USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Bridging Bitcoin to Second Layers via BitVM2

Robin Linus Woll, *Stanford University and ZeroSync Association;* Lukas Aumayr, *University of Edinburgh and Common Prefix;* Zeta Avarikioti, *TU Wien and Common Prefix;* Matteo Maffei, *TU Wien;* Andrea Pelosi, *University of Pisa, University of Camerino, and TU Wien;* Orfeas Stefanos Thyfronitis Litos, *Imperial College London and Common Prefix;* Christos Stefo, *TU Wien;* David Tse, *Stanford University and Byzantine Research;* Alexei Zamyatin, *BOB*

https://www.usenix.org/conference/usenixsecurity26/presentation/woll

This paper is included in the Proceedings of the 35th USENIX Security Symposium.

August 12–14, 2026 • Baltimore, MD, USA

ISBN 978-1-939133-58-8

Open access to the Proceedings of the 35th USENIX Security Symposium is sponsored by

جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology

ARTIFACT EVALUATED
USENIX
AVAILABLE

Bridging Bitcoin to Second Layers via BITVM2

Robin Linus Woll
Stanford University
ZeroSync

Lukas Aumayr
University of Edinburgh
Common Prefix

Zeta Avarikioti
TU Wien
Common Prefix

Matteo Maffei
TU Wien

Andrea Pelosi
University of Pisa, University of Camerino, TU Wien

Orfeas Stefanos Thyfronitis Litos
Imperial College London, Common Prefix

Christos Stefo
TU Wien

David Tse
Stanford University, Byzantine Research

Alexei Zamyatin
BOB

Abstract

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

1 Introduction

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

Limitations of the base blockchain (L1) can often be mitigated by adding second layers (L2s). For example, Bitcoin scalability has been addressed by L2 protocols such as payment channels [27]. Payment channels enable the Bitcoin asset to be transacted off the Bitcoin chain and settled back on Bitcoin in a completely trustless manner. However, payment channels serve to scale only one application: payments, Bitcoin's original application. To make Bitcoin support a broader range of applications (e.g., DeFi) Bitcoin needs L2s with general-purpose capabilities. To that end, the holy grail is a *bridge* that allows Bitcoin assets to be trustlessly transferred back and forth between such an L2 and Bitcoin, where a wrapped version of the Bitcoin asset is used on the L2.

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

Figure 1: Honest requirements of the BITVM2-BRIDGE as compared to a multi-signature bridge, e.g., [8, 24]. Peg-in refers to the transfer of bitcoins from the main chain to the L2. Peg-out refers to the transfer back to the main chain. Precise definitions of security properties are found in the full version [22]. Security analysis is given in Section 6.

Trustless bridges between other blockchains are typically realized through *light clients* on the destination chain, which check if a certain transaction happened in the source chain before releasing tokens in the destination chain. The Inter-Blockchain Communication protocol (IBC) in Cosmos [18] is a notable example. Such light clients are typically implemented as smart contracts on the destination chains. Since Bitcoin lacks smart contracts, existing bridges typically use multi- or threshold signature schemes, where a group of t-of-n signers is entrusted with safekeeping BTC. If these signers collude or are bribed, user deposits are lost.

In this paper, we present the first light-client-based Bitcoin bridge, the BITVM2-BRIDGE. The BITVM2-BRIDGE is run by n permissioned signers, m permissioned operators, and an arbitrary number of *permissionless* challengers that watch over the operators. As shown in Fig. 1, BITVM2-BRIDGE achieves stronger security guarantees than standard multi-signature bridges. In particular, it reduces the peg-out safety and liveness assumptions from honest majorities to *existential honesty*: the presence of just one honest party per role (signer, operator, challenger) suffices.

The signers act as an emulation of a *covenant*, a set of pre-

signed transactions forming a contract on how and to whom the locked pegged-in funds can be released in a peg-out. Unlike in a multi-signature bridge, the signers only need to be present at peg-in, and once the transactions are presigned, they can delete their keys. The peg-out process is achieved in a trust-minimizing way through the light client. A user first burns the wrapped asset on the L2. Then one of the operators fronts the bitcoins to that user, and using a proof of burn, the operator claims the locked funds that were pegged in. The on-chain light client is used to verify the correctness of the proof and ensures that an honest operator can always claim the funds. Precisely because of the light client, no action by the signers is needed at peg-out, and as long as at least one of the signers deleted its key after peg-in, the covenant is sealed, and the peg-out process is trustless. This is the strong distinction between the BITVM2-BRIDGE and a standard multi-signature bridge.

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

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

Related work. A number of works approach the problem of bridging assets between different chains with minimal trust. Similarly to our work, zkBridge [32] leverages zkSNARKs to prove to one chain the state of another chain, thus achieving trustless bridging. Unfortunately, verifying zkSNARKs directly on Bitcoin is impossible due to its maximum block size, so zkBridge is incompatible with Bitcoin.

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

BitVMX [20] achieves a lower cost in terms of transactions compared to BitVM in exchange for requiring a permissioned set of operators. TOOP [12] extends BitVMX by moving funds to an address controlled by all active operators at the time of peg-out and having them send their private keys to the peg-out party. Unfortunately, its practicality is questionable as it needs off-chain communication exponential in the number of operators in the worst case.

ColliderVM [16] focuses on providing stateful computation on Bitcoin. It improves on BitVM by (i) permitting permissionless operators and by (ii) replacing the bisection game with executing stateful computation directly on-chain. Its main limitations are (i) the high energy requirements needed to find hash collisions and (ii) its need to trust a permissioned set of parties both for liveness and for safety as in BitVM.

The BITVM2-BRIDGE relies on a group of signers to emulate *covenants*: constraints on when UTXOs are spendable that are inherited by descendant UTXOs. True covenants can remove our reliance on existentially honest signers. Unfortunately, covenants are not available in today's Bitcoin Script. As exemplified by [9], the few additional opcodes available in the Bitcoin fork BSV are enough to express covenants. Furthermore, PIPE [17] describes a way to simulate covenants on Bitcoin without a consensus change.

Real-world impact. BITVM2-BRIDGE has already influenced the industry: several startups, including BOB, Babylon, Citrea, Alpen, Bitlayer, and Fiamma, have emerged to build Bitcoin bridges and L2s based on this work. Each has introduced practical improvements and trade-offs, some of which are detailed in Section 8.

2 Model and Protocol Overview

We next describe the setting, assumptions, and actors in our construction, and provide a high-level overview of the BITVM2-BRIDGE. Our model includes two ledgers, the parties interacting with them, and the cryptographic and system assumptions we rely on.

Our main application is a trust-minimizing Bitcoin bridge. Along the way we develop two components of independent interest: (1) BITVM2-CORE, a protocol for verifying arbitrary computations on Bitcoin, and (2) a Bitcoin light client built on BITVM2-CORE. These serve as building blocks for the bridge and are analyzed formally in [22].

2.1 Ledger and Network Model

We consider two ledgers: Bitcoin ($\mathcal{L}_A$) and a secondary ledger ($\mathcal{L}_B$). Both ledgers are assumed to satisfy the usual safety and liveness properties [13], i.e., honest parties' views of each ledger are consistent (no two honest nodes confirm conflicting histories), and any transaction broadcast by an honest party will be confirmed within a bounded time.

We assume parties know each other's identities and communicate via authenticated channels. We also assume messages are delivered within a bounded message delay Δ, i.e., we adopt a synchronous round-based model, as in the Bitcoin backbone framework [13]. Formal definitions can be found in [22].

2.2 UTXO Model Overview

In Bitcoin, coins are modeled in the standard *unspent transaction output* (UTXO) paradigm: coins reside in transaction outputs, each carrying a value and a *locking script*. An output can be spent by providing a *witness* that satisfies its script. Transactions consume existing outputs as inputs and create new outputs, preserving value.

We rely on standard spending conditions used in Bitcoin, such as signature locks (CheckSig), multisignature locks (CheckMultiSig), time-based conditions (timelocks), and Taproot trees that enable complex, private policies (denoted $\langle \cdots \rangle$), which we detail in Appendix B.2.

We additionally use spending conditions based on Lamport signatures, a one-time signature scheme verifiable on Bitcoin (cf. Appendix B.1). We leverage the scheme's one-time security property to construct a commitment mechanism for messages (such as the intermediate states of a computation). In this construction, given a message m, the public key pk_m serves as its binding commitment instance, constraining the signer to a single future choice. The corresponding signature c_m represents the value selection: by revealing c_m, the user irrevocably selects m and satisfies the commitment. Consequently, subsequent locking scripts can enforce this selection via the $\mathsf{CheckLampComm}_{pk_m}(m, c_m)$ spending condition.

