Cardinal: Bridging Bitcoin with Ownership Preservation

Lukas Aumayr[1], Jesus Diaz[2], Dimitar Jetchev[3], and Aggelos Kiayias[4]

[1]University of Edinburgh and Common Prefix and Input Output, Edinburgh, UK, laumayr@ed.ac.uk
[2]Input Output, Madrid, Spain, jesus.diazvico@iohk.io
[3]Microsoft and Input Output, Lausanne, Switzerland and EPFL, dimitar.jetchev@iohk.io
[4]University of Edinburgh and Input Output, Edinburgh, UK, aggelos.kiayias@ed.ac.uk

December 3, 2025

Abstract

Cross-chain bridges have emerged as key components of blockchain infrastructure, enabling digital assets to flow across chains and benefit from non-native features. While early bridge designs were fraught with brittle trust assumptions, more recent designs, even within Bitcoin's script limits, can operate in a *trust-minimized* setting. Given this ever-increasing expansion of blockchain bridge systems, a critical question regarding *custody* arises: When a user's coins cross a bridge back and forth, is there a possibility that they are substituted with other coins upon return? Naturally, if cryptocurrencies were truly fungible, this would not be a consequential event. Nevertheless, chain analytics algorithms as applied to Bitcoin, Ethereum, and other popular chains can trace coins' provenance up to their minting event, thereby highlighting the possibility that users may end up with "tainted" coins upon receiving their coins from a peg-out bridge operation. Similarly, this also highlights the possibility that bridges could become an attraction for money laundering.

To address these considerations, we put forward the concept of *ownership preservation* for blockchain bridges and we observe that existing multi-sig and BitVM bridges fail to satisfy it. We then present a novel BitVM-based bridge that enables Bitcoin to connect bidirectionally with another DeFi supporting chain in an ownership-preserving and trust-minimized manner. We also observe that our ownership-preserving design is the first Bitcoin bridge to facilitate the transfer of Bitcoin NFTs, Ordinals, across chains, extending in this way their potential value and use cases.

Contents

1 Introduction **3**

2 Related Work **5**

3 Preliminaries **7**
3.1 Modeling Framework 7
3.2 Distributed Ledgers, Decentralized Apps and Pegged Ledgers 7
3.3 Cryptographic Building Blocks 9
3.4 (E)UTxO transaction syntax 10
3.5 Definitions for Sidechains Systems 10

4 Definitions **12**
4.1 ChainVM 12
4.2 Ownership-Preserving Validity Language 13

5 Our Pegged Ledgers Construction **14**
5.1 Defining the Asset 14
5.2 The Pegging Protocol 16
5.3 Protocol Transactions 18
5.4 Computation Run Inside ChainVM 21
5.5 Peg-In and Peg-Out Protocols 22

6 Security **23**

7 Discussion **24**
7.1 Improvements for Cardinal 24
7.2 Vanilla BitVM bridges are not ownership preserving 25

8 Conclusion **25**

A Deferred proofs **29**
A.1 Merge, effect, and asset validity proofs 29

B Full proof of pegging security **30**
B.1 Firewall Property During $\mathcal{L}_D$ Failure 32
B.2 Firewall Property During $\mathcal{L}_S$ Failure 34
B.3 Firewall Property 36

C BitVMX instantiation of ChainVM **37**
C.1 BitVMX blockchain protocols 41

1 Introduction

Blockchains minimize trust through decentralization and have expanded far beyond financial use cases. Following Bitcoin's success, many blockchains emerged with different tradeoffs in decentralization, privacy, and expressiveness. As ecosystems grow, the ability to move assets across chains—via cross-chain bridges—has become crucial. While expressive platforms like Ethereum host most bridges, Bitcoin has been constrained by limited scripting, leaving existing bridges reliant on trusted intermediaries. Recent advances like BitVM [6, 28, 26, 27, 19] are changing this, enabling trust-minimized Bitcoin bridges and potentially unlocking over 2 trillion USD [3] for DeFi across sidechains, rollups, and other blockchains.

Blockchain bridges landscape. Many bridges rely on threshold or multisignatures [9, 37, 16, 32, 7], where small operator sets jointly approve transfers. Security is bolstered by requiring collateral, but this raises costs and limits operator participation, weakening honest-majority assumptions—as seen in the Ronin [9] and Harmony [16] hacks, which lost over 700M USD. Trustless alternatives use light clients [24, 31] or SNARKs [39], but these are expensive and require smart contracts, making them unsuitable for Bitcoin. BitVM offers a middle ground, simulating SNARK verification on Bitcoin, though with drawbacks: long disputes, costly introspection, and the need for operators to temporarily front withdrawals.

