_O, \sigma^C_{H,i}, \sigma^A_{H,i}$

7 Security Analysis

We summarize the main security guarantees of BitVM3-core, the Bitcoin light client of Section 5, and the two BitVM3-bridge instantiations of Section 6. Full definitions, formal theorem statements, and proofs are deferred to Section C (BitVM3-core), Section D.2 (light client), and Section E (BitVM3-bridge), respectively.

7.1 BitVM3-core

BitVM3-core instantiates a *ChainVM proof system* [4]. Accordingly, its security is captured by *completeness* and *soundness* (Definitions 4 and 5). Completeness requires that an honest prover holding a witness w for $(\phi, w) \in \mathcal{R}$ can cause the on-chain success predicate S to hold. Soundness requires that whenever S holds in the view of

an honest verifier, an efficient extractor can recover from that view a proof π satisfying the predicate Q given to the parties.

Theorem 1 (Completeness of BitVM3-core, informal). *BitVM3-core is complete in the random oracle model, assuming that the prover follows BitVM3-core honestly during setup and proving, the ledger is safe and live, the SNARK has perfect completeness, the garbling scheme is correct and authentic, the garbled encoding extractable signature scheme is correct and extractable, and the digital signature scheme is EUF-CMA secure.*

Proof sketch. An honest prover holding a valid witness w computes $\pi \leftarrow \mathsf{SNARK.Prove}(\mathcal{R}, \mathsf{crs}, \phi, w)$, which verifies by SNARK completeness. By correctness of the garbling scheme, evaluating GC on $L_\pi = \mathsf{En}(e, \pi)$ decodes to True, so the false-output label is never exposed. By authenticity, no challenger can forge such a label except with negligible probability, and by EUF-CMA security no challenger can spend the timeout path of Assert.

Theorem 2 (Soundness of BitVM3-core, informal). *BitVM3-core is sound in the random oracle model, assuming that the prover follows BitVM3-core honestly during setup but may behave arbitrarily during proving, at least one verifier is honest, the ledger is safe and live, the garbling scheme is correct, the garbled encoding extractable signature scheme is extractable, and the SNARK is knowledge-sound.*

Proof sketch. A malicious prover that publishes Assert with witness σ_π reveals, by extractability of GS, the encoding $L_\pi = \mathsf{En}(e, \pi)$. Since setup is honest, GC correctly implements the SNARK verifier, and by correctness of the garbling scheme, $\mathsf{De}(d, \mathsf{Ev}(GC, L_\pi)) = \mathsf{SNARK.Vrfy}(\mathcal{R}, \mathsf{crs}, \phi, \pi)$. If this value were False, an honest verifier would obtain the false-output label and publish Disprove before timeout. Hence whenever S holds in the view of an honest verifier, the asserted proof must verify. Knowledge soundness of the SNARK then yields a witness w such that $(\phi, w) \in \mathcal{R}$.

7.2 Light Client

The Bitcoin light client outputs a (u, k)-admissible block (Definition 9), namely a block that, after u rounds, appears in every honest party's chain at depth at least k, while not yet being k blocks deep at the time it is accepted.

Theorem 3 (Light Client Security, informal). *For any $m > (2/\delta - 1)x$, the light client of Section 5 accepts a (u, k)-admissible block, where k is the common-prefix parameter, u is the number of rounds in which any honest chain grows by at least $m + x$ blocks, and $x = m_t + k$ is the operator's time advantage over the honest chain. The result holds assuming that the underlying ledger satisfies common-prefix, chain-quality, and chain-growth [20], and that BitVM3-core is sound and complete.*

Proof sketch. The proof proceeds by induction over checkpoints. The base case (Lemma 2) follows from BitVM3-core completeness and soundness applied to the chainstate-proof relation $\mathcal{R}_{\text{lc-btc}}$ (Algorithm 3), implying that any accepted witness is a valid 2017-block segment anchored at the previous admissible block. The induction step (Lemma 4) requires that every block in the witness has approximately correct difficulty, up to ϵ_D (Lemma 3). This follows from the `OP_CLTV`-based timestamp binding: if a malicious operator commits to a timestamp T_i that is too far ahead of real chain time, any honest challenger can slash it, thereby bounding the possible difficulty deviation. Liveness follows because once chain time catches up with T_i, an honest operator can post a valid checkpoint.

7.3 BitVM3-bridge

We analyze BitVM3-bridge in two settings: an Ethereum bridge using Ethereum sync-committee finality certificates (Section 6.2) and a Bitcoin rollup bridge using the on-chain Bitcoin light client of Section 5 (Section 6.3). The two variants share the same peg-in structure. They differ only in the light-client component and in the user-side exit mechanism: the Ethereum variant additionally assumes a secure atomic-swap protocol.

Theorem 4 (Peg-In Safety, informal). *The peg-in bridge $\Lambda_{R_1}^{PegIn}$ satisfies safety with parameter u_s, assuming that the signer committee is honest during setup, there exists an honest party monitoring both ledgers, the digital signature scheme is EUF-CMA secure, and $\Pi_{\mathcal{L}_A}$ and $\Pi_{\mathcal{L}_B}$ are safe and live.*

Intuition. A confirmed mint on $\mathcal{L}_B$ without a corresponding Deposit on $\mathcal{L}_A$ within the safety window would require one of the following: a malformed pre-signature generated during setup, a forged signature authorizing an invalid spend of Request, a ledger reorganization removing Request or Deposit, or failure to relay the cancellation secret s_a within the delay window. These are ruled out respectively by honest setup, EUF-CMA security, ledger safety, and the combination of an honest party with ledger liveness.

Theorem 5 (Peg-In Liveness, informal). *The peg-in bridge $\Lambda_{R_1}^{PegIn}$ satisfies liveness with parameter u_ℓ, assuming that the signer committee is honest during setup, $\Pi_{\mathcal{L}_B}$ is safe and live, and hash functions are modeled as random oracles.*

Intuition. Once Deposit confirms on $\mathcal{L}_A$, the only remaining abort path is a relay of a preimage of $\mathcal{H}(s_a)$ to $\mathcal{L}_B$. A non-canceling user reveals such a preimage only with negligible probability in the random oracle model. Honest setup ensures that the destination-side logic is correctly configured, safety and liveness of $\mathcal{L}_B$ ensures that the corresponding mint is eventually included and persists. A user who has only posted Request also obtains Deposit and the mint, provided the committee signs and an operator runs the setup (Appendix E.2.1).

Theorems 6 and 9 (Peg-Out Safety, informal). *The peg-out bridge $\Lambda_{R_2}^{PegOut}$ satisfies safety with parameter u_s, assuming that the signer committee is honest during setup, at least one signer remains honest after setup, and $\mathcal{L}_A$ and $\mathcal{L}_B$ are safe. The Ethereum variant additionally requires that the underlying atomic-swap protocol is secure.*

Intuition. A violation of peg-out safety implies a confirmed user-side peg-out on $\mathcal{L}_A$ without a corresponding valid transfer on $\mathcal{L}_B$. In the Ethereum variant, this is prevented by the security of the atomic-swap protocol. In the rollup variant, Burn was included and removed, violating $\mathcal{L}_B$ safety.

Theorems 7 and 10 (Peg-Out Liveness, informal). *The peg-out bridge $\Lambda_{R_2}^{PegOut}$ satisfies liveness with parameter u_ℓ, assuming that the signer committee is honest during setup, at least one signer remains honest after setup, $\Pi_{\mathcal{L}_A}$ and $\Pi_{\mathcal{L}_B}$ are safe and live, and at least*

one honest, capitalized operator participates. The Ethereum variant additionally requires that the underlying atomic-swap protocol is secure.

Intuition. In the Ethereum variant, correctness of the atomic swap together with ledger liveness ensures completion of the user-side exit. In the rollup variant, an honest operator broadcasts the user's pre-signed PegOut on $\mathcal{L}_A$, and ledger liveness ensures its inclusion within bounded time.

Theorems 8 and 11 (Operator Safety, informal). *BitVM3-bridge satisfies Operator Safety with recovery parameter u_Λ, assuming that $\Pi_{\mathcal{L}_A}$ and $\Pi_{\mathcal{L}_B}$ are safe and live, the relevant light client is safe and live, the signer committee is honest during setup, at least one signer remains honest after setup, an honest challenger exists, the digital signature scheme is EUF-CMA secure, hash functions are modeled as random oracles, and BitVM3-core is sound (Theorem 2) and complete (Theorem 1).*

Intuition. The reserve can be unlocked only through Withdraw bound to a valid Assert. A malicious operator cannot claim it without either forging a light-client proof or with a fraudulent BitVM3-core assertion, both excluded by the light-client and BitVM3-core guarantees in the presence of an honest challenger. Conversely, an honest operator obtains a valid light-client witness and a valid SNARK proof, and therefore by BitVM3-core completeness and ledger liveness can confirm Assert and Withdraw.

The full statements and proofs appear in Theorems 8 and 11.

8 Evaluation

We implemented BitVM3-bridge and evaluated both its on-chain and off-chain costs. Our prototype uses the garbled circuit implemented by BitVM Alliance [9], which is a Yao-style Boolean circuit of the Groth16 SNARK verifier. We report the transaction sizes of Assert and Disprove, as well as the cost of garbling and evaluating the corresponding circuit off-chain.

8.1 On-Chain Costs

Adaptor signature encoding. The prover commits the SNARK proof on-chain using *adaptor signatures*. The Groth16 proof size is $|\pi| = 128$ bytes ($A, C \in \mathbb{G}_1$ at 32 B each, and $B \in \mathbb{G}_2$ at 64 B). We split the proof into $N = |\pi| = 128$ digits of 8 bits each. For each digit, the witness contains one 65-byte Schnorr adaptor signature. The locking script uses a single public key with `OP_CODESEPARATOR` between successive `OP_CHECKSIGVERIFY` (signature check) operations, binding each signature to its digit position. The optimized script size is $33 + 3(N-1) + 1$ bytes. Since $N = 128$ is below the tapscript stack limit of 1000 items, all signatures fit in a single input.

Assert **transaction.** The Assert transaction has one input, a script-path spend containing N adaptor signatures, the leaf script, and a 33-byte control block, and two P2TR outputs, namely the connector and anchor. Its weight is $W = 4 \cdot B + S_w$, where $B = 137$ bytes is the non-witness size and $S_w = \mathsf{varint}(N{+}2) + 66N + \mathsf{varint}(|s|) + |s| + 34$ is the witness size, with $|s|$ denoting the script length. For $N = 128$, the witness totals 9159 B and the transaction weight is 9707 WU, or approximately 2.4 kvB. At a fee rate of 2 sat/vB, this costs roughly 4853 sats, or about $5 at current prices.

Comparison with Lamport signatures. A naive instantiation using Lamport signatures [31] requires 74 bytes per bit in the leaf script and a 20-byte preimage per bit in the witness. It must also be chunked across multiple inputs due to the tapscript 1000-item stack limit. For a 128-byte proof, this yields approximately 25 kvB—about 10× larger than the adaptor-signature approach.

Disprove **transaction.** The Disprove transaction has one input, a script-path spend revealing a 32-byte hash preimage corresponding to the false-output wire label, and one `OP_RETURN` output that burns the connector. Its total weight is 370 WU, or approximately 93 vB. At 2 sat/vB, this costs fewer than 200 sats (< $0.20), matching the approximately 200-byte fraud-proof cost reported in Table 9.

Comparison with BitVM2. Overall, the on-chain footprint is substantially smaller than BitVM2: whereas BitVM2 disputes can fill an entire 4 MB block, our Assert transaction is approximately 2.4 kvB for a 128-byte Groth16 proof, a reduction of roughly 1700×. The fraud proof itself is only approximately 93 vB, and both transactions remain well below Bitcoin's 100 kvB standard transaction limit.

8.2 Off-Chain Costs

For consistency with the literature, we quote here the same baseline numbers for the BitVM Alliance circuit used in [23] to compare their protocol against.

Per garbled circuit. The BitVM Alliance garbled circuit has 2.7 billion non-free gates. Using half-gates garbling and interpreting this count as half-gates (one 16-byte ciphertext per non-free gate), this yields $2.7 \times 10^9 \times 16$ B $\approx$ 41.2 GB. On a Ryzen 7 7840U (16 cores), the garbling time and the evaluation time are benchmarked to be 353.7 seconds and 352.1 seconds, respectively.

Cut-and-choose setup verification. As discussed in Section 4.2, cut-and-choose can be used to verify the correctness of the setup. A standard cut-and-choose parameter choice is $(181, 7)$, i.e., 7 unopened instances out of a total of 181 instances are kept for evaluation, with the guarantee that with probability at most 2^{-40} all unopened instances are incorrect. (As long as one of the instances is correct, a challenger can evaluate it to open the hash lock.) Under this parameter choice, the total storage is $7 \times 41.2 = 288.4$ GB. With parallelization across the garbling of the 181 instances on the 16 cores, the setup time is found to be 3 hours and 24 minutes. To avoid putting the labels of all 7 finalized instances on chain, Verifiable Shamir Secret Sharing (VSSS) across the labels can be used such that revealing one set of labels on chain is sufficient for a challenger to obtain the labels of all the other 6 instances for evaluation [17].

9 Related Work

Most deployed Bitcoin bridges rely on external key holders: institutional custodians, threshold-signature committees, federations, or HSM-assisted pegs. On expressive chains, trust-minimized alternatives exist via light-client bridges that verify source-chain consensus in a smart contract [3, 12, 40, 43], and PoS sidechains formalize cross-chain asset transfers under committee-based consensus [24]. None of these approaches transfer directly to Bitcoin, whose scripting limitations preclude on-chain verification of the required proofs. XClaim [44] formalized cross-chain claims using collateralized intermediaries that are economically incentivized

Table 9: Comparison of garbled-circuit schemes by cost, storage, compute, and properties. This table was compiled using numbers reported in the respective papers. Computing time is on varying commodity hardware, e.g., Apple Silicon M4 or AMD Ryzen 7 7840U, and is therefore indicative. The third scheme is the one used in our evaluations in Section 8.

Scheme	Proof size	Fraud proof	Off-chain storage	Off-chain compute	Permissionless	Post-quantum
Delbrag [39] + Groth16 [27]	128 B	~200 kB	~80 GB	30 min	yes	no
BitVM3s [34] + Groth16	128 B	~20 kB	~80 GB	30 min	yes	no
Yao-style [8] + Groth16 [9]	128 B	~200 B	~40 GB	6 min	yes	no
Glock [17] + DV-SNARK	128 B	~200 B	~100 MB	10 min	no	no
Argo* + Groth16 [18]	256 B	~200 B	~25 MB (est.)	200 ms	yes	no
BABE + Groth16 [23]	64 B	~200 B	~22 MB	100 ms	no	no
Yao-style + STARK	50–100 kB	~200 B	~10 GB	5 min	yes	yes

*These numbers are estimates reported in the Argo MAC [18] paper.

rather than trusted, but requires the destination chain to verify Bitcoin SPV proofs on-chain, a capability Bitcoin itself lacks in the reverse direction. We therefore focus our comparison on BitVM-style bridges, which achieve trust minimization on Bitcoin through optimistic off-chain computation. The closest prior constructions are BitVM2 [35], Clementine [7], and the recent Cardinal [4]. All three share BitVM3-bridge's high-level optimistic reimbursement architecture but differ along three axes, which we discuss in turn.

Verification engine. BitVM [2, 32] introduced optimistic computation on Bitcoin via pre-signed challenge–response transaction graphs, resolving disputes by an $O(\log C)$-round binary search over an execution trace with a designated verifier online throughout. BitVM2 removes the designated-verifier bottleneck by committing intermediate verifier states on-chain, enabling any challenger to dispute a single transition; Clementine retains this mechanism. In both cases, a dispute re-executes a chunk of the computation in Bitcoin Script, with intermediate states authenticated via Lamport-style one-time signatures, costing roughly $16,000 in worst-case mainnet fees [35]. BitVM3-core moves verifier execution off-chain: a challenger evaluates the garbled verifier locally and posts a compact fraud witness if the claim is invalid, reducing the challenge transaction to ~93 vB and the full on-chain dispute path to ~$5.

Several prior constructions use garbled circuits to move BitVM-style verification off-chain. We instantiate BitVM3-core with the Yao-style Boolean garbled circuit for the Groth16 verifier implemented by the BitVM Alliance [9]. Early work expressed Boolean functions in Bitcoin contracts via Discrete Log Contracts [33]. Delbrag [39] carried this approach into the BitVM setting, and BitVM3s [34] reduced dispute footprints from megabytes to tens of kilobytes at the cost of ~80 GB of off-chain storage. Subsequent work improves either the verifier being garbled or the garbling scheme itself. Glock [17] garbles a designated-verifier SNARK over a binary curve, shrinking off-chain data to ~100 MB at the expense of permissionless challenging. OHMG [19] introduces arithmetic-oriented garbling via one-hot encodings. Argo [18] introduces homomorphic elliptic-curve MACs and reports over 1000× reduction in garbling table size relative to Yao-style approaches. This yields an estimated ~25 MB circuit, though no public Groth16 verifier implementation is available. BABE [23] combines the Argo MAC primitive with linear witness encryption to achieve over 1000× improvement in Groth16 verification, but restricts dispute to a permissioned challenger set. DutyFree Bits [30] reduces the off-chain data of Argo-style constructions to under 1 MB. BitPriv [1] extends the same enforcement pattern beyond proof verification to privacy-preserving Bitcoin DeFi. Table 9 summarizes this design space.

Our contribution differs in scope. The constructions above improve specific dimensions of the garbled circuit enforcement primitive, whereas BitVM3-core is a modular on-chain proof system that abstracts this pattern, separating the bridge protocol and its security proof from the concrete garbling backend. We adopt the ChainVM proof-system abstraction formalized by Cardinal [4] and prove that BitVM3-core is a secure instantiation, so any advance in the garbling layer can be incorporated as a drop-in backend without modifying the bridge protocol or its security proofs.

On-chain light client. The light client embedded in BitVM2 is analyzed in the static-difficulty setting, where the target is not recalculated every 2016 blocks as in the mainnet Bitcoin protocol. Clementine addresses Bitcoin's dynamic-difficulty setting and proves security against adversaries controlling less than half of Bitcoin's hash rate, but its light-client protocol relies on a permissioned set of verifiers. Our light client is, to our knowledge, the first on-chain Bitcoin light client for the variable-difficulty setting with permissionless challenging, and is the key primitive enabling the rollup variant of BitVM3-bridge.

Bridge complexity and operator economics. BitVM2-style designs require multiple auxiliary optimistic paths to avoid invoking the costly verification path in the common case. This leads to a complex pre-signed transaction graph and large operator collateral requirements. BitVM3-bridge's verification path is itself cheap, so these auxiliary paths are unnecessary. Clementine addresses operator-collateral cost from a complementary angle: it introduces payoff rounds that allow operator collateral to be reused across deposits and enable a single successful challenge to slash multiple misbehaviors. This optimization is orthogonal to ours and the two could be composed in a future deployment.