Furthermore, we employ the *connector outputs* technique to enforce mutual exclusivity within a set of transactions. This mechanism leverages Bitcoin's SIGHASH flags primitive to bind multiple candidate transactions to a single, shared unspent transaction output. Once one transaction consumes this connector output, the remaining candidates become invalid double-spends. We provide a more complete description of SIGHASH flags and connector outputs in [22].

Example. We illustrate the notation we use in the rest of this work with the following example: Let tx be a transaction that consumes outputs from two previous transactions, tx1 and an arbitrary transaction $*$, and creates two new outputs:

	tx
Inputs	(0) $(\mathsf{tx1}, 0, \mathsf{CheckSig}_{\mathsf{pk}_A} \wedge \mathsf{RelTimelock}(10))$
	(1) $(*)$
Outputs	(0) $(4\ \text{BTC},\ \mathsf{AbsTimelock}(500))$
	(1) $(2\ \text{BTC},\ \mathsf{CheckSig}_{\mathsf{pk}_B})$
Witness	(0) σ_A
	(1) $*$

Here, the transaction tx spends (i) the first output of tx1 which requires a signature from user A and is locked under a relative timelock of 10 blocks, and (ii) an additional output denoted by $*$. We use $*$ to denote valid inputs and witnesses that carry coins needed for the protocol but have logic irrelevant to the protocol. The witness for the first input includes σ_A, authorizing the spend. The transaction creates two outputs: one holding 4 BTC, locked until block height 500, and another holding 2 BTC, spendable by user B.

2.3 Bitcoin Covenant Emulation

Although Bitcoin Script lacks native support for covenants, they can be emulated via pre-defined transaction trees. Consider the tree of all possible transaction sequences originating from the outputs of a root transaction Tx. A *Covenant Emulation System* (CES) for Tx is defined as a specific subtree rooted at Tx. By limiting valid execution paths to this subtree, a CES constrains how funds are spent not only for Tx, but recursively for all its descendants. We refer to the Bitcoin spending condition that enforces the rules of a CES as CheckCovenant.

In this work, we emulate covenants using a signer committee. In the "Signer Committee" Subsection of Section 3 we give an informal description of our covenant emulation protocol. We formalize the notion of CES and prove that our instantiation realizes it in Appendix A. Finally, we formally describe our covenant emulation protocol in [22].

2.4 System and Threat Model

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

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

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

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

Signer committee ($S_1, \ldots, S_n$) : A set of n signers is responsible for setting up the bridge instance by signing necessary transactions and generating ephemeral keys used to authorize minting and define valid burn conditions for wBTC. For liveness, we assume that the committee is honest during setup for each bridge instance. For safety against dynamic adversaries, we assume that after setup at least *one signer behaves honestly and deletes the key*.

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

We assume that at least one honest operator has sufficient liquidity to front the user. We argue that this assumption is reasonable: since the required amount is known at the time of the peg-in, operators can anticipate their future liquidity needs and, if necessary, borrow funds to meet them. Moreover, operators with sufficient funds can reuse the same liquidity to serve multiple users sequentially, as they regain the funds each time.

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

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

Cryptographic assumptions. All participants are computationally bounded. We rely on standard primitives: collision-resistant hashes, EUF-CMA secure signatures (e.g., Schnorr, Lamport), and succinct non-interactive arguments of knowledge (SNARKs) with perfect completeness and knowledge soundness. Full definitions are given in [22].

2.5 BitVM2-bridge Protocol Overview

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

2.5.1 Strawman: a bridge with native covenants

To build intuition, consider a natural bridge design in a hypothetical version of Bitcoin with native covenant support. Alice wishes to bridge BTC to a sidesystem and locks BTC into a Bitcoin UTXO whose spending conditions fully encode the bridge logic. Alice transfers the corresponding asset on the sidesystem to Bob, who later wishes to peg out by burning the sidesystem coins. Bob can unlock the original Bitcoin UTXO by publishing a transaction, which we call Payout, that includes a proof that the sidesystem coins were burned. A covenant script ensures that the recipient of Payout is the same public key specified in the burn transaction.

This design cannot be implemented in Bitcoin today for two reasons: (a) Bitcoin does not support native covenants and therefore cannot restrict how the Payout transaction spends the locked coins. (b) Bitcoin Script cannot verify a proof of burn from an external sidesystem. To address the first limitation, we use *covenant emulation* via a signer committee and *a set of permissioned operators* that front funds to users. To address the second, we rely on an *on-chain dispute mechanism* that allows any party to force verification when a withdrawal is published. We describe this Bitcoin-compatible design next.

2.5.2 High-level BitVM2-bridge flow

Peg-in. BitVM2-bridge begins when a user, Alice, initiates a peg-in by creating (but not yet posting on-chain) a transaction, PegIn, that locks BTC on Bitcoin (ledger $\mathcal{L}_A$). Alice sends the peg-in request to the signer committee, which in response, generates ephemeral keys and collaborates with a set of designated operators to prepare *presigned* Bitcoin transactions. These transactions are designed to enforce correct behavior at peg-out. Looking ahead, the presigned transactions are Assert, Payout, PayoutOptimistic, Claim, Challenge (Section 3), and PegOut (Section 5). Once the setup completes, the signers send transactions to Alice. After Alice verifies that she has received all the necessary presigned transactions, ensuring that the PegOut procedure can be executed successfully when needed, she posts PegIn on $\mathcal{L}_A$. Subsequently, she is credited the equivalent amount of wrapped BTC (denoted wBTC) on an external ledger ($\mathcal{L}_B$). This step is done trustlessly with the use of a light client [7] or atomic swap protocol [14, 29].

Transfer and payment. Once minted, the wrapped BTC can be transferred across users on $\mathcal{L}_B$ as regular coins. These coins may change hands arbitrarily,[1] and any user holding wBTC can later initiate a peg-out back to Bitcoin.

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

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

Operator reimbursement and dispute resolution. The operator must now reclaim the BTC they provided to Bob. This is where BitVM2-core comes into play: The operator publishes the presigned Claim transaction on $\mathcal{L}_A$, which triggers a dispute window. If no one challenges the Claim during the dispute window, the operator thereafter claims the PegIn coins by publishing the presigned PayoutOptimistic transaction. However, if the Claim is fraudulent, i.e., if Bob never burned the wrapped BTC or the operator failed to pay him, an honest challenger will submit a dispute (presigned Challenge transaction) to prevent the reimbursement.

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

A valid Disprove will prevent the *malicious operators* from taking the PegIn coins. Since the PegIn output remains unspent, honest operators can later claim the PegIn coins as reimbursement after successfully fronting coins to a user.

If no successful Dispute is published on-chain before the deadline, the PegIn coins are awarded to the *(honest) operator* by publishing the presigned Payout transaction. Malicious challengers cannot construct a valid Disprove, and thereby the operator will reclaim the PegIn coins.

A high-level illustration of the BitVM2-core protocol architecture and challenge flow is shown in Fig. 2.

2.5.3 Intuition on BitVM2-core and light client

Implementing Assert in Bitcoin Script. At the heart of our construction lies an interactive protocol, termed BitVM2-core, which is realized using Bitcoin Script. BitVM2-core enables honest operators to reclaim their funds (via Assert), while preventing malicious operators from withdrawing funds from the bridge (via Disprove). *But how can* Assert *express such a complex statement on Bitcoin Script?*

The key idea is to use Assert to commit to the output of a SNARK verifier attesting that the burn and payment indeed occurred. The core technical challenge lies in expressing the SNARK verifier logic on Bitcoin Script, which is inherently constrained. To overcome this, we observe that Bitcoin Script can express arbitrary total functions with statically bounded control flow,[2] that is, deterministic computations that are guaranteed to terminate and whose execution bounds are fixed at script construction time.

Based on this observation, we implement the verifier logic as a SNARK verifier circuit and compile it to Bitcoin Script. Since a full SNARK verifier typically exceeds Bitcoin's block size,[3] we decompose it into sequential chunks, each small enough to fit in a Bitcoin transaction (max 4MB). The Assert transaction commits to each chunk's code and output. During a successful dispute, a Disprove transaction reveals the true output of a specific chunk by re-executing the chunk on-chain and comparing it against the operator's claim. This enables a full on-chain fraud proof mechanism without any changes to Bitcoin. To ensure that Assert encodes the desired program, it is presigned by the signer committee for each operator during setup. Additionally, we assume the program is public knowledge, e.g., it may be available in a public repository on the Internet. This ensures its availability to challengers and any other participants who can independently verify it. The precise design of Assert is iteratively refined in strawman designs S1–S5 and finalized in the Optimistic BitVM2-core description in Section 3.

