BitVM2: Bridging Bitcoin to Second Layers

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

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

Abstract. BitVM2 is a novel paradigm that enables arbitrary program execution in Bitcoin, thereby combining Turing-complete expressiveness with the security of Bitcoin consensus. At its core, BitVM2 leverages optimistic computation, assuming operators are honest unless proven otherwise by challengers through fraud proofs, and SNARK proof verification scripts, which are split into sub-programs that are executed within Bitcoin transactions. As a result, BitVM2 ensures program correctness with just three on-chain transactions. BitVM2 significantly improves over prior BitVM designs by enabling, for the first time, permissionless challenging and by reducing the complexity and number of on-chain transactions required to resolve disputes. Our construction requires no consensus changes to Bitcoin.
BitVM2 enables the design of an entirely new class of applications in Bitcoin. We showcase that by presenting BitVM Bridge, a protocol that enhances prior Bitcoin bridges by reducing trust assumptions for the safety of deposits from an honest majority (t-of-n) to existential honesty (1-of-n) during setup. To guarantee liveness, we only require one active rational operator (while the others can be malicious). Any user can act as challenger, facilitating permissionless verification of the protocol.

1 Introduction

Bitcoin, similar to other Layer 1 blockchains, faces significant scalability challenges, driving the need for Layer 2 (L2) solutions to enhance network performance. In Bitcoin's case, L2s can also serve to extend scripting functionality to match the one of expressive blockchains like Ethereum. While other L1s have successfully integrated L2s, most notably Ethereum, Bitcoin's implementation is hindered by the limitations of its scripting language. Although we can express any program in Bitcoin script, contrary to common misconceptions in the community, these programs quickly exceed the Bitcoin block size and stack limits due to inefficiencies stemming from a very limited set of natively available operations.

Bridges that facilitate the transfer of BTC from Bitcoin to other blockchains as "wrapped" assets are an essential component of L2 architectures. These bridges are necessary for managing deposits and withdrawals from and to the underlying L1. At the core of every bridge is the critical issue of custody – how BTC is securely stored while its wrapped counterpart is utilized on other blockchains. Currently, almost all Bitcoin bridges rely on multi- or threshold signature schemes, where a group of t-of-n signers is entrusted with safekeeping BTC. Although some bridges employ economic security through collateralization [25], these designs face scalability challenges due to high capital requirements and have thus achieved limited adoption in practice.

Contribution. This paper brings two fundamental contributions, the design of BitVM2, a new paradigm that enables execution of arbitrary programs in Bitcoin, and, based on that, BitVM Bridge, a novel protocol to bridge BTC to Bitcoin layer 2 systems.

- Similar to Truebit[21] and Arbitrum [10], BitVM2 makes use of *optimistic computation* where the computing party (operator) is assumed to be honest until proven guilty using so-called *fraud proofs*. In

BitVM2 a set of operators commits (on-chain) to correctly executing a SNARG [5] verifier (off-chain) for an arbitrary program, such that any observing party (challenger) can disprove a faulty computation within a pre-defined challenge period. We implement a Groth16 SNARK [7] verifier program in Bitcoin script [26] and split it into sequential sub-programs, each of which is small enough to be executed in a Bitcoin block. An operator challenged by a challenger must reveal on-chain the intermediary results of these sub-programs, alongside the input and output states of the SNARK verifier. By design, if the operator is trying to cheat, one of the published intermediary states will be incorrect, i.e., different from that of the sub-program the operator committed to. To disprove a faulty operator's claim, a challenger must simply execute the respective sub-program on-chain in a Bitcoin transaction, showing a mismatch between the computation and the operator's claim. Following these design principles, BitVM2 ensures that *anyone* can challenge and disprove a faulty operator within only 3 on-chain transactions.

BitVM2 is inspired by the BitVM paradigm proposed by Linus [14] and makes significant improvements in terms of computational and communication complexity, as well as security. While in BitVM and similar designs, only a fixed set of operators can perform challenges, BitVM2 supports for the first time *permissionless challenging*, i.e., any user with a Bitcoin full node can challenge faulty operators. Furthermore, the original BitVM design required up to 70 on-chain transactions (over a period of multiple weeks or even months in practice) to disprove a faulty operator, whereas BitVM2 requires only 3 on-chain transactions that can be executed within 1-2 weeks, achieving similar practical properties as Ethereum L2s.

– We showcase the groundbreaking design opportunities enabled by BitVM2 by presenting BitVM Bridge, a novel protocol to bridge BTC to Bitcoin layer 2 systems. BitVM Bridge significantly improves over existing Bitcoin bridge designs, reducing trust assumptions from an honest majority of signers to existential honesty while necessitating only one active rational operator and rational challengers to safeguard the protocol.

Organization. We first present an (informal) model, protocol goals, and protocol overview in Section 2. We then introduce the necessary background, notation, and building blocks in Sections 3 and 4. We combine these building blocks to introduce BitVM2, a general function verification protocol operating securely on Bitcoin in Section 5. Subsequently, we outline a bridge protocol based on BitVM2, termed BitVM Bridge, in Section 6. In the current draft, Sections 7 and 8 sketch our plan for the security analysis and limitations and extensions sections, respectively, that will be completed at a later time in the full version.

2 Model and Protocol Overview

In this section, we present our model and assumptions and provide a high-level overview of the BitVM2-based bridge protocol. The security properties of BitVM2 may be of independent interest and will be discussed and proven in the security analysis as a stepping stone to the security of the bridge.

2.1 Assumptions

A blockchain or distributed ledger protocol takes as input transactions (provided by users) and outputs an immutable transaction order that eventually includes all provided transactions. We assume Bitcoin and a sidesystem maintain robust public transaction ledgers that have *persistence* and *liveness*, as defined in [4]. We denote as Δ_L the liveness parameter, i.e., the upper bound for how long it takes an honest actor to include a transaction in the underlying blockchain.

We assume that the sidesystem leverages Bitcoin for its consensus, for example, is a so-called *roll-up* that uses data commitments posted to the Bitcoin blockchain for consensus on the transaction order, paired with some form of state transition verification. The strawman protocol is illustrated in Fig. 5.

We assume a synchronous network, i.e. , all messages broadcast to the network will be delivered to all parties within a known time bound. We make the usual cryptographic assumptions: the participants are computationally bounded and cryptographically secure communication channels, hash functions, signatures, and encryption schemes exist. We abuse the notion of blockchains to also refer to their transaction ledgers.

2.2 System Model and Goals

In our setting, a bridge protocol pertains two main operations: 1) The Peg-In, where a user, Alice, deposits funds u₿ on Bitcoin, which are then represented in a sidesystem as "wrapped BTC" (₿$_s$); 2) The Peg-Out, where another user, say Bob, claims these funds u from the sidesystem to the Bitcoin blockchain.

Besides the roles of depositor (Alice) and beneficiary (Bob), in our bridge protocol we additionally have the following participant roles:

- **Operators ($O_1, ..., O_m$)** [7] : The operators are responsible for executing a pre-agreed program f; in the case of our bridge, this is the peg-out process. The operators are assumed to be rational, i.e., they are profit-maximizing agents.
- **Challengers**: The challengers ensure the safety of the peg-out process by challenging an operator in case of misbehavior. Anyone can act as a challenger, including the operators. We assume that challengers are rational.
- **Signers ($S_1, ..., S_n$)**: A committee of n signers that is responsible for the correct setup of a BitVM2 instance for a pre-agreed program f. One of the n signers is assumed to be honest (existential honesty). As we explain in Section 4.2, this (committee and thus this) assumption is not necessary if the underlying blockchain supports covenants.

We assume all participants are computationally bounded, i.e., cryptographic primitives are secure. We first show that the setup of the bridge is secure given a signer committee with existential honesty. Under this assumption, we prove the BitVM2-bridge is secure when all challengers are rational and at least one active operator is rational (the rest can be malicious). We underscore here that while operators are predefined entities (permissioned), anyone can act as a challenger (permissionless) and thus any party wishing to exit the sidesystem can act as a challenger to ensure safety.

Protocol Goals. The following refers to the bridge protocol and encompasses both its safety and liveness.

- **Balance security:** Any user holding v coins in the sidesystem, can burn them and then, and only then, eventually claim $v - f_O$ coins on the main blockchain, where $f_O \geqslant 0$ is a fee charged to cover the bridge's operating costs.

2.3 BitVM2 Protocol Overview

The core idea of BitVM2 is to enable arbitrary computations to occur *optimistically*, i.e., the computing party (operator) is assumed to be honest unless proven guilty by a challenger using so-called fraud proofs. This design choice stems from the limited storage and computational capabilities enabled by the Bitcoin protocol. We utilize several tricks in BitVM2 to achieve this goal while inheriting Bitcoin security.

1. We start by compressing the program using SNARKs. In BitVM2 we do not verify the program execution itself but the correct execution of the SNARK verifier verifying a proof for the program execution.
2. We implement the SNARK verifier in Bitcoin Script. The program size (e.g. 3GB for Groth16) exceeds the block size by magnitudes.
3. We split the verifier into sub-program chunks, each at most 4MB in size. These chunks *can* be executed in a Bitcoin transaction/block. Thereby, the sub-programs are *sequential*: program 2 takes as input the output of program 1 and so on.
4. The operator commits to executing the program (i.e., SNARK verifier) correctly during setup. Pre-signing carefully crafted Bitcoin transactions and Taproot trees ensures the operator can only withdraw funds in a way that can be challenged in case of misbehavior.
5. When the operator wants to withdraw funds from BitVM2, e.g. take out BTC as part of the peg-out process of a bridge, they must post the output of the SNARK verifier on-chain.

[7] We expect the cardinality of this committee to depend on the maximum amount of xBTC held by the bridge and the fees required for the operators.

6. Anyone can check this data against their local execution of the SNARK verifier, given the publicly known inputs (data availability problem aside). If they disagree, anyone can post an on-chain challenge, forcing the operator to reveal more data on the computation.
7. If challenged, the operator must reveal all *intermediary outputs* of the sub-programs in an on-chain transaction.
8. Now, anyone can find the incorrect intermediary result and prove that the operator cheated by executing the corresponding sub-program chunk on-chain, showing that the results do not match. This invalidates the operators BitVM2 withdrawal transaction and slashes (some) limited amount of collateral (reimbursing on-chain costs).

In summary, BitVM2 allows anyone to challenge and slash a faulty operator with 3 on-chain transactions, with a delay of no more than 2-3 weeks. A high-level overview of the BitVM2 protocol is provided in Fig. 1.

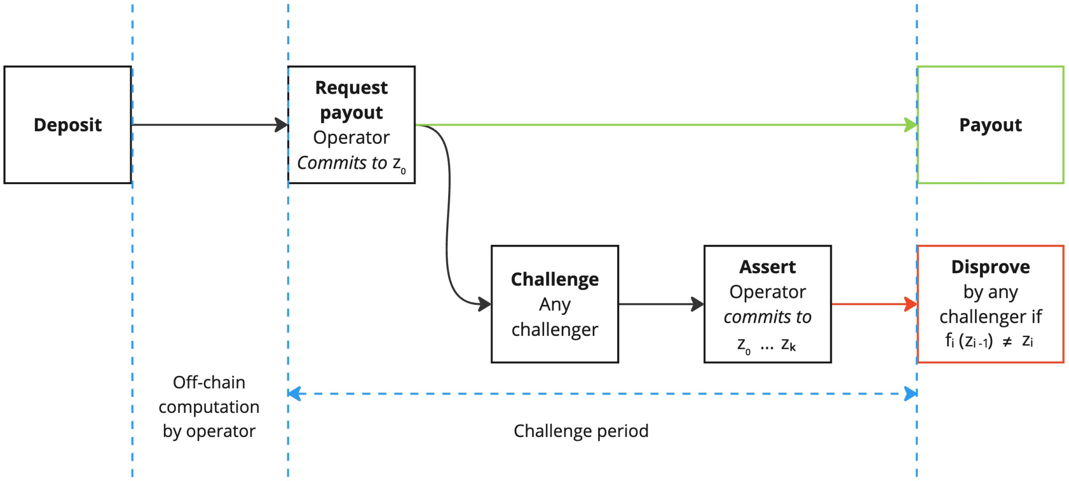

Fig. 1. Simplified overview of the BitVM2 transaction flow. The SNARK verifier program $f(x) = y$ is split into $f_1, f_2, ..., f_k$ with intermediary results $z_0, ..., z_k$. A challenged operator must reveal the intermediary states in the Assert transaction. This allows a challenger to disprove a false claim of an operator by executing the disputed sub-program f_i in a Bitcoin transaction.