10 Discussion and Limitations

Challenger costs and dispute data availability. BitVM3-core moves verification off-chain, so the work that BitVM2 pays for in Bitcoin fees is instead borne off-chain by whoever plays the challenger role: a challenger must hold the garbled circuits published at setup, and evaluate the disputed claim within the challenge window. The storage requirement is not trivial for the backend used in

our evaluation (Section 8), though it is a property of the garbling backend rather than of BitVM3-bridge. Evaluation time is not a constraint: roughly six minutes against a challenge window of days. Our model already assumes that the signer committee behaves honestly during setup (Section 6.1), and the committee verifies the operator's garbled circuit as part of that setup, so the signers are in a position to serve this data. In practice the requirement is weaker still: the circuit can be distributed over a public bulletin board or torrent, and any party that has obtained it can reseed it, so availability does not depend on any particular signer remaining online. Stakeholders of the side system have a direct interest in reseeding, since the data is what allows an invalid claim against their reserve to be refuted.

Challenger incentives. Our analysis assumes an honest challenger is online at all times, but does not model why a party is incentivized to take on that role. The natural candidate is a holder of wBTC on $\mathcal{L}_B$. An invalid reimbursement claim attempts to spend the BTC backing their wrapped holdings, so a holder who has verified the provenance of the relevant Deposit has a direct stake in monitoring it. The on-chain cost of challenging is low, as a successful Disprove is approximately 93 vB (< \$0.20), but requires maintaining off-chain data as discussed above. Designing an explicit reward mechanism that compensates challengers for the off-chain storage and evaluation costs of the garbling backend (Section 8), in the spirit of watchtower and optimistic-rollup dispute mechanisms [6, 29], is orthogonal to our construction and remains future work.

Peg-out denominations. Each Deposit is a single UTXO bound at setup to a specific Withdraw, so a peg-out redeems the full deposited amount (minus the operator fee f_O). BitVM3-bridge therefore does not support redeeming part of a deposit. A user who wants finer-grained liquidity must peg in at a finer granularity and peg out only the deposits they need. Under BitVM2's cost structure this was a substantial burden, since an operator had to post collateral covering a worst-case dispute of roughly \$16,000 for each denomination it served. Our reduction of dispute costs to approximately \$5 makes smaller denominations practical, which mitigates but does not remove the restriction. Supporting native amount-splitting would require a different transaction structure, which we leave open.

Reliance on the signer committee. The most significant limitation of our construction, shared with all BitVM-family designs, is the reliance on the signer committee during setup. Safety rests on existential honesty (one honest signer deletes their key), but liveness requires the committee to be available whenever a new operator is onboarded or a new deposit is registered. A committee that goes offline can prevent new bridge activity even when existing deposits remain safe. Mitigating this requires either a rotating committee with well-defined handoff procedures, a rationally incentivized committee whose members are bonded against unavailability, or a cryptographic construction that eliminates the setup committee altogether. We view rational and dynamic committee designs as the most pressing direction for future work.

11 Conclusion

Until recently, trust-minimized Bitcoin bridging was not known to be achievable without changes to Bitcoin's consensus rules: the base layer's limited scripting is unable to verify the light-client proofs that trust-minimized bridges require. The BitVM line of work changed this, showing that optimistic off-chain computation can extend Bitcoin's expressiveness without a fork. BitVM2 turned the idea into a concrete bridge, but remained impractical at scale: its dispute mechanism is costly and complex, and its rollup bridge assumes static mining difficulty.

BitVM3-bridge addresses both limitations. By replacing on-chain verifier execution with off-chain garbled-circuit evaluation, BitVM3-core reduces the worst-case dispute from multi-megabyte transactions to roughly 2.4 kvB (approximately \$5 at current fee rates), an improvement of over three orders of magnitude that simultaneously simplifies the transaction graph and settles disputes in two on-chain rounds rather than four. Our on-chain Bitcoin light client, secure in the variable-difficulty setting, completes the picture by enabling permissionless chain introspection on Bitcoin mainnet. Together, these components yield the first trust-minimized bridge architecture that supports both chains with finality certificates and Bitcoin rollups while retaining permissionless challenging. Alongside the bridge, we formalize BitVM3-core as a ChainVM proof system with standard completeness and soundness guarantees, providing a modular framework into which future garbling backends can be plugged as drop-in instantiations. A major practical implication of the cost reduction is economic: smaller worst-case disputes mean smaller operator bonds, which lowers the capital barrier to participation and admits smaller deposit denominations.

Acknowledgments. This work was partially supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement 101141432-BlockSec), by the Austrian Science Fund (FWF) through the SFB SpyCode project (F8510-N and F8512-N), by the Vienna Science and Technology Fund (WWTF) through the projects 10.47379/ICT25056 and 10.47379/ICT22045, and by the Austrian Research Promotion Agency (FFG) through the SBA-K1 NGC COMET center. The research of Robin Linus Woll and David Tse was supported by Input Output Global through the Stanford IOG Blockchain Hub. Lukas Aumayr was supported by Input Output through their funding of the Edinburgh Blockchain Technology Lab.

We thank the BitVM Alliance for implementing the garbled circuit used in this work. We also thank Ekrem Bal for helpful discussions and contributions to an early design of the light client. This paper was edited for grammar using Claude (Anthropic).

References

[1] Ioannis Alexopoulos, Zeta Avarikioti, Paul Gerhart, Matteo Maffei, and Dominique Schröder. 2025. BitPriv: A Privacy-Preserving Protocol for DeFi Applications on Bitcoin. Cryptology ePrint Archive, Paper 2025/1575. https://eprint.iacr.org/2025/1575

[2] Lukas Aumayr, Zeta Avarikioti, Robin Linus, Matteo Maffei, Andrea Pelosi, Christos Stefo, and Alexei Zamyatin. 2024. BitVM: Quasi-Turing Complete Computation on Bitcoin. Cryptology ePrint Archive, Paper 2024/1995. https://eprint.iacr.org/2024/1995

[3] Lukas Aumayr, Zeta Avarikioti, Matteo Maffei, Giulia Scaffino, and Dionysis Zindros. 2025. Blink: An Optimal Proof of Proof-of-Work. In *Financial Cryptography*.

[4] Lukas Aumayr, Jesus Diaz, Dimitar Jetchev, and Aggelos Kiayias. 2025. Cardinal: Bridging Bitcoin with Ownership Preservation. Cryptology ePrint Archive, Paper 2025/2196. https://eprint.iacr.org/2025/2196

[5] Lukas Aumayr, Xinshu Dong, Runchao Han, Srivatsan Sridhar, David Tse, and Nikhil Vanjani. 2026. Trustless Bitcoin Vaults. Cryptology ePrint Archive, Paper 2026/2466. https://eprint.iacr.org/2026/2466

[6] Zeta Avarikioti, Orfeas Stefanos Thyfronitis Litos, and Roger Wattenhofer. 2020. Cerberus Channels: Incentivizing Watchtowers for Bitcoin. In *Financial Cryptography and Data Security (FC 2020)*. Springer, 346–366.

[7] Ekrem Bal, Lukas Aumayr, Atacan İyidoğan, Giulia Scaffino, Hakan Karakuş, Cengiz Eray Aslan, and Orfeas Stefanos Thyfronitis Litos. 2025. Clementine: A Collateral-Efficient, Trust-Minimized, and Scalable Bitcoin Bridge. https://citrea.xyz/clementine_whitepaper.pdf.

[8] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. 2012. Foundations of garbled circuits. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, Ting Yu, George Danezis, and Virgil D. Gligor (Eds.). ACM, 784–796. doi:10.1145/2382196.2382279

[9] BitVM. 2025. Garbled SNARK Verifier. https://github.com/BitVM/garbled-snark-verifier. GitHub repository. Last accessed: 2026-01-11.

[10] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. 2003. Aggregate and verifiably encrypted signatures from bilinear maps. In *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 416–432.

[11] Dan Boneh, Ben Lynn, and Hovav Shacham. 2001. Short signatures from the Weil pairing. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 514–532.

[12] Benedikt Bünz, Lucianna Kiffer, Loi Luu, and Mahdi Zamani. 2019. Flyclient: Super-Light Clients for Cryptocurrencies. *IACR Cryptol. ePrint Arch.* 2019 (2019), 226. https://eprint.iacr.org/2019/226

[13] Vitalik Buterin and Virgil Griffith. 2017. Casper the Friendly Finality Gadget. arXiv:1710.09437. https://arxiv.org/abs/1710.09437

[14] R. Canetti. 2001. Universally composable security: a new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*. 136–145. doi:10.1109/SFCS.2001.959888

[15] Chainalysis Team. 2026. Inside the KelpDAO Bridge Exploit: How ~$292 Million in rsETH Was Released Against a Non-Existent Burn. Chainalysis Blog. https://www.chainalysis.com/blog/kelpdao-bridge-exploit-april-2026/.

[16] DefiLlama. 2026. Bridge Hacks and Total Value Locked. https://defillama.com/protocols/Bridge. Accessed: 2026-04-28.

[17] Liam Eagen. 2025. Glock: Garbled Locks for Bitcoin. Cryptology ePrint Archive, Paper 2025/1485. https://eprint.iacr.org/2025/1485

[18] Liam Eagen and Ying Tong Lai. 2026. Argo MAC: Garbling with Elliptic Curve MACs. Cryptology ePrint Archive, Paper 2026/049. https://eprint.iacr.org/2026/049

[19] Ariel Futoransky, Fadi Barbàra, Ramses Fernandez, Gabriel Larotonda, and Sergio Demian Lerner. 2025. OHMG: One hot modular garbling. Cryptology ePrint Archive, Paper 2025/2338. https://eprint.iacr.org/2025/2338

[20] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. 2015. The Bitcoin Backbone Protocol: Analysis and Applications. In *Advances in Cryptology – EUROCRYPT 2015*. Springer, 281–310. doi:10.1007/978-3-662-46803-6_10

[21] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. 2016. The Bitcoin Backbone Protocol with Chains of Variable Difficulty. Cryptology ePrint Archive, Paper 2016/1048. https://eprint.iacr.org/2016/1048

[22] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. 2020. How Does Satoshi Set His Clock? Full Analysis of Nakamoto Consensus. Cryptology ePrint Archive, Paper 2020/277. https://eprint.iacr.org/2020/277

[23] Sanjam Garg, Dimitris Kolonelos, Mikhail Sergeevitch, Srivatsan Sridhar, and David Tse. 2026. BABE: Verifying Proofs on Bitcoin Made 1000x Cheaper. Cryptology ePrint Archive, Paper 2026/065. https://eprint.iacr.org/2026/065

[24] Peter Gaži, Aggelos Kiayias, and Dionysis Zindros. 2019. Proof-of-Stake Sidechains. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 139–156. doi:10.1109/SP.2019.00040

[25] Craig Gentry and Daniel Wichs. 2011. Separating succinct non-interactive arguments from all falsifiable assumptions. In *Proceedings of the Forty-Third Annual ACM Symposium on Theory of Computing* (San Jose, California, USA) *(STOC '11)*. Association for Computing Machinery, New York, NY, USA, 99–108. doi:10.1145/1993636.1993651

[26] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. 1988. A Digital Signature Scheme Secure Against Adaptive Chosen-Message Attacks. *SIAM J. Comput.* 17, 2 (04 1988), 281–28. https://www.proquest.com/scholarly-journals/digital-signature-scheme-secure-against-adaptive/docview/919828126/se-2 Copyright - Copyright] © 1988 Society for Industrial and Applied Mathematics; Last updated - 2023-12-04.

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

[28] David Harding and Mike Schmidt. 2024. Bitcoin Optech: Covenants. https://bitcoinops.org/en/topics/covenants/ Accessed: 2026-09-11.

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

[30] Nakul Khambhati, Anwesh Bhattacharya, and David Heath. 2026. Duty-Free Bits: Projectivizing Garbling Schemes. Cryptology ePrint Archive, Paper 2026/476. https://eprint.iacr.org/2026/476

[31] Leslie Lamport. 1979. *Constructing Digital Signatures from a One Way Function*. Technical Report CSL-98. SRI International. https://www.microsoft.com/en-us/research/publication/constructing-digital-signatures-one-way-function/

[32] Sergio Demian Lerner, Ramon Amela, Shreemoy Mishra, Martin Jonas, and Javier Álvarez Cid-Fuentes. 2024. BitVMX: A CPU for Universal Computation on Bitcoin. arXiv:2405.06842 [cs.CR] https://arxiv.org/abs/2405.06842

[33] Robin Linus. 2022. Boolean Functions in Discrete Log Contracts. https://gist.github.com/RobinLinus/4035ced3fa04cc3745a30dcda09e2367#boolean-functions. Accessed: 2026-03-13.

[34] Robin Linus. 2025. BitVM 3s – Garbled Circuits for Efficient Computation on Bitcoin. Manuscript, July 22, 2025. https://bitvm.org/bitvm3.pdf

[35] Robin Linus, Lukas Aumayr, Zeta Avarikioti, Matteo Maffei, Andrea Pelosi, Orfeas Thyfronitis Litos, Christos Stefo, David Tse, and Alexei Zamyatin. 2025. Bridging Bitcoin to Second Layers via BitVM2. Cryptology ePrint Archive, Paper 2025/1158. https://eprint.iacr.org/2025/1158

[36] Malte Möser, Ittay Eyal, and Emin Gün Sirer. 2016. Bitcoin Covenants. In *FC '16: Proceedings of the 20th International Conference on Financial Cryptography*. http://fc16.ifca.ai/bitcoin/papers/MES16.pdf

[37] Satoshi Nakamoto. 2008. Bitcoin: A Peer-to-Peer Electronic Cash System. https://bitcoin.org/bitcoin.pdf Accessed: 2026-04-01.

[38] Russell O'Connor and Marta Piekarska. 2017. Enhancing Bitcoin Transactions with Covenants. In *Financial Cryptography and Data Security*, Michael Brenner, Kurt Rohloff, Joseph Bonneau, Andrew Miller, Peter Y.A. Ryan, Vanessa Teague, Andrea Bracciali, Massimiliano Sala, Federico Pintore, and Markus Jakobsson (Eds.). Springer International Publishing, Cham, 191–198.

[39] Jeremy Rubin. 2025. Delbrag. https://rubin.io/public/pdfs/delbrag.pdf

[40] Giulia Scaffino, Lukas Aumayr, Zeta Avarikioti, and Matteo Maffei. 2023. Glimpse: on-demand PoW light client with constant-size storage for DeFi. In *Proceedings of the 32nd USENIX Conference on Security Symposium* (Anaheim, CA, USA) *(SEC '23)*. USENIX Association, USA, Article 42, 18 pages.

[41] Pieter Wuille, Jonas Nick, and Anthony Towns. 2020. BIP 0341, Taproot: SegWit version 1 spending rules.

[42] Pieter Wuille, Jonas Nick, and Anthony Towns. 2020. BIP 341: Taproot: SegWit version 1 spending rules. Bitcoin Improvement Proposal. https://bips.dev/341/ Section "Constructing and spending Taproot outputs". Status: Final. License: BSD-3-Clause.

[43] Tiancheng Xie, Jiaheng Zhang, Zerui Cheng, Fan Zhang, Yupeng Zhang, Yongzheng Jia, Dan Boneh, and Dawn Song. 2022. zkBridge: Trustless Cross-chain Bridges Made Practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 3003–3017. doi:10.1145/3548606.3560652

[44] Alexei Zamyatin, Dominik Harz, Joshua Lind, Panayiotis Panayiotou, Arthur Gervais, and William J. Knottenbelt. 2018. XCLAIM: Trustless, Interoperable Cryptocurrency-Backed Assets. Cryptology ePrint Archive, Paper 2018/643. https://eprint.iacr.org/2018/643

A Modeling Framework

We use the modeling approach of [4], which builds on [21, 24], and follows a UC-style formulation of real-world executions [14]. Time advances in discrete rounds (or slots), with communication (semi-)synchronously via a diffusion of messages, where clock drift is absorbed into the round length to ensure approximate synchronization.

We use $\{\text{exec}_{\Pi,\mathcal{A},\mathcal{Z}}(z)\}_{z\in\{0,1\}^*}$ to denote the random variable ensemble determining $\mathcal{Z}$'s output on input z, executing Π in presence of adversary $\mathcal{A}$. We use $\{\text{VIEW}^P_{\Pi,\mathcal{A},\mathcal{Z}}(z)\}_{z\in\{0,1\}^*}$ to denote the random variable ensemble that describes the "view" of party P after an execution of $\mathcal{Z}$ running Π, and adversary $\mathcal{A}$ on input z has completed. This view includes everything that P "has seen", including any message that P has received, and its internal state. We write $\text{VIEW}^{P,r}_{\Pi,\mathcal{A},\mathcal{Z}}(z)$ to denote such a view up to a specific round r, i.e., everything that this party has seen up until and including the round r. We denote $\text{VIEW}_{\Pi,\mathcal{A},\mathcal{Z}}(z)$ as the concatenation of $\text{VIEW}^P_{\Pi,\mathcal{A},\mathcal{Z}}(z)$ for every $P \in \mathcal{P}$. This concatenation can be specified up to a specific round as well, i.e., $\text{VIEW}^r_{\Pi,\mathcal{A},\mathcal{Z}}(z)$. Later, we define properties of protocols as predicates over the random variable $\text{VIEW}^{P,r}_{\Pi,\mathcal{A},\mathcal{Z}}(z)$,

quantifying over all PPT adversaries $\mathcal{A}$, all PPT environments $\mathcal{Z}$, and all inputs $z \in \{0,1\}^*$.

A.1 Ledger and Network Model

We consider two ledgers, denoted $\mathcal{L}_A$ and $\mathcal{L}_B$, which are run by the corresponding ledger protocols $\Pi_{\mathcal{L}_A}$ and $\Pi_{\mathcal{L}_B}$. In our primary application, $\mathcal{L}_A$ is Bitcoin, while $\mathcal{L}_B$ is a second-layer system or an alternative chain. Each ledger protocol maintains an ordered sequence of confirmed transactions that is visible to participants called full nodes. We represent the ledger by the function ${}^{r}\mathcal{L}_X^P$, which denotes party P's view of ledger $\mathcal{L}_X$ at round r.

Definition 1 (Safety). A distributed ledger protocol is *safe* if the following two properties hold: (i) For every honest party P and any rounds $r_1 \le r_2$, we have ${}^{r_1}\mathcal{L}^P \preceq {}^{r_2}\mathcal{L}^P$ (*Self-Consistency*); and (ii) For any honest parties P_1, P_2 and any round r, 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*).

Definition 2 (Liveness). A distributed ledger protocol execution in a synchronous network is *live*(u) if every transaction written to the ledger of any honest party at round r appears in the ledgers of all honest parties by round $r+u$, denoted by ${}^{r+u}\mathcal{L}^{\cap}$.

We assume a global round-based clock shared by all parties and ledgers; rounds capture time as discrete steps. We further assume a synchronous network, i.e., every message broadcast to the network is delivered to all parties within a known delay bound Δ. Ledgers and blockchains are modeled as in the Bitcoin backbone model [20], which, in particular, works in the q-bounded setting.

A.2 Transactions in the UTXO model

We identify a user U on a ledger L by the key pair $(\mathsf{pk}_\mathsf{U}, \mathsf{sk}_\mathsf{U})$ of a signature scheme Σ, which is used to demonstrate ownership of coins. We write $\sigma_\mathsf{U}(m)$ for U's digital signature on a message $m \in \{0,1\}^*$. When the signed message is clear from context, we sometimes abbreviate this as σ_U.