Verifying transaction inclusion. Currently, BitVM2-core must verify two key events: the inclusion of the peg-out transaction on Bitcoin (where the operator pays Bob), and the inclusion of Bob's burn transaction on $\mathcal{L}_B$. We assume that $\mathcal{L}_A$ (Bitcoin) includes state commitments of $\mathcal{L}_B$ and discuss alternative designs in Section 8. This reduces the task of verifying transaction inclusion on $\mathcal{L}_B$ to verifying a Bitcoin transaction against a known state commitment; thereby enabling both verifications to be handled uniformly within Bitcoin. To realize this, we present in Section 4, to our knowledge, the first *on-chain light client for Bitcoin*, built using BitVM2-core.

2.6 Protocol Goals

In the rest of this section we introduce our primitive in stages: we begin by exposing informally the BitVM2-core protocol goals, then outline the requirements for the light client protocol, and finally define the desiderata for the full BitVM2-bridge system. We defer formal definitions to [22].

BitVM2-core. BitVM2-core is a protocol run between a prover P ("operator" in BitVM2-bridge) and a set of verifiers V ("challengers" in BitVM2-bridge), who take part in a distributed ledger protocol $\Pi_{\mathcal{L}}$ as full nodes.

BitVM2-core enables P to prove on-chain the output of a known function f. The prover is entitled to publish a specific transaction conditioned on successful verification of the computation's result. For example, P may lock a deposit to qualify as a service provider for the computation of f in

[2] This can be done via opcodes such as OP_ADD and OP_IF.
[3] For example, 900MB for Groth16.

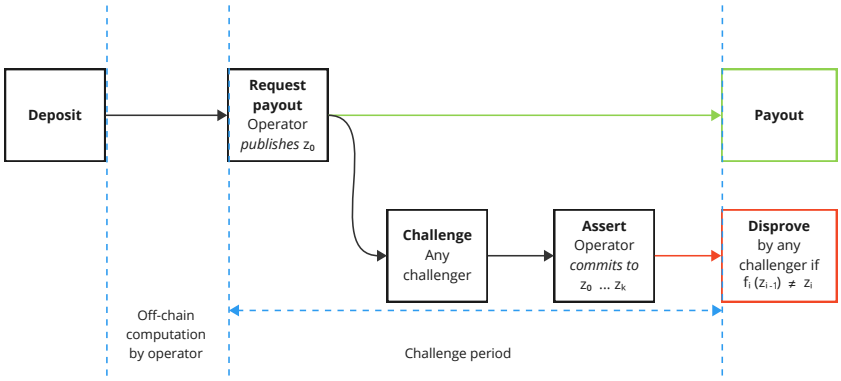

Figure 2: Simplified overview of the BITVM2-CORE transaction flow. The SNARK verifier program $f(x) = y$ is split into $f_1, f_2, ..., f_k$ with intermediary results $z_0, ..., z_k$. A challenged operator must reveal the intermediary states in the Assert transaction. This allows a challenger to disprove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

exchange for a service fee. In such a case, P can only reclaim their funds upon V verifying the result of the computation f.

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

Light client. The security definition of the light client protocol is based on BITVM2-CORE. Intuitively, the light client is secure if upon termination, it outputs a block that: (i) is guaranteed to appear in the future chains of all honest parties (**safety**), and (ii) is recent enough, i.e., reflects the honest network's progress (**liveness**). We term this block as **admissible**.

BITVM2-BRIDGE. At a high level, a *bridge* connects two ledgers, say $\mathcal{L}_A$ and $\mathcal{L}_B$, and ensures that actions on one ledger (the *source*) trigger corresponding actions on the other (the *destination*). For example, locking coins on $\mathcal{L}_A$ should lead to minting coins on $\mathcal{L}_B$. To keep track of such cross-chain transfers, we associate each with a unique identifier `id`, which links all related transactions across both ledgers.

A secure bridge must satisfy two fundamental properties:

- **Bridge liveness.** If a valid transaction (e.g., deposit or other action) appears on the source ledger $\mathcal{L}_A$, then within bounded time the corresponding transaction (identified by `id`) will appear on the destination ledger $\mathcal{L}_B$.
- **Bridge safety.** If a transaction appears on the destination ledger $\mathcal{L}_B$, then the corresponding transaction (identified by `id`) must appear on the source ledger $\mathcal{L}_A$ (possibly observed slightly earlier or within a bounded delay).

Bridge liveness ensures that honest users who lock coins on $\mathcal{L}_A$ can be confident that they will eventually receive the corresponding coins on $\mathcal{L}_B$. Bridge safety on the other hand prevents "printing coins out of thin air", ensuring that the total supply across the two systems is preserved. If both properties hold, we call the bridge *robust*.

Finally, we distinguish between a *unidirectional bridge*, where assets move only one way (e.g., from $\mathcal{L}_A$ to $\mathcal{L}_B$), and a *wrapped-asset bridge*, which combines two unidirectional bridges to enable bidirectional transfers. In the latter, a *peg-in* procedure locks coins on $\mathcal{L}_A$ and mints wrapped coins on $\mathcal{L}_B$, while a *peg-out* procedure burns wrapped coins on $\mathcal{L}_B$ and unlocks the original coins on $\mathcal{L}_A$. Bidirectional robustness ensures that assets can safely circulate between ledgers.

Bridge operator safety. In our design, peg-outs are executed by operators who advance their own liquidity to clients. Hence, peg-out liveness in practice requires that honest operators can always recover their funds, even against Byzantine adversaries. We refer to this guarantee as *Bridge Operator Safety*. It refines peg-out liveness by ensuring honest operators face no risk of loss and thus remain incentivized to service exits; an essential condition for deployability in practice.

3 BITVM2-CORE: General Function Verification on Bitcoin

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

[4] https://tinyurl.com/mt7ex7xh

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

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

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

(S1) Direct function verification

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

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

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

(S2) Function verification in a SNARK

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

(S3) Splitting the computation

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

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

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

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

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

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

(S4) Signer committee

To enforce that f is fixed at setup, we employ a signer committee of n signers $S_1, \ldots, S_n$. The signers are assumed to be honest during setup to ensure it completes, but they cannot steal the operator's funds.

During setup, the operator O contacts the signer committee to authorize the validity of the Assert transaction, thereby

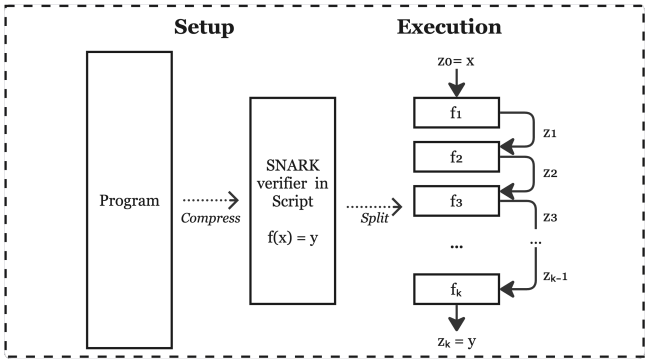

Figure 3: A SNARK verifier, representing an arbitrary program, is compiled to Bitcoin Script and split into subprograms $f_1, \ldots, f_k$, each small enough to run in a block.

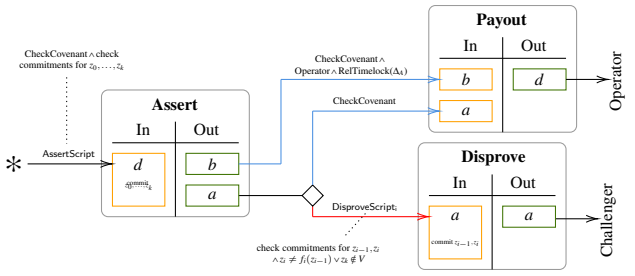

Figure 4: Illustration of the transactions Assert, Disprove, and Payout. The input of Assert can be any UTXO (∗), but its spending script must be a Covenant including commitments to $z_0, \ldots, z_k$. For readability, we use: gray rounded rectangles for transactions, green for BTC in outputs, and orange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

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

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

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

(S5) Lamport signatures

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

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

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

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

Assert **Transaction**	
Inputs	(0) $(*, *, \mathsf{AssertScript} := \mathsf{CheckCovenant} \wedge \mathsf{CheckLampComm}_{\mathsf{pk}_{z_0}} \wedge \cdots \wedge \mathsf{CheckLampComm}_{\mathsf{pk}_{z_k}})$
Outputs	(0) $(b$ BTC$, \langle \mathsf{RelTimelock}(\Delta_A) \wedge \mathsf{CheckCovenant} \wedge \mathsf{CheckSig}_{\mathsf{pk}_O} \rangle)$ (1) $(a$ BTC$, \langle \mathsf{CheckCovenant}, \mathsf{DisproveScript}_1, \ldots, \mathsf{DisproveScript}_k, \rangle)$
Witness	(0) $*, \mathsf{Covenant}, z_0, \ldots, z_k, c_{z_0}, \ldots, c_{z_k}$