3 Background and Notation

3.1 Digital signatures

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

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

In this work, we make use of signature schemes that are EUF-CMA secure [6].

3.2 Succinct non-interactive arguments (SNARGs)

For this definition, we closely follow, e.g., [5,7]. Let $R \leftarrow \mathcal{R}(\lambda)$ be a relation generator that takes as input a security parameter λ, and returns a polynomial time decidable binary relation R. We denote ϕ as the statement and w as the witness for the pairs $(\phi, w) \in R$. We define an efficient publicly verifiable non-interactive argument for $\mathcal{R}$ as a tuple of three PPT algorithms: Setup, Prove, and Vrfy.

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

$(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Vrfy})$ is a non-interactive argument for $\mathcal{R}$ if it has *perfect completeness* and *computational soundness*, as defined in [7,5]. On a high level, the former means that given any true statement ϕ, an honest prover can convince an honest verifier with overwhelming probability; the latter means that it is not possible to prove a false statement, i.e., convince the verifier if no witness exists, also with overwhelming probability. Finally, a non-interactive argument, where the verifier runs in polynomial time in $\lambda + |\phi|$ and the proof size is polynomial in λ, is denoted as (preprocessing) succinct non-interactive argument (SNARG). Note that while we later use an implementation of [7], we do are not interested nor do we utilize the zero-knowledge property or the stronger notion of succinct non-interactive arguments of knowledge (SNARKs).

3.3 Lamport digital signature scheme

Let $h : X \to Y$ be a one-way function, where $X := \{0,1\}^*$ and $Y := \{0,1\}^\lambda$, for a given security parameter λ. Let $m \in \{0,1\}^\ell$ be a ℓ-bit message, with $\ell \in \mathbb{N}_{>0}$. A *Lamport digital signature scheme* [12] 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 1), 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 2), 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 3), is a DPT algorithm parameterized by a public key $pk_{\mathcal{M}}$ that takes as input a message m, a signature c_m, and outputs True iff c_m is a valid signature for m generated by the secret key $sk_{\mathcal{M}}$, corresponding to $pk_{\mathcal{M}}$, i.e., $(pk_{\mathcal{M}}, sk_{\mathcal{M}})$ is a key pair generated by Lamp.KeyGen.

Lamport signatures are secure one-time signatures. 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.

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., [2]. Lamport signatures, and in particular Algorithm 3, is implementable in Bitcoin Script; [26] contains a sample implementation.

Since we leverage Lamport signatures to enable a party to commit to a bit (and thus to a message), from now on, we will refer to the Algorithm 2 as LampComm instead of Lamp.Sig and to the Algorithm 3 as CheckLampComm instead of Lamp.Vrfy.