In the *unspent transaction output* (UTXO) model, each transaction output is assigned a coin value (in BTC). An output is specified as an attribute tuple $\mathsf{out} := (a \text{ BTC}, \mathsf{lockScript})$; that is, it contains an amount $\mathsf{out}.a \in \mathbb{R}_{\ge 0}$ of coins BTC together with the condition(s) $\mathsf{out}.\mathsf{lockScript}$ under which those coins may be spent. A transaction Tx consumes a non-empty list of existing unspent outputs and creates a non-empty list of new outputs Tx.outputs. We call the consumed outputs the transaction inputs, denoted Tx.inputs. An input, $\mathsf{in} := (\mathsf{PrevTx}, \mathsf{outIndex}, \mathsf{lockScript})$, uniquely identifies an existing output by referring to a transaction PrevTx and an output index outIndex; for convenience, it also repeats the output's spending condition lockScript. We refer to the input and output lists of a transaction as its *transaction skeleton*.

Formally, we define a transaction as $\mathsf{Tx} := (\mathsf{inputs}, \mathsf{witnesses}, \mathsf{outputs})$. In addition to the inputs $\mathsf{Tx.inputs} := [\mathsf{in}_1, \ldots, \mathsf{in}_n]$ and outputs $\mathsf{Tx.outputs} := [\mathsf{out}_1, \ldots, \mathsf{out}_m]$, it includes witness data $\mathsf{Tx.witnesses} := [\mathsf{w}_1, \ldots, \mathsf{w}_n]$, namely the list of tuples that satisfy the spending conditions of the transaction inputs, with one witness per input. The locking script, written in the ledger's scripting language, is run with the corresponding witness as input to the script. If execution returns False, the transaction is invalid. If it returns True, the spending condition is satisfied.

For a transaction to be valid, every witness must satisfy the locking condition of its corresponding input; every input must be unspent; and the total value of the outputs must be at most the total value of the inputs. If the output value is strictly smaller, the difference is paid to the miners.

Transaction spending conditions. We are mainly interested in Bitcoin, whose scripting language is stack-based. We now describe the subset of Bitcoin spending conditions used in this paper. These conditions can be combined with the logical operators $\wedge$ (and) and $\vee$ (or) to form more complex spending conditions.

- **Signature locks.** An output locked by $\mathsf{CheckSig}_{\mathsf{pk}_\mathsf{U}}$ can be spent only if the spending transaction is signed using the secret key corresponding to the key pair $(\mathsf{sk}_\mathsf{U}, \mathsf{pk}_\mathsf{U})$.
- **Multisignature locks.** A multisignature spending condition is satisfied by providing a prescribed threshold k of n signatures. For example, for users A and B, a 2-of-2 multisignature spending condition is written as $\mathsf{CheckMultiSig}_{\mathsf{pk}_{A,B}}$, and the corresponding joint signature is denoted by $\sigma_{A,B}$.
- **Hashlocks.** A hashlock binds an output to a hash value and can be satisfied only by revealing a corresponding preimage. We write $\mathsf{Hashlock}(\mathcal{H}(s))$ for the spending condition that requires the spender to provide a value s such that its hash equals $\mathcal{H}(s)$.
- **Timelocks** prevent a transaction output from being spent until a specified future time (absolute timelock), or until a specified delay has elapsed after the transaction is included on-chain (relative timelock). We denote the former by $\mathsf{AbsTimelock}(\Delta)$ and the latter by $\mathsf{RelTimelock}(\Delta)$. In what follows, timelocks are used together with other spending conditions. For example, if the UTXO $\mathsf{Tx.out}_1$ has locking script $\mathsf{lockScript} := \mathsf{RelTimelock}(\Delta) \wedge \mathsf{CheckSig}_{\mathsf{pk}_\mathsf{U}}$, then user U can spend $\mathsf{Tx.out}_1$ once a certain amount of time T has elapsed since $\mathsf{Tx.out}_1$ was published on-chain.
- **Taproot Trees** [41], or Taptrees, allow a UTXO to be spent by satisfying one of several alternative spending conditions. The spending conditions are encoded as (Tap)leaves of a Merkle tree. To spend a UTXO whose locking script is a Taptree, a user provides a witness for one leaf together with a Merkle inclusion proof for that leaf in the Taptree. In the following, we write the Tapleaves of a Taptree locking script as $\langle \mathsf{leaf}_1, \ldots, \mathsf{leaf}_r \rangle$; when a user satisfies script leaf_i to unlock the j-th UTXO of transaction Tx, we write the corresponding input as $(\mathsf{Tx}, j, \langle \mathsf{leaf}_i \rangle)$. Whenever a user spends a UTXO through a Tapleaf of a Taptree, we assume that the user also provides a valid Merkle inclusion proof for that Tapleaf.
- **Other conditions.** We use True for a condition that is always satisfied and False for a condition that can never be satisfied. In the latter case, the coins cannot be redeemed and are instead *burnt*.

We also use $*$ to denote a transaction input, witness, or output that may be arbitrary, as long as it is valid under Bitcoin consensus rules, but is irrelevant to the protocol.

Combining spending conditions. When expressing long spending conditions, we provide their pseudocode explicitly, combining the

spending conditions above with other standard language constructs expressible in Bitcoin Script, such as the *if-then-else* construction. In particular, when a long script LongScript appends the keyword Verify to one of its sub-spending conditions, say script, which returns either True or False, we intend to model the behavior of Bitcoin's `OP_VERIFY` opcode: if script returns True, then True is popped from the stack and execution continues with the rest of the script; if script returns False, the transaction is marked invalid and therefore fails to satisfy LongScript.

A.3 Ledger Interface.

During protocol execution, parties interact with a robust ledger protocol $\Pi_{\mathcal{L}}$ that provides safety and liveness guarantees. The ledger protocol offers the following functionalities.

- $\Pi_{\mathcal{L}}.\mathsf{read}()$: When called at the current time t_{cur}, this functionality returns the current ledger view $L_{t_{\mathsf{cur}}}$, namely the ordered list of transactions that have been written to the ledger and are confirmed in the common view of all honest parties by time t_{cur}.
- $\Pi_{\mathcal{L}}.\mathsf{write}(tx)$: When called at the current time t_{cur}, this functionality attempts to append transaction tx to the ledger.
- $\mathsf{Valid}_{\mathcal{L}}(tx)$: Returns 1 if and only if transaction tx is valid with respect to the current ledger state, and returns 0 otherwise. Let

  $$tx := (\mathsf{inputs}, \mathsf{witnesses}, \mathsf{outputs})$$

  Then $\mathsf{Valid}_{\mathcal{L}}(tx) = 1$ if and only if all of the following conditions hold:
  (i) **Referenced outputs exist and remain unspent.** For each input that references an output of a previous transaction, that output appears in the current ledger state and has not been spent.
  (ii) **All spending conditions are fulfilled.** For each input, the associated witness satisfies the locking condition of the referenced output.
  (iii) **Value is preserved.** The aggregate value of the outputs created by tx is no greater than the aggregate value of the outputs consumed by its inputs.

B Cryptographic primitives

B.1 Succinct Non-Interactive Arguments of Knowledge (SNARKs)

This definition follows [25, 27] closely. Let $\mathcal{R} \leftarrow \hat{\mathcal{R}}(\lambda)$ be a relation generator that, on input a security parameter λ, outputs a polynomial-time decidable binary relation $\mathcal{R}$. We use ϕ to denote the statement and w to denote the witness for pairs $(\phi, w) \in \mathcal{R}$.

An efficient publicly verifiable non-interactive argument of knowledge for $\hat{\mathcal{R}}$ is defined as a tuple of three PPT algorithms: Setup, Prove, and Vrfy.

- $crs \leftarrow \mathsf{SNARK.Setup}(\mathcal{R})$ takes a relation $\mathcal{R}$ as input and outputs a common reference string crs.
- $\pi \leftarrow \mathsf{SNARK.Prove}(\mathcal{R}, crs, \phi, w)$ takes as input the common reference string crs and a pair $(\phi, w) \in \mathcal{R}$, and outputs an argument π.
- $\{\mathsf{True}, \mathsf{False}\} \leftarrow \mathsf{SNARK.Vrfy}(\mathcal{R}, crs, \phi, \pi)$ takes as input the common reference string crs, a statement ϕ, and an argument π, and returns either True or False, depending informally on whether π is a valid argument.

The tuple (Setup, Prove, Vrfy) is a non-interactive argument of knowledge for $\hat{\mathcal{R}}$ if it satisfies *perfect completeness* and *knowledge soundness*, as defined in [25, 27]. At a high level, perfect completeness says that for any true statement ϕ, an honest prover can convince an honest verifier with overwhelming probability. Knowledge soundness says that any prover who can convince an honest verifier must know a valid witness w, also with overwhelming probability.

Finally, a non-interactive argument in which the verifier runs in time polynomial in $\lambda + |\phi|$ and the proof size is polynomial in λ is called a preprocessing succinct non-interactive argument of knowledge, or SNARK. Our implementation uses [27]; we neither require nor use the zero-knowledge property.

B.2 Garbling Schemes

We use a slightly modified definition of a secure, privacy-free garbling scheme [8]. We define a garbling scheme with respect to a space of functions $\mathcal{F} \subseteq \mathcal{X} \to \mathcal{Y}$.

The scheme consists of a tuple of efficient, deterministic algorithms $\mathcal{G} = (\mathsf{Gb}, \mathsf{En}, \mathsf{Ev}, \mathsf{De})$ defined as follows:

- $(F, e, d) \leftarrow \mathsf{Gb}(1^\kappa, f)$: A probabilistic garbler that takes the security parameter 1^κ and a function $f \in \mathcal{F}$. It outputs a garbled circuit F, encoding information e, and decoding information d.
- $L_x := \mathsf{En}(e, x)$: An encoder that takes the encoding information e and a plaintext input $x \in \mathcal{X}$, and outputs the encoded input L_x.
- $L_y := \mathsf{Ev}(F, L_x)$: An evaluator that takes the garbled circuit F and the encoded input L_x, and produces the encoded output L_y.
- $y := \mathsf{De}(d, L_y)$: A decoder that uses the decoding information d to recover the plaintext result $y \in \mathcal{Y}$ from the encoded output L_y.

A secure privacy-free garbling scheme under this definition must satisfy the following properties:

- **Correctness**: For all f, all $(F, e, d) \leftarrow \mathsf{Gb}(1^\kappa, f)$, and all $x \in \mathcal{X}$:

  $$\Pr\left[\mathsf{De}(d,\ \mathsf{Ev}(F,\ \mathsf{En}(e, x))) = f(x)\right] = 1.$$

- **Authenticity**: For all efficient adversaries $\mathcal{A}$, all f, and all x, let $(F, e, d) \leftarrow \mathsf{Gb}(1^\kappa, f)$, $L_x := \mathsf{En}(e, x)$, and $L'_y \leftarrow \mathcal{A}(1^\kappa, f, F, L_x)$. Then:

  $$\Pr\left[L'_y \neq \mathsf{Ev}(F, L_x) \ \wedge\ \mathsf{De}(d, L'_y) \neq \perp\ \right] \leq \mathsf{negl}(\kappa).$$

B.3 Digital Signatures

A digital signature scheme Σ consists 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 outputs a key pair, consisting of a secret, or private, key 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, called a *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 $m \in \{0,1\}^*$, and outputs True if and only if σ is a valid signature on m under the secret key sk corresponding to pk, i.e., where (pk, sk) is a key pair generated by $\Sigma.\mathsf{KeyGen}$. For consistency with Bitcoin Script, we denote this predicate by $\mathsf{CheckSig}_{\mathsf{pk}}(\sigma)$, with the transaction serving as the signed message.

In this work, we use signature schemes that are EUF-CMA secure [26].

B.4 Garbled Encoding Extractable Signature

Let $\mathcal{G} = (\mathsf{Gb}, \mathsf{En}, \mathsf{Ev}, \mathsf{De})$ be a projective garbling scheme as defined in Section B.2, and let f be a Boolean circuit with input length ℓ. Consider a garbled circuit $(GC, e, d) \leftarrow \mathsf{Gb}(1^\kappa, f)$ generated honestly. A *garbled encoding extractable signature scheme* (GS) for $\mathcal{G}$ is a tuple of deterministic polynomial-time algorithms

$$\mathsf{GS} = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Vrfy}, \mathsf{Extract})$$

defined as follows.

- $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{GS.KeyGen}(1^\kappa, e)$: On input a security parameter κ and the encoding information e of the garbled circuit, output a public/secret key pair $(\mathsf{pk}, \mathsf{sk})$. Intuitively, pk commits to the valid input-label pairs of the garbled circuit, while sk contains the corresponding underlying labels.
- $\sigma \leftarrow \mathsf{GS.Sign}(\mathsf{sk}, x)$: On input a secret key sk and a message $x \in \{0,1\}^\ell$, output a signature σ.
- $b \leftarrow \mathsf{GS.Vrfy}(\mathsf{pk}, \sigma)$: On input a public key pk and a signature σ, output $b \in \{\mathsf{True}, \mathsf{False}\}$. The algorithm outputs True iff σ is a valid signature under pk. For consistency with Bitcoin Script, we refer to the corresponding on-chain check as $\mathsf{CheckGS}_{\mathsf{pk}}(\sigma)$.
- $L_x \leftarrow \mathsf{GS.Extract}(\mathsf{pk}, \sigma)$: On input a public key pk and a valid signature σ, output the encoded input state L_x revealed by σ.

We say that GS is secure if it satisfies the following properties.

- *Correctness.* For every $x \in \{0,1\}^\ell$, let

  $$(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{GS.KeyGen}(1^\kappa, e),\ \sigma \leftarrow \mathsf{GS.Sign}(\mathsf{sk}, x)$$

  Then $\Pr\left[\mathsf{GS.Vrfy}(\mathsf{pk}, \sigma) = \mathsf{True}\right] = 1$.
- *Extractability.* For every $x \in \{0,1\}^\ell$, let

  $$(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{GS.KeyGen}(1^\kappa, e),\ \sigma \leftarrow \mathsf{GS.Sign}(\mathsf{sk}, x).$$

  Then $\Pr\left[\mathsf{GS.Extract}(\mathsf{pk}, \sigma) = \mathsf{En}(e, x)\right] = 1$. That is, every honestly generated signature reveals exactly the garbled encoding of the signed message.
- *Unforgeability.* For every PPT adversary $\mathcal{A}$, let

  $$(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{GS.KeyGen}(1^\kappa, e),\ \sigma^* \leftarrow \mathcal{A}(1^\kappa, \mathsf{pk}).$$

  Then

  $$\Pr\left[\mathsf{GS.Vrfy}(\mathsf{pk}, \sigma^*) = \mathsf{True}\right] \le \mathsf{negl}(\kappa).$$

B.5 BitVM3-core pseudocode

We give a full protocol description of BitVM3-core in Figure 7. For simplicity, the protocol description treats the case where the statement ϕ is fixed during setup and only the proof π is left as a free input to the garbled SNARK verifier circuit.

Helper Functions. We use the following auxiliary functions in the protocol description.

- $\mathsf{Circuit}(f)$: Given a deterministic function f, outputs a Boolean circuit f^{C} that computes the same functionality. In our protocol, $\mathsf{Circuit}(\mathsf{SNARK.Vrfy}(\mathcal{R}, \mathsf{crs}, \phi, \cdot))$ denotes the Boolean circuit for the SNARK verifier with the proof left as a free input, while $\mathsf{Circuit}(\mathsf{SNARK.Vrfy}(\mathcal{R}, \mathsf{crs}, \cdot, \cdot))$ denotes the SNARK verifier with both the public statement ϕ and the proof π left as free inputs.
- $\mathsf{CheckExpired}(t)$: Returns 1 if and only if the relative timelock parameter t has expired for the relevant unspent output. Concretely, this helper uses the ledger interface to compare the current block height with the block height at which the referenced output was created.
- $\mathsf{CurrentRound}(\Pi_{\mathcal{L}})$: Returns the current block height, or round, of the ledger protocol.

C Formal Security Analysis (Honest Setup)

In this section, we formalize the ChainVM abstraction, define completeness and soundness as game-based experiments, and prove that BitVM3-core satisfies both properties under the honest-setup assumption.

C.1 ChainVM proof system

Definition 3 (ChainVM proof system). A ChainVM proof system is a dApp parameterized by a distributed ledger L and a parameter v, executed by a prover P and multiple verifiers V who are full nodes in L. It consists of two decentralized protocols:

- $\mathsf{ChainVM.Setup}(1^\kappa, Q) \to (\mathsf{sid}, S, \mathsf{aux})$: Given a security parameter 1^κ and a predicate $Q\colon \{0,1\}^* \to \{0,1\}$, the prover outputs a session identifier sid, a success predicate S over the ledger L, and auxiliary data aux.
- $\mathsf{ChainVM.Execute}(\mathsf{sid}, \pi, \mathsf{aux}) \to r$: Given a string π, the prover outputs $r \in \mathbb{N} \cup \{\perp\}$ (a round number, or failure).

Notation. We write $X(\ldots) \downarrow_{\mathsf{view}} y$ to mean that the invocation $X(\ldots) \to y$ appears in view, and $\mu(\kappa)$ to denote a negligible function in κ.

Completeness and soundness are defined via the experiments Complete and Sound (Algorithms 1 and 2). Completeness requires that an honest prover holding a valid witness always succeeds; soundness requires that whenever execution completes successfully, an efficient extractor can recover from any honest verifier's view a string π satisfying Q.

Definition 4 (Completeness). A ChainVM proof system is *complete* if for all PPT $\mathcal{A}, \mathcal{Z}$ and any $z \in \{0,1\}^*$,

$$\Pr\left[\mathsf{Complete}\left(\mathrm{VIEW}_{\Pi,\mathcal{A},\mathcal{Z}}(z)\right) = 0\right] \le \mu(\kappa).$$

Definition 5 (Soundness). A ChainVM proof system is *sound* if there exists an efficient Extract such that for all PPT $\mathcal{A}, \mathcal{Z}$ and any

BitVM3-core implementing Definition 3

$\underline{\textbf{Setup}}(1^\kappa, Q) \to (\text{sid}, S, \text{aux})$

Public parameters. The following are public knowledge: the relation $\mathcal{R}$, the statement ϕ, the SNARK scheme, the common reference string crs, the garbling scheme $\mathcal{G}$, and the timeout t which is bigger than the ledger's liveness parameter u.