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

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

Payout **Transaction**	
Inputs	(0) $(\mathsf{Assert}, 0, \langle \mathsf{RelTimelock}(\Delta_A) \wedge \mathsf{CheckCovenant} \wedge \mathsf{CheckSig}_{\mathsf{pk}_O} \rangle)$ (1) $(\mathsf{Assert}, 1, \langle \mathsf{CheckCovenant} \rangle)$
Outputs	(0) $(d\ \mathrm{BTC}, \mathsf{CheckSig}_{\mathsf{pk}_O})$
Witness	(0) $\mathsf{Covenant}, \sigma_O$ (1) $\mathsf{Covenant}$

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

1: **function** $\mathsf{DisproveScript}_i(\sigma_C, z_i, z_{i-1}, c_{z_i}, c_{z_{i-1}})$
2: $\quad \mathsf{CheckCovenant}()$
3: $\quad \mathsf{CheckLampComm}_{pk_{z_{i-1}}}(z_{i-1}, c_{z_{i-1}})$ *Both $pk_{z_{i-1}}$ and pk_{z_i} are hard-coded in the script*
4: $\quad \mathsf{CheckLampComm}_{pk_{z_i}}(z_i, c_{z_i})$
5: $\quad$ **return** $z_i \neq f_i(z_{i-1}) \vee (i = k \wedge z_k \notin V)$ *f_i, V and k are hard-coded in the script*
6: **end function**

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

Disprove **Transaction**	
Inputs	(0) $(\mathsf{Assert}, 1, \langle \mathsf{DisproveScript}_i \rangle)$
Outputs	(0) $(a\ \mathrm{BTC}, \mathsf{True})$
Witness	(0) $\mathsf{Covenant}, z_{i-1}, z_i, c_{z_{i-1}}, c_{z_i}$

The output determines the fate of the operator's deposit d BTC: a portion b BTC is burned (i.e., cannot be reclaimed), and the remainder $a = d - b$ may be claimed by the challenger. We partially burn the operator's deposit d BTC to ensure that any unsuccessful claim incurs a real penalty. Without burning part of the deposit, a malicious operator could submit PayoutOptimistic claims for free: if challenged, they could simply disprove their own false claim and recover both the challenge fees and their full deposit[5]. For simplicity, we denote this in the above definition as $(a\ \mathrm{BTC}, \mathsf{True})$. We provide the script logic in Algorithm 1.

[5]This mechanism is enforced using the `SIGHASH_SINGLE` flag [30], which commits to specific input-output pairs, while allowing more outputs to be added without a signature (e.g., bounty payouts, cover transaction fees).

Optimistic BitVM2-core

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

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

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

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

Claim **Transaction**	
Inputs	(0) $(*)$
Outputs	(0) $(d\ \mathrm{BTC}, \langle \mathsf{RelTimelock}(\Delta_B) \wedge \mathsf{CheckCovenant}, \mathsf{AssertScript} \rangle)$ (1) $(0\ \mathrm{BTC}, \mathsf{CheckSig}_{\mathsf{pk}_O})$
Witness	(0) $\sigma_O \wedge *$ (1) z

During the challenge period Δ_B, any party may dispute Claim by posting a presigned Challenge transaction, forcing the operator to respond with a corresponding Assert transaction, which can then be contested via a Disprove. The second output of Claim serves as a *connector output* (for the definition of connector outputs, see see [22]): it must remain unspent for the operator to use PayoutOptimistic and reclaim their deposit unchallenged.

Challenge **Transaction**	
Inputs	(0) $(\mathsf{Claim}, 1, \mathsf{CheckSig}_{\mathsf{pk}_O})$ (1) $(*)$
Outputs	(0) $(c\ \mathrm{BTC}, \mathsf{CheckSig}_{\mathsf{pk}_O})$
Witness	(0) σ_O (1) $*$

Phase 4: resolution. If unchallenged after Δ_B, the operator

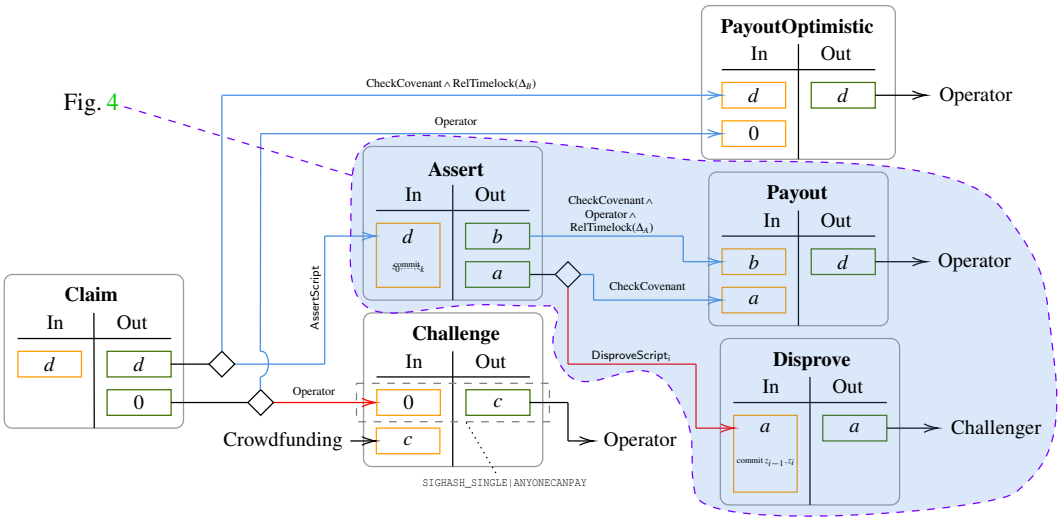

Figure 5: Illustration of the optimistic function verifier of BITVM2-CORE.

posts PayoutOptimistic to claim the funds and disable the dispute logic.

PayoutOptimistic **Transaction**	
Inputs	(0) $(\mathsf{Claim}, 0, \langle \mathsf{RelTimelock}(\Delta_B) \wedge \mathsf{CheckCovenant} \rangle)$
	(1) $(\mathsf{Claim}, 1, \mathsf{CheckSig}_{\mathsf{pk}_O})$
Outputs	(0) $(d\ \mathrm{BTC}, \mathsf{CheckSig}_{\mathsf{pk}_O})$
Witness	(0) Covenant
	(1) σ_O

We prove BITVM2-CORE satisfies both completeness and soundness (Definitions D.4 and D.5 in [22]) assuming existential honesty of signers, operators, and challengers in [22].

Reward and collateral. To deter a malicious challenger from griefing an honest operator by repeatedly triggering costly Assert transactions, the protocol requires the challenger to post collateral c BTC that covers the operator's on-chain costs. This collateral can be crowdfunded, allowing multiple users to jointly contribute inputs to the Challenge transaction. See Section 8 for details.

4 On-chain Bitcoin Light Client

We now build an on-chain Bitcoin light client for static difficulty using BITVM2-CORE. The goal is to let Bitcoin verify that a given block belongs to the canonical proof-of-work chain (i.e., stable chain adopted by honest nodes) and is sufficiently recent. Note that this is a light client of Bitcoin on the Bitcoin chain, i.e., it verifies to Bitcoin the presence of a block in its consensus. As such, it provides an *introspection capability* to Bitcoin, something that it does not have natively.

To this end, we instantiate BITVM2-CORE's AssertScript and DisproveScripts with a statement that checks whether a candidate block is part of a valid PoW chain from genesis to the declared tip (cf. Algorithm 2). At a high level, the operator commits to the chain tip and provides a succinct proof attesting to its validity, which is embedded in the Assert transaction. More concretely, the operator signs a sequence of intermediate states $z_0, \ldots, z_k$ with Lamport keys. The first state z_0 encodes the chain tip, its height, and a SNARK proof that this tip extends a valid proof-of-work chain from genesis. From this commitment, the admissible block is set to the block $m+k$ positions before the tip, ensuring that it is both recent and stable (i.e., the light client protocols is live and safe, respectively).

Assert **Transaction**	
Inputs	(0) $(\mathsf{Claim}, 0, \mathsf{AssertScript})$
Outputs	(0) $(b\ \mathrm{BTC}, \mathsf{CheckCovenant})$
	(1) $(a\ \mathrm{BTC}, \langle \mathsf{CheckCovenant} \wedge \mathsf{RelTimelock}(\Delta),$ $\mathsf{DisproveScript}_1, \ldots, \mathsf{DisproveScript}_k,$ $\mathsf{CheckLampComm}_{\mathsf{pk}_h} \wedge \mathsf{AbsTimelock}(h+3\Delta) \rangle)]$
Witness	(0) $*, \mathsf{Covenant}, z_0, \ldots, z_k, c_{z_0}, \ldots, c_{z_k}$

The timelock $\mathsf{AbsTimelock}(h+3\Delta)$ ensures freshness: if an operator tries to commit to an outdated block, challengers can disprove it before payout. The only change from the standard protocol is this added spending condition (see Fig. 6). Setting the timelock to 3Δ (Δ being the block confirmation window) gives an honest operator sufficient time to post Assert, wait for confirmation, and publish Payout.