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

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

Algorithm 2 The Lamport signature algorithm $\mathsf{Lamp.Sig}$, 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 3 The Lamport verification algorithm $\mathsf{Lamp.Vrfy}$, 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
```

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

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

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

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

- **Signature locks.** An output locked with $\mathsf{CheckSig}_{\mathsf{pk}_\mathsf{U}}$ can only be spent, if the spending transaction is signed with the secret key corresponding to 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 $\mathbf{Tx}.\mathsf{out}_1$ has locking script $\mathsf{lockScript} := \mathsf{RelTimelock}(\Delta) \wedge \mathsf{CheckSig}_{\mathsf{pk}_\mathsf{U}}$, the user U can spend the UTXO $\mathbf{Tx}.\mathsf{out}_1$ after that a certain amount of time T has passed from the moment that $\mathbf{Tx}.\mathsf{out}_1$ has been published on-chain.
- **Taproot Trees** [23], or Taptrees, make a UTXO spendable by satisfying one among multiple spending conditions. The spending conditions are (Tap)leaves of a Merkle tree. To spend a UTXO that has a Taptree as a locking script, a user needs to provide a witness for one of the leaves along with a Merkle inclusion proof of such leaf into the Taptree. In the following, we denote the Tapleaves of a Taptree locking script as $\langle \mathsf{leaf}_1, \ldots, \mathsf{leaf}_r \rangle$; when a user fulfills script leaf_i to unlock the j-th UTXO of the transaction $\mathbf{Tx}$, we write the corresponding input as $(\mathbf{Tx}, j, \langle \mathsf{leaf}_i \rangle)$. 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 $\ast$ to denote a transaction input, witness, or output that can be anything (valid according to Bitcoin consensus rules), but is irrelevant to our protocol.

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

SIGHASH flags. SIGHASH flags specify which part of the transaction data is included in the hash that is signed as part of a signature lock. These flags are primarily used to help coordinate multiple users in creating and signing a transaction.[8]

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

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

4 BitVM2 Building Blocks

In this section, we present a set of Bitcoin script primitives that we make use of in the BitVM2 design.

4.1 Stateful Bitcoin scripting via one-time signature public key hard-coding

Even though the Bitcoin scripting language is stateless, clever use of one-time digital signatures schemes, e.g., Lamport signatures, makes it possible to preserve a state over different Bitcoin transactions. Consider the following example. Let a user U hold a Lamport key pair $(sk_{\mathcal{M}}, pk_{\mathcal{M}})$ associated with $\mathcal{M}$, the set of all v-bit messages. We can think of $\mathcal{M}$ as a variable that can hold any v-bit string. Using this mental model, U can assign a value to m to $\mathcal{M}$, by creating the commitment $c_m \leftarrow \mathsf{LampComm}_{\mathcal{M}}(m)$.

Hard-coding $\mathsf{CheckLampComm}_{pk_{\mathcal{M}}}$ for a public key $pk_{\mathcal{M}}$ in the locking script of multiple outputs, such a variable assignment can not only be checked but also carried over from one output to another, thus enabling a (global) state in Bitcoin. This is done by reading m and c_m from the unlocking script of one output and passing them via the unlocking script to another output.

4.2 Emulating covenants using presigned transactions

In Bitcoin Script, covenants (e.g., [16,19,8]) are a *proposed* class of spending constraints that would allow a transaction's locking script to impose restrictions on how the coins locked in a UTXO can be spent in the future. At the time of writing, covenants have not yet been added to Bitcoin. If added to Bitcoin Script, covenants would allow to restrict the outputs of subsequent transactions. A simple example of a covenant is to restrict a UTXO spendable by Bob such that it can only be spent if the spending transaction allocates 5 BTC to a user Alice. Among other things, covenants enable the storage of a state and execution of a state machine through a series of different transactions. This provides for more expressive smart contract capabilities and would enable more complex applications in Bitcoin.

In BitVM2, we need to restrict how UTXOs can be spent such that operators spending from BitVM2 can be challenged by the challengers. In the absence of covenants, we can emulate their functionality using a committee of n signers $S_1, \ldots S_n$, where at least one is honest. In practice, this idea can be implemented

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

such that anyone is allowed to join the committee as long as they are active (inactive users are kicked from the committee to prevent denial of service), and thus honest users could convince themselves of this existential honesty assumption by joining the committee [3]. Specifically, during the setup phase of *every* BitVM2 instance, we introduce the following steps:

1. Each signer generates a fresh key pair.
2. For each transaction output that should be spendable only in a specific way, we introduce an additional n-of-n multi-signature spending condition $\mathsf{CheckMultiSig}_{\mathsf{C}}$, i.e., all signers must collaboratively create the signature σ_{C} to spend the UTXO.
3. For each of these UTXOs, the signers pre-sign the specific transactions that should be used to spend the output. Each operator receives a dedicated set of pre-signed transactions, spendable by them under certain conditions.
4. Finally, the signers delete their keys.

This mechanism ensures that as long as one of the signers is honest and deletes their key, the UTXOs can only ever be spent using one of the pre-signed transactions, i.e., in the intended way of the protocol. We can further use signature aggregation schemes to improve the efficiency of the setup, reducing the on-chain footprint. As a side note, if all protocol parties are known upfront, e.g., in a 2-party or in a permissioned protocol, they can form the committee.

For readability and to highlight that enabling covenants can eliminate the need for the committee, we abstract this emulation and henceforth use CheckCovenant to refer to a spending condition on outputs. Our transaction graphs and formal transaction definition show, which transactions can spend those outputs: No transactions other than the ones we defined can spend them. In other words, whenever we write CheckCovenant (e.g., in Fig. 2), this can be replaced with the n-of-n multi-signature spending condition $\mathsf{CheckMultiSigVerify}_{\mathsf{pk}_{\mathsf{C}}}(\sigma_{\mathsf{C}})$ paired with a transaction pre-signed by the signer committee or by an actual covenant introduced to Bitcoin in the future. We use Covenant in the witness of transactions to indicate that the conditions of the Covenant were met.

4.3 Connector outputs

Connector Outputs[9] is a technique to ensure that only one of a given set of Bitcoin transactions is valid and can be included in the Bitcoin blockchain. Specifically, we create multiple transactions that require as input the same connector output. We achieve this by introducing an additional input (that references the connector output) to each transaction and set the `SIGHASH_ALL` flag, requiring all inputs to be present for the transaction to be valid. As soon as one of these transactions is included in the Bitcoin blockchain, spending the connector output, the other transactions become invalid. The connector output can thereby specify a variety of custom spending conditions. In BitVM2 we use connector outputs to invalidate a faulty operator's attempt to spend funds from the BitVM2 instance after a successful challenge by a challenger.

5 BitVM2: General Function Verification on Bitcoin

In this section, we show how arbitrary functions can be computed in Bitcoin using Bitcoin Script without the need for consensus changes. We use this towards building the BitVM2-based bridge protocol, but note that such a building block is interesting as a standalone construction. Suppose that a party O, the operator (acting as prover), wants to prove on Bitcoin that for a given program f written in Bitcoin Script, an input x and an output $y \in V$ out of a set of possibly multiple valid outputs V, the assertion $f(x) = y$ holds.

Contrary to common beliefs in the community, Bitcoin Script does support universal computation, i.e., we can represent any program in (a likely very large) Bitcoin Script. However, Bitcoin's consensus rules impose some limitations on the size of a valid block, and, in turn, on the size of a script.[10] Since f is a

[9] For example, described in `https://tinyurl.com/2p566ynp`.

[10] At the time of writing, after the SegWit soft-fork, the maximum block size is 4 MB (as discussed in `https://tinyurl.com/5e4kudyj`); as a consequence, it is not possible to write scripts bigger than 4 MB.

Bitcoin Script program, executing f (or any sub-program f_i, for some $i \in \{1, \ldots, k\}$) means manipulating data in a stack that can store a fixed number of elements.[11] Let the content of the stack at any given point in the program execution be the *state* of the program. Using such terminology, the input x of f and the output y are the initial and the final state of the program, respectively.

In BitVM2, some party stakes a deposit a deposit of d₿, where $d \geqslant 0$, which the operator can claim after successful verification. The deposit d can be one of two things, depending on the use case of f:

- A deposit by a third party, e.g., requesting a computation, that is paid to the operator after successful execution;
- A collateral deposit by the operator that is reclaimed by the operator after correct execution, in cases where the payment for the computation is processed outside of BitVM2 such as the BitVM Bridge described in Section 6.

5.1 Naive Function Verification

In the following, we leverage Lamport signatures and Taptrees to enable the operator O to assert on Bitcoin that $f(x) = y$ even if the script needed to encode the program does not fit into a block. To achieve this, we devise an optimistic verification mechanism that allows any user (challenger) to *disprove* the operator O if the provided assertion is incorrect.

1. **Setup**. First, we split the program f into k sub-programs $f_1, \ldots, f_k$, where each sub-program is below the script size limit, i.e., can be executed in a Bitcoin block. Let z_i be the intermediate state after the execution of the sub-program f_i; we have that

$$\begin{aligned} z_1 &:= f_1(z_0), \\ z_2 &:= f_2(z_1), \\ &\ldots, \\ z_k &:= f_k(z_{k-1}) \end{aligned}$$

   where $z_0 := x$ and $z_k := y$. For each element in $\{z_0, z_1, \ldots, z_k\}$, the operator O creates a fresh Lamport keypair $(sk_{z_0}, pk_{z_0}), (sk_{z_1}, pk_{z_1}), \ldots (sk_{z_k}, pk_{z_k})$. Next, the operator creates three transactions where they hard-code these public keys in the locking scripts: **Assert**, **Disprove**, and **Payout** that are defined below in Eqs. (1) to (3) and illustrated in Fig. 2. Using Covenants, we restrict the way the involved outputs can be spent, i.e., only via these specific transactions. As explained in Section 4.2, we emulate Covenants by having the signer committee pre-sign **Assert**, **Disprove** and **Payout** during the setup phase and sharing them with the involved parties, who check the validity of such signatures.
2. **Execution**. The operator O executes the sub-programs $f_1, \ldots, f_k$ off-chain, given input $x = z_0$ and in doing so, produces intermediary states $z_1, \ldots, z_{k-1}$ as well as the final state $y = z_k$. This output z_k should be in the set of valid outputs V, e.g., V could be $\{\mathsf{True}\}$.
3. **Commit and Challenge Period.** The operator then publishes an **Assert** transaction (specified in Eq. (1)) on-chain, committing to $z_0, \ldots, z_k$, claiming correct execution of the program. The input to the **Assert** transaction can be any output (indicated by $\ast$) holding d₿ that must require at least the following conditions for spending: (i) the requirements of the Covenant, i.e., can only be spent by the **Assert** transaction, (ii) the operator's signature, and (iii) a valid Lamport commitment $c_{z_0}, \ldots, c_{z_k}$ for each of the public keys $pk_{z_0}, \ldots, pk_{z_k}$. We will use this in constructions presented later in this paper.

$$\begin{aligned} \textbf{Assert} := \Big(&\mathit{input} = [(\ast)], \\ &\mathit{witness} = [(\ast, \mathsf{Covenant}, z_0, \ldots, z_k, c_{z_0}, \ldots, c_{z_k})], \\ &\mathit{output} = [(d₿, \langle \mathsf{DisproveScript_1}, \ldots, \mathsf{DisproveScript_k}, \\ &\mathsf{RelTimelock}(\Delta_A) \wedge \mathsf{CheckCovenant} \wedge \mathsf{CheckSig}_{\mathsf{pk}_O} \rangle)]\Big), \end{aligned} \tag{1}$$

[11] Bitcoin protocol rules limit the size of the stack to 1000 elements (`https://tinyurl.com/mt7ex7xh`), where each element can be up to 5 bytes (`https://en.bitcoin.it/wiki/Script`).

The **Assert** transaction creates a new output with the following script. Let $k + 1$ Tapleaves form a Taptree, specifying the following spending conditions.

- **Happy path (Payout)**. The last, $(k+1)$-th leaf can be spent after the pre-defined challenge period Δ_A by (i) fulfilling the condition of the Covenant and (ii) providing the operator signature.
- **Unhappy path (Disprove)**. The first k leaves handle dispute cases. The leaf at position i (for $i \in \{1, \ldots, k\}$) hard-codes the sub-program f_i, the public key $pk_{z_{i-1}}$ for the program input z_{i-1} and the public key pk_{z_i} for the program output z_i. Spending such a leaf requires (i) fulfilling the conditions of the Covenant (ii) correct commitments for the input and output public keys of the program f_i, (iii) and that $z_i \neq f_i(z_{i-1})$ holds for the values corresponding to these commitments, i.e., the operator committed to an incorrect intermediary state z_i for a sub-program f_i. We present the detailed output script (the $\mathsf{DisproveScript}$) in Algorithm 4.

Given this information, an incorrect claim made by a faulty operator, i.e., if $z_i \neq f_i(z_{i-1})$ for some $i \in \{1, \ldots, k\}$, can be disproved by any observing party within the challenge period Δ_A by publishing a **Disprove** transaction.

4. (i) **Payout**. If no one publishes the **Disprove** transaction, after some time Δ_A, the operator can spend the output of the **Assert** transaction by publishing the **Payout** transaction and claiming the funds for themselves.

$$\begin{aligned}\textbf{Payout} := \Big(&\mathit{input} = [(\textbf{Assert}, 0, \langle \mathsf{RelTimelock}(\Delta_A) \wedge \mathsf{TakeScript}\rangle)],\\ &\mathit{witness} = [(\mathsf{Covenant}, \sigma_O)],\\ &\mathit{output} = [(d\text{₿}, \mathsf{CheckSig}_{\mathsf{pk}_O})]\Big),\end{aligned} \tag{2}$$

(ii) **Disprove**. Using the operator's commitments to $z_0, \ldots, z_k$ in the witness of the **Assert** transaction, anyone can disprove an incorrect claim about the intermediary states of the program execution, i.e., if $z_i \neq f_i(z_{i-1})$ for some $i \in \{1, \ldots, k\}$. This is done by publishing the **Disprove** transaction specified in Eq. (3) that spends the output of the **Assert** transaction. Specifically, a locking script of one of the first k leaves must be fulfilled. This is possible if and only if the operator made an incorrect claim about one of the intermediary states $z_0, \ldots, z_k$. Anyone can read the commitments made by the operator O from the chain and provide them as inputs to the script in one of the leaves. At this point, the script is verified on-chain and all (honest) Bitcoin full nodes come to consensus that the operator was faulty.

$$\begin{aligned}\textbf{Disprove} := \Big(&\mathit{input} = [(\textbf{Assert}, 0, \langle \mathsf{DisproveScript_i}\rangle)],\\ &\mathit{witness} = [(\mathsf{Covenant}, z_{i-1}, z_i, c_{z_{i-1}}, c_{z_i})],\\ &\mathit{output} = [(b\text{₿}, \mathsf{False}), (a\text{₿}, \mathsf{True})]\Big),\end{aligned} \tag{3}$$

The outputs of this transaction determine what happens with the d₿ provided by the operator. Specifically, the **Disprove** transaction must create a new output that burns part b₿ of the deposit d₿, where $0 \leqslant b \leqslant d$. By setting the SIGHASH flag [22] to `SIGHASH_SINGLE`, the aforementioned input and output become immutable. This allows any user to publish the **Disprove** transaction while adding further outputs where they can, e.g., claim the remaining a₿ as bounty or to cover transaction fees, where $a = d - b$. For simplicity, we denote this in the above definition as $(a\text{₿}, \mathsf{True})$. This mechanism provides an incentive for users to publish this transaction and allows indeed anyone to do so and claim the money.

We illustrate the transactions involved in the function verification mechanism on Bitcoin in Fig. 2.

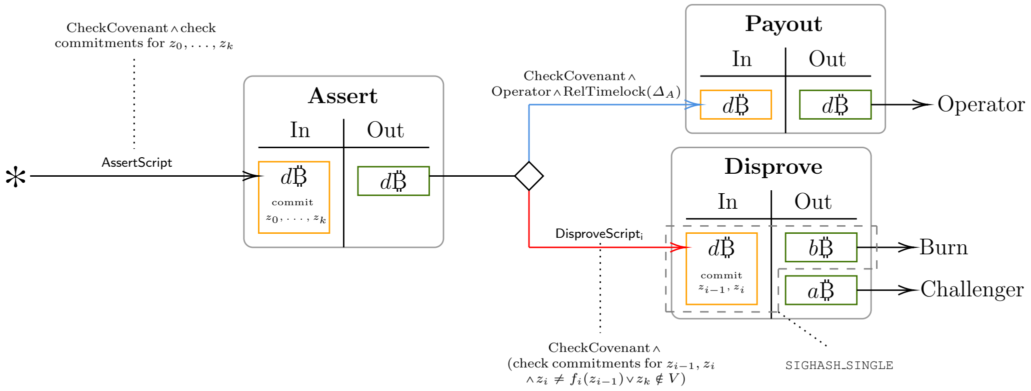
CheckCovenant ∧ check commitments for $z_0, \ldots, z_k$
AssertScript
∗
Assert
In
Out
d₿
commit $z_0, \ldots, z_k$
d₿
CheckCovenant ∧ Operator ∧ RelTimelock(Δ_A)
Payout
In
Out
d₿
d₿
Operator
Disprove
In
Out
DisproveScript$_i$
d₿
commit z_{i-1}, z_i
b₿
Burn
a₿
Challenger
CheckCovenant ∧ (check commitments for z_{i-1}, z_i $\wedge z_i \neq f_i(z_{i-1}) \vee z_k \notin V$)
SIGHASH_SINGLE

Fig. 2. Illustration of the transactions **Assert**, **Disprove** and **Payout**. The input of **Assert** can be any UTXO, signified by $\ast$, however, we require the spending script to be a Covenant and to include commitments for $z_0, \ldots, z_k$. To increase readability, we introduce the following color-coding: gray rounded rectangles depict the transactions. We denote the BTC locked in a transaction output by drawing a green rectangle and the BTC spent by a transaction input by drawing an orange rectangle. Blue arrows denote spending paths taken by the operator and red arrows denote spending paths taken by someone else. Above the arrows, we write the conditions to spend the outputs from which the arrows start. Gray dashed rectangles around some of the inputs and outputs of a transaction denote which portion of the transaction is hashed and pre-signed whenever a SIGHASH flag different from `SIGHASH_ALL` is used.

Algorithm 4 The $\mathsf{DisproveScript_i}$ for each Tapleaf $\mathsf{i} \in \{1, \ldots, k\}$. The algorithm takes as input the committee's multisignature σ_C, the intermediate states z_{i-1}, z_i of the program f, along with the Lamport commitments $c_{z_{i-1}}, c_{z_i}$.