Prover

(1) $(\text{pk}_P, \text{sk}_P) \leftarrow \Sigma.\text{KeyGen}(1^\lambda)$ ▷ Generate prover key pair
(2) $f^C \leftarrow \text{Circuit}(\text{SNARK.Vrfy}(\mathcal{R}, \text{crs}, \phi, \cdot))$
(3) $(GC, e, d) := \text{Gb}(1^\kappa, f^C)$
(4) $(\text{pk}_{GS}, \text{sk}_{GS}) \leftarrow \text{GS.KeyGen}(1^\kappa, e)$
(5) Let L^* be such that $\text{De}(d, L^*) = \text{False}$
(6) Construct Assert as follows:
  (i) $\text{Assert.inputs}[0] \leftarrow (txid, idx, \text{GS.Vrfy}(\text{pk}_{GS}, \sigma))$ ▷ pk_{GS} is hard-coded in the locking script
  (ii) $\text{Assert.outputs}[0] \leftarrow \text{Taptree}(\text{Hashlock}(\mathcal{H}(L^*)), \text{CheckSig}_{\text{pk}_P} \wedge \text{RelTimelock}(t))$ ▷ Challenge path and timeout path
(7) $\text{sid} \leftarrow \text{Assert.txid}$
(8) Send $(GC, d, \text{pk}_{GS}, \text{Assert}, \text{sid})$ to all verifiers
(9) Define $S := \text{Assert} \in \Pi_{\mathcal{L}}.\text{read}() \wedge \text{CheckExpired}(t)$ ▷ Assert is on-chain and the timeout has expired
(10) $\text{aux} \leftarrow (\mathcal{R}, \phi, \text{crs}, \mathcal{G}, f^C, GC, e, d, L^*, \text{pk}_{GS}, \text{sk}_{GS}, \text{Assert})$
(11) **return** $(\text{sid}, S, \text{aux})$

Verifier

(1) Receive $(GC, d, \text{pk}_{GS}, \text{Assert}, \text{sid})$ from the prover
(2) Check that $\text{Assert.txid} = \text{sid}$
(3) Check that $\text{Assert.inputs}[0]$ contains $\text{GS.Vrfy}(\text{pk}_{GS}, \sigma)$
(4) Define $S := \text{Assert} \in \Pi_{\mathcal{L}}.\text{read}() \wedge \text{CheckExpired}(t)$
(5) $\text{aux} \leftarrow (\mathcal{R}, \phi, \text{crs}, \mathcal{G}, f^C, GC, d, \text{pk}_{GS}, \text{Assert})$
(6) **return** $(\text{sid}, S, \text{aux})$

$\underline{\textbf{Execute}}(\text{sid}, \pi, \text{aux}) \to r$

Prover

(1) $(\mathcal{R}, \phi, \text{crs}, \mathcal{G}, f^C, GC, e, d, L^*, \text{pk}_{GS}, \text{sk}_{GS}, \text{Assert}) \leftarrow \text{aux}$
(2) $\pi \leftarrow \text{SNARK.Prove}(\mathcal{R}, \text{crs}, \phi, w)$
(3) $\sigma_\pi \leftarrow \text{GS.Sign}(\text{sk}_{GS}, \pi)$
(4) $\text{Assert.witness} \leftarrow \sigma_\pi$
(5) if $\neg\text{Valid}_{\mathcal{L}}(\text{Assert})$, then **return** $\perp$
(6) $\Pi_{\mathcal{L}}.\text{write}(\text{Assert})$
(7) While true:
  (i) if $\text{Disprove} \in \Pi_{\mathcal{L}}.\text{read}()$, then **return** $\perp$
  (ii) if $\text{Assert} \in \Pi_{\mathcal{L}}.\text{read}() \wedge \text{CheckExpired}(t)$, then **return** $\text{CurrentRound}(\Pi_{\mathcal{L}})$

Verifier

(1) $(\mathcal{R}, \phi, \text{crs}, \mathcal{G}, f^C, GC, d, \text{pk}_{GS}, \text{Assert}) \leftarrow \text{aux}$
(2) Wait until $\text{Assert} \in \Pi_{\mathcal{L}}.\text{read}()$. Let r_1 be the round that Assert was first read.
(3) $\sigma_\pi \leftarrow \text{Assert.witness}$
(4) $L_\pi \leftarrow \text{GS.Extract}(\text{pk}_{GS}, \sigma_\pi)$
(5) $L_y \leftarrow \text{Ev}(GC, L_\pi)$
(6) $y \leftarrow \text{De}(d, L_y)$
(7) If $y = \text{False}$, then:
  (i) $\text{Disprove.inputs}[0] \leftarrow (\text{Assert.txid}, 0, \text{Hashlock}(\mathcal{H}(L^*))$
  (ii) $\text{Disprove.outputs}[0] \leftarrow (v, \text{True})$
  (iii) $\text{Disprove.witness} \leftarrow L_y$
  (iv) $\Pi_{\mathcal{L}}.\text{write}(\text{Disprove})$
  (v) **return** $\perp$
(8) Otherwise, **return** $r = r_1 + t$

Figure 7: Protocol description of BitVM3-core implementing Definition 3. The setup phase prepares the garbled verifier and the Assert **transaction, while the execution phase shows how a verifier can refute an invalid proof by evaluating the garbled circuit off-chain and spending the challenge path with** Disprove**.**

Algorithm 1 Complete(view)

for all $Q, S, \mathsf{sid}, \mathsf{aux}, \pi \in \{0,1\}^*$ **do**
  if $\mathsf{ChainVM.Setup}(1^\kappa, Q) \downarrow_{\text{view}} (\mathsf{sid}, S, \mathsf{aux})$ **and** $\mathsf{ChainVM.Execute}(\mathsf{sid}, \pi, \mathsf{aux}) \downarrow_{\text{view}} r$ **and** $Q(\pi)$ **and** $\big(r = \bot \ \vee \ (\exists\, r' > r + v \ : \ \neg S(\mathcal{L}^P_{r'}))\big)$ **then**
    return 0
return 1

Algorithm 2 Sound(view)

for all $Q, S, \mathsf{sid} \in \{0,1\}^*,\ V \in \mathcal{V}_H,\ r$ **do**
  if $S(\mathcal{L}^V_r)$ **and** $\neg Q\big(\mathsf{Extract}(\text{view}^V_r, \mathsf{sid})\big)$ **then**
    return 0
return 1

$z \in \{0,1\}^*$,

$$\Pr\big[\mathsf{Sound}\big(\text{view}_{\Pi,\mathcal{A},\mathcal{Z}}(z)\big) = 0\big] \le \mu(\kappa).$$

C.2 Completeness and soundness proofs

We now prove completeness and soundness of BitVM3-core under the honest-setup assumption. For completeness, we use a short sequence of hybrids followed by a reduction to an auxiliary game that isolates the remaining cryptographic attack surface.

Throughout this subsection, we interpret $\Pi_{\mathcal{L}}.\mathsf{write}(tx)$ as the ledger submission interface returning the first round at which tx appears in the caller's local ledger view.

Lemma 1 (Write-to-Liveness). *Let P be an honest party and let $r_0 := \mathsf{CurrentRound}(\Pi_{\mathcal{L}})$ immediately before P invokes $\Pi_{\mathcal{L}}.\mathsf{write}(tx)$. If tx is valid and the call returns round r, then*

$$r \le r_0 + u$$

except with negligible probability.

Proof. By the interpretation above, invoking $\Pi_{\mathcal{L}}.\mathsf{write}(tx)$ at round r_0 means that the honest party P submits tx to the ledger at round r_0, and the returned value r is the first round at which tx appears in P's local ledger view. Since tx is valid, ledger liveness implies that tx is included in P's view by round $r_0 + u$, except with negligible probability. Therefore the first inclusion round satisfies $r \le r_0 + u$, as claimed. □

Theorem 1 (Completeness of BitVM3-core). *If the prover follows BitVM3-core honestly during setup and proving, the ledger protocol is safe and live, the SNARK scheme has perfect completeness, the garbling scheme satisfies correctness and authenticity, the garbled encoding extractable signature scheme satisfies correctness and extractability, and the underlying digital signature scheme is EUF-CMA secure, then BitVM3-core is complete in the random oracle model.*

We prove completeness via a game between a challenger (the honest prover) and an adversary $\mathcal{A}$ (the malicious verifier). At the beginning of the experiment, the challenger sends $\mathcal{A}$ the output of the setup phase together with the information needed to reconstruct an honest verifier's local state just before execution. The adversary is given oracle access to the random oracle $\mathcal{H}$ and can interact with the ledger protocol $\Pi_{\mathcal{L}}$.

The setup phase defines an on-chain success predicate S capturing the event that the Assert transaction is posted on-chain and remains unspent. During setup, Assert is constructed so that its spending conditions encode the predicate Q given to the parties. Concretely, the challenge path is tied to the garbled-circuit evaluation of the same SNARK-verifier computation corresponding to Q, while the timeout path captures the case in which no successful challenge is produced. A completeness violation occurs if, despite honest execution by a prover holding a valid witness and producing a proof π such that $Q(\pi)$ holds, the predicate S nevertheless fails to hold.

Proof. We proceed through a sequence of hybrid games starting from the completeness experiment. In these games, the command $\mathsf{assert}(\cdot)$ denotes that the challenger aborts the game if the condition specified in the argument is not met.

Proof idea. We begin with the real completeness experiment, in which the honest prover executes the protocol using a valid witness. The hybrids then condition on three properties that should hold in an honest execution: the generated SNARK proof verifies, the posted signature extracts to the intended garbled input encoding, and the honest Assert transaction is included on-chain within the liveness window. Once these properties are fixed, a completeness violation can occur only if the adversary spends output 0 of Assert. There are two ways this can happen: either the adversary spends the transaction by forging the required signature, or it spends the challenge path before expiry by producing a valid witness for the hashlock. We isolate the latter event in the auxiliary game Game $0'$.

Game 0: Real Completeness Experiment

$(\mathsf{sid}, S, \mathsf{aux}) \leftarrow \mathbf{Setup}(1^\kappa, Q)$
$(\mathsf{pk}_P, \mathsf{sk}_P, \mathsf{ctx}) \leftarrow \mathsf{aux}$
$(\mathcal{R}, \phi, \mathsf{crs}, \mathcal{G}, \mathsf{GC}, e, d, L^*, \mathsf{pk}_{\mathsf{GS}}, \mathsf{sk}_{\mathsf{GS}}, \mathsf{Assert}) \leftarrow \mathsf{ctx}$
$\mathsf{state}_{\mathcal{A}} \leftarrow (\mathcal{R}, \phi, \mathsf{crs}, \mathcal{G}, \mathsf{GC}, d, \mathcal{H}(L^*))(L^*), \mathsf{pk}_{\mathsf{GS}}, \mathsf{Assert})$
$\pi \leftarrow \mathsf{SNARK.Prove}(\mathcal{R}, \mathsf{crs}, \phi, w)$
$\sigma_\pi \leftarrow \mathsf{GS.Sign}(\mathsf{sk}_{\mathsf{GS}}, \pi)$
$\mathsf{Assert.witness} \leftarrow \sigma_\pi$
$r \leftarrow \Pi_{\mathcal{L}}.\mathsf{write}(\mathsf{Assert})$
$r' \leftarrow \mathcal{A}^{\Pi_{\mathcal{L}}, \mathcal{H}}(\mathsf{state}_{\mathcal{A}})$
return $\big(r' \ge r \implies (\mathsf{Assert} \in \Pi_{\mathcal{L}}.\mathsf{read}(r') \wedge$
  $\wedge \nexists tx \in \Pi_{\mathcal{L}}.\mathsf{read}(r')$ s.t. $\mathsf{Assert} \in tx.\mathsf{inputs})\big)$

Figure 8: The real completeness experiment (Game_0) for BitVM3-core.

Game 0 in Figure 8 is the real completeness experiment corresponding to the completeness condition formalized in Definition 4 via the experiment Complete in Algorithm 1. As in Complete, the violating outcome is 0: a completeness violation occurs if, despite honest proving with a valid witness, the adversary causes the success predicate S to fail.

We define the hybrids Game 1, Game 2, and Game 3 from Game 0 by adding the boxed statements labeled [G1], [G2], and [G3] shown inline in Figure 9.

Game 0 with Hybrid Additions

$(\mathsf{sid}, S, \mathsf{aux}) \leftarrow \mathbf{Setup}(1^\kappa, Q)$
$(\mathsf{pk}_P, \mathsf{sk}_P, \mathsf{ctx}) \leftarrow \mathsf{aux}$
$(\mathcal{R}, \phi, \mathsf{crs}, \mathcal{G}, \mathsf{GC}, e, d, L^*, \mathsf{pk}_{\mathsf{GS}}, \mathsf{sk}_{\mathsf{GS}}, \mathsf{Assert}) \leftarrow \mathsf{ctx}$
$\mathsf{state}_{\mathcal{A}} \leftarrow (\mathcal{R}, \phi, \mathsf{crs}, \mathcal{G}, \mathsf{GC}, d, \mathcal{H}(L^*), \mathsf{pk}_{\mathsf{GS}}, \mathsf{Assert})$
$\pi \leftarrow \mathsf{SNARK.Prove}(\mathcal{R}, \mathsf{crs}, \phi, w)$
$\mathsf{assert}(\mathsf{SNARK.Vrfy}(\mathcal{R}, \mathsf{crs}, \phi, \pi) = \mathsf{True})$ [G1]
$\sigma_\pi \leftarrow \mathsf{GS.Sign}(\mathsf{sk}_{\mathsf{GS}}, \pi)$
$\mathsf{assert}(\mathsf{GS.Extract}(\mathsf{pk}_{\mathsf{GS}}, \sigma_\pi) = \mathsf{En}(e, \pi))$ [G2]
$\mathsf{Assert.witness} \leftarrow \sigma_\pi$
$r_0 \leftarrow \mathsf{CurrentRound}(\Pi_{\mathcal{L}})$ [G3]
$r \leftarrow \Pi_{\mathcal{L}}.\mathsf{write}(\mathsf{Assert})$
$\mathsf{assert}(r \le r_0 + u)$ [G3]
$r' \leftarrow \mathcal{A}^{\Pi_{\mathcal{L}}, \mathcal{H}}(\mathsf{state}_{\mathcal{A}})$
return $\big(r' \ge r \implies (\mathsf{Assert} \in \Pi_{\mathcal{L}}.\mathsf{read}(r') \wedge$
$\nexists tx \in \Pi_{\mathcal{L}}.\mathsf{read}(r')$ s.t. $\mathsf{Assert} \in tx.\mathsf{inputs})\big)$

Figure 9: The real completeness experiment together with the boxed statements that are added to obtain the hybrids Game 1, Game 2, and Game 3.

From Game 0 to Game 1. Game 1 is obtained from Game 0 by adding the boxed statement labeled [G1] in Figure 9, which explicitly checks that the honestly generated SNARK proof π verifies.

Claim 1. $|\Pr[\mathbf{Game}_0 = 0] - \Pr[\mathbf{Game}_1 = 0]| \le \mathsf{negl}(\kappa)$.

Proof. Since $(\phi, w) \in \mathcal{R}$ and the prover computes

$$\pi \leftarrow \mathsf{SNARK.Prove}(\mathcal{R}, \mathsf{crs}, \phi, w),$$

perfect completeness of the SNARK, as specified in Section B.1, implies that

$$\Pr[\mathsf{SNARK.Vrfy}(\mathcal{R}, \mathsf{crs}, \phi, \pi) = \mathsf{True}] = 1 - \mathsf{negl}(\kappa).$$

Hence the additional abort in Game 1 occurs only with negligible probability.

From Game 1 to Game 2. Game 2 is obtained from Game 1 by adding the boxed statement labeled [G2] in Figure 9, which requires that the signature posted in Assert extracts to the correct garbled encoding of the proof.

Claim 2. $|\Pr[\mathbf{Game}_1 = 0] - \Pr[\mathbf{Game}_2 = 0]| \le \mathsf{negl}(\kappa)$.

Proof. The witness of Assert is computed honestly as

$$\sigma_\pi \leftarrow \mathsf{GS.Sign}(\mathsf{sk}_{\mathsf{GS}}, \pi).$$

By the correctness and extractability properties of GS from Section B.4, it holds that

$$\mathsf{GS.Extract}(\mathsf{pk}_{\mathsf{GS}}, \sigma_\pi) = \mathsf{En}(e, \pi)$$

except with negligible probability. Therefore the additional abort in Game 2 changes the experiment only negligibly.

From Game 2 to Game 3. Game 3 is obtained from Game 2 by adding the two boxed statements labeled [G3] in Figure 9, which condition on timely inclusion of the honest Assert transaction.

Claim 3. $|\Pr[\mathbf{Game}_2 = 0] - \Pr[\mathbf{Game}_3 = 0]| \le \mathsf{negl}(\kappa)$.

Proof. The transaction Assert is well formed, since it is generated during the honest setup and posted by the honest prover. Let r_0 denote the round immediately before the invocation of $\Pi_{\mathcal{L}}.\mathsf{write}(\mathsf{Assert})$, and let r be the round returned by that call. By Lemma 1, we have $r \le r_0 + u$ except with negligible probability. Therefore, the additional abort condition in Game 3 is triggered only with negligible probability.

It remains to analyze Game 3. By construction of the previous hybrids, the following all hold in Game 3 except with negligible probability: (i) the proof π is valid, (ii) the witness σ_π reveals exactly the garbled input encoding $L_\pi := \mathsf{En}(e, \pi)$, and (iii) the transaction Assert is posted on-chain within the liveness window.

Therefore, a completeness violation in Game 3 can occur only if the adversary causes output 0 of Assert to be spent. There are two ways this can happen in the protocol: either the adversary spends output 0 via the timeout path after expiry by forging the required signature, or it spends output 0 via the challenge path before expiry by producing a valid witness for the hashlock. We denote these bad events by $\mathsf{Bad}_{\mathsf{sig}}$ and $\mathsf{Bad}_{\mathsf{false}}$, respectively. We capture the latter event by the auxiliary game Game 0′ in Figure 10.

Game 0′: False-Label Challenge Game

$f^{\mathsf{C}} \leftarrow \mathsf{Circuit}(\mathsf{SNARK.Vrfy}(\mathcal{R}, \mathsf{crs}, \phi, \cdot))$
$(\mathsf{GC}, e, d) \leftarrow \mathsf{Gb}(1^\kappa, f^{\mathsf{C}})$
$L^* \leftarrow$ the unique output label such that $\mathsf{De}(d, L^*) = \mathsf{False}$
$\pi \leftarrow \mathsf{SNARK.Prove}(\mathcal{R}, \mathsf{crs}, \phi, w)$
$L_\pi \leftarrow \mathsf{En}(e, \pi)$
$L_y \leftarrow \mathcal{A}(1^\kappa, \mathcal{R}, \phi, \mathsf{crs}, \mathsf{GC}, L_\pi, d, \mathcal{H}(L^*))$
return $\big(L_y \ne \mathsf{Ev}(\mathsf{GC}, L_\pi) \wedge \mathsf{De}(d, L_y) = \mathsf{False}\big)$

Figure 10: Game$_{0'}$: Auxiliary game isolating the event that the adversary produces a valid witness for the challenge branch despite the proof being valid.

Claim 4. If there exists a PPT adversary that violates completeness in Game 3 with non-negligible probability, then at least one of the two bad events $\mathsf{Bad}_{\mathsf{sig}}$ or $\mathsf{Bad}_{\mathsf{false}}$ occurs with non-negligible probability.

Proof. In Game 3, all honest-setup and liveness conditions already hold except with negligible probability. Thus, if completeness is still violated, the adversary must cause output 0 of Assert to be spent in a way that prevents the honest prover from succeeding. Since output 0 has exactly two spending branches in the protocol, this can only happen if the adversary either (i) spends the timeout branch after expiry without the honest prover's authorization, which is precisely $\mathsf{Bad}_{\mathsf{sig}}$, or (ii) spends the challenge branch by providing a valid hashlock witness before expiry, which is precisely $\mathsf{Bad}_{\mathsf{false}}$.

Claim 5. $\Pr[\mathsf{Bad}_{\mathsf{sig}}] \le \mathsf{negl}(\kappa)$.