Intuitively, this construction guarantees that the admissible

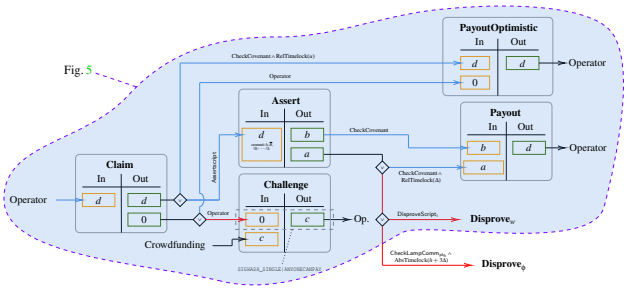

Figure 6: Illustration of the on-chain Bitcoin light client. Its goal is to verify to Bitcoin that a block B is in the ledger of Bitcoin. It does so by verifying that 1) the B is in a valid PoW chain $\mathcal{C}$, 2) the chain $\mathcal{C}$ is a prefix of the stable chain adopted by honest Bitcoin miners. The former is achieved using BitVM2-core (Fig. 5) and the latter is achieved by verifying that the height h of $\mathcal{C}$ is close to the length of the longest Bitcoin chain. This latter verification is piggy-backed onto BitVM2-core by adding a Lamport signed height h in Assert, and adding a Disprove path to allow challenge if the height is not close to the height of the longest chain.

block is both *live*, since it is chosen close to the tip of the chain (within m+k blocks), and *safe*, since an adversary cannot extend a competing chain fast enough to replace it. We formalize this in Theorem 6.3 and prove it in [22].

Algorithm 2 The chainstate proof Π_{lc} is the statement checked by the SNARK. It consists of (ϕ, w), where $\phi = (\overline{B}, h)$ is a block and its height, and w is a chain of blocks of length $|w| > m+k$. The algorithm verifies that w is a valid chain (each block B has a correct hashpointer $B.parent$, and a hash $\mathcal{H}(B)$ below the PoW-target T) that connects the (hard-coded) genesis block B_0 with $\overline{B}$ at height h.

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

5 The BitVM2-bridge Design

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

Bridges typically require trust assumptions, with current Bitcoin bridges relying heavily on multisignatures and honest majority assumptions [33]. While ideally, peg-in/out transactions could rely on native blockchain light clients, Bitcoin's limited scripting capability and lack of covenants restrict direct implementation: since the receiver (Bob) of the peg-out is not known at the time of the peg-in, it is impossible to enforce that only Bob can unlock the v BTC.

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

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

Algorithm 3 The chainstate proof Π, which defines the statement over which we define our SNARKs. It uses Algorithm 2 with input (ϕ, w), to verify that w is a valid PoW chain. In addition, it checks (1) that a peg-out is embedded in a Burn on the sidesystem, and (2) that this Burn appears in the sidesystem state ommitted in w $\mathsf{SidesystemState}(w)$ up to the admissible block.

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

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

PegIn **Transaction**	
Inputs	(0) $(*, *, \mathsf{CheckSig}_{\mathsf{pk}_A})$
Outputs	(0) $(v\,\mathrm{BTC}, \mathsf{CheckCovenant})$
Witness	(0) σ_A

Alice shares PegIn with all operators who execute the setup of the Bitcoin light client (cf. Section 4), including PegIn as an additional input. Specifically, both the Payout

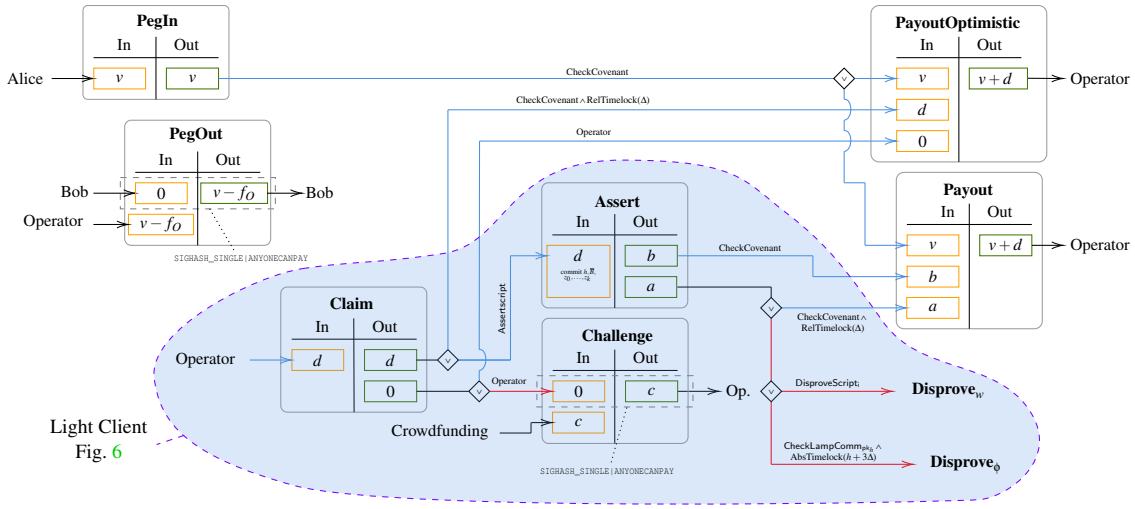

Figure 7: The BITVM2-BRIDGE construction.

and PayoutOptimistic transactions now require an extra input, spending the output of PegIn. This modified transaction graph is depicted in Fig. 7. Once operators complete this step, Alice publishes PegIn on Bitcoin, which results in minting equivalent wrapped BTC (v wBTC) on $\mathcal{L}_B$. This trustless minting can be handled via light clients on $\mathcal{L}_B$ [7] or atomic swaps [14, 29] and is thus omitted here.

Transfer and payment. Wrapped BTC circulate freely on $\mathcal{L}_B$ as regular coins. Let Bob be the last owner of the wBTC before peg-out.

Peg-out. Bob constructs and signs a PegOut transaction that takes as input a fixed, arbitrary zero-value output and an unspecified second input $*$, sending $v - f_O$ coins to himself. He signs the first input and output (using `SIGHASH_SINGLE|ANYONECANPAY`, see [22]). Bob then posts Burn on $\mathcal{L}_B$, burning his v wBTC and embeds the presigned PegOut (e.g., via an `OP_RETURN` output), denoted PegOut $\in$ Burn in Algorithm 3. This embedding commits Bob to a specific $\mathcal{L}_A$ transaction: by tying the burn on $\mathcal{L}_B$ to a unique presigned PegOut, Bob cannot reuse the same burn to claim multiple PegOut.

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

Operators monitoring $\mathcal{L}_B$ extract the PegOut, complete it by adding the second input using their own funds, and post it to $\mathcal{L}_A$. Because the first input of PegOut spends a unique connector output from Bob, up to a single operator can successfully publish a valid PegOut avoiding concurrency issues. Operators compete, since successful completion yields a fee f_O from the transaction.

Operator reimbursement. The operator whose PegOut is included on $\mathcal{L}_A$ then executes the Bitcoin light client protocol, i.e., post Claim to recover their funds from the bridge. Due to our design, an honest operator always manages to claim the locked BTC (v BTC), recovering their fronted $v - f_O$ BTC and netting a profit of f_O (see Section 6 and [22]).

We emphasize that after a successful dispute of a malicious operator's Claim, PegIn's output remains unspent. An honest operator can later publish a valid Claim, and receive the PegIn coins, as if no prior claim had been made. Thus, honest Bob will not lose his coins in case of a successful dispute.

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

6 Analysis Overview

In this section, we informally state the main security theorems for our three protocols. Formal statements and full proofs appear in [22].

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

Theorem 6.1 (Soundness)**.** *If the SNARK has knowledge soundness, the ledger is safe and live, and a set of signers and a set of verifiers with existential honesty,* BITVM2-CORE *satisfies soundness.*

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

Intuition. Soundness holds because an invalid proof would either require breaking SNARK soundness or a violation of ledger safety/liveness. Completeness holds because if the witness is valid, the payout must appear unless the SNARK is not complete or the ledger itself fails.

Bitcoin light client. We next analyze the light client of Section 4.