```
function DisproveScript_i($\sigma_C$, $z_i$, $z_{i-1}$, $c_{z_i}$, $c_{z_{i-1}}$)
    CheckCovenant();
    CheckLampCommVerify_{$pk_{z_{i-1}}$}($z_{i-1}, c_{z_{i-1}}$);        ▷ Both the public keys $pk_{z_{i-1}}$ and $pk_{z_i}$ are hard-coded in the script
    CheckLampCommVerify_{$pk_{z_i}$}($z_i, c_{z_i}$);
    if $z_i \neq f_i(z_{i-1}) \vee (i = k \wedge z_k \notin V)$ then        ▷ $f_i$, $V$ and $k$ are hard-coded in the script
        return True
    else
        return False
    end if
end function
```

5.2 Cost-effective, Optimistic Function Verification

In the current design, both the **Assert** and **Disprove** transactions are significant in size, resulting in high transaction fees when posted on-chain. We address this by extending the construction to an optimistic model, significantly reducing the on-chain footprint under honest operation. Thereby, the operator at first commits to the input x of the program f and must only reveal the output and intermediary states if challenged by a challenger. To this end, we introduce additional transactions to the **Commit and Challenge Period**, as well as the **Payout** phases. **Setup** and **Execution** remain unchanged. The improved construction is illustrated in Fig. 3.

1. **Setup**. If using a committee to emulate covenants, the $\mathsf{PayoutOptimistic}$ transaction also needs to be created and pre-signed by the signer committee. Otherwise, it remains unchanged.
2. **Execution**. Remains unchanged.
3. **Commit and Challenge Period**. When committing to the execution of program f, the operator first posts a **Claim** transaction (specified in Eq. (4)), indicating that for some $y = z_k$ in a predefined set V, they know some $x = z_0$ such that $f(x) = y$. The operator can convince any potential challenger that they know such an x either by posting it on-chain or by sending it to them via any other communication channel, e.g., using a third party data availability layer.

$$\begin{aligned}\textbf{Claim} := \Big(&\mathit{input} = [\ast],\\ &\mathit{witness} = [(\sigma_O \wedge \ast), x],\\ &\mathit{output} = [(d\text{₿}, \langle \mathsf{RelTimelock}(\varDelta_B) \wedge \mathsf{CheckCovenant}, \mathsf{AssertScript}\rangle), (0\text{₿}, \mathsf{CheckSig}_{\mathsf{pk}_O})]\Big),\end{aligned} \tag{4}$$

Again, note that x can be omitted from the witness if shared off-chain, and the script $\mathsf{AssertScript}$ is defined as

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

During the following challenge period $\varDelta_B$, anyone can dispute the claim by publishing a **Challenge** transaction (specified in Eq. (6)), which forces the operator to respond by publishing the **Assert** transaction, which in turn can be disproved. The second output of the **Claim** transaction thereby acts as a so-called *connector output*: it must remain be unspent for the operator use **PayoutOptimistic** to withdraw funds from $\mathsf{BitVM2}$ unchallenged.

4. (i) **Payout**. If the operator's claim goes unchallenged, the operator can post the **PayoutOptimistic** transaction (specified in Eq. (5)) after period $\varDelta_B$ to spend both outputs of the **Claim** transaction. This way, the operator claims funds from $\mathsf{BitVM2}$ and disables the dispute logic.

$$\begin{aligned}\textbf{PayoutOptimistic} := \Big(&\mathit{input} = [(\textbf{Claim}, 0, \langle \mathsf{RelTimelock}(\varDelta_B) \wedge \mathsf{CheckCovenant}\rangle),\\ &(\textbf{Claim}, 1, \mathsf{CheckSig}_{\mathsf{pk}_O})],\\ &\mathit{witness} = [(\mathsf{Covenant}), (\sigma_O)],\\ &\mathit{output} = [(d\text{₿}, \mathsf{CheckSig}_{\mathsf{pk}_O})]\Big),\end{aligned} \tag{5}$$

   (ii) **Challenge and Disprove**. Any challenger can dispute the claim by publishing a **Challenge** transaction (specified in Eq. (6)), which spends the second (connector) output of the **Claim** transaction and disables the **PayoutOptimistic** transaction, forcing the operator into the dispute. Now, the operator only can access the $\mathsf{BitVM2}$ funds d₿ through the **Payout** transaction, which in turn requires the **Assert** transaction to be published. The **Assert** transaction can thereby be disproved in case of a faulty claim, as discussed in Section 5.3.

$$\begin{aligned}\textbf{Challenge} := \Big(&\mathit{input} = [(\textbf{Claim}, 1, \mathsf{CheckSig}_{\mathsf{pk}_O}), (\ast)],\\ &\mathit{witness} = [(\sigma_O), (\ast)],\\ &\mathit{output} = [(c\text{₿}, \mathsf{CheckSig}_{\mathsf{pk}_O})]\Big),\end{aligned} \tag{6}$$

5.3 BitVM2: SNARG Verifier on Bitcoin

We can use this technique to encode a SNARG verifier (see Section 3.2) on Bitcoin, in particular, the Groth16 [7] verifier (which even fulfills the stronger notion of a SNARK verifier). The goal is to execute

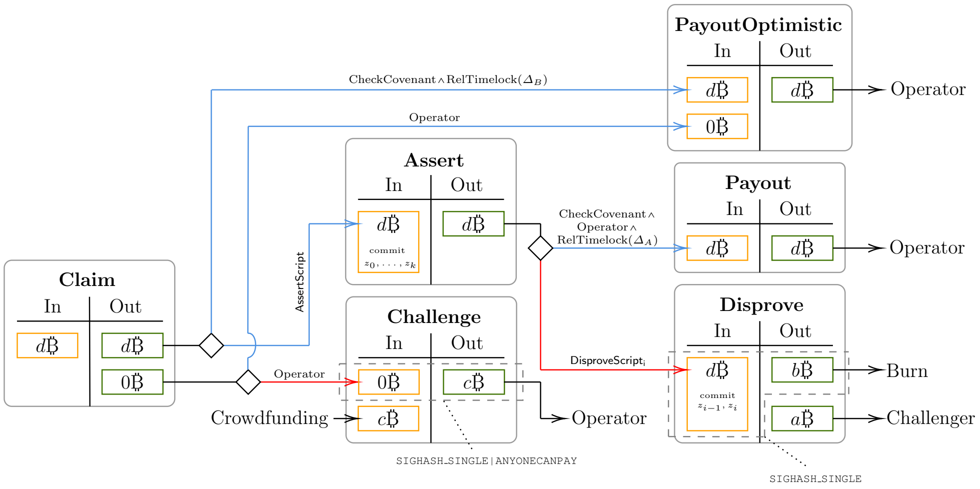

Fig. 3. Illustration of the optimistic function verifier.

transactions contingent on providing a valid SNARG proof. To achieve this, the **Assert** transaction is created with f as the SNARG verifier. This verifier is split up into k chunks and set up as Tap leaves as explained in Section 5. An operator can execute the **Payout** transaction only if they provide a correctly verifying proof. Otherwise, anyone can post the **Disprove** transaction. An example implementation of a Groth16 [7] verifier written in Bitcoin Script is provided in [26], showing that it is possible to verify SNARKs (and thus also the weaker notion, SNARGs) on Bitcoin. An overview of the resulting BitVM2 protocol is shown in Fig. 4.

5.4 Protecting against malicious challengers via collateral

Currently, any user can always challenge the operator, forcing them to reveal the **Assert** transaction, which results in increased on-chain costs, even if the operator is honest. We disincentivize such griefing attacks against the operator by requiring the challenger to put up some collateral c₿. This collateral c₿ should be parameterized such that it covers the transaction fees of **Assert**.

Crowdfunding Collateral for Challenges. While the collateral protects operators, it may deter challengers from triggering the challenge, should it be too expensive for individual users. We can mitigate this by crowdfunding the challenge by setting the SIGHASH flag [22] of the **Challenge** transaction to `SIGHASH_SINGLE|ANYONECANPAY`. The operator can effectively pre-sign the first input and output, broadcast this signature, and thus allow challengers to add more inputs. Consequently, instead of a single challenger covering the upfront costs for **Challenge**, the collateral can be shared among users, amortizing costs. Each participating challenger simply adds their input and sends the transaction along until enough inputs are provided to cover c₿.

6 BitVM Bridge: A Trust-Minimized Bitcoin Bridge

One of the main practical use cases of BitVM2 is to enable trust-minimized bridges between Bitcoin (and similar blockchains) and other blockchain networks, sidesystems, and in particular so-called Layer 2 networks. In the following, we refer to such networks collectively as *sidesystems*. As outlined in Section 2, the goal of a bridge is to create ("peg-in") a "wrapped" representation[12] of BTC on a sidesystem (₿$_s$) that can later be redeemed for BTC (₿) at a 1:1 ratio ("peg-out").