Proof. Any successful occurrence of $\mathsf{Bad}_{\mathsf{sig}}$ yields a forgery against the digital signature scheme required to spend output 0 of Assert via the timeout path. Since this signature is not available to the adversary in the honest execution transcript, producing it constitutes a successful EUF-CMA forgery. By the EUF-CMA security of the

signature scheme from Section B.3, this occurs only with negligible probability.

Claim 6. $\Pr[\mathsf{Bad}_{\mathsf{false}}] \le \Pr[\mathbf{Game}_{0'} = 1] + \mathsf{negl}(\kappa)$.

Proof. In Game 3, the adversary can realize $\mathsf{Bad}_{\mathsf{false}}$ only by causing output 0 of Assert to be spent through the challenge path before the timeout expires. By construction of BitVM3-core, this requires a valid witness for the hashlock branch, namely a value $\widetilde{L}$ such that

$$\mathcal{H}(\widetilde{L}) = \mathcal{H}(L^*).$$

In the random oracle model, any such value must satisfy $\widetilde{L} = L^*$.

By the previous hybrids, the adversary receives the honest encoding $L_\pi = \mathsf{En}(e, \pi)$ of a valid proof π. Since π is valid, correctness of the garbling scheme implies that

$$\mathsf{De}\big(d, \mathsf{Ev}(\mathsf{GC}, L_\pi)\big) = \mathsf{True}.$$

On the other hand, by definition of L^*, we have $\mathsf{De}(d, L^*) = \mathsf{False}$. Therefore, except with negligible probability, any successful challenge-path spend yields an adversarially produced encoded output

$$L_y := L^*$$

such that

$$L_y \neq \mathsf{Ev}(\mathsf{GC}, L_\pi) \quad \text{and} \quad \mathsf{De}(d, L_y) = \mathsf{False}.$$

This is precisely a winning output for $\mathbf{Game}_{0'}$. Hence any adversary that causes $\mathsf{Bad}_{\mathsf{false}}$ can be transformed into an adversary for $\mathbf{Game}_{0'}$, with only a negligible loss.

Claim 7. $\Pr[\mathbf{Game}_{0'} = 1] \le \mathsf{negl}(\kappa)$.

Proof. Since π is a valid proof, we have

$$\mathsf{SNARK.Vrfy}(\mathcal{R}, \mathsf{crs}, \phi, \pi) = \mathsf{True}.$$

By the correctness property of the garbling scheme from Section B.2,

$$\mathsf{De}\big(d, \mathsf{Ev}(\mathsf{GC}, L_\pi)\big) = \mathsf{True}.$$

Hence the honestly evaluated output does not decode to False.

A winning adversary in $\mathbf{Game}_{0'}$ must therefore produce a distinct encoded output $L_y \neq \mathsf{Ev}(\mathsf{GC}, L_\pi)$ such that $\mathsf{De}(d, L_y) = \mathsf{False}$. This contradicts the authenticity of the garbling scheme from Section B.2. Moreover, even if the adversary attempts to bypass this by inverting the hashlock commitment directly, it must find a valid preimage for $\mathcal{H}(L^*)$, which occurs only with negligible probability in the random oracle model as $\mathcal{A}$ needs to query the random oracle for L^* which has κ bits and is allowed to do polynomial number of queries. Therefore $\Pr[\mathbf{Game}_{0'} = 1]$ is negligible.

Combining the above bounds, we obtain

$$\begin{aligned}\Pr[\mathbf{Game}_0 = 0] &\le \Pr[\mathbf{Game}_3 = 0] + \mathsf{negl}(\kappa)\\ &\le \Pr[\mathsf{Bad}_{\mathsf{sig}}] + \Pr[\mathsf{Bad}_{\mathsf{false}}] + \mathsf{negl}(\kappa)\\ &\le \Pr[\mathbf{Game}_{0'} = 1] + \mathsf{negl}(\kappa)\\ &\le \mathsf{negl}(\kappa).\end{aligned}$$

This shows that the completeness-violation event occurs only with negligible probability. Hence BitVM3-core is complete. □

Theorem 2 (Soundness of BitVM3-core). *If the prover follows BitVM3-core honestly during setup but may behave arbitrarily during proving, at least one verifier is honest, the ledger protocol is safe and live, the garbling scheme satisfies correctness, the garbled encoding extractable signature scheme satisfies extractability, and the SNARK scheme is knowledge-sound, then BitVM3-core is sound in the random oracle model.*

We prove soundness via a game between a challenger (the honest verifier) and an adversary $\mathcal{A}$ (the malicious prover). The challenger runs the setup phase honestly and gives $\mathcal{A}$ the state that an honest prover would hold after setup; from that point onward, $\mathcal{A}$ controls the proving phase. Here Q is the predicate induced by the underlying relation, namely $Q(w) = 1$ iff $(\phi, w) \in \mathcal{R}$. As in Algorithm 2, the violating outcome is 0: a soundness violation occurs if the success predicate S holds in the view of an honest verifier, yet $Q(\mathsf{Extract}(\mathsf{view}_r^V, \mathsf{sid}))$ does not.

Proof. We proceed through a sequence of hybrid games starting from the real soundness experiment and conditioning on the expected behavior of the honest verifier. In these games, $\mathsf{assert}(\cdot)$ denotes an immediate abort.

Proof idea. The honest verifier observes the witness posted in Assert and extracts from it a garbled encoding L_π. For the analysis, the challenger uses the setup information to interpret this encoding as the one corresponding to some proof π, evaluates the garbled circuit on L_π, and, if the decoded output is False, posts Disprove. The hybrids condition on the correctness of these steps. Once they are fixed, the only remaining way for the adversary to win is if the recovered proof π satisfies the SNARK verifier circuit. If so, by the SNARK's knowledge soundness, a valid witness w must exist, which the experiment extracts.

$\mathbf{Game}_0$: Real Soundness Experiment

$(\mathsf{sid}, S, \mathsf{aux}) \leftarrow \mathbf{Setup}(1^\kappa, Q)$
$(\mathcal{R}, \phi, \mathsf{crs}, \mathcal{G}, f^{\mathsf{C}}, \mathsf{GC}, e, d, L^*, \mathsf{pk}_{\mathsf{GS}}, \mathsf{sk}_{\mathsf{GS}}, \mathsf{Assert}) \leftarrow \mathsf{aux}$
$\mathsf{state}_{\mathcal{A}} \leftarrow (\mathcal{R}, \phi, \mathsf{crs}, \mathcal{G}, f^{\mathsf{C}}, \mathsf{GC}, e, d, L^*, \mathsf{pk}_{\mathsf{GS}}, \mathsf{sk}_{\mathsf{GS}}, \mathsf{Assert})$
$\sigma \leftarrow \mathcal{A}(1^\kappa, \mathsf{state}_{\mathcal{A}})$
$\mathsf{Assert.witness} \leftarrow \sigma$
wait until $\mathsf{Assert} \in \mathcal{L}_{r_0}^V$
$L_\pi \leftarrow \mathsf{GS.Extract}(\mathsf{pk}_{\mathsf{GS}}, \sigma)$
$L_y \leftarrow \mathsf{Ev}(\mathsf{GC}, L_\pi)$
$y \leftarrow \mathsf{De}(d, L_y)$
if $y = \mathsf{True}$ **then return** 1
$\mathsf{Disprove.inputs}[0] \leftarrow (\mathsf{Assert.txid}, 0, \mathcal{H}(L^*))$
$\mathsf{Disprove.outputs}[0] \leftarrow (v, \mathsf{True})$
$\mathsf{Disprove.witness} \leftarrow L_y$
$\Pi_{\mathcal{L}}.\mathsf{write}(\mathsf{Disprove})$
let r_1 *denote the first round such that* $\mathsf{Disprove} \in \mathcal{L}_{r_1}^V$
return $\Big(S(\mathcal{L}_{r_1}^V) \wedge \neg Q(\mathsf{Extract}(\mathsf{view}_{r_1}^V, \mathsf{sid}))\Big)$

Figure 11: The real soundness experiment, denoted $\mathbf{Game}_0$, for BitVM3-core.

Game 0 in Figure 11 is the real soundness experiment corresponding to the soundness condition formalized in Definition 5 via the experiment Sound in Algorithm 2. We define the hybrids

$\mathbf{Game}_0$ with Hybrid Additions

$(\mathsf{sid}, S, \mathsf{aux}) \leftarrow \mathbf{Setup}(1^\kappa, Q)$
$(\mathcal{R}, \phi, \mathsf{crs}, \mathcal{G}, f^{\mathsf{C}}, \mathsf{GC}, e, d, L^*, \mathsf{pk}_{\mathsf{GS}}, \mathsf{sk}_{\mathsf{GS}}, \mathsf{Assert}) \leftarrow \mathsf{aux}$
$\mathsf{state}_{\mathcal{A}} \leftarrow (\mathcal{R}, \phi, \mathsf{crs}, \mathcal{G}, f^{\mathsf{C}}, \mathsf{GC}, e, d, L^*, \mathsf{pk}_{\mathsf{GS}}, \mathsf{sk}_{\mathsf{GS}}, \mathsf{Assert})$
$\sigma \leftarrow \mathcal{A}(1^\kappa, \mathsf{state}_{\mathcal{A}})$
$\mathsf{Assert.witness} \leftarrow \sigma$
wait until $\mathsf{Assert} \in \mathcal{L}^V_{r_0}$
$L_\pi \leftarrow \mathsf{GS.Extract}(\mathsf{pk}_{\mathsf{GS}}, \sigma)$
let π *be the unique bitstring such that* $L_\pi = \mathsf{En}(e, \pi)$
$\mathsf{assert}(L_\pi = \mathsf{En}(e, \pi))$ [G1]
$L_y \leftarrow \mathsf{Ev}(\mathsf{GC}, L_\pi)$
$y \leftarrow \mathsf{De}(d, L_y)$
$\mathsf{assert}(y = \mathsf{SNARK.Vrfy}(\mathcal{R}, \mathsf{crs}, \phi, \pi))$ [G2]
if $y = \mathsf{True}$ **then return** 1
$\mathsf{Disprove.inputs}[0] \leftarrow (\mathsf{Assert.txid}, 0, \mathcal{H}(L^*))$
$\mathsf{Disprove.outputs}[0] \leftarrow (v, \mathsf{True})$
$\mathsf{Disprove.witness} \leftarrow L_y$
$r' \leftarrow \mathsf{CurrentRound}(\Pi_{\mathcal{L}})$ [G3]
$r_1 \leftarrow \Pi_{\mathcal{L}}.\mathsf{write}(\mathsf{Disprove})$
$\mathsf{assert}(r_1 \le r' + u)$ [G3]
let r *denote the first round such that* $\mathsf{Disprove} \in \mathcal{L}^V_r$
return $\left(S(\mathcal{L}^V_r) \wedge \neg Q(\mathsf{Extract}(\mathsf{VIEW}^V_r, \mathsf{sid}))\right)$

Figure 12: The real soundness experiment together with the boxed statements added to obtain hybrids $\mathbf{Game}_1$, $\mathbf{Game}_2$, and $\mathbf{Game}_3$.

Game 1, Game 2, and Game 3 from Game 0 by adding the boxed statements labeled [G1], [G2], and [G3] shown inline in Figure 12.

From Game 0 to Game 1. Game 1 is obtained from Game 0 by adding the boxed statement labeled [G1] in Figure 12, which conditions on consistency between the extracted garbled encoding L_π and the unique bitstring π whose encoding under e is L_π.

Claim 1. $|\Pr[\mathbf{Game}_0 = 0] - \Pr[\mathbf{Game}_1 = 0]| \le \mathsf{negl}(\kappa)$.

Proof. Since Assert appears in the honest verifier's ledger view, its input script has accepted σ under $\mathsf{pk}_{\mathsf{GS}}$. Therefore σ is a valid GS signature, and by extractability the verifier recovers the encoding of some message π.

From Game 1 to Game 2. Game 2 is obtained from Game 1 by adding the boxed statement labeled [G2] in Figure 12, which conditions on correct garbled evaluation of the extracted encoding.

Claim 2. $|\Pr[\mathbf{Game}_1 = 0] - \Pr[\mathbf{Game}_2 = 0]| \le \mathsf{negl}(\kappa)$.

Proof. By correctness of the garbling scheme, for every bitstring π and corresponding encoding $L_\pi = \mathsf{En}(e, \pi)$, it holds that $\mathsf{De}(d, \mathsf{Ev}(\mathsf{GC}, L_\pi)) = \mathsf{SNARK.Vrfy}(\mathcal{R}, \mathsf{crs}, \phi, \pi)$. Hence the additional abort in $\mathbf{Game}_2$ occurs only with negligible probability.

From Game 2 to Game 3. Game 3 is obtained from Game 2 by adding the boxed statements labeled [G3] in Figure 12, which condition on timely inclusion of the honest verifier's Disprove transaction whenever the decoded output is False.

Claim 3. $|\Pr[\mathbf{Game}_2 = 0] - \Pr[\mathbf{Game}_3 = 0]| \le \mathsf{negl}(\kappa)$.

Proof. Whenever the verifier computes $y = \mathsf{False}$, the transaction Disprove is valid by construction since L^* is exactly L_{False} by the honest setup assumption. Let r' be the round immediately before the honest verifier invokes $\Pi_{\mathcal{L}}.\mathsf{write}(\mathsf{Disprove})$, and let r_1 be the round returned by that call. By Lemma 1, we have $r_1 \le r' + u$ except with negligible probability. Therefore, the additional abort condition in Game 3 changes the experiment only negligibly.

It remains to analyze Game 3. By construction of the previous hybrids, if the success predicate S holds on the ledger view $\mathcal{L}^V_r$ of the honest verifier, then the experiment has extracted from the witness of Assert a garbled encoding L_π, evaluated the garbled circuit correctly on L_π, and successfully posted Disprove whenever the decoded output is False. In addition, the setup state retained by the experiment determines the unique proof π such that $L_\pi = \mathsf{En}(e, \pi)$.

Therefore, if $S(\mathcal{L}^V_r)$ still holds in Game 3, the decoded output cannot be False, since otherwise Disprove would be posted and included within the liveness bound. Hence the only remaining possibility is $y = \mathsf{True}$. By the condition fixed in [G2], it follows that $\mathsf{SNARK.Vrfy}(\mathcal{R}, \mathsf{crs}, \phi, \pi) = \mathsf{True}$.

By SNARK knowledge soundness, there exists an efficient extractor $\mathsf{Ext}_{\mathsf{SNARK}}$ that recovers a witness w such that $(\phi, w) \in \mathcal{R}$. We therefore define the BitVM3-core extractor required by Definition 5 as:

$$\mathsf{Extract}(\mathsf{VIEW}^V_r, \mathsf{sid}) := \mathsf{Ext}_{\mathsf{SNARK}}(\mathcal{R}, \mathsf{crs}, \phi, \pi),$$

where π is the unique bitstring such that $L_\pi = \mathsf{En}(e, \pi)$, for the value $L_\pi = \mathsf{GS.Extract}(\mathsf{pk}_{\mathsf{GS}}, \sigma)$ extracted from the witness σ of the Assert transaction appearing in VIEW^V_r.

Consequently, whenever $S(\mathcal{L}^V_r)$ holds, $Q(\mathsf{Extract}(\mathsf{VIEW}^V_r, \mathsf{sid})) = 1$ except with negligible probability. Hence $\Pr[\mathbf{Game}_3 = 0] \le \mathsf{negl}(\kappa)$.

Combining the above bounds, we obtain

$$\begin{aligned}\Pr[\mathbf{Game}_0 = 0] &\le \Pr[\mathbf{Game}_1 = 0] + \mathsf{negl}(\kappa)\\ &\le \Pr[\mathbf{Game}_2 = 0] + \mathsf{negl}(\kappa)\\ &\le \Pr[\mathbf{Game}_3 = 0] + \mathsf{negl}(\kappa)\\ &\le \mathsf{negl}(\kappa).\end{aligned}$$

Thus the soundness-violation event occurs only with negligible probability, and BitVM3-core is sound. □

D Light Client Protocol and Analysis

D.1 Light Client Protocol

The light client protocol involves two parties: an *operator*, who maintains and advances the checkpoint chain, and any number of permissionless *challengers*, who monitor the chain and penalize misbehavior. The protocol is parameterized by the number of checkpoints c (so that $c/26$ is the lifetime of the light client in years), an initial input UTXO $txin_0$, and an initial checkpoint C_{-1}. The full pseudocode is given in Figure 14.

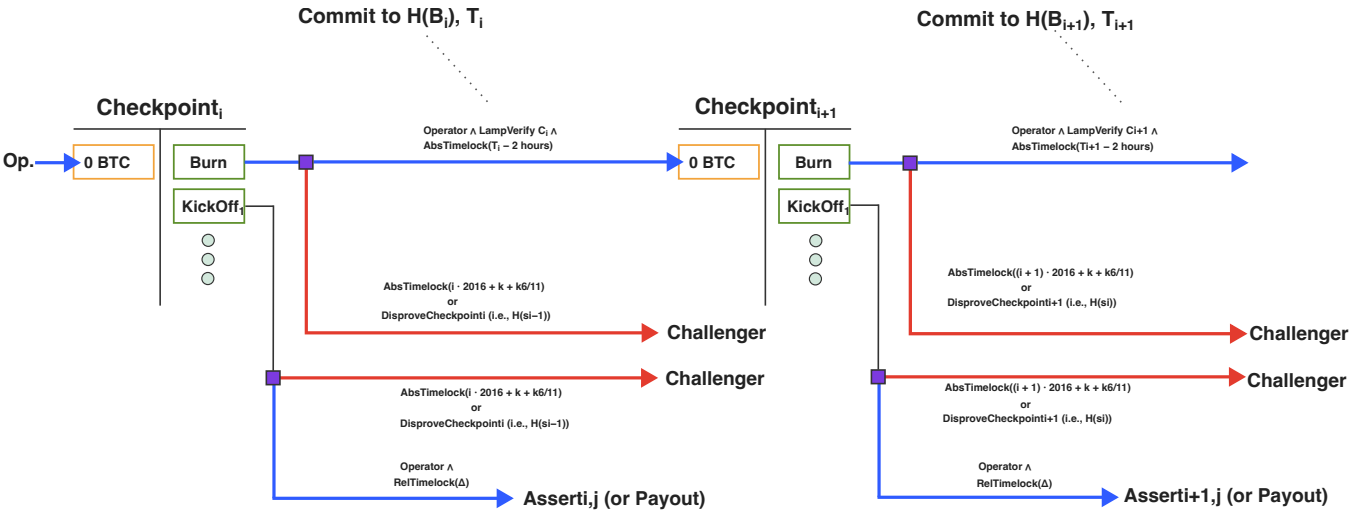

Figure 13: Light Client transaction graph

Light Client

Public parameters. c, the number of checkpoints. Note that $c/26$ is the lifetime of the light client in years; $txin_0$, the input given to the first checkpoint; C_{-1}, the initial checkpoint.