Theorem 6.3 (Informal Light Client Security)**.** *The light client accepts only* admissible blocks*: with overwhelming probability, the accepted block is both recent (within $m+k$ blocks of the tip) and will remain in all honest chains.*

Intuition. Liveness follows because the chosen block is always close to the tip; safety follows from the Bitcoin's immutability (common-prefix property [13]), which prevents an adversary from replacing such a block.

BITVM2-BRIDGE. Finally, we establish bridge security for both directions of transfer and for operator incentives.

Theorem 6.4 (Peg-in Safety)**.** *The peg-in bridge is safe under the assumption that at least one signer is honest during setup and that $\Pi_{\mathcal{L}_A}$ satisfies ledger safety.*

Intuition. A forged mint would require all signers to collude or a valid peg-in to be removed from Bitcoin; both violate our assumptions.

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

Intuition. An honest operator triggers the mint, honest signers authorize it, and ledger liveness ensures eventual inclusion on the sidesystem.

Theorem 6.6 (Informal Peg-out Safety)**.** *The peg-out bridge satisfies safety assuming at least one honest signer post-setup, and $\mathcal{L}_B$ satisfies safety.*

Intuition. A peg-out on Bitcoin is possible only if a corresponding Burn exists on the sidesystem: all users are honest during setup, an honest post-setup signer additionally prevents fraudulent presigning, and sidesystem safety ensures valid Burns are not removed.

Theorem 6.7 (Informal Peg-out Liveness)**.** *The peg-out bridge satisfies liveness, assuming all signers are honest during setup while at least one remains honest post-setup, $\mathcal{L}_A$ is safe and live, and an honest operator monitors $\mathcal{L}_B$.*

Intuition. Once a valid Burn appears on the sidesystem, an honest operator will complete the peg-out, presigned transactions (from setup with at least one honest signer) allow redemption, and Bitcoin liveness ensures inclusion.

Assuming an honest challenger, BITVM2-BRIDGE also satisfies Bridge Operator Safety.

Theorem 6.8 (Informal Bridge Operator Safety)**.** *An honest operator who performs a valid peg-out is guaranteed to recover their funds, assuming the signer committee is honest during setup and one member remains honest after, an honest challenger exists, both ledgers are live and safe, the light client is secure, and* BITVM2-CORE *is complete and sound.*

Intuition. An honest operator can fail to recover funds only if (i) another operator wrongfully claims the payout, or (ii) no payout appears despite a valid witness. The former cannot happen because BITVM2-CORE is sound, the light client is safe, and the signers are honest during setup and at least one signer is honest after (deleting their key). The latter cannot occur because the signers are honest during setup, the light client is live, and BITVM2-CORE is complete.

7 Evaluation

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

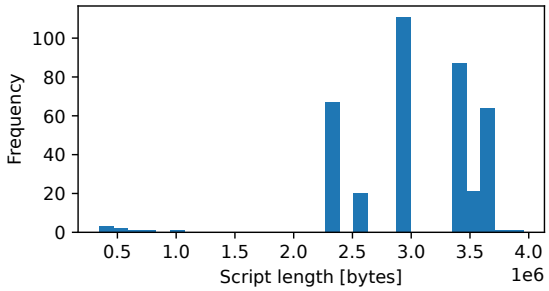

Figure 8: Histogram of script lengths in our implementation

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

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

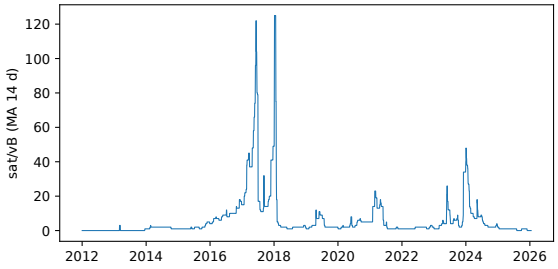

Figure 9: Fee-rate that gives a $\geqslant 95\%$ chance of confirmation within 14 days (mempool.space data).

On June 3 2025 we validated the full unhappy path on Bitcoin mainnet: starting with the Claim [4], followed by the Challenge [3], then the Assert [2], and finally the (largest) Disprove [5] transaction. The sequence confirmed in 42 blocks (7h 36min), demonstrating that a challenge period is viable even under mainnet congestion. The total cost was 14.9M sat (≈ \$16,,000 as of June 2025), with Assert and Disprove accounting for 3.89M and 10.99M sat, respectively. This path is unlikely in practice, as honest operators avoid losing collateral and challengers must pay these fees in case their challenge fails (cf. Section 3). Rather, this amount reflects the collateral required per peg-in in BITVM2-BRIDGE. Reducing the size of the Assert and Disprove transactions remains an active area of research.

The happy path, which is the one normally executed, involves instead only PegOut, Claim, and PayoutOptimistic that remain below 16.75 vkB, i.e., around 50k sat (roughly \$ 53). We discuss how to further reduce this cost to 534 vbytes, or around 1.6k sat (roughly \$ 1.66) in Section 8.

In practice, users will not directly interact with BitVM deposits and withdrawals. Instead, they will likely swap BTC for wBTC on L2s, previously bridged by professional operators and sold for a convenience fee[6].

8 Practicality, Limitations, and Extensions

We conclude by discussing key practical aspects of our design as well as limitations, design trade-offs, and extensions.

Funding challenge. Any user can challenge an operator's Claim, forcing them to reveal the Assert transaction, which results in increased on-chain costs even for an honest operator. To prevent such griefing attacks, challengers must post collateral covering the fees of an Assert transaction.

While this collateral protects operators, it may be costly for a single challenger to provide. To address this, we allow the challenge collateral to be crowdfunded to amortize costs. The operator presigns the first input and output of Challenge using the SIGHASH flag [30] `SIGHASH_SINGLE|ANYONECANPAY`, enabling multiple challengers to contribute inputs until the required collateral is reached.

Operator liquidity. By design, when a user initiates a peg-out, one of the operators must front the equivalent BTC to the user immediately, before that operator can reclaim the locked funds from the Bitcoin side. The upside of this design is that users receive their BTC payouts without waiting for the dispute period to elapse – the withdrawal is effectively instant from the user's perspective. The downside is that operators need sufficient BTC liquidity on hand to cover user withdrawals on the spot. This introduces a capital cost for operators, who must tie up funds to serve the bridge. To compensate, the protocol includes operator fees (and potentially additional incentives from the sidesystem, such as a share of L2 transaction fees) so that operating the bridge remains profitable despite the liquidity demands. This liquidity-provider model is already common in cross-chain bridges, and such incentives ensure that operators remain willing to serve exits. Future additions to Bitcoin's consensus could allow for a design in which the identity of the spender of the PegOut transaction does not need to be determined at the time of setup, removing the need for operators altogether.

Large, non-standard transactions. Some transactions, notably Assert and Disprove, exceed Bitcoin's 400 kB "standardness" relay policy. While this prevents them from propagating through the default peer-to-peer network, miners can still include them if received directly. Several mining pools already provide public submission channels for valid non-standard transactions,[7] making inclusion feasible in practice. We discuss how we could reduce the transaction sizes in Section 8.

Fixed deposit amounts. Our current BITVM2-CORE instantiation only supports fixed deposit sizes, meaning users must hold the exact wrapped amount to peg out. In practice, this can be mitigated by deploying multiple bridge instances with different denominations (e.g., 0.5–100 BTC). We also expect peg-ins/outs to be handled by professional liquidity providers who aggregate and rebalance deposits on behalf of users.

Covenant emulation via a signer committee. One of the limitations of BITVM2-CORE stems from the need to emulate covenants using a signer committee. Since we only use n-of-n multisignatures, i.e., without threshold signing, the signer set can be scaled far beyond the "vanilla" Bitcoin multisignature to 100+ signers using protocols like MuSig [23, 25, 26], at

[6] This behavior is common on chains like Ethereum and Solana, with multi-billion annual rebalancing volumes.

[7] See, e.g., https://slipstream.mara.com/.

the cost of increased off-chain communication and coordination effort. While the implementation of this setup is beyond the scope of this paper, we note that such setups have been successfully executed at a much larger scale, for example, Ethereum's KZG trusted setup ceremony[8] which involved over 140,000 participants, executed via a simple website.

For each bridge peg-in instance, we assume that the signer committee behaves honestly and participates in the BITVM setup protocol. This leads to a fundamental design choice: one may either reuse the same signer committee across multiple bridge instances, requiring the signers to remain continuously available, or instantiate a fresh signer set per bridge instance, which only needs to be honest and active during its one-time setup.

Signature optimization. Replacing Lamport with Winternitz signatures [10] halves the signature size.