[12] Also referred to as cryptocurrency-backed assets, see definition in [25].

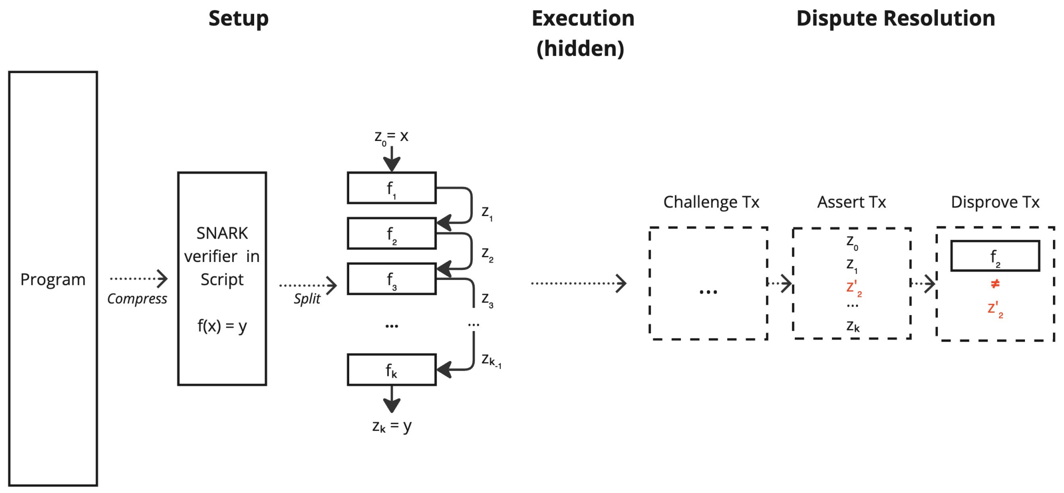

Fig. 4. High-level overview of the BitVM2 protocol. A SNARG (or SNARK) verifier (in the generic case this is an arbitrary program) is first implemented in Bitcoin Script, and then split into sub-programs $f_1, \dots, f_k$, such that each could be executed in a Bitcoin block. When closing the BitVM2 instance, operators commit to the program result. If challenged, the operator reveals the start, intermediary and end states $z_0, \dots z_k$. As a result, anyone can show that $z'_2 \neq f_2(z_1)$, disproving the faulty operator's claim.

Cross-chain bridges, by design, require a trusted third party [24] and are inherently difficult to build, which is why all Bitcoin bridges today rely on multisignature designs and honest majority assumptions. In the following, we show how we can reduce the additional trust assumptions (on top of Bitcoin and the sidesystem being secure) to the existence of one active rational operator and rational challengers, for achieving balance security. We note that the following holds assuming a correct setup (CheckCovenant) as described in Section 4.2, i.e., demands existential honesty in the signer committee used to emulate covenants. In particular, we outline a so-called *light client* bridge protocol, delineated below and illustrated in Fig. 5.

1. **PegIn**. A user Alice (A) deposits v₿ on Bitcoin into a BitVM2 instance via a **PegIn** transaction (specified in Eq. (7)). The single output of this transaction uses CheckCovenant as spending condition to enforce that it can only be spent via a transaction that can be challenged by *any challenger*. The sidesystem verifies that the **PegIn** transaction was included in the Bitcoin blockchain (via a Bitcoin light client function) and mints v wrapped ₿$_s$ to Alice's account in the sidesystem. This sidesystem verification and minting logic are enforced by the sidesystem's consensus, e.g., as a smart contract[13]

$$\begin{aligned}\textbf{PegIn} := \Big(&\mathit{input} = [(*, *, \mathsf{CheckSig}_{\mathsf{pk}_A})],\\ &\mathit{witness} = [(\sigma_A)],\\ &\mathit{output} = [(v₿, \mathsf{CheckCovenant})]\Big),\end{aligned} \tag{7}$$

   Alice can now freely use ₿$_s$ on the sidesystem. To simulate a real-world setting, we assume Alice transferred her v₿$_s$ to a user Bob (B), i.e., Bob has v₿$_s$ in his sidesystem account.
2. **PegOut**. Bob wants to withdraw the v₿$_s$ back to Bitcoin. For this, he burns the v₿$_s$ using a **Burn** transaction on the sidesystem, which makes the ₿$_s$ unspendable.

[13] See, e.g., [25] for a specification of such sidesystem mint and Bitcoin light client functionality.

Ideal functionality. In an ideal scenario, Bob should now be able to spend the v₿ previously locked by Alice via the **PegIn** transaction using a transaction $\textbf{PegOut}_{ideal}$. This ideal peg-out transaction would verify that the **Burn** transaction was correct and included in the sidesystem using a SidesytemLCScript that implements a light client functionality for the sidesystem.

$$\begin{aligned}\textbf{PegOut}_{ideal} := \Big(&input = [(\textbf{PegIn}, 0, \mathsf{CheckCovenant}), (\ast, \ast, \mathsf{SideSystemLCScript})],\\ &witness = [(\sigma_{\mathsf{C}}), (\ast)],\\ &output = [(v\text{₿}, \mathsf{CheckSig}_{\mathsf{pk}_B})]\Big),\end{aligned} \tag{8}$$

As of this writing, this ideal functionality cannot be implemented natively in Bitcoin Script, inter alia due to the fact that Bob – the user withdrawing from the bridge – is not known at the time of the peg-in, making it impossible to enforceably link the **PegIn** transaction to $\textbf{PegOut}_{ideal}$.

6.1 Strawman Bridge Design

In this section, we outline a strawman design for a BitVM2 bridge that emulates the ideal functionality of $\textbf{PegOut}_{ideal}$ using the optimistic SNARG verifier. Here, we assume that the sidesystem operates as a rollup, relying on Bitcoin for its consensus. In particular, the sidesystem uses data commitments posted to the Bitcoin blockchain to determine the order of transactions, combined with a method for verifying state transitions. This design simplifies verification by focusing only on validating transactions included in the Bitcoin blockchain. Consequently, BitVM2 only needs to implement a Bitcoin light client rather than both a sidesystem and a Bitcoin light client. We illustrate the strawman protocol in Fig. 5.

A fully operational bridge consists of multiple BitVM2 instances, each with a pre-defined amount of v₿ deposited though **PegIn** transactions. By design, wrapped BTC ₿$_s$ is fungible across different BitVM2 instances, i.e., it does not matter against which BitVM2 instance Bob is executing the **PegOut** (the selection can be determined by the sidesystem).

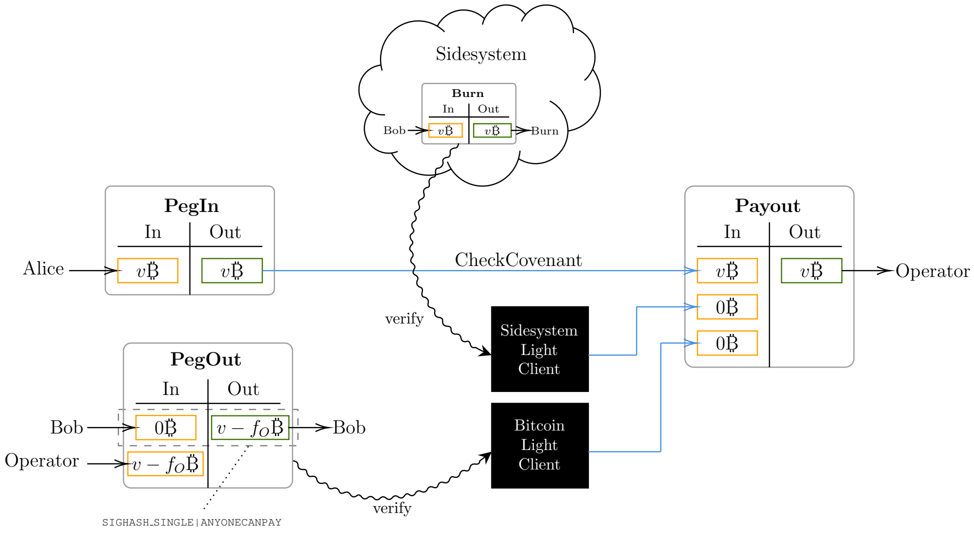

Fig. 5. Illustration of the strawman bridge protocol in the generic case. In our bridge protocol, the sidesystem light client collapses to the Bitcoin light client.

Initializing the SNARG. Recall from Section 3.2 that the SNARG is defined over a relation R (via algorithm SNARG.Setup). In our case, we are interested in a relation R with pairs (ϕ, w), where the statements ϕ consist of two block hashes, and the witness w consists of a blockchain. A statement ϕ is correct, i.e., $(\phi, w) \in R$ if there exists a blockchain w, such that:

1. The blockchain's first and last block hashes are ϕ;
2. The blockchain is valid, i.e., each block references its parent and its hash is smaller than the inscribed target;
3. The blockchain contains a **PegOut** transaction of the operator O, i.e., a transaction where O pays the amount of the **PegIn** transaction to Bob (possibly referencing the hash of the **PegIn** transaction in an `OP_RETURN` output);
4. The blockchain contains an inscribed sidesystem state in one of its blocks, with the sidesystem containing the **Burn** transaction of Bob.

More formally, let $\mathcal{H} : \{0,1\}^* \to \{0,1\}^\lambda$ be a hash function (modeled as a Random Oracle), let $\phi \in \{0,1\}^\lambda \times \{0,1\}^\lambda$ be two (block) hashes, let $B \in \{0,1\}^{bl}$ be a bl-length bit-string representing a block, that contains the field $B.parent \in \{0,1\}^\lambda$, let $w \in (\{0,1\}^{bl})^*$ be a list of blocks (a blockchain). We define $R := \{(\phi, w) : \mathit{\Pi}(\phi, w) = \mathsf{True}\}$, where $\mathit{\Pi}$ is defined in Algorithm 5. Note that we abstract the extraction of the sidesystem's state with the function $\mathsf{SidesystemState}(w)$, which returns a state of the sidesystem, inscribed on the Bitcoin blockchain w. The actual implementation depends on the sidesystem's operation.

Algorithm 5 The chainstate proof $\mathit{\Pi}$, which defines the relation R over which we define our SNARGs.