Setup

Operator

(1) For each $i \in [0, c]$:
- Define the predicate $Q_i(\pi) \triangleq \mathsf{SNARK.Vrfy}(\mathcal{R}_{\text{btc-lc}}, \mathsf{crs}, \phi_i, \pi)$, where $\phi_i := (H(B_i), H(B_{i-1}), T_i, \mathsf{sid})$ and $\mathcal{R}_{\text{btc-lc}}$ is defined in Algorithm 3.
- If $i = 0$, let $in \leftarrow txin_0$; otherwise let $in \leftarrow \mathsf{Checkpoint}_{i-1}[0]$.
- Run BitVM3-core.**Setup**$(1^\kappa, Q_i) \rightarrow (\mathsf{sid}_i, S_i, \mathsf{aux}_i)$, where aux_i contains the 288 + 160 bit pk_{GS} together with the transaction template $\mathsf{Checkpoint}_i$ constructed with input in and locking script $\mathsf{CheckSig}_{\mathsf{pk}_{\text{GS}}}$, and where $S_i := \mathsf{Checkpoint}_i \in \Pi_{\mathcal{L}}.\mathsf{read}() \wedge \mathsf{CheckExpired}(t)$.
- Store $(\mathsf{sid}_i, \mathsf{aux}_i)$.

Execute

Operator (every round)

(1) Let $L \leftarrow \Pi_{\mathcal{L}}.\mathsf{read}()$
(2) If $|L| \not\equiv 0 \mod 2016$ go idle.
(3) Let $B \leftarrow L[-1]$
(4) Let $k = L/2016$
(5) Let T_k be $L[-2]$.timestamp.
(6) Let $\phi_k := H(B)||T_k$
(7) Compute $\pi_k \leftarrow \mathsf{SNARK.Prove}(\mathcal{R}_{\text{btc-lc}}, \mathsf{crs}, \phi_k, w_k)$.
(8) Execute the BitVM3-core proving procedure for checkpoint k
- **Operator:** Construct and broadcast $\mathsf{Checkpoint}_k$, embedding in its witness the garbled encoding extractable signature on $\phi_k||\pi_k$.
- **Challenger:** Evaluate the garbled circuit off-chain. If the claimed proof is invalid, obtain the false-output label L^* and broadcast $\mathsf{DisproveCheckpoint}_k$ before timeout t.

Challenger (every round)

(1) If $\Pi_{\mathcal{L}}.\mathsf{read}()$ contains a $\mathsf{Checkpoint}_i$ transaction for any $i \in [0, c]$, where $\mathsf{Checkpoint}_i$'s outputs are contains a spending path locked only with timelocks that have expired, create a transaction tx' spending this output, and post tx' to $\Pi_{\mathcal{L}}$.

Figure 14: LC Protocol description.

where $\mathsf{BurnScript}_i$ is defined as $\mathsf{CheckGS}_{\mathsf{pk}_{\text{GS},i}} \wedge \mathsf{CheckSig}_{pk_O} \wedge \mathsf{AbsTimelock}(T_i - 2\text{h})$. Here, $\mathsf{DisproveCheckpoint}_i$ denotes the dispute branch for checkpoint instance i; its witness must contain the false-output label L_i^* of the garbled circuit for checkpoint i, namely the unique label satisfying $\mathsf{De}(d_i, L_i^*) = \text{False}$.

Setup. Before any checkpoint is posted on-chain, the operator performs a one-time setup for each epoch $i \in [0, c]$. For each epoch, the operator defines the SNARK verification predicate Q_i for the relation $\mathcal{R}_{\text{btc-lc}}$ (see Algorithm 3) with public inputs $\phi_i =$

Table 10: The $\mathsf{Checkpoint}_i$ transaction template.

$\mathsf{Checkpoint}_i$ **Transaction**	
Inputs	(0) $(\mathsf{Checkpoint}_{i-1},\ 0,\ \mathsf{BurnScript}_{i-1})$
Outputs	(0) $\big(0\ \text{BTC},\ \langle \mathsf{BurnScript}_i,\ \mathsf{AbsTimelock}(i \cdot 2016 + k + k_{6/11}) \vee \mathsf{DisproveCheckpoint}_i \rangle\big)$ (1) ... $(j)^9$ $\big(0\ \text{BTC},\ \langle \mathsf{AbsTimelock}(i \cdot 2016 + k + k_{6/11}) \vee \mathsf{DisproveCheckpoint}_i,\ \mathsf{CheckSig}_{pk_O} \wedge \mathsf{RelTimelock}(t) \rangle\big)$
Witness	(0) $\sigma_O,\ \sigma_{\mathrm{GS},i}$

Algorithm 3 The chainstate proof $\Pi_{\text{lc-btc}}$, defining the SNARK relation R. $\phi = (\mathcal{H}(B_i), \mathcal{H}(B_{i-1}), T_i, \mathsf{sid})$ and $w := [B_0, ..., B_{2016}]$. The algorithm checks that there is a valid Bitcoin chain connecting the last checkpoint to the current checkpoint, with correct difficulty calculation and timestamp matching T_i. Blocks have fields ts (timestamp), prev (hash of previous block), and difficulty.

function $\Pi_{\text{lc-btc}}(\phi, w)$
  parse ϕ as $((\mathcal{H}(B_i), \mathcal{H}(B_{i-1}), T_i, \mathsf{sid})$
  parse w as $[B_0, ..., B_{2016}]$
  assert $|w| = 2017$
  assert $\mathcal{H}(w[0]) = \mathcal{H}(B_0)$
  assert $\mathcal{H}(w[2016]) = \mathcal{H}(B_{2016})$
  assert $w[2015].\mathrm{ts} = T_i$
  $\mathrm{difficulty}_{old} \leftarrow w[0].\mathrm{difficulty}$
  for $i \in [1, 2016]$ **do**
    assert $w[i].\mathrm{prev} = \mathcal{H}(w[i-1])$
    if $i \neq 2016$ **then assert** $w[i].\mathrm{difficulty} = \mathrm{difficulty}_{old}$
  assert $w[2016].\mathrm{difficulty} = \mathrm{difficulty}_{old} \frac{2\ \text{weeks}}{w[2015].\mathrm{ts} - w[0].\mathrm{ts}}$

$(\mathcal{H}(B_i), \mathcal{H}(B_{i-1}), T_i, \mathsf{sid})$, and runs the BitVM3-core setup procedure to obtain a session identifier sid_i and auxiliary data aux_i. The auxiliary data encodes the transaction template $\mathsf{Checkpoint}_i$, whose input is either $txin_0$ (for $i = 0$) or Output 0 of $\mathsf{Checkpoint}_{i-1}$ (for $i > 0$), and whose locking script embeds the garbled-signature verification key $\mathsf{pk}_{\mathsf{GS}}$. The operator stores $(\mathsf{sid}_i, \mathsf{aux}_i)$ for all i and distributes the transaction templates and pre-signed Burn_i witnesses to all challengers.

Execution. During execution the operator monitors the Bitcoin chain every round. Whenever the chain length is a multiple of 2016, a new epoch boundary has been reached. The operator reads the hash $\mathcal{H}(B)$ of the last block and the timestamp T_k of the second-to-last block (which Bitcoin uses for the difficulty adjustment), constructs the public input $\phi_k = \mathcal{H}(B) \| T_k$, and computes a SNARK proof π_k for $\mathcal{R}_{\text{btc-lc}}$. It then runs the BitVM3-core proving procedure, broadcasting $\mathsf{Checkpoint}_k$ with a garbled-encoding extractable signature on $\phi_k \| \pi_k$ as witness.

Challengers evaluate the garbled circuit off-chain upon seeing $\mathsf{Checkpoint}_k$. If the embedded proof is invalid, a challenger obtains the false-output label L^* and broadcasts $\mathsf{DisproveCheckpoint}_k$ before the BitVM3-core timeout t expires, slashing the operator. Additionally, challengers monitor all $\mathsf{Checkpoint}_i$ outputs: if any output becomes spendable via a timelock-only path (i.e., the operator has missed the liveness deadline or failed to spend the timestamp-enforcement output before Burn_i fires), any challenger broadcasts a transaction spending that output, permanently invalidating the checkpoint chain.

D.2 Light Client Analysis

Recall the following definitions from [3, 20].

Definition 6 (Common Prefix Property [20]). 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 $r_1 \leq r_2$ in $\text{view}^{t,n}_{\Pi,A,Z}$ respectively, it holds that $C_1^{\lceil k \rceil} \preceq C_2$.

Definition 7 (Chain Quality Property [20]). 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 8 (Chain Growth Property [20]). 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.

Definition 9 ((u,k)-Admissible Block at r [3]). Consider $u, k \in \mathbb{N}$. A block B that, at round r, fulfils the following properties is an *admissible block at r*:

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

Proof strategy. We will prove this theorem in two steps. First, we will argue that for one epoch, i.e., going from one checkpoint to the next, the block returned by the light client is admissible if the last block of the previous epoch was admissible. Then we will argue that no adversary can commit to a fake timestamp and succeed in providing the next checkpoint (thus leaving the difficulty intact). Finally, we will show that given the admissible block of the previous epoch and the fact that the next checkpoint has been posted, the last block of the previous epoch must be admissible. Recall c is the parameter how many checkpoints there are.

Lemma 2. *For any $i \in [0, c]$, let $\phi = (\mathcal{H}(B_i), \mathcal{H}(B_{i-1}), T_i)$ be the statement corresponding to the BitVM3-core instance of $\mathsf{Checkpoint}_i$, i.e., the Lamport signatures revealed at $\mathsf{Checkpoint}_i$ $(\mathcal{H}(B_i), T_i)$ and the Lamport signature revealed at $\mathsf{Checkpoint}_{i-1}$ $(\mathcal{H}(B_{i-1}))$.*

(i) There exists a valid chain of length 2017, i.e., $w = [B_0, B_1, \ldots, B_{2016}]$, such that $\mathcal{H}(B_0) = \mathcal{H}(B_{i-1})$ and $\mathcal{H}(w[2016]) = \mathcal{H}(w[i])$.

(ii) If B_0 is admissible, then for any $m > (\frac{2}{\delta} - 1)x$, $w[2016 - m]$ is a (u,k)-admissible B, where k is the common prefix parameter, $\delta \in (0,1]$ the honest advantage, i.e., the adversary controls at most a $(1-\delta)$ fraction of the honest hashing power, and u the time it takes for the chain of any honest party to grow by more than $m + x$ blocks.

Proof. (i) holds directly from completeness and soundness of ChainVM (implemented by BitVM3-core) and the fact that it is instantiated with a relation that check precisely that, i.e., Algorithm 3.

(ii) was proven in [35, Theorem G.4]. Note that x is the time advantage a malicious operator has with respect to the honest chain, until he needs to produce the checkpoint. In our case $x = m_t + k$. □

Lemma 3. *Let $\phi = (\mathcal{H}(B_i), \mathcal{H}(B_{i-1}), T_i)$ be the statement corresponding to the BitVM3-core instance of* $\mathsf{Checkpoint}_i$*, i.e., the Lamport signatures revealed at* $\mathsf{Checkpoint}_i$ *$(\mathcal{H}(B_i), T_i)$ and the Lamport signature revealed at* $\mathsf{Checkpoint}_{i-1}$ *$(\mathcal{H}(B_{i-1}))$.*

Let w be the corresponding witness to ϕ. If B_0 is admissible, then each block $B \in w$ has correct *difficulty up to ϵ_D, i.e., the same difficulty as the block with the same height in the honest chain, i.e., $C^{\cap}$.*

Proof. Since B_0 is admissible, it has the correct difficulty by definition of admissible block. We know that all block in $w[:-1]$ must have the same difficulty, by completeness and soundness of ChainVM (implemented by BitVM3-core) and the fact that it is instantiated with a relation that check precisely that, i.e., Algorithm 3.

For the remaining block $w[-1]$, we know that this block will have the same (or harder) difficulty up to ϵ_D, if it has the *correct* (or smaller[10]) timestamp, i.e., its timestamp matches the timestamp of the block with the same height in the honest chain, i.e., $C^{\cap}$. It remains to show this.

Suppose an operator tries to commit to a valid, but fraudulent $w[-1]$ with a timestamp T_i that is in the future (which would result in a lower difficulty in the next period). Note that by construction of the $\mathsf{Checkpoint}_i$ transaction, and its input spending a script that checks an absolute timelock of $T_i - 2$ hours, we know that T_i cannot be in the future with respect to the time of the honest chain by more than 2 hours $+ m_t + k$, where $m_t + k$ is the time in the slashing condition of that output. If an operator sets T_i to be more than 2 hours $+ m_t + k$ into the future, any challenger can and will slash this output, preventing $\mathsf{Checkpoint}_i$ from going on-chain.

Note that setting T_i to be 2 hours $+ m_t + k$ in the future will yield the maximum allowable difficulty decrease of ϵ_D, which a malicious operator can achieve. □

Lemma 4. *Let $\phi = (\mathcal{H}(B_i), \mathcal{H}(B_{i-1}), T_i)$ be the statement corresponding to the BitVM3-core instance of* $\mathsf{Checkpoint}_i$*, i.e., the Lamport signatures revealed at* $\mathsf{Checkpoint}_i$ *$(\mathcal{H}(B_i), T_i)$ and the Lamport signature revealed at* $\mathsf{Checkpoint}_{i-1}$ *$(\mathcal{H}(B_{i-1}))$.*

If $\mathsf{Checkpoint}_{i+1}$ *goes on-chain with the corresponding statement $\phi' = (\mathcal{H}(B_{i+1}), \mathcal{H}(B_i), T_{i+1})$, and let w and w' be the witnesses corresponding to ϕ and ϕ' respectively, and Given that the blocks in w and w' have the* correct *(or harder) difficulty up to ϵ_D, then $w[2016 - m]$ admissible implies $w[2016]$ admissible.*

Proof. This lemma follows from [35, Theorem G.4] and Lemma 2. □

Observation. Note that Lemmas 2 and 4 imply that also $w'[2016 - m]$ is admissible. We also know that B_0 was chosen at setup time between the operator and signers, such that they agree that B_0 is admissible. We can see that this proof follows an induction-like shape.

Theorem 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, $\delta \in (0, 1]$ the honest advantage, i.e., the adversary controls at most a $(1 - \delta)$ fraction of the honest hashing power, u the time it takes for the chain of any honest party to grow by more than $m + x$ blocks, and $x = m_t + k$ the operator's time advantage over the honest chain, assuming that the underlying ledger satisfies common prefix, chain quality, and chain growth [20], and that BitVM3-core is sound and complete.*

Proof. Lemmas 2, 3, 4 directly imply the safety part of Theorem 3. It remains to show that the operator *can* always commit to a valid statement ϕ at every $\mathsf{Checkpoint}_i$.

The only problem that can arise is that the operator is unable to post $\mathsf{Checkpoint}_i$. This could happen, if for instance T_i is correct, but the chain time has not yet caught up with T_i. From [22] and in particular, the *concentrated chain quality* property, we know that there exists a number of blocks, m_t, after which we know that the chain time will catch up with T_i. And since the locking condition of the slashing path is locked by m_t, we know that the operator will be able to post the transaction. k ensures that additionally, the checkpoint will be finalized. □

E Bridge Security Analysis

Algorithm 4 The chainstate proof $\Pi_{\text{eth-bridge}}$, defining the SNARK relation R. $\phi = (C_{\text{anchor}}, \text{txid}, \text{sid})$ and $w := \big((C_1, \sigma_1), \ldots, (C_n, \sigma_n), B, \sigma_B, \text{burn}, \pi_{\text{inc}}\big)$. The algorithm checks that there is a valid chain of sync committee handoffs from C_{anchor} to a committee C_n that finalizes block B, that burn is included in B, and that burn references the deposit $(\text{txid}, \text{sid})$.

```
function Π_eth-bridge(φ, w)
    parse φ as (C_anchor, txid, sid)
    parse w as ((C_1, σ_1), ..., (C_n, σ_n), B, σ_B, burn, π_inc)
    C ← C_anchor
    for i ∈ {1, ..., n} do
        if ¬VerifyHandoff(C, C_i, σ_i) then
            return False
        C ← C_i
    if ¬VerifyFinality(C, B, σ_B) then
        return False
    if ¬VerifyInclusion(burn, B, π_inc) then
        return False
    return (burn.deposit_ref = txid ‖ sid)
```

In this section, we formalize the security properties of BitVM3-bridge, closely following [35].

E.1 Formal Definitions

We assume a global clock governs both ledger protocols, $\Pi_{\mathcal{L}_A}$ and $\Pi_{\mathcal{L}_B}$. To consistently track assets across chains, we associate each Peg-In event with a unique identifier `id`. This identifier links all transactions across both ledgers that originate from the same Peg-In, allowing the bridge to reason about asset continuity.

Definition 10 (Unidirectional Bridge). Let $X \in \{A, B\}$ and let $V_{\mathcal{L}_{X,\text{id}}}$ be the set of syntactically valid transactions w.r.t. assets with identifier `id` in $\Pi_{\mathcal{L}_X}$. Parameterized by a binary relation $R(\text{id}) \subseteq \wp(V_{\mathcal{L}_{A,\text{id}}}) \times \wp(V_{\mathcal{L}_{B,\text{id}}})$, a *bridge* $\Lambda_{R(\text{id})}(\Pi_{\mathcal{L}_A}, \Pi_{\mathcal{L}_B})$ is a protocol that relays requests (e.g., coin transfers) issued on a source ledger $\mathcal{L}_A$ to a destination ledger $\mathcal{L}_B$.

[10] If the operator maliciously tries to set the timestamp of $w[-1]$ to something smaller, this will result in a higher difficulty.

Ethereum BitVM3-bridge Protocol Execution

Phase 1 : Peg − in (Setup & Deposit Swap)

User (Alice)

(1) Sample a high-entropy cancellation secret s_a and compute its hash $\mathcal{H}(s_a)$.
(2) Broadcast Request on Bitcoin locking u BTC. ▷ Paths: deposit (requires n-of-n committee signature) and cancel (requires s_a after a timelock).

Operator (Prover) & Signer Committee (Verifiers)

(1) Define predicate $Q(\pi) \triangleq \mathsf{SNARK.Vrfy}(\mathcal{R}_{\mathsf{eth-bridge}}, \mathsf{crs}, \phi, \pi)$ for public inputs $\phi = (C_{\mathsf{anchor}}, \mathsf{txid}, \mathsf{sid})$ with $\mathcal{R}_{\mathsf{eth-bridge}}$ as defined in Algorithm 4.
(2) Run BitVM3-core.**Setup**$(1^\kappa, Q) \to (\mathsf{sid}, S, \mathsf{aux})$ to generate the garbled payload and Assert skeleton.
(3) Signers verify aux and confirm Request is included in Bitcoin.
(4) Signers generate signatures for Deposit (spending Request to the bridge reserve) and Withdraw (spending Deposit with the Assert as a connector input).

User (Alice) & Smart Contract (Ethereum)

(1) Alice registers $\mathcal{H}(s_a)$ and the Deposit committee signature on the Ethereum smart contract.
(2) Any party broadcasts the pre-signed Deposit transaction on Bitcoin.
(3) The smart contract initiates a delay window $\Delta > u_{\mathcal{L}_A} + u_{\mathcal{L}_B}$. If no party relays s_a to the contract before Δ expires, the contract mints u wBTC to Alice.

Phase 2 : Transfer (Swap Markets)

Users & Operators

(1) Users trade wBTC freely on Ethereum.
(2) Operators provide liquidity and process client peg-outs via atomic swaps, acquiring u wBTC and paying $u - f_O$ BTC to the client.

Phase 3 : Peg − out (BitVM3-core Execute & Claim)

Operator (Prover)

(1) Burn u wBTC on Ethereum, referencing Deposit.txid and sid.
(2) Generate a valid SNARK proof π satisfying Q for the finality certificate of the burn.