Tuning Assert **and** Disprove **Sizes**. The current design commits to the full SNARK execution trace in a single Assert transaction. In principle, the computation could be split across multiple Assert transactions that could be used in parallel, each committing to a smaller execution segment, thereby reducing the size of individual Assert and Disprove transactions. However, under covenant emulation, this approach incurs a fundamental trade-off: with n parallel Assert segments, the signer committee must presign $O(2^n)$ Payout transactions to ensure an honest operator can reclaim funds, regardless of which of the n Assert transactions a malicious challenger forces them to reveal.

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

Atomic swaps. To address both fixed deposit sizes and the requirement to run additional software for initiating a BITVM2-CORE deposit, we expect that peg-ins and peg-outs in BITVM2-BRIDGE be handled primarily by professional users offering these operations as a service. In particular, bridge operators may naturally assume this role and act as market makers. Under this model, end users do not interact with peg-in and peg-out transactions directly. Instead, users enter and exit the sidesystem via cross-chain swaps [15]. A market maker (Alice) maintains balances in both BTC on Bitcoin and wBTC on the sidesystem, and offers users (e.g., Bob) the ability to swap between the two assets against a fee. The market maker periodically rebalances her inventories by executing peg-in and peg-out operations on BITVM2-BRIDGE as needed. This design abstracts away denomination constraints and protocol complexity from end users, while concentrating bridge interactions on actors equipped to manage liquidity and operational overhead. Similar architectures have been deployed at scale on Ethereum and its layer-2 systems via so-called "liquidity bridges."[9]

Rotating operators. In our design, we assume that each BITVM2-BRIDGE instance has a pre-defined operator set. While it is possible to add and remove operators during runtime, the implementation of a such mechanism depends on the Covenant model. In our case, where we emulate covenants via a signer committee, the m signers cannot add or remove operators later on, as this would contradict our protocol and model that assumes at least one honest signer that deletes their key. Even if such modifications were possible, for example, by employing key-evolving cryptography, our signer set is designed to be large for safety reasons. As a result, the coordination overhead may be significant and the failure rate (e.g. a signer being offline) high, possibly making ad-hoc operator rotation impractical for an existing bridge instance.

Instead, we can introduce a pre-defined, periodic rotation schedule embedded in the BITVM2-BRIDGE ruleset, i.e., the SNARK verifier. Under this model, as long as there exists one honest operator per BITVM2-BRIDGE instance, we could rotate operators for existing instances by executing a peg-out transaction that at the same time acts as a peg-in transaction for a new BITVM2-BRIDGE instance with a different operator set. A similar rotation design is implemented by tBTCv2[10].

The design of mechanisms for selecting operators for a BITVM2-BRIDGE instance is orthogonal to our construction and outside the scope of this paper. Existing approaches include random sampling based on staked assets, as commonly employed in Proof-of-Stake systems (e.g., staking BTC [11, 21]), as well as sampling based on Proof-of-Work contributions over a fixed time window, for instance in sidesystems that are merge-mined with Bitcoin.

Alternative light client designs. Beyond the approach adopted in this work, one may consider alternative Bitcoin light client constructions based on explicit block commitments by operators.

One such design requires operators to periodically commit on-chain to the latest Bitcoin block they observe, using Lamport or Winternitz signatures that enable challenges against incorrect commitments. While this approach provides verifiability, it incurs substantial communication overhead and on-chain costs, making it impractical.

A refinement of this approach replaces periodic on-chain commitments with optimistic off-chain publication, for example via a bulletin board or a blockchain with inexpensive data availability. This makes the optimistic case of full honesty cheaper. If an operator omits or publishes an incorrect commitment, a challenger can force the commitment on-chain, where it can be disputed using a protocol similar to Section 3.

However, in the worst case, a malicious challenger can repeatedly trigger on-chain commitments, effectively forcing all

[8] https://github.com/ethereum/kzg-ceremony

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

[10] https://github.com/keep-network/tbtc-v2/tree/main/docs

data on-chain. While one could require challengers to cover or share the resulting on-chain costs, the associated capital requirements undermine the practicality of this approach.

Consensus changes enhancing BitVM2. Various proposed consensus changes would enable more efficient bridge designs: (a) Native big-integer arithmetic [28] would substantially reduce the SNARK verifier's script size from gigabytes to megabytes, potentially enabling full verification in a single transaction and significantly simplifying the design. (b) Native support for signature verification of arbitrary stack data (e.g., `OP_CHECKSIGFROMSTACK`) would reduce the cost of one-time signature schemes, improving the efficiency of Winternitz signatures by avoiding indirect message encoding. (c) Any covenant proposal that allows committing to transaction inputs would remove the need for a signer committee, guaranteeing unconditional deposits safety. Proposals include introspection opcodes (e.g., `OP_INSPECTINPUTOUTPOINT`) or `TXHASH`, with introspection-based approaches being particularly efficient in terms of transaction fees.

Beyond rollups. In this work we focus on Bitcoin rollups, which naturally fit our design by using Bitcoin consensus through data commitments and verifiable state transitions. More generally, BitVM2-core can also support bridges to blockchains with independent consensus, by embedding their light client inside the SNARK verifier.

Acknowledgments

The work was partially supported by the European Research Council (ERC) under the European Union's Horizon 2020 research (grant agreement 771527-BROWSEC), by the Austrian Science Fund (FWF) through the SFB SpyCode project F8510-N and F8512-N, by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development and the Christian Doppler Research Association through the Christian Doppler Laboratory Blockchain Technologies for the Internet of Things (CDL-BOT), by the WWTF through the project 10.47379/ICT22045 and by Ministero dell'Università e della Ricerca (MUR), issue D.M. 352/2022 "Borse di Dottorato" - Dottorato di Ricerca di Interesse Nazionale in "Blockchain & Distributed Ledger Technology", under the National Recovery and Resilience Plan (NRRP), by Input Output (https://iohk.io) through their funding of the Edinburgh Blockchain Technology Lab, and by the Stanford IOG Blockchain Research Hub. We also thank the members of the BitVM Alliance for implementing BitVM2 and Babylon for performing the mainnet experiment (Section 7).

Ethical Considerations

We here discuss the ethical considerations of our work. Our contribution is twofold: BitVM2-core, a protocol for proving on Bitcoin the output of arbitrary functions, and BitVM2-bridge, a protocol for transferring bitcoins to and from sidesystems. It is expected to more closely integrate Bitcoin with other blockchains and cryptocurrencies. As such, the relevant stakeholders are individuals and organizations that invest directly or indirectly in Bitcoin and other cryptocurrencies, entities that develop blockchain infrastructure and provide bridging services, and the public at large.

Many Bitcoin users choose to bridge their bitcoins to other systems for a variety of reasons. Our work improves the state of the art by building Bitcoin bridges with minimal trust to third parties. Such bridges lead to loss of funds under strictly fewer scenarios than alternatives, expanding the scope of beneficial outcomes of Bitcoin bridging. Reducing the risks of Bitcoin bridging can have second-order benefits, including the integration of Bitcoin, the most secure and established cryptocurrency, into complex smart contracts, an increase in Bitcoin price, and wider Bitcoin adoption, benefiting all stakeholders involved with Bitcoin.

Potential harm for bridge users may come in two forms: Firstly, if our trust assumptions fail, loss of funds is still possible. We consider this risk to be acceptable, as alternatives involve strictly higher risks of failure. Furthermore, different bridge users may choose to trust different sets of operators, meaning that a trust failure for one user does not necessarily lead to loss of funds for another user, i.e., trust failures are local. Secondly, the reduced trust requirements of our protocol may prompt currently passive Bitcoin holders that are unfamiliar with the risks of cryptocurrency investment vehicles of other blockchains to engage in risky investment strategies, potentially causing loss of savings. We judge that this portion of the population is sufficiently small, and that, in line with the Menlo report, sufficient harm reduction measures against reckless investment are in place, as well as that individuals have the right to take autonomous, informed decisions regarding their financial behavior, even if it leads to harm.

Our work involves no human subjects. Furthermore, we do not provide a complete end-user product. Still, we urge any entity that provides our protocol to end-users to respect the autonomy of persons, while providing sufficient protections against harmful use, e.g., reckless investment decisions.

Bitcoin does not offer financial privacy and our work does not change the status quo in this respect. Future end-users should be informed that third parties may potentially deduce the users' financial history by observing Bitcoin and the relevant sidesystem.

In the interest of transparency, all outcomes of our work, including code implementations and experimental data, are made publicly available. We refer the reader to our statement on Open Science for more details.

Open Science

Our full implementation of BitVM2 is available at https://doi.org/10.5281/zenodo.17949747. The transactions involved in the on-chain validation of the unhappy path, which was conducted on June 3, 2025, are the following: Claim [4], Challenge [3], Assert [2] and Disprove [5] — we provide their transaction IDs and links to them in a blockchain explorer webpage in the references.