```
function $\mathit{\Pi}(\phi, w)$
    for $i \in \{0, \dots, length(w) - 2\}$ do
        if $w[i+1].parent \neq \mathcal{H}(w[i])$ then
            return False
        end if
    end for
    if $\phi[0] \neq \mathcal{H}(w[0]) \lor \phi[1] \neq \mathcal{H}(w[-1]) \lor$ PegOut $\notin w \lor$ Burn $\notin \mathsf{SidesystemState}(w)$ then
        return False
    end if
    return True
end function
```

Emulating a Bitcoin Light Client via `OP_BLOCKHASH`. A key component of the bridge is the ability to introspect and verify the Bitcoin state within the BitVM2 instance. This verification is essential to confirm that the **PegOut** transaction was executed correctly and aligns with the **Burn** transaction on the sidesystem. Since Bitcoin currently lacks built-in support for this functionality, it necessitates the implementation of a Bitcoin light client within BitVM2. We stress here that using connector outputs to ensure the atomicity of these two steps, rather than relying on the light client, is not feasible. Specifically, adding a connector output to the **PegOut** transaction and using it as input to the **Payout** transaction is not possible. This is because, in a bridge setting, we do not know the identity of the withdrawing user (Bob) in advance and, therefore, cannot determine the necessary transaction hash during setup.

For simplicity, we start with a strawman construction that makes use of a hypothetical Bitcoin Script opcode `OP_BLOCKHASH` and proceed to demonstrate a practical light client construction on Bitcoin without additional opcodes in Section 6.2. We assume `OP_BLOCKHASH` takes an element from the Bitcoin Script execution stack, interprets it as a number, and then fetches the block at that height and puts its hash on the stack.

In this strawman scheme, the operator includes the height h_{end} of a block that has occurred *after* the **PegOut** transaction in the **Assert** transaction's input along with commitments to $z_0, \dots, z_k$ (thus, a slightly modified version of Eq. (1)). Using `OP_BLOCKHASH`, the hash of the block corresponding to h_{end}, i.e., $\mathcal{H}(B_{end})$, is fetched. Then, it is checked that $\mathcal{H}(B_{end})$ is part of z_0. This is to ensure correct parameterization of the SNARG, such that we can verify that the block in which the **PegOut** was included is indeed part of the Bitcoin blockchain. Recall that the commitment for z_0 is verified in the $\mathsf{AssertScript}$, and in $\mathsf{DisproveScript}_1$. Thus, we use `OP_BLOCKHASH` to ensure that the block hash $\mathcal{H}(B_{end})$ that is part of z_0 and is fed as input to $\mathsf{SNARG.Vrfy}$ is part of the Bitcoin blockchain. The Genesis block B_{start} at height $h_{start} := 0$ (or some other block in the Bitcoin blockchain agreed upon during the bridge setup) can be hard-coded during the setup. Then we have $\phi = (\mathcal{H}(B_{start}), \mathcal{H}(B_{end}))$. Note that the SNARG will only return True, if the provided statement ϕ is correct. Thus, if the result z_k is True, this means that the **Burn** and **PegOut** transactions have indeed happened.

Strawman Protocol. We proceed to describe the phases of the strawman bridge protocol. Thereby, we emulate the ideal functionality of $\textbf{PegOut}_{ideal}$ using BitVM2 and employ operators and challengers to ensure the peg-out is processed correctly. The protocol is outlined below while the transactions with their dependencies are visualized in Figure 6.

1. **Setup (Peg-In)**. Following the BitVM2 design, SNARG.Vrfy represents the program f that is split into sub-programs. Recall that SNARG.Vrfy takes as input (R, crs, ϕ, π). As the relation R and the common reference string crs are known at setup time, they can be hard-coded, leaving our function with input of the form $z_0 = (\phi, \pi)$, consisting of a statement ϕ and a SNARG proof π. Otherwise, the setup remains unchanged to that of a standard BitVM2 instance as per Section 5.2. A user Alice (A) deposits v₿ via the **PegIn** transaction as per Eq. (7).
2. **Execution (Peg-out)**. First, Bob publishes the **Burn** transaction on the sidesystem. Next, Bob creates an *incomplete* **PegOut** Bitcoin transaction (specified in Eq. (9)), with a zero-value input from Bob and one output allocating $v - f_O$₿ to himself, where f_O is a pre-agreed fee charged by the operator for processing withdrawals. Bob thereby sets the `SIGHASH_SINGLE|ANYONECANPAY` flag for the transaction, which allows another user to complete the transaction by providing the outstanding v₿ as input.
At this stage, the operators come into play: One of the m operators effectively fronts the v₿ from their own funds. As a result, all of the operators compete to publish the **PegOut** transaction in order to claim the associated fees, but only one of the transactions will be accepted into the Bitcoin blockchain. The operator who successfully facilitates the peg-out then reclaims the fronted v₿ from BitVM2 by spending the output of **PegIn** transaction. Since the set of m operators is pre-defined for each BitVM2 instance, we can enforce the reclaiming process via the CheckCovenant. In the absence of the necessary Bitcoin opcodes, the CheckCovenant is emulated through a committee n-of-n multisig (cf. Section 4.2) that pre-signs the transactions during the **Setup** phase.

$$\begin{aligned}\textbf{PegOut} := \Big(&input = [(*, *, \mathsf{CheckSig}_{\mathsf{pk}_B}), (*, *, \mathsf{CheckSig}_{\mathsf{pk}_O})],\\ &witness = [(\sigma_B), (\sigma_O)],\\ &output = [(v - f_O\text{₿}, \mathsf{CheckSig}_{\mathsf{pk}_O})]\Big),\end{aligned} \tag{9}$$

The operator proceeds to publish the **PegOut** transaction to the Bitcoin blockchain, completing the withdrawal for Bob.
3. **Commitment and Challenge Period**. Next, the operator initiates a claim to recover the pre-funded amount of v₿ from BitVM2 (the **PegIn** output). This is done by publishing the **Claim** transaction (introduced earlier in Eq. 4). Before the operator can spend these funds, we must verify that the operator correctly processed the peg-out for Bob, i.e., the **PegOut** transaction that corresponds to the

Burn transaction on the sidesystem was included in the Bitcoin blockchain. All challengers (including other operators) perform this verification on their local machines and, in case of a dispute, publish the **Challenge** transaction within timeout Δ_B. This ensures that (i) the peg-out only happens when the **Burn** transaction has happened on the sidesystem, and (ii) prevents an operator from stealing the ₿ locked in the bridge.

4. **Payout or Disprove**. After the challenge period has elapsed, the operator finalizes the withdrawal by publishing **PayoutOptimistic**. Disputes are handled the same way as with any other BitVM2 program: a challenger publishes the **Challenge** transaction, the operator must respond by publishing the **Assert** whereupon the challenger can use the **Disprove** transaction to slash the operator (or an honest operator can use **Payout** to claim v₿ after timeout Δ_A).

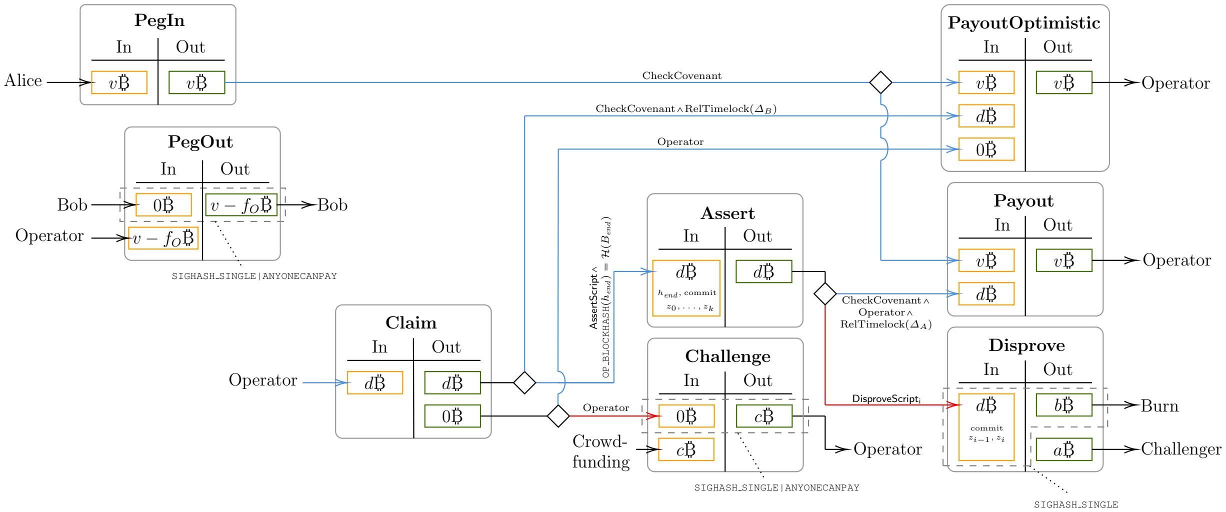

Fig. 6. Implementing transaction verification with a hypothetical opcode OP_BLOCKHASH. The input $z_0 := (\phi, \pi)$ consists of the statement and the snarg proof. The former $\phi = (\mathcal{H}(B_{start}), \mathcal{H}(B_{end}))$ consists of two block hashes. In the **Assert** transaction, the operator additionally commits to a height h_{end}, and it is checked that OP_BLOCKHASH$(h_{end}) = \mathcal{H}(B_{end})$.

6.2 BitVM Bridge: Bitcoin-compatible Bridge Protocol

In this section we show how to implement a Bitcoin light client using the BitVM2 optimistic SNARG verifier. Combining such a light client with the strawman bridge design that we described in Section 6.1, allows us to create the first practical, trust-minimized bridge that is compatible with today's Bitcoin.

To secure our bridge design, it is essential to revisit the necessary Bitcoin light client functionality. We need to verify two key points: (i) that the **Burn** transaction was correctly included in the sidesystem, and (ii) that the corresponding **PegOut** transaction was included in the Bitcoin blockchain before the operator O requests a refund through the **Claim** transaction. This ensures that the amount of burned v₿$_s$ by Bob matches the v₿ sent to Bob by the operator O on Bitcoin (minus fees f_O).

The main challenge for our light client is verifying that block B_{end}, whose hash is part of the witness w passed to the SNARG verifier, is part of the same blockchain as the **Assert** transaction. In other words, we need to emulate our hypothetical opcode OP_BLOCKHASH. Note that the SNARG verifier can check that a given chain is valid and that the transaction of interest is part of it, but it cannot perform blockchain introspection. As a result, a malicious operator could attempt to mine a private Bitcoin fork that contains the **PegOut** transaction, thus generating a valid SNARG proof. This proof could then be used to take the money via a **Assert** and **Payout** transaction, while not posting **PegOut** on the main chain.

PowPV: Light Client using Superblocks. We propose a practical light client model, taking advantage of so-called superblocks [11], i.e., Bitcoin blocks that exceed the minimum required difficulty target.

Superblocks. Recall that in Bitcoin, a valid block must exceed a certain difficulty target during each difficulty epoch of 2016 blocks (approximately 2 weeks). The more leading zeroes a block's hash has, the *heavier* the block is in terms of Proof-of-Work difficulty. As a result, we can order Bitcoin blocks by their block hash. Furthermore, for a given time period, $\varDelta_C$, there will be exactly one block, the *heaviest* block, that has the smallest block hash (except with negligible probability, assuming a collision-resistant hash function). The probability of a Bitcoin miner (consensus participant) finding this block is equal to their relative mining power (to the rest of the network).

Replacing `OP_BLOCKHASH`. Without an `OP_BLOCKHASH` opcode, we need to ensure that the blocks, whose hashes $\mathcal{H}(B_{start})$ and $\mathcal{H}(B_{end})$ provided as input ϕ to the $\mathsf{SNARG.Vrfy}$ function, are on the same chain as the **Assert** transaction in which the operator commits to ϕ. It is sufficient to show that B_{end} is on the longest chain. This ensures the $\mathsf{SNARG.Vrfy}$ function outputs False if the correct **PegOut** was not included in the main chain, allowing challengers to easily invalidate private fork submissions. Since start block B_{start} is hard coded during BitVM2 setup, e.g. as the Genesis block, and is certainly on the main chain, we focus our attention on h_{end} provided by the operator during runtime in the **Assert** transaction.

We solve this by ensuring that B_{end} is a block that was mined *after* the operator initiates the process to reclaim funds from BitVM2. The selection of the block must be deterministic but also such that the operator cannot easily predict or manipulate it. This is where we make use of the aforementioned *superbloks*.

Light Client Protocol. We slightly modify our SNARG setup with the modified relation $R' := \{(\phi, w) : \varPi'(\phi, w) = \mathsf{True}\}$ which is defined over $\varPi'$, see Algorithm 6, which is slightly modified from $\varPi$ Algorithm 5. For example, it takes as input a statement $\phi := (\mathcal{H}(B_{start}), \mathcal{H}(B_{end}), T_S, SB_O)$, consisting of two Block hashes, a start time T_S (as block height) and a superblock SB_O. The witness w remains the same. Similar to before, the function input is $z_0 := (\phi, \pi)$, where π is a SNARG proof π. We then insert the logic enumerated below at the start of the **Commit and Challenge Period** phase of the BitVM Bridge protocol. When emulating covenants with a committee, all transactions that spend outputs containing the CheckCovenant condition must be pre-signed during the **Setup** phase. The rest of the strawman bridge protocol remains the same. We illustrate the resulting full BitVM Bridge protocol in Fig. 7.

1. The operator initiates the reclaim process by publishing a new **KickOff** transaction (cf. Eq. 10) on Bitcoin.