Operator (Prover) & Challenger (Verifier)

(1) Execute the BitVM3-core.**Execute**$(\mathsf{sid}, \pi, \mathsf{aux})$ protocol on Bitcoin:
   - **Prover:** Broadcasts Assert, locking the operator's bond and committing to the garbled proof labels.
   - **Challenger:** Evaluates the garbled circuit off-chain. If $Q(\pi) = 0$, it obtains the fraud witness label L^* and broadcasts Disprove to slash the operator before timeout t.
   - **Resolution:** If no Disprove occurs before timeout t expires, the predicate S holds.

Operator (Prover)

(1) Broadcast the pre-signed Withdraw transaction, consuming both the Deposit reserve UTXO and the Assert connector output, reclaiming u BTC from the reserve.

Figure 15: The Ethereum instantiation of BitVM3-bridge using BitVM3-core.

A bridge protocol is *robust* if it satisfies both *bridge liveness* and *bridge safety* in every execution I. Liveness ensures honest parties can successfully bridge their funds, while safety ensures coins are never minted or unlocked without a corresponding valid action on the opposing chain, thereby preserving the total supply.

Definition 11 (Bridge Liveness). A bridge protocol $\Lambda_{R(\mathtt{id})}(\Pi_{\mathcal{L}_A}, \Pi_{\mathcal{L}_B})$ is *live* with parameter $u_\ell \in \mathbb{N}$ during execution 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 $r_2 \geq r_1 + u_\ell$, the following holds: for all $(\mathcal{A}, \mathcal{B}) \in R(\mathtt{id})$, if $\mathtt{tx} \in \mathcal{A}$ is included in ${}^{r_1}\mathcal{L}_A^{P_1}$, then there exists $\mathtt{tx}' \in \mathcal{B}$ included in ${}^{r_2}\mathcal{L}_B^{P_2}$ with overwhelming probability.

Definition 12 (Bridge Safety). A bridge protocol $\Lambda_{R(\mathtt{id})}(\Pi_{\mathcal{L}_A}, \Pi_{\mathcal{L}_B})$ is *safe* with parameter $u_s \in \mathbb{N}$ during execution 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 $r_2 \leq r_1 + u_s$, the following holds: for all $(\mathcal{A}, \mathcal{B}) \in R(\mathtt{id})$, if $\mathtt{tx}' \in \mathcal{B}$ is included in ${}^{r_2}\mathcal{L}_B^{P_2}$, then there exists $\mathtt{tx} \in \mathcal{A}$ included in ${}^{r_1}\mathcal{L}_A^{P_1}$ with overwhelming probability.

Definition 13 (Wrapped Asset Bridge). A *wrapped asset bridge* between ledgers $\mathcal{L}_A$ and $\mathcal{L}_B$ is defined by two unilateral bridge instances:

$$\Lambda_{R_1}^{\mathrm{PegIn}}(\Pi_{\mathcal{L}_A}, \Pi_{\mathcal{L}_B}) \quad \text{and} \quad \Lambda_{R_2}^{\mathrm{PegOut}}(\Pi_{\mathcal{L}_B}, \Pi_{\mathcal{L}_A}),$$

both associated with a common asset identifier id. The relation $R_1(\mathtt{id})$ captures *Peg-In* behavior (locking on $\mathcal{L}_A$ implies minting on $\mathcal{L}_B$), and $R_2(\mathtt{id})$ captures *Peg-Out* behavior (burning on $\mathcal{L}_B$ implies unlocking on $\mathcal{L}_A$).

Definition 14 (Swap Operator Safety). A bridge protocol satisfies *Swap Operator Safety* with recovery parameter $u_\Lambda \in \mathbb{N}$ if, for any honest operator p: If there is a transaction $\mathtt{tx} \in \mathsf{Burn}(\mathtt{id})$ executed by p that is included in ${}^{r_1}\mathcal{L}_B^{P_B}$ for all honest parties P_B at some round r_1, then there must exist a transaction $\mathtt{tx}' \in \mathsf{Withdraw}(\mathtt{id})$ paying p such that for every round $r_2 \geq r_1 + u_\Lambda$, $\mathtt{tx}' \in {}^{r_2}\mathcal{L}_A^{P_A}$ for all honest parties P_A.

Definition 15 (Front Operator Safety). A bridge protocol satisfies *Front Operator Safety* with recovery parameter $u_\Lambda \in \mathbb{N}$ if, for any

honest operator p: If there is a transaction $\mathtt{tx} \in \mathsf{PegOut}(\mathtt{id})$ executed by p that is included in ${}^{r_1}\mathcal{L}_B^{P_B}$ for all honest parties P_B at some round r_1, then there must exist a transaction $\mathtt{tx}' \in \mathsf{Withdraw}(\mathtt{id})$ paying p such that for every round $r_2 \geq r_1 + u_\Lambda$, $\mathtt{tx}' \in {}^{r_2}\mathcal{L}_A^{P_A}$ for all honest parties P_A.

E.2 Security Proofs for the Ethereum Bridge

We now instantiate these definitions for our Ethereum wrapped asset bridge. Let $\mathcal{L}_A$ be Bitcoin and $\mathcal{L}_B$ be Ethereum. The bridge relations are defined precisely by the transaction sets introduced in Section 6.2:

$$R_1(\mathtt{id}) = \{(\mathsf{Deposit}(\mathtt{id}), \mathsf{Mint}(\mathtt{id}))\}$$
$$R_2(\mathtt{id}) = \left\{\left(\mathsf{Swap}_{\mathcal{L}_\mathrm{A}}(\mathtt{id}), \mathsf{Swap}_{\mathcal{L}_\mathrm{B}}(\mathtt{id})\right)\right\}$$

E.2.1 Peg-In Bridge Security.

Theorem 4 (Ethereum Peg-In Safety). *The Peg-In bridge $\Lambda_{R_1}^{PegIn}$ satisfies safety with parameter u_s, assuming the signer committee behaves honestly during setup, there exists an honest party who monitors both ledgers, the signature scheme is EUF-CMA secure, and $\Pi_{\mathcal{L}_A}$ and $\Pi_{\mathcal{L}_B}$ satisfy ledger safety and liveness.*

Proof. Assume, for contradiction, that $\mathtt{tx}' \in \mathsf{Mint}(\mathtt{id})$ is included in ${}^{r_2}\mathcal{L}_B^{P_B}$ for an honest party P_B, but no $\mathtt{tx} \in \mathsf{Deposit}(\mathtt{id})$ exists in ${}^{r_1}\mathcal{L}_A^{P_A}$ for any honest party P_A at any round $r_1 \geq r_2 - u_s$.

By the protocol construction, before providing the signature for $\mathtt{tx} \in \mathsf{Deposit}(\mathtt{id})$, the signer committee verifies that $\mathtt{tx}_{req} \in \mathsf{Request}(\mathtt{id})$ is included in ${}^{r}\mathcal{L}_A^{P_A}$ for some round r. Since $\mathtt{tx}$ is not present in the required window, this implies one of the following cases:

Case 1: The Request(id) transaction was incorrectly formed or failed to persist on $\mathcal{L}_A$.

- *Setup failure.* The signer committee pre-signed a spending condition for $\mathtt{tx}_{req}$ diverting funds to a path other than $\mathtt{tx} \in \mathsf{Deposit}(\mathtt{id})$. This violates the assumption that signers behave honestly during setup.
- *Safety failure of $\mathcal{L}_A$.* $\mathtt{tx}_{req}$ was included in ${}^{r}\mathcal{L}_A^{P_A}$, but at some round $r' > r$, it was dropped without being spent. This violates the self-consistency and therefore safety of $\mathcal{L}_A$.

Case 2: Request(id) persisted on $\mathcal{L}_A$, but Deposit(id) failed to execute without canceling the mint on $\mathcal{L}_B$.

- *Invalid state transition or Forgery.* $\mathtt{tx}_{req}$ was spent using a condition other than the deposit or cancel paths. This requires either a consensus failure on $\mathcal{L}_A$ or an adversary forging a signature to authorize an invalid spend, which violates the EUF-CMA security of the underlying signature scheme.
- *Liveness failure during the delay window.* The peg-in request on $\mathcal{L}_B$ at round r_{req} initiates a delay window $\Delta > u_{\mathcal{L}_A} + u_{\mathcal{L}_B}$. If $\mathtt{tx}_{req}$ is not canceled by the client, the failure to include $\mathtt{tx}$ on $\mathcal{L}_A$ within $u_{\mathcal{L}_A}$ rounds constitutes a liveness failure of $\mathcal{L}_A$. Alternatively, to prevent the inclusion of $\mathtt{tx}$, $\mathtt{tx}_{req}$ must be spent via its cancel path by round $r'' \leq r_{req} + u_{\mathcal{L}_A}$, revealing s_a. This leaves at least $u_{\mathcal{L}_B}$ rounds to relay s_a to $\mathcal{L}_B$. If the honest monitoring party submits this relay transaction but it fails to be included and abort the mint before Δ expires, this constitutes a liveness failure of $\mathcal{L}_B$.
- *Unrelayed cancellation.* If $\mathtt{tx}_{req}$ is canceled on $\mathcal{L}_A$ revealing s_a, but no party submits a transaction to $\mathcal{L}_B$ to relay s_a and abort the mint, this violates the assumption of the existential honesty of a party who monitors both ledgers.

In all cases, we reach a contradiction. Thus, $\mathtt{tx} \in \mathsf{Deposit}(\mathtt{id})$ must exist, satisfying Peg-In safety. □

Theorem 5 (Ethereum Peg-In Liveness). *The Peg-In bridge $\Lambda_{R_1}^{PegIn}$ satisfies liveness with parameter u_ℓ, assuming the signer committee behaves honestly during setup, and $\Pi_{\mathcal{L}_B}$ satisfies ledger liveness and safety, and hash functions are modeled as random oracles.*

Proof. Assume, for contradiction, that $\mathtt{tx} \in \mathsf{Deposit}(\mathtt{id})$ is included in ${}^{r}\mathcal{L}_A^{P_A}$ for an honest party P_A, but no corresponding $\mathtt{tx}' \in \mathsf{Mint}(\mathtt{id})$ is included in ${}^{r'}\mathcal{L}_B^{P_B}$ for an honest party P_B at any round $r' \geq r + u_\ell$. This implies one of the following cases:

- *Setup failure.* The mint smart contract on $\mathcal{L}_B$ was incorrectly configured. This violates the assumption that signers behave honestly during setup.
- *False cancellation.* The mint on $\mathcal{L}_B$ was aborted because an adversary submitted s_a to $\mathcal{L}_B$, although the cancel path on $\mathcal{L}_A$ was not spent. Assuming s_a has κ bits, a pre-image attack succeeds with $2^{-\kappa}$, i.e., negligible probability in the random oracle model.
- *Liveness failure of $\mathcal{L}_B$.* $\mathtt{tx}' \in \mathsf{Mint}(\mathtt{id})$ was submitted to $\mathcal{L}_B$ but was delayed beyond the ledger liveness parameter $u_{\mathcal{L}_B}$. This contradicts the liveness of $\mathcal{L}_B$.
- *Safety failure of $\mathcal{L}_B$.* $\mathtt{tx}' \in \mathsf{Mint}(\mathtt{id})$ was temporarily included in $\mathcal{L}_B$ but was subsequently dropped due to a chain reorganization and did not persist at $r' \geq r + u_\ell$. This violates the self-consistency and thus safety of $\mathcal{L}_B$.

In all cases, we reach a contradiction. Thus, $\mathtt{tx}'$ must eventually be included and persist in $\mathcal{L}_B$, satisfying Peg-In liveness. □

Remark (Peg-in correctness). Peg-in liveness begins with Deposit, which the committee signs. It therefore provides no guarantee to a user whose Request is never converted into a Deposit. We capture this guarantee separately, following the setup correctness property for Bitcoin vaults [5].

Specifically, if an honest user includes $\mathtt{tx}_{req} \in \mathsf{Request}(\mathtt{id})$ in $\mathcal{L}_A$ at round r and does not cancel the request, then some $\mathtt{tx} \in \mathsf{Deposit}(\mathtt{id})$ is included in $\mathcal{L}_A$, and some $\mathtt{tx}' \in \mathsf{Mint}(\mathtt{id})$ is included in $\mathcal{L}_B$, both for all honest parties by round

$$r + \delta_{\text{setup}} + u_{\mathcal{L}_A} + u_\ell,$$

provided that the committee is available to sign, an operator executes the BitVM3-core setup, and $\Pi_{\mathcal{L}_A}$ is safe and live.

The setup and the committee's signature complete within δ_{setup}, after which the deposit is included within $u_{\mathcal{L}_A}$ rounds by the liveness of $\mathcal{L}_A$. Since only the user knows s_a, no other party can cancel the request. Finally, Theorem 5 guarantees inclusion of the corresponding mint within an additional u_ℓ rounds.

E.2.2 Peg-Out Bridge Security.

Theorem 6 (Ethereum Peg-Out Safety). *The Peg-Out bridge $\Lambda_{R_2}^{PegOut}$ satisfies Peg-Out Safety with safety parameter u_s, assuming*

$\Pi_{\mathcal{L}_A}$ *and* $\Pi_{\mathcal{L}_B}$ *are safe and live, and the underlying atomic swap protocol is secure.*

Proof. Assume, for contradiction, that $\texttt{tx}' \in \text{Swap}_{\mathcal{L}_A}(\texttt{id})$ paying the user is included in ${}^{r_2}\mathcal{L}_A^{P_A}$ for an honest party P_A, but no corresponding transaction $\texttt{tx} \in \text{Swap}_{\mathcal{L}_B}(\texttt{id})$ locking or transferring the user's wBTC exists in ${}^{r_1}\mathcal{L}_B^{P_B}$ for any honest party P_B at any valid round $r_1 \geq r_2 - u_s$.

This scenario implies one of the following cases:

- *Atomic swap failure.* An adversarial client bypassed the swap condition on $\mathcal{L}_A$ without fulfilling the required cryptographic lock (e.g., revealing a hash pre-image or signature) on $\mathcal{L}_B$. This violates the assumed security of the atomic swap protocol.
- *Safety failure of* $\mathcal{L}_B$. The atomic swap protocol executed correctly and $\texttt{tx} \in \text{Swap}_{\mathcal{L}_B}(\texttt{id})$ was initially included in $\mathcal{L}_B$ but tx subsequently failed to persist due to a chain reorganization. This violates the self-consistency and therefore safety of $\mathcal{L}_B$.

In all cases, we reach a contradiction. Thus, $\texttt{tx} \in \text{Swap}_{\mathcal{L}_B}(\texttt{id})$ must exist, satisfying Peg-Out Safety. □

Theorem 7 (Ethereum Peg-Out Liveness). *The Peg-Out bridge* $\Lambda_{R_2}^{PegOut}$ *satisfies Peg-Out Liveness with parameter* u_ℓ*, assuming* $\Pi_{\mathcal{L}_A}$ *and* $\Pi_{\mathcal{L}_B}$ *are safe and live, the existence of at least one honest operator, and a secure atomic swap protocol.*

Proof. Assume, for contradiction, that an honest user posts a valid transaction $\texttt{tx} \in \text{Swap}_{\mathcal{L}_B}(\texttt{id})$ initiating the swap on $\mathcal{L}_B$ at round r, but no corresponding transaction $\texttt{tx}' \in \text{Swap}_{\mathcal{L}_A}(\texttt{id})$ paying the user appears in ${}^{r'}\mathcal{L}_A^{P_A}$ for an honest party P_A at any round $r' \geq r + u_\ell$.

We stress that the assumption of an existentially honest operator to facilitate this swap is identical to the assumption made in prior trust-minimized designs, such as [35]. The absence of $\texttt{tx}'$ implies one of the following cases:

- *Operator unavailability.* No operator fulfilled the swap request. This contradicts the assumption that an existentially honest, capitalized operator exists and actively participates.
- *Atomic swap failure.* An adversarial operator received the client's funds through a $\texttt{tx} \in \text{Swap}_{\mathcal{L}_B}(\texttt{id})$, but the conditions to post $\texttt{tx}' \in \text{Swap}_{\mathcal{L}_A}(\texttt{id})$ were not fulfilled. This violates the security of the atomic swap protocol.
- *Safety failure of* $\mathcal{L}_A$. tx was temporarily included in $\mathcal{L}_A$ but was subsequently dropped. This violates the self-consistency and thus safety of $\mathcal{L}_B$.

In all cases, we reach a contradiction. Thus, $\texttt{tx}' \in \text{Swap}_{\mathcal{L}_A}(\texttt{id})$ must eventually be included in $\mathcal{L}_A$, satisfying Peg-Out Liveness. □

Theorem 8 (Ethereum Operator Safety). *The BitVM3-bridge protocol Ethereum variant satisfies* Swap Operator Safety *with recovery parameter* u_Λ*, assuming* $\Pi_{\mathcal{L}_A}$ *and* $\Pi_{\mathcal{L}_B}$ *are safe and live, the Ethereum light client protocol is safe and live, the signer committee is honest during setup, at least one signer remains honest after setup, there exists an honest challenger, the underlying signature scheme is EUF-CMA secure, hash functions are modeled as random oracles, and the BitVM3-core protocol is sound (Theorem 2) and complete (Theorem 1).*

Proof. Assume, for contradiction, that an honest operator p posts a valid transaction $\texttt{tx} \in \text{Burn}(\texttt{id})$ on $\mathcal{L}_B$ at round r_1, but no corresponding transaction $\texttt{tx}' \in \text{Withdraw}(\texttt{id})$ paying the operator appears in $\mathcal{L}_A$ at any round $r_2 \geq r_1 + u_\Lambda$. This implies either the output of Deposit(id) was previously spent by another party, or the operator was unable to spend Withdraw(id) and have it persist on $\mathcal{L}_A$. This scenario can occur only under one of the following cases:

Case 1: The output of Deposit**(id) was spent by another party** $p' \neq p$**.**

- *Setup failure.* The signer committee incorrectly pre-signed a withdrawal transaction that does not consume the connector output of an $\texttt{tx}_{ast} \in \text{Assert}(\texttt{id})$. This allows a malicious party p' to spend Deposit(id) without a valid burn, violating the honest setup assumption of the signer committee.
- *Forged Authorization.* An adversary p' bypassed the protocol by directly authorizing a withdrawal. This requires either the entire signer committee becoming corrupted after setup to collude, or an adversary successfully forging the signature, which violates the existential honesty of a signer or the EUF-CMA security of the signature scheme respectively.
- *Safety failure of the Ethereum Light Client.* The Ethereum light client protocol verified by the BitVM3-core predicate accepted an invalid or forged state transition (e.g., due to a corrupted sync committee maliciously finalizing a fake state). This allows an attacker to generate a valid SNARK proof π for a non-existent burn, violating the assumed safety of the light client protocol.
- *BitVM3-core Soundness failure.* A malicious operator p' posted an invalid $\texttt{tx}_{ast} \in \text{Assert}(\texttt{id})$ that successfully bypassed the challenge window and spent the output of Deposit(id) before the honest operator p. Assuming an honest challenger exists, this implies either the disprove transaction was censored (violating the liveness of $\mathcal{L}_A$) or the underlying BitVM3-core predicate evaluated to True without a valid witness. The latter constitutes a direct violation of BitVM3-core soundness (Theorem 2).