References

[1] BitVM reference implementation: Zenodo code repository. https://doi.org/10.5281/zenodo.17949747.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[18] Jae Kwon and Ethan Buchman. Cosmos whitepaper. *A Netw. Distrib. Ledgers*, 27:1–32, 2019.

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

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

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

[22] Robin Linus, Lukas Aumayr, Zeta Avarikioti, Matteo Maffei, Andrea Pelosi, Orfeas Thyfronitis Litos, Christos Stefo, David Tse, and Alexei Zamyatin. Bridging bitcoin to second layers via BitVM2. Cryptology ePrint Archive, Paper 2025/1158, 2025.

[23] Gregory Maxwell, Andrew Poelstra, Yannick Seurin, and Pieter Wuille. Simple schnorr multi-signatures with

applications to bitcoin. *Designs, Codes and Cryptography*, 87(9):2139–2164, 2019.

[24] Threshold Network. tbtc: Threshold network documentation. https://docs.threshold.network/, 2020. Accessed: 2025-09-19.

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

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

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

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

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

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

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

[32] Tiancheng Xie, Jiaheng Zhang, Zerui Cheng, Fan Zhang, Yupeng Zhang, Yongzheng Jia, Dan Boneh, and Dawn Song. zkbridge: Trustless cross-chain bridges made practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, 2022.

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

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

A Covenant Emulation

In this section, we formally define covenant emulation and prove that the method described at a high level in Section 3, "Signer Committee" Subsection (and its algorithm, described in [22]) implements this interface. We first need some preliminary definitions.

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

(i) Tx is the root of T,

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

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

In a spending tree with root Tx, we call *spending path* between Tx and an intermediate node Tx' the nodes on the path from Tx to Tx'. By writing $\mathsf{Tx} < \mathsf{Tx}'$, we denote that there is a spending path between Tx and Tx', and by $\mathsf{Tx} \nless \mathsf{Tx}'$ we denote that no such path exists.

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

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

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

$$\Pr[\mathsf{Tx}' \in {}^{r}\mathcal{L}^{\bigcap} : \mathsf{Tx} < \mathsf{Tx}' \wedge \forall i, T'.leaf(i) \nless \mathsf{Tx}' \mid T' \leftarrow CES(\lambda, \mathsf{Tx})] \leqslant \mathsf{negl}(\lambda). \quad (1)$$

In the Setup procedure of the BitVM2-core protocol (cf. Algorithm E.1 of [22]), we implement a Covenant Emulation System. The BitVM2-core prover collaborates with a set of signers to create a covenant tree to condition the spending of the Claim transaction. Specifically, the prover creates the transaction skeletons that are the nodes of the covenant tree with root Claim, and sends them to the signers. They receive the transactions and sign some of them only if the covenant tree is constructed as expected. The resulting covenant tree is shown in Fig. 10.

Lemma A.1. *If the signature scheme Σ is EUF-CMA secure and there is an honest signer, then the procedure described in Algorithm E.1 of [22] is a restrictive Covenant Emulation System.*

We defer the proof to [22].

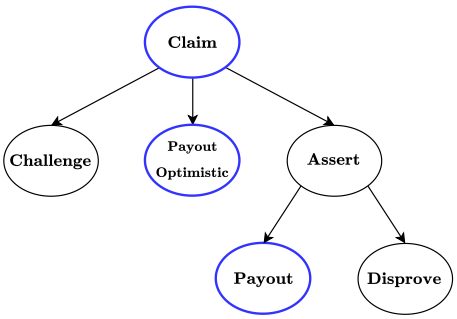

Figure 10: The covenant tree for the Claim transaction built by the Setup procedure of the BITVM2-CORE protocol. Transactions depicted with a blue border are signed by the signing committee during protocol execution.

B Background and Notation

B.1 Lamport digital signature scheme

Let $h : X \to Y$ be a one-way function, where $X := \{0,1\}^*$ and $Y := \{0,1\}^\lambda$, for a given security parameter λ. Let $m \in \{0,1\}^\ell$ be a ℓ-bit message, with $\ell \in \mathbb{N}_{>0}$. A *Lamport digital signature scheme* [19] Lamp consists of a triple of algorithms $(\mathsf{KeyGen}, \mathsf{Sig}, \mathsf{Vrfy})$, where:

- $(pk_\mathcal{M}, sk_\mathcal{M}) \leftarrow \mathsf{Lamp.KeyGen}(\ell)$ (Algorithm 4), is a PPT algorithm that takes as input a positive integer ℓ and returns a key pair, consisting of a secret key $sk_\mathcal{M}$ and a public key $pk_\mathcal{M}$ which can be used for one-time signing a ℓ-bit message. For readability, $\mathcal{M} = \{0,1\}^\ell$ is an alias for the ℓ-bit message space.
- $c_m \leftarrow \mathsf{Lamp.Sig}_{sk_\mathcal{M}}(m)$ (Algorithm 5), is a DPT algorithm parameterized by a secret key $sk_\mathcal{M}$, that takes as input a message $m \in \mathcal{M}$ and outputs the signature c_m, which we also call (Lamport) commitment.
- $\{\mathsf{True}, \mathsf{False}\} \leftarrow \mathsf{Lamp.Vrfy}_{pk_\mathcal{M}}(m, c_m)$ (Algorithm 6), is a DPT algorithm parameterized by a public key $pk_\mathcal{M}$ that takes as input a message m, a signature c_m, and outputs True iff c_m is a valid signature for m generated by the secret key $sk_\mathcal{M}$, corresponding to $pk_\mathcal{M}$, i.e., $(pk_\mathcal{M}, sk_\mathcal{M})$ is a key pair generated by Lamp.KeyGen.

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

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

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

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

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

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

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

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

For a formal definition of one-time security and the proof that Lamport signatures are one-time secure digital signature schemes (assuming the existence of one-way functions), refer to, e.g., [10]. Lamport signatures, and in particular Algorithm 6, are implementable in Bitcoin Script; a sample implementation is available at [34].

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

B.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, used to prove ownership over coins. We let $\sigma_\mathsf{U}(m)$ be the digital signature of U over a message $m \in \{0,1\}^*$. If it is clear what message is signed, we sometimes use σ_U as shorthand.

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

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

A transaction is valid when all witnesses fulfill the locking condition of their corresponding input; all of the transaction's inputs are unspent; the sum of the value of the outputs is smaller or equal to the sum of the value of the inputs (the difference is given to the miners).

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

- **Signature locks.** An output locked with $\mathsf{CheckSig}_{\mathsf{pk}_\mathsf{U}}$ can only be spent if the spending transaction is signed with the secret key of the key pair $(\mathsf{sk}_\mathsf{U}, \mathsf{pk}_\mathsf{U})$.
- **Multisignature locks.** To fulfill a multisignature spending condition, a certain number k out of n signatures are required. For example, for users A and B, a 2-of-2 multisignature spending condition is denoted as $\mathsf{CheckMultiSig}_{\mathsf{pk}_{A,B}}$ and the respective signature as $\sigma_{A,B}$.
- **Timelocks** lock a transaction output until a specified time in the future (absolute timelock) or until a specific time after the transaction is included on-chain (relative timelock). We denote the former as $\mathsf{AbsTimelock}(\Delta)$, and the latter as $\mathsf{RelTimelock}(\Delta)$. In the following, we use timelocks in conjunction with other spending conditions. For instance, if the UTXO $\mathsf{Tx.out}_1$ has locking script $\mathsf{lockScript} := \mathsf{RelTimelock}(\Delta) \wedge \mathsf{CheckSig}_{\mathsf{pk}_\mathsf{U}}$, the user U can spend the UTXO $\mathsf{Tx.out}_1$ after that a certain amount of time T has passed from the moment that $\mathsf{Tx.out}_1$ has been published on-chain.
- **Taproot trees** [31], or Taptrees, make a UTXO spendable by satisfying one among multiple spending conditions. The spending conditions are (Tap)leaves of a Merkle tree. To spend a UTXO that has a Taptree as a locking script, a user needs to provide a witness for one of the leaves along with a Merkle inclusion proof of such leaf into the Taptree. We denote the Tapleaves of a Taptree locking script as $\langle \mathsf{leaf}_1,\ldots,\mathsf{leaf}_r\rangle$; when a user fulfills script leaf_i to unlock the j-th UTXO of the transaction Tx, we write the corresponding input as $(\mathsf{Tx}, j, \langle \mathsf{leaf}_i\rangle)$. Every time that a user spends a UTXO via a Tapleaf of a Taptree, we assume that the user has provided a valid Merkle inclusion proof for the Tapleaf.
- **Other conditions.** We denote with True a condition that is always fulfilled and with False a condition that can never be fulfilled. In the latter case, the coins can not be redeemed, and they are *burnt* instead.

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