$$\begin{aligned}
\textbf{KickOff} := \Big(&\mathit{input} = [\ast],\\
&\mathit{witness} = [(\sigma_O \wedge \ast)],\\
&\mathit{output} = [\\
&(d\text{₿}, \langle(\mathsf{RelTimelock}(\varDelta_C) \wedge \mathsf{CheckCovenant} \wedge \mathsf{CheckLampComm}_{SB_O}),\\
&(\mathsf{RelTimelock}(\varDelta_L) \wedge \mathsf{CheckCovenant}),\\
&(\mathsf{RelTimelock}(\varDelta_C + \varDelta_L) \wedge \mathsf{CheckCovenant})\rangle),\\
&(0\text{₿}, \langle(\mathsf{RelTimelock}(\varDelta_L) \wedge \mathsf{CheckCovenant}),\\
&(\mathsf{CheckSig}_{\mathsf{pk}_O} \wedge \mathsf{CheckLampComm}_{\mathsf{pk}_{T_S}} \wedge \mathsf{AbsTimelock}(T_S))\rangle)\\
&]\Big),
\end{aligned} \tag{10}$$

2. Immediately, the operator must publish the **StartTime** transaction, committing to the current time T_S.

$$\begin{aligned}\textbf{StartTime} := \Big(&\mathit{input} = [(\textbf{KickOff}, 1, \langle(\mathsf{CheckSig}_{\mathsf{pk}_O} \wedge \mathsf{CheckLampComm}_{\mathsf{pk}_{T_S}} \wedge \mathsf{AbsTimelock}(T_S))\rangle)],\\ &\mathit{witness} = [(\sigma_O, T_S, c_{T_S})],\\ &\mathit{output} = [(0\text{₿}, \mathsf{False})]\Big),\end{aligned} \tag{11}$$

This time T_S marks the start of a *superblock measurement* period that lasts for the period $\mathit{\Delta}_C$ (e.g. 2000 blocks), during which the operator must observe all blocks on the main chain and identify the *heaviest superblock* SB.

3. The two connector outputs of the **KickOff** transaction can be spent in three ways.

3.a After $\mathit{\Delta}_L$ if the operator has not published the **StartTime** transaction. A challenger then spends both outputs via the **TimeTimeout** transaction (cf. Eq 12). This invalidates the **Claim** transaction and ensures the operator cannot pre-mine a private chain ahead of the measurement.

$$\begin{aligned}\textbf{TimeTimeout} := \Big(&\mathit{input} = [(\textbf{KickOff}, 0, \langle\mathsf{RelTimelock}(\mathit{\Delta}_L) \wedge \mathsf{CheckCovenant}\rangle),\\ &(\textbf{KickOff}, 1, \langle\mathsf{RelTimelock}(\mathit{\Delta}_L) \wedge \mathsf{CheckCovenant}\rangle)],\\ &\mathit{witness} = [(\mathsf{Covenant}), (\mathsf{Covenant})],\\ &\mathit{output} = [(b\text{₿}, \mathsf{False}), (a\text{₿}, \mathsf{True})]\Big),\end{aligned} \tag{12}$$

3.b After $\mathit{\Delta}_C$ but before $\mathit{\Delta}_C + \mathit{\Delta}_L$ has passed. Assuming the operator correctly published the **StartTime** transaction, they now publish the **ClaimLC** transaction (cf. Eq. 13, a modified version of **Claim**), where they commit to what they believe to be the heaviest superblock SB_O.

$$\begin{aligned}\textbf{ClaimLC} := \Big(&\mathit{input} = [(\textbf{KickOff}, 0, \langle\mathsf{RelTimelock}(\mathit{\Delta}_C) \wedge \mathsf{CheckCovenant} \wedge \mathsf{CheckLampComm}_{SB_O}\rangle)],\\ &\mathit{witness} = [(\mathsf{Covenant}, SB_O, c_{SB_O})],\\ &\mathit{output} = [(d\text{₿}, \langle\mathsf{RelTimelock}(\mathit{\Delta}_B) \wedge \mathsf{CheckCovenant}, \mathsf{AssertScript},\\ &\mathsf{CheckLampComm}_{\mathsf{pk}_{SB_O}} \wedge \mathsf{CheckLampComm}_{\mathsf{pk}_{T_S}} \wedge SB_O.\mathit{weight} < SB_V.\mathit{weight} \wedge\\ &SB_V.\mathit{time} > T_S \vee SB_V.\mathit{time} < T_S + \mathit{\Delta}_C)\rangle), (0\text{₿}, \mathsf{CheckSig}_{\mathsf{pk}_O})]\Big),\end{aligned} \tag{13}$$

3.c After $\mathit{\Delta}_C + \mathit{\Delta}_L$ if the operator has not published the **ClaimLC** transaction. A challenger then spends the first output via the **ClaimTimeout** transaction (cf. Eq. 14). This invalidates the **ClaimLC** transaction and ensures that a malicious operator has at most the same amount of time $\mathit{\Delta}_C$ (plus liveness parameter $\mathit{\Delta}_L$[14]) as the honest miners to attempt mining a private fork.

$$\begin{aligned}\textbf{ClaimTimeout} := \Big(&\mathit{input} = [(\textbf{KickOff}, 0, \langle\mathsf{RelTimelock}(\mathit{\Delta}_C + \mathit{\Delta}_L) \wedge \mathsf{CheckCovenant}\rangle)],\\ &\mathit{witness} = [(\mathsf{Covenant})],\\ &\mathit{output} = [(b\text{₿}, \mathsf{False}), (a\text{₿}, \mathsf{True})]\Big),\end{aligned} \tag{14}$$

4. Once the **ClaimLC** transaction has been included in the Bitcoin blockchain, the challengers review the superblock SB_O and compare it against their observed heaviest superblock SB_V. Any challenger can then dispute the operator's claim using the **DisputeChain** transaction if $SB_O \neq SB_V$ and $SB_O.\mathit{weight} <$

[14] Recall that $\mathit{\Delta}_L$ is the liveness parameter, i.e., the upper bound on the time it takes an honest actor to include a transaction in the underlying blockchain. This delay ensures that an honest operator has sufficient time to include the **ClaimLC** transaction in a block after the superblock measurement period $\mathit{\Delta}_C$ has passed.

$SB_V.weight$, i.e., the operator committed to a different block with a lower weight than the heaviest superblock on the Bitcoin main chain mined during period Δ_C. To protect honest operators from malicious challengers attempting incorrect disputes, a challenger can only publish the **DisputeChain** transaction if they provide a block SB_V that was mined between T_S and after $T_S + \Delta_C$.

$$\begin{aligned}\textbf{DisproveChain} := \Big(&\mathit{input} = [(\textbf{Claim}, 0, \langle \mathsf{CheckLampComm}_{\mathsf{pk}_{SB_O}} \wedge \mathsf{CheckLampComm}_{\mathsf{pk}_{T_S}} \wedge \\ &SB_O.\mathit{weight} < SB_V.\mathit{weight} \wedge SB_V.\mathit{time} > T_S \vee SB_V.\mathit{time} < T_S + \Delta_C)\rangle)], \\ &\mathit{witness} = [(\mathsf{Covenant}, SB_V, SB_O, c_{SB_O}, T_S, c_{T_S})], \\ &\mathit{output} = [(b₿, \mathsf{False}), (a₿, \mathsf{True})]\Big),\end{aligned} \tag{15}$$

5. If the challengers do not dispute the superblock commitment, we proceed down the usual protocol path, i.e., challengers can challenge the bridge execution by publishing the **Challenge** transaction or the operator finalizes claiming funds from BitVM2 after Δ_B using transaction **PayoutOptimistic**. We note that the input $z_0 := (\phi, \pi)$ to the SNARG verifier consists of the statement and proof, where $\phi := (\mathcal{H}(B_{start}), \mathcal{H}(B_{end}), T_S, SB_O)$ contains T_S and SB_O. The operator commits to z_0' in the **Assert** transaction, which is the same as z_0, but missing T_S and SB_O. $\mathsf{DisproveScript}_1$ is slightly modified: It checks the Lamport commitments for the public keys for z_0', SB, and T_S. Then, z_0 is reconstructed from z_0', SB and T_S, before executing $f_1(z_0)$.

To attack this light client protocol, a malicious operator would have to mine a longer chain than the main Bitcoin blockchain during the period $\Delta_C + \Delta_L$ and find the heaviest superblock. The probability of success scales with the hashrate of the operator. The longer the measurement period Δ_C, the more precise the success probability is due to reduction in variance. The intuition is that this offers sufficient economic security to discourage an attacker as they would have to waste of lot of hashrate on a fork that is not part of the main chain thereby forgoing their block rewards.

Dealing with Bitcoin Difficulty Adjustments For simplicity, our PowPV light client design currently assumes a constant Bitcoin difficulty target $\mathcal{T}$. We note that Π' can easily be modified to encode the complex difficulty readjustment for the blockchain w. The crucial part for security is the difficulty of blocks with heights between T_S and $T_S + \Delta_C$. First of all, it is advisable for operators that T_S is chosen such that there is no difficulty change between T_S and $T_S + \Delta_C$. Secondly, we cannot know the difficulty in the period between T_S and $T_S + \Delta_C$ upfront, as during setup time we only know the difficulty target at that time $\mathcal{T}_{setup}$.

To deal with this, we can hard-code $\mathcal{T}_{setup}$ (perhaps scaled by some constant in $\mathbb{R}_{\geqslant 0}$) in Π' for the blocks with heights between T_S and $T_S + \Delta_C$. Choosing it higher or lower is always a trade-off between safety (a lower difficulty makes it easier for operators to fake a proof) and liveness (a difficulty higher than the actual difficulty makes it impossible for operators to complete the withdrawal until the difficulty is high enough again). We are working on a more extensive version of this paper, where we propose (i) a way of estimating a future difficulty based on historic difficulty changes and the time difference between the time of the setup and T_S, and (ii) a way of reading the difficulty from the suberblock SB.

7 Analysis

Note that this is a work in progress, and by no means a formal analysis. Instead, for this draft we content ourselves with outlining the type of analysis and proof strategy we are currently working on for a more extensive version of this paper.

Under our threat model, there are two main cases where either the operator or the challenger may deviate from the protocol specification. Our analysis will mainly focus on the case where operators and challengers have access to computational power, i.e., they are either miners or collude with miners, as proving security

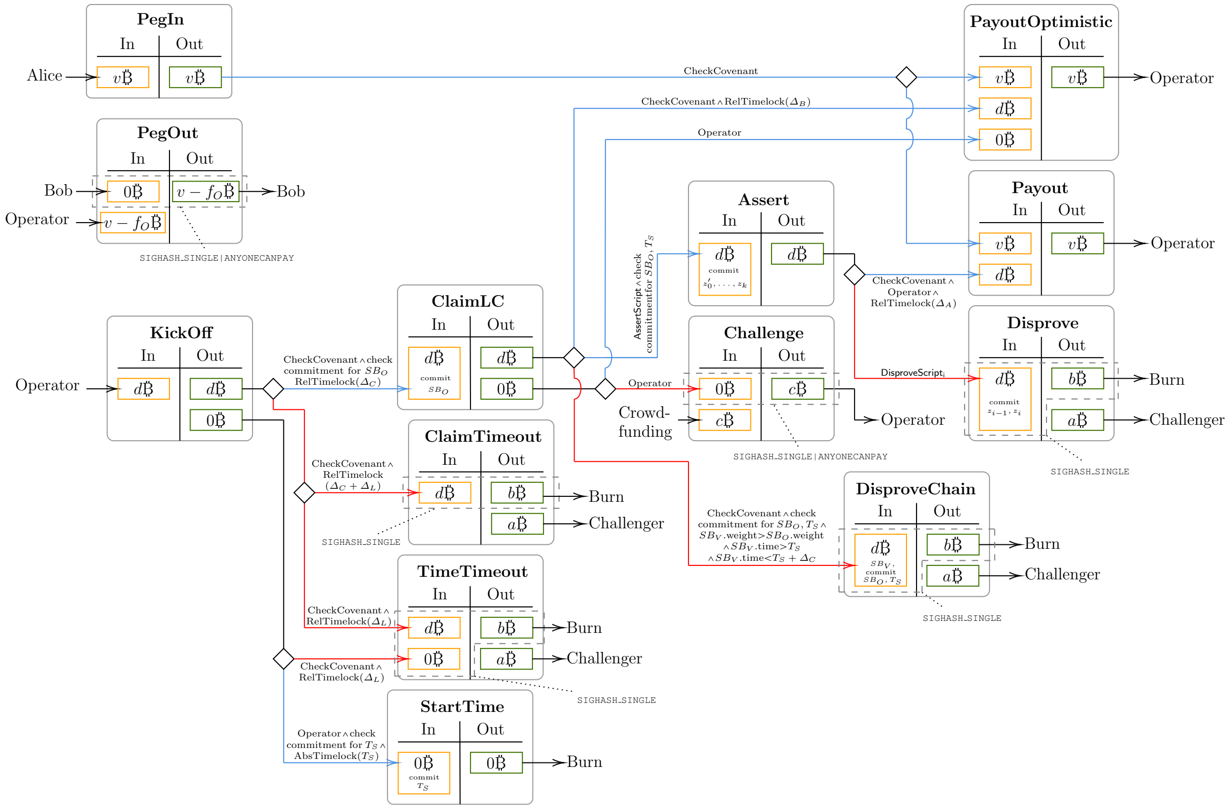

Fig. 7. Transactions of the BitVM Bridge protocol. The transactions **ClaimLC**, **PayoutOptimistic**, **Assert**, **Challenge**, **Payout**, and **Disprove** are essentially the same as Fig. 3. Together with the **KickOff**, **StartTime**, **TimeTimeout**, **ClaimTimeout**, and **DisproveChain**, they make up the PowPV light client, i.e., the component shown as a black-box in Fig. 5. The input $z_0 := (\phi, \pi)$ consists of the statement and proof, where $\phi := (\mathcal{H}(B_{start}), \mathcal{H}(B_{end}), T_S, SB_O)$ contains T_S and SB_O. z'_0, which the operator commits to in the **Assert** transaction, is the same as z_0, but missing T_S and SB_O. $\mathsf{DisproveScript}_1$ is slightly modified: It checks the Lamport commitments for the public keys for z'_0, SB, and T_S. Then, z_0 is reconstructed from z'_0, SB and T_S, before executing $f_1(z_0)$.

when either has mining power is straightforward. We can disregard the attack where a malicious user, Bob (colluding with an operator), tries to withdraw the coins of a BitVM2 instance via a **Payout** transaction without posting the proper **Burn** transaction on the sidesystem. This is because we assume that the state of the sidesystem is inscribed in the Bitcoin blockchain and, as such, would result in a failed execution of the SNARG.Vrfy function, which in turn will lead to a challenge.

Deviating operator. An operator can try and steal the money v if they manage to publish either the **Payout** or **PayoutOptimistic** transaction without posting the **PegOut** transaction (which would give the money to Bob). We argue here that a rational challenger would prevent an operator from posting **PayoutOptimistic** by posting the **Challenge** transaction. Parties that are interested in this are Bob (who loses money otherwise) and other users who have staked money in the bridge. This leaves the operator trying to publish **Payout**, which is only possible if no challenger posts a **Disprove** transaction, i.e., the operator is able to post the Assert transaction without being challenged, even though they did not post the **PegOut** transaction.

Without having `OP_BLOCKHASH` at our disposal, the light client we introduced in Section 6.2 brings some risk. Specifically, the operator has to (i) find the heaviest block SB_O in the period between T_S and $T_S + \Delta_C$, and (ii) mine as many blocks of sufficient difficulty as fall in the same period. In the full version, we will

Algorithm 6 The modified chainstate proof Π' extending Π (cf. Algorithm 5), which defines the relation R' over which we define the SNARGs used in our PowPV construction. Here, the statement $\phi := (\mathcal{H}(B_{start}), \mathcal{H}(B_{end}), T_S, SB_O)$ consists of two Block hashes, a start time T_S (as block height) and a superblock SB_O. Here, $\mathcal{T} \in \{0,1\}^\lambda$ denotes the difficulty target of blocks. For simplicity, here, it is modeled as static. We discuss dynamic difficulty in Section 6.2.