Case 2: The honest operator's withdrawal transactions were incorrectly constructed at setup or fail to finalize on $\mathcal{L}_A$**.**

- *Operator Setup failure.* The signer committee failed to construct the transactions correctly during setup for this specific operator, e.g., they pre-signed an invalid transaction spending $\texttt{tx} \in \text{Assert}(\texttt{id})$ or did not behave honestly during the setup of BitVM3-core. This violates the assumption that signers behave honestly during setup.
- *Safety failure of* $\mathcal{L}_B$. The honest operator's transaction $\texttt{tx} \in \text{Burn}(\texttt{id})$ was included in $\mathcal{L}_B$, but due to a chain reorganization, it was dropped and did not persist. This prevents the valid generation of a finality certificate, violating the self-consistency and safety of $\mathcal{L}_B$.
- *Liveness failure of the Ethereum Light Client.* The Ethereum sync committee failed to produce valid finality certificates within the necessary timeframe. Consequently, the honest operator p cannot obtain the witness necessary to generate a valid SNARK proof π for their burn, violating the assumed liveness of the light client protocol.

- *Liveness failure of $\mathcal{L}_A$.* The honest operator successfully generated a valid SNARK proof π and submitted the necessary transactions $\mathtt{tx}_{ast} \in \mathsf{Assert}(\mathtt{id})$ or $\mathtt{tx}' \in \mathsf{Withdraw}(\mathtt{id})$ to Bitcoin, but they were not included by miners within the required liveness bound $u_{\mathcal{L}_A}$. This contradicts the assumed liveness property of $\mathcal{L}_A$.
- *BitVM3-core Completeness failure.* The operator posted a valid transaction $\mathtt{tx}_{ast} \in \mathsf{Assert}(\mathtt{id})$ on $\mathcal{L}_A$, but it did not remain unspent before the timelock t expired. This directly violates the completeness of BitVM3-core.

In all cases, we reach a contradiction. Thus, the operator is guaranteed to recover their funds via $\mathtt{tx}' \in \mathsf{Withdraw}(\mathtt{id})$, satisfying Operator Safety. □

E.3 Security Proofs for the Rollup Bridge

We instantiate the bridge definitions for our Bitcoin rollup bridge with $\mathcal{L}_A$ Bitcoin and $\mathcal{L}_B$ the rollup. The bridge relations are induced by the transaction sets introduced in Section 6.3:

$$R_1(\mathtt{id}) = \{(\mathsf{Deposit}(\mathtt{id}), \mathsf{Mint}(\mathtt{id}))\},$$
$$R_2(\mathtt{id}) = \{(\mathsf{PegOut}(\mathtt{id}), \mathsf{Burn}(\mathtt{id}))\}.$$

The peg-in bridge $\Lambda_{R_1}^{\mathrm{PegIn}}$ uses the same deposit-swap construction as the Ethereum bridge (Section 6.2); Theorem 4 and Theorem 5 therefore apply verbatim. We focus below on the peg-out and operator-safety properties, which differ because the BitVM3-bridge rollup variant i) employs non-interactive user exits with operator fronting the client with a PegOut(id) transaction and ii) uses the Bitcoin light client Figure 13 inside BitVM3-core.

E.3.1 Peg-Out Bridge Security.

Theorem 9 (Rollup Peg-Out Safety). *The Peg-Out bridge $\Lambda_{R_2}^{PegOut}$ satisfies safety with parameter u_s, assuming $\Pi_{\mathcal{L}_B}$ satisfies safety.*

Proof. Assume, for contradiction, that $\mathtt{tx}' \in \mathsf{PegOut}(\mathtt{id})$ is included in ${}^{r_2}\mathcal{L}_A^{P_A}$ for an honest party P_A, but no $\mathtt{tx} \in \mathsf{Burn}(\mathtt{id})$ appears in ${}^{r_1}\mathcal{L}_B^{P_B}$ for any honest party P_B at any round $r_1 \geq r_2 - u_s$. This implies the following:

- *Safety failure of $\mathcal{L}_B$.* A burn $\mathtt{tx} \in \mathsf{Burn}(\mathtt{id})$ was included in $\mathcal{L}_B$ at some round $r < r_1$ and triggered the operator's $\mathtt{tx}' \in \mathsf{PegOut}(\mathtt{id})$ on $\mathcal{L}_A$, but a chain reorganization on $\mathcal{L}_B$ subsequently dropped tx, violating the self-consistency and therefore the safety of $\mathcal{L}_B$.

In all cases, we reach a contradiction. Hence $\mathtt{tx} \in \mathsf{Burn}(\mathtt{id})$ must exist, satisfying Peg-Out safety. □

Theorem 10 (Rollup Peg-Out Liveness). *The Peg-Out bridge $\Lambda_{R_2}^{PegOut}$ satisfies liveness with parameter u_ℓ, assuming the signer committee is honest during setup and at least one signer remains honest after setup, $\Pi_{\mathcal{L}_A}$ is safe and live, and an honest operator monitors $\mathcal{L}_B$.*

Proof. Let $\mathtt{tx} \in \mathsf{Burn}(\mathtt{id})$ be included in ${}^{r_1}\mathcal{L}_B^{P_B}$ for an honest party P_B at round r_1. By the protocol, an honest operator p observing this event posts the fronting transaction $(\mathtt{tx}', p) \in \mathsf{PegOut}(\mathtt{id})$ as embedded in the Burn(id) transaction by the client to $\mathcal{L}_A$, which should appear in ${}^{r_1+u_\ell}\mathcal{L}_A^{P_A}$ for all honest P_A. Failure can occur only for:

- *Setup failure.* The signer committee failed to construct the protocol transactions correctly during setup, making it unsafe for an honest operator to front PegOut. This violates the honest-setup assumption of the signer committee.
- *Liveness failure of $\mathcal{L}_A$.* The honest operator submitted $(\mathtt{tx}', p)$ to $\mathcal{L}_A$ but it failed to be included within $u_{\mathcal{L}_A}$ rounds, contradicting the liveness of $\mathcal{L}_A$.
- *Safety failure of $\mathcal{L}_A$.* $(\mathtt{tx}', p)$ was included in $\mathcal{L}_A$ at some round $r' \leq r_1 + u_\ell$ but did not persist due to a chain reorganization, violating the self-consistency and therefore the safety of $\mathcal{L}_A$.

In all cases, we reach a contradiction. Thus, $\mathtt{tx}' \in \mathsf{PegOut}(\mathtt{id})$ must appear on $\mathcal{L}_A$, satisfying Peg-Out liveness. □

Theorem 11 (Rollup Operator Safety). *The BitVM3-bridge protocol satisfies Front Operator Safety with recovery parameter u_Λ, assuming $\Pi_{\mathcal{L}_A}$ and $\Pi_{\mathcal{L}_B}$ are safe and live, the Bitcoin light client is safe and live, the signer committee is honest during setup, at least one signer remains honest after setup, there exists an honest challenger, the underlying signature scheme is EUF-CMA secure, hash functions are modeled as random oracles, and the BitVM3-core protocol is both sound (Theorem 2) and complete (Theorem 1).*

Proof. Assume, for contradiction, that an honest operator p_i's fronting transaction $(\mathtt{tx}, p_i) \in \mathsf{PegOut}(\mathtt{id})$ appears in ${}^{r_1}\mathcal{L}_A^{P_A}$ for all honest parties P_A at round r_1, but no reimbursement $(\mathtt{tx}', p_i) \in \mathsf{Withdraw}(\mathtt{id})$ appears in $\mathcal{L}_A$ by round $r_1 + u_\Lambda$.

Since Withdraw(id) consumes the Deposit(id) output and the latter can be spent only once under the UTXO model, at most one transaction in Withdraw(id) can ever be confirmed. We distinguish two cases.

Case 1: A transaction $(\mathtt{tx}', p_j) \in \mathsf{Withdraw}(\mathtt{id})$ with $p_j \neq p_i$ is included in $\mathcal{L}_A$ at some round $r' \leq r_1 + u_\Lambda$. The reserve has been claimed by a different operator p_j without a corresponding $(\mathtt{tx}, p_j) \in \mathsf{PegOut}(\mathtt{id})$ ever appearing on-chain. This can occur only for:

- *Setup failure.* The signer committee incorrectly authorized some transaction tied to the BitVM3-core instance during setup, e.g., by pre-signing a $\mathtt{tx}'$ that lets a dishonest operator post Withdraw(id) without a valid $\mathtt{tx}_{ast} \in \mathsf{Assert}(\mathtt{id})$ as a second input. This violates either the honest-setup assumption or the existential-honesty assumption that at least one honest signer deletes their signing key after setup.
- *Light client safety failure.* Suppose p_j produced a pair (ϕ, w) satisfying the chainstate-proof relation $R := \{(\phi, w) : \Pi(\phi, w) = \mathsf{True}\}$, where ϕ contains the checkpoint hash $\mathcal{H}(B_i)$ used by the reimbursement claim, even though the alleged fronting transaction $(\mathtt{tx}, p_j) \in \mathsf{PegOut}(\mathtt{id})$ does not appear in the Bitcoin ledger view of any honest party at round r'. Since $\Pi(\phi, w) = \mathsf{True}$, the witness w certifies a Bitcoin history extending the light-client checkpoint and containing a block in which $(\mathtt{tx}, p_j)$ is included. Thus the light client has accepted, relative to $\mathcal{H}(B_i)$, a history that is inconsistent with the honest Bitcoin ledger view. This means that the block accepted by the light client is not admissible, contradicting light client safety

Bitcoin Rollup BitVM3-bridge Protocol Execution

Phase 1 : Peg − in (Setup & Deposit Swap)

User (Alice)

(1) Sample a high-entropy cancellation secret s_a and compute its hash $\mathcal{H}(s_a)$.
(2) Broadcast Request on Bitcoin locking u BTC. ▷ Paths: deposit (requires n-of-n committee signature) and cancel (requires s_a after a timelock).

Operator (Prover) & Signer Committee (Verifiers)

(1) Set up the Bitcoin light-client checkpoint instances for the intended lifetime of the bridge. For each checkpoint index i, prepare the corresponding $\mathsf{Checkpoint}_i$ transaction, checkpoint commitment keys, and dispute/timeout paths so that future reimbursement claims can be made relative to checkpoints accepted by the light client.
(2) Define predicate $Q(\pi) \triangleq \mathsf{SNARK.Vrfy}(\mathcal{R}_{\mathsf{rollup-bridge}}, \mathsf{crs}, \phi, \pi)$ for public inputs $\phi = (\mathcal{H}(B_i), \mathsf{txid}, \mathsf{sid})$, where $\mathcal{R}_{\mathsf{rollup-bridge}}$ validates, relative to a Bitcoin light-client checkpoint, the burn in the rollup state and the corresponding L1 PegOut fronting payment.
(3) Run BitVM3-core.**Setup**$(1^\kappa, Q) \to (\mathsf{sid}, S, \mathsf{aux})$ to generate the garbled payload and the corresponding $\mathsf{Assert}_{i,j}$ skeletons for reimbursement claims.
(4) Signers verify aux, including the consistency of the generated $\mathsf{Assert}_{i,j}$ skeletons with the checkpoint instances, and confirm Request is included in Bitcoin.
(5) Signers generate signatures for Deposit, spending Request to the bridge reserve.
(6) Signers generate signatures for the corresponding $\mathsf{Withdraw}_{i,j}$ transactions, each spending Deposit together with its matching $\mathsf{Assert}_{i,j}$ connector input.

User (Alice) & Rollup State Machine

(1) Alice registers $\mathcal{H}(s_a)$ and the Deposit committee signature on the Rollup.
(2) Any party broadcasts the pre-signed Deposit transaction on Bitcoin.
(3) The Rollup initiates a delay window $\Delta > u_{\mathcal{L}_A} + u_{\mathcal{L}_B}$. If no party relays s_a to the Rollup before Δ expires, it mints u wBTC to Alice.

Phase 2 : Transfer (L2 Execution)

Users & Operators

(1) Users trade and transfer wBTC freely on the Rollup. Operators monitor the L2 state for exit requests.

Phase 3 : Peg − out (Non − Interactive Exit & Reimbursement)

User (Alice)

(1) Post a Burn transaction on the Rollup destroying u wBTC and embedding a presigned PegOut transaction as metadata, delegating fulfillment of the peg-out to any active operator.

Operator (Prover)

(1) Broadcast PegOut on Bitcoin L1, fronting $u - f_O$ BTC to Alice's specified address.
(2) Select a valid Bitcoin light-client checkpoint $\mathsf{Checkpoint}_i$ and generate a valid SNARK proof π satisfying Q, demonstrating the Rollup state inclusion of Burn and the Bitcoin L1 inclusion of PegOut relative to the checkpoint hash $\mathcal{H}(B_i)$.

Operator (Prover) & Challenger (Verifier)

(1) Execute the BitVM3-core.**Execute**$(\mathsf{sid}, \pi, \mathsf{aux})$ protocol on Bitcoin, using the light-client checkpoint block hash as part of the public statement:
   - **Prover:** Broadcasts $\mathsf{Assert}_{i,j}$, consuming the corresponding checkpoint connector output and committing to the garbled proof labels.
   - **Challenger:** Evaluates the garbled circuit off-chain. If $Q(\pi) = 0$ or labels are invalid, it obtains the fraud witness label L^* and broadcasts Disprove before timeout t.
   - **Resolution:** If no Disprove occurs before timeout t expires, the success predicate S of BitVM3-core holds.

Operator (Prover)

(1) Broadcast the pre-signed Withdraw transaction after t rounds, consuming both the Deposit UTXO and the Assert connector output, reclaiming u BTC from the reserve.

Figure 16: The Bitcoin Rollup instantiation of BitVM3-bridge, featuring non-interactive user exits and operator fronting.

- *BitVM3-core soundness failure.* If neither of the above holds, then p_j does not know any pair $(\phi, w) \in R$, yet $(\mathtt{tx}', p_j) \in \mathsf{Withdraw}(\mathtt{id})$ is included in $\mathcal{L}_A$. This implies that an $\mathsf{Assert}(\mathtt{id})$ was included on $\mathcal{L}_A$ at $r'' \le r'$ whose connector output remained unspent at $r'' + t$, contradicting BitVM3-core soundness (Theorem 2).

Case 2: No transaction in $\mathsf{Withdraw}(\mathtt{id})$ appears on $\mathcal{L}_A$ for any round $r \le r_1 + u_\Lambda$. The honest operator p_i failed to claim the reserve. This can occur only for:

- *Setup failure.* The signer committee did not construct the transactions as specified during setup, e.g., they pre-signed an invalid transaction spending $\mathsf{Deposit}(\mathtt{id})$ or deviated from BitVM3-core's setup. This makes it infeasible for an honest operator to reclaim funds, violating the honest-setup assumption.

- *Light client liveness failure.* Although $(\mathtt{tx}, p_i) \in \mathsf{PegOut}(\mathtt{id})$ appears in $\mathcal{L}_A$ at round r_1, p_i does not yet know a witness accepted by the light-client relation for any checkpoint that can be used in the reimbursement claim. Once the Bitcoin chain has progressed far enough that the block containing $(\mathtt{tx}, p_i)$ is stable and lies behind an admissible checkpoint, an honest operator can form a witness w consisting of the corresponding Bitcoin header chain and inclusion proof for $(\mathtt{tx}, p_i)$. The public statement ϕ contains the checkpoint hash $\mathcal{H}(B_i)$ and the peg-out identifier $\mathtt{id}$, and the pair (ϕ, w) satisfies the chainstate-proof relation used by the rollup-bridge predicate. If no such checkpoint and witness ever become available, then the light client has failed to output an admissible block, contradicting light-client liveness.
- *Liveness failure of $\mathcal{L}_A$.* The honest operator generated a valid SNARK proof π and submitted $\mathtt{tx}_{ast} \in \mathsf{Assert}(\mathtt{id})$ and $\mathtt{tx}' \in \mathsf{Withdraw}(\mathtt{id})$, but they were not included within $u_{\mathcal{L}_A}$ rounds, contradicting the liveness of $\mathcal{L}_A$.
- *BitVM3-core completeness failure.* If none of the above holds, p_i holds a valid $(\phi, w) \in R$ and an honestly produced proof π, yet $\mathtt{tx}_{ast} \in \mathsf{Assert}(\mathtt{id})$ does not remain unspent until the timelock t expires. This directly contradicts BitVM3-core completeness (Theorem 1).

In both cases we reach a contradiction. Hence the honest operator recovers the funds via $\mathtt{tx}' \in \mathsf{Withdraw}(\mathtt{id})$ within $u_\Lambda = u_{\mathcal{L}_A} + t$ rounds, where $u_{\mathcal{L}_A}$ accounts for inclusion of Assert and t for completion of the BitVM3-core challenge timelock. □

F Open Science

A Rust implementation of BitVM3-bridge is available at https://github.com/idealgroup/bridge, archived at https://doi.org/10.6084/m9.figshare.33716479. It contains a standalone Lamport one-time signature library with Bitcoin Script verification, shared bridge types and spending-condition scripts, crates implementing transaction construction and signing for the committee, operator, depositor, and challenger roles, and end-to-end integration tests exercising the deposit, withdraw, cancel, and fraud-proof flows against a live Bitcoin regtest node. The implementation is generic over the garbled-circuit backend through a `BitVMEngine` trait, reflecting the separation of BitVM3-core from any specific garbling scheme. Our tests use a mock engine.

The on-chain transaction sizes of Section 8.1 are derived analytically from the transaction templates and can be verified against the script-construction logic in the repository. The off-chain figures of Section 8.2 and Table 9 are based on the BitVM Alliance's Yao-style Groth16 verifier circuit [9], an independently maintained open-source project whose repository provides instructions for reproducing the reported circuit sizes, garbling times, and evaluation times. The Argo implementation [18] is not yet open source. The variable-difficulty Bitcoin light client of Section 5 is specified formally in the paper but not included in the artifact. It is required only for the rollup variant of BitVM3-bridge.

G Ethical Considerations

This work studies trust-minimized Bitcoin bridges. Its intended benefit is to reduce the trust placed in custodians and federations when moving BTC to secondary systems: existing bridges typically rely on designated key holders, threshold committees, or federated custody, whose failure can freeze or steal user funds. BitVM3-bridge makes invalid withdrawals disputable by any challenger and lowers the collateral and on-chain costs of operating, which may broaden participation and reduce concentration in bridge infrastructure. The main stakeholders are Bitcoin users, operators, challengers, and infrastructure providers.

The protocol does not eliminate risk. Its security rests on the assumptions stated throughout the paper, in particular correct setup, one honest signer deleting its setup key, one honest challenger monitoring the protocol, and ledger safety and liveness. If these fail, users or operators may lose funds, as they may through implementation bugs, poor parameter choices, fee spikes, or failures in the connected secondary system. Cheaper bridging may also increase users' exposure to applications whose risk profiles differ substantially from holding BTC on Bitcoin. We regard user autonomy as important, but deployments should not present the bridge as risk-free and should make the challenge windows, withdrawal delays, and residual risks explicit. Bridge activity is observable on both chains, so users should further be warned that third parties may correlate it across systems.

This work involves no human subjects and collects no personal data. It presents a protocol and research artifacts rather than an end-user product. Any production deployment should undergo independent security audits, careful parameter selection, and staged rollout with clear incident-response procedures.