```
function $\Pi$'($\phi$, $w$)
    $(\mathcal{H}(B_{start}), \mathcal{H}(B_{end}), T_S, SB_O) := \phi$
    for $i \in \{0, \ldots, length(w) - 1\}$ do
        if $\mathcal{H}(w[i]) > \mathcal{T}$ then
            return False
        end if
    end for
    if $length(w) < T_S + \Delta_C \vee SB_O \notin w \vee$ PegOut $\notin w[: T_S] \vee$ Burn $\notin$ SidesystemState$(w[: T_S]) \vee SB_O.height < T_S \vee SB_O.height > T_S + \Delta_C$ then
        return False
    end if
    return $\Pi((\mathcal{H}(B_{start}), \mathcal{H}(B_{end})), w)$
end function
```

quantify this risk and further provide an economic analysis, i.e., determine how much money an attacker would need to spend to carry out such an attack and compare it to how much would they earn. We will conclude our analysis by showing that a rational operator will correctly follow the protocol specification.

Deviating challenger. In principle, a challenger can grief an honest operator, i.e., burn most of the operator's deposit d and get a small fee a themselves, by posting a **DisproveChain** transaction, even though the operator has posted the **PegOut** transaction. In that case, the operator would lose additionally $v - f_o$ coins from the **PegOut** transaction. In order to pull this off, the challenger would need to generate a fake heaviest block in the period T_S and $T_S + \Delta_C$, which is heavier than the heaviest block that was honestly mined in that time. Again, we will quantify this risk and analyze how much money an attack would need to spend to carry out this attack. Here, note that the attacker can only get a very small amount a, which makes this attack mostly a griefing attack. We will conclude our analysis by showing that any rational challenger will thus follow the protocol correctly.

8 Limitations and Extensions

In this section, we discuss the practical limitations of our work, alternative approaches of specific design components such as the Bitcoin light client, as well as possible extensions.

8.1 Practical Limitations

BitVM2 makes use of several tricks and workarounds to enable optimistic verification with the existing Bitcoin Script functionality, i.e., without requiring a fork.

- **Covenant emulation via a Signer committee**. One of the limitations of BitVM2 stems from the need to emulate covenants using a signer committee as described in Section 4.2. Since we only use m-of-m

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 [15,17,18] – at the cost of increased off-chain communication and coordination effort. While the implementation of this setup is beyond the scope of this paper, we note that such setups have been successfully executed at a much larger scale, for example, Ethereum's KZG trusted setup ceremony[15] which involved over 140,000 participants, executed via a simple website.

- **Large, non-standard transactions**. As of this writing, the **Assert** and **Disprove** transactions in the BitVM2 implementation exceed the 400kB "standardness" rule, i.e., these transactions will not be relayed by non-modified Bitcoin full nodes. As a result, we cannot use the Bitcoin P2P network to deliver these transactions to miners and must establish other communication channels, ideally as direct as possible to enable timely inclusion in the Bitcoin blockchain. Currently, some mining pools offer public services to include valid but non-standard transactions.[16] We discuss how we could reduce the transaction sizes in Section 8.4.
- **Fixed deposit amounts**. By design, BitVM2 only supports fixed deposit amounts. This limits the flexibility of BitVM Bridge as users must collect the exact amount of wrapped BTC on the sidesystem to initiate a peg-out. A simple mitigation is to spread the bridge deposits across multiple BitVM2 instances of different sizes, ranging from, e.g., 0.5 BTC to 100 BTC. In practice, however, we expect that peg-in and peg-out transactions will be performed by professional users as a service, as discussed in Section 8.5.
- **Operators must front BTC during peg-out.** By design, operators must front the BTC to the withdrawing user from their own balance. The benefit of this design is that users do not have to wait for the challenge period(s) to pass and receive their BTC right away. The drawback is that this introduces a capital cost for operators considering they must maintain or be able to source sufficient BTC on Bitcoin to perform peg-outs in a timely manner. This cost must be accounted for in the operator fee f_O, or additional incentives paid by the sidesystem, such as sharing sidesystem transaction fees revenues. 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.
- **Economic light client security**. Finally, as discussed in Section 7, the PowPV light client design offers economic security but can still be attacked by non-rational adversaries with sufficient hashrate to inflict damage on the bridge.
- **Sidesystem light client complexity**. In this paper, we assumed that the sidesystem is a Bitcoin roll-up, i.e., uses the Bitcoin blockchain as its consensus by posting data commitments and verifying state transitions. BitVM2 can theoretically be used to bridge BTC to blockchains with consensus protocols completely independent from Bitcoin. The implementation complexity, thereby, is specific to each blockchain protocol. Bitcoin-related consensus protocols, such as merged mining or Stakechains [13], are likely easier to verify than completely independent Proof-of-Stake networks like Ethereum.

8.2 Using Winternitz instead of Lamport Signatures

Winternitz signatures [2] can replace Lamport signatures, which reduces the size of data commitments by about 50%.

8.3 Other Bitcoin Light Clients Approaches

A Light Client using Regular On-chain Commitments. The straightforward approach to implementing a secure light client functionality would involve requiring operators to regularly commit to the latest Bitcoin block they are aware of, using Lamport signatures that enable challenges from a challenger. However, this method is impractical due to the substantial communication overhead and the associated on-chain costs.

[15] `https://github.com/ethereum/kzg-ceremony?tab=readme-ov-file`

[16] See e.g. `https://slipstream.mara.com/`.

Using Optimistic On-chain Commitments. A slight improvement over the design of the previous paragraph is making the commitments optimistic: the operators do not have to commit to the latest Bitcoin block on-chain but rather share this data with challengers via some off-chain bulletin board (which could also be another public blockchain where data storage is cheap). Challengers can challenge the operator in case of missing data or a dispute, requiring an on-chain commitment, which in turn can be challenged, using a similar protocol as shown in Section 5.2.

However, in the worst-case scenario, a malicious challenger can force operators to commit all data on-chain if they decide to issue continuous challenges. While a possible mitigation could be to require the challengers to cover or split the operator's on-chain costs, the additional capital requirements imposed on the challengers render this scheme impractical.

8.4 Balancing Assert and Disprove Transaction Sizes

So far, we assumed that there is only a single **Assert** transaction in which the operator must commit to the intermediary states of the program execution. Given the Bitcoin block size limit, we estimate that we can split a program into maximum 960 sub-programs of 4MB each[17] using a single 4MB **Assert** transaction. In theory, it is possible to utilize n multiple **Assert** transactions in parallel which would allow us to either (i) reduce the size of each **Assert** transaction, or (ii) keep the **Assert** transaction sizes at 4MB but reduce the size of the sub-programs and hence the size of the **Disprove** transaction. Thereby, in the **Claim** transaction the operator must commit to the state of the computation at the end each of the n sub-program groups as split across **Assert** transactions. A notable drawback of this approach is that the required number of pre-signed transactions (when emulating covenants via a signer committee) increases exponentially. Specifically we would have to pre-sign 2^n **Payout** transaction to ensure an honest operator that was challenged by a malicious challenger can recover funds, no matter which of the n **Assert** transactions they had to reveal.

8.5 Fast and Flexible Bridging via Atomic Swaps

Considering the fixed deposit sizes, as well as the need to run additional software to initiate a BitVM2 deposit, we expect that peg-ins and peg-outs in BitVM Bridge will be performed by professional users as a service, potentially by operators themselves acting as market makers. In turn, we anticipate that the majority of users enter and exit the sidesystem using cross-chain swaps such as [9]. Specifically, in our BitVM Bridge protocol, Alice would not only peg-in but also handle the peg-outs. To this end, Alice maintains balances in both ₿ on Bitcoin and ₿_s on the sidesystem, offering Bob to swap in and out against a fee, and re-balancing as needed by performing peg-in and peg-out operations. We observe that this model has been implemented at scale on Ethereum and Ethereum L2s via the so-called "liquidity bridges"[18].

8.6 Rotating Operators

In our design we assume that each BitVM2 instance has a pre-defined operator set. While it is possible to add and remove operators during runtime, the implementation of a such mechanism depends on the Covenant model. In our case, where we emulate covenants via a signer committee, the m signers cannot add or remove operators later on, as this would contradict our protocol and model that assumes at least one honest signer that deletes their key. Even if such modifications were possible, for example, by employing key-evolving cryptography, our signer set is designed to be large for safety reasons. As a result, the coordination overhead may be significant and the failure rate (e.g. a signer being offline) high, possibly making ad-hoc operator rotation impractical for an existing BitVM2 instance. Instead, we can introduce a pre-defined, periodic rotation schedule embedded in the BitVM2 ruleset, i.e., the SNARG verifier. Under this model, as long as there exists one honest operator per BitVM2 instance, we could rotate operators for existing instances by

[17] https://bitvm.org/snark

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

executing a peg-out transaction that at the same time acts as a peg-in transaction for a new BitVM2 instance with a different operator set. A similar rotation design is implemented by tBTCv2[19].

The design of mechanisms for selecting operators for a BitVM2 instance is outside the scope of this paper. A design implemented across Proof-of-Stake systems, for example, is random sampling based staked tokens, e.g. BTC [13] [3]. Similarly, one could sample operators based on Proof-of-Work performed over a certain period, e.g. in sidesystems merge-mined with Bitcoin.

8.7 Consensus Changes Enhancing BitVM2

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

Having big integer arithmetic, as proposed by the *Great Script Restoration*[20], would substantially reduce the SNARK verifier's script size from gigabytes down to megabytes, which could potentially allow for SNARK verification in a single transaction, drastically simplifying the overall design.

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

Additionally, with `OP_CAT`, the SNARK verifier could be replaced with a STARK verifier, avoiding the need for a trusted setup [1] for the proof system. Furthermore, the **Assert** and **Disprove** transactions could be reduced in size improving on-chain costs. Finally, `OP_CAT` would make the emulation of `OP_BLOCKHASH` simpler and more robust, however ideally, it would be natively supported.

References

1. Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. *Cryptology ePrint Archive*, 2018.
2. Dan Boneh and Victor Shoup. A graduate course in applied cryptography. *Draft 0.6*, 2023.
3. Xinshu Dong, Orfeas Stefanos Thyfronitis Litos, Ertem Nusret Tas, David Tse, Robin Linus Woll, Lei Yang, and Mingchao Yu. Remote staking with economic safety, 2024.
4. 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.
5. Craig Gentry and Daniel Wichs. Separating succinct non-interactive arguments from all falsifiable assumptions. In *Proceedings of the Forty-Third Annual ACM Symposium on Theory of Computing*, STOC '11, page 99–108, New York, NY, USA, 2011. Association for Computing Machinery.
6. Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281–28, 04 1988. Copyright - Copyright] © 1988 Society for Industrial and Applied Mathematics; Last updated - 2023-12-04.
7. Jens Groth. On the size of pairing-based non-interactive arguments. In *Advances in Cryptology–EUROCRYPT*, pages 305–326. Springer, 2016.
8. David Harding and Mike Schmidt. Bitcoin optech: Covenants, 2024.
9. Maurice Herlihy. Atomic cross-chain swaps. arXiv:1801.09515, 2018. Accessed:2018-01-31.
10. Harry Kalodner, Steven Goldfeder, Xiaoqi Chen, S Matthew Weinberg, and Edward W Felten. Arbitrum: Scalable, private smart contracts. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1353–1370, 2018.
11. Aggelos Kiayias, Andrew Miller, and Dionysis Zindros. Non-interactive proofs of proof-of-work. In *Financial Cryptography and Data Security*, pages 505–522. Springer International Publishing, 2020.
12. Leslie Lamport. Constructing digital signatures from a one way function. Technical Report CSL-98, October 1979.
13. Robin Linus. Stakechain: A bitcoin-backed proof-of-stake. In *International Conference on Financial Cryptography and Data Security*, pages 3–14. Springer, 2022.
14. Robin Linus. BitVM: Compute anything on bitcoin, dec 2023. `https://bitvm.org/bitvm.pdf`.

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

15. 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.
16. Malte Möser, Ittay Eyal, and Emin Gün Sirer. Bitcoin covenants. In *FC '16: Proceedings of the the 20th International Conference on Financial Cryptography*, February 2016.
17. Jonas Nick, Tim Ruffing, and Yannick Seurin. Musig2: Simple two-round schnorr multi-signatures. In *Annual International Cryptology Conference*, pages 189–221. Springer, 2021.
18. 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.
19. Russell O'Connor and Marta Piekarska. Enhancing bitcoin transactions with covenants. In Michael Brenner, Kurt Rohloff, Joseph Bonneau, Andrew Miller, Peter Y.A. Ryan, Vanessa Teague, Andrea Bracciali, Massimiliano Sala, Federico Pintore, and Markus Jakobsson, editors, *Financial Cryptography and Data Security*, pages 191–198, Cham, 2017. Springer International Publishing.
20. Rusty Russell. The great script restoration, jan 2024. `https://github.com/bitcoin/bips/blob/c2f268e83031b9b67e798c5c72a1171bfc463d1f/bip-unknown-var-budget-script.mediawiki`.
21. Jason Teutsch and Christian Reitwießner. A scalable verification solution for blockchains. In *Aspects of Computation and Automata Theory with Applications*, pages 377–424. World Scientific, 2024.
22. Bitcoin Wiki. Contract: Sighash flags, sep 2023.
23. Pieter Wuille, Jonas Nick, and Anthony Towns. Bip 0341, taproot: Segwit version 1 spending rules, jan 2020.
24. 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.
25. Alexei Zamyatin, Dominik Harz, Joshua Lind, Panayiotis Panayiotou, Arthur Gervais, and William J. Knottenbelt. Xclaim: Trustless, interoperable cryptocurrency-backed assets. Cryptology ePrint Archive, Report 2018/643, 2018. `https://eprint.iacr.org/2018/643`.
26. ZeroSync. BitVM Github repository, dec 2023. `https://github.com/BitVM/BitVM`.