Fungibility vs. chain of custody. Even though cryptocurrencies are usually considered fungible, the transparency and traceability enabled by their respective ledgers make it possible to distinguish and even track the provenance of any individual digital coin up to the smallest possible denomination. This lack of fungibility can have important repercussions for end users. In fact, for cryptocurrencies like Bitcoin, Ethereum, Cardano, Solana, and others, it is possible to demonstrate a simple but powerful *chain of custody* property where any transfer of a certain digital coin from one address to another takes place if and only if it can be explained by a transaction that is digitally signed by the spending address. This property is exactly what facilitates chain analysis algorithms that trace stolen funds.

Can blockchain services ignore the tainted coins dilemma? A quick search returns many cases in which blockchain analysis tools using the aforementioned principle play a core role in subsequent legal requirements to exchanges or custodial wallets to seize or freeze coins involved in illegitimate activities. For instance, in *United States v Sterlingov*[1], a US Court ruled as valid evidence the results obtained via Chainalysis to determine that specific bitcoins where linked to illegitimate behavior. In the UK, in the *Jones v Persons Unknown* case[2], Huobi was required to return some of the bitcoins stolen from Mr Jones, after they were associated to the parties that stole them originally. Huobi was similarly required to freeze part of the funds stolen in the Harmony bridge hack[3]. It therefore seems reasonable to expect that this practice of tracing coins related to illegitimate activities will become a de facto process for service providers at different layers of the blockchain ecosystem, to avoid regulatory issues – similar to what happened in Web2, where service providers now routinely block or flag requests originating from IP addresses related to anonymizing networks or VPNs, and advertise it as an added-value service[4]. In fact, services

[1] `https://assetforfeiturelaw.us/wp-content/uploads/2024/03/United-States-v-Sterlingov.pdf` (last access, 2025/11/11).

[2] `https://www.hsfkramer.com/notes/cf/2022-12/high-court-orders-delivery-up-of-stolen-bitcoin-against-crypto-exchange` (last access, 2025/11/11).

[3] `https://www.elliptic.co/blog/analysis/elliptic-collaborates-with-binance-and-huobi-to-freeze-lazarus-group-hack-proceeds` (last access, 2025/11/11).

[4] See, AWS `https://docs.aws.amazon.com/waf/latest/developerguide/aws-managed-rule-groups-ip-rep.html`,

such as `https://explorer.money/blacklists` already exist, that directly and explicitly give a list of blockchain addresses related to illegitimate activities. Moreover, regulations are already in place hinting towards this as a legal requirement: US CFR 594.201[5], prohibiting transactions involving property owned by entities in OFAC's Specially Designated Nationals list – which includes many examples of Bitcoin addresses[6], or EU's 2023/1113 regulation, by means of which authorities can request the freezing or sequestration of crypto-assets when there is suspicion of illegitimate activities[7]. Likely as a consequence, exchanges such as Coinbase state in their EULAs that they reserve the right to suspend accounts and freeze assets when a *prohibited use* is detected, and this includes "Transactions involving proceeds of any unlawful activity" [2].

Ownership preservation and blockchain bridges. The above considerations highlight the crucial question of what happens to the ownership status of digital assets when they cross a bridge to another platform and back. For illustration purposes, let us consider a frequently quoted use case for Bitcoin bridges: a Bitcoin holder transfers her coins to a chain that supports DeFi, such as Ethereum or Cardano, in order to borrow USD stablecoins using her Bitcoin as collateral. At a later point, she pays the loan back and transfers her coins back to the Bitcoin blockchain to continue her self-custody. *Is it possible she may end up with different coins compared to the ones she originally held?* Examining the details of existing multi-sig and BitVM bridges, one can see that this is entirely possible, as bridge operators quite often front the asset(s) being swapped, meaning that the user ends up receiving an asset that is not the same that was locked at first during peg-in. That is, existing bridge designs for Bitcoin are not *ownership-preserving*. As stated, this can be a significant disincentive for utilizing a bridge, as users may find themselves suddenly with "tainted" coins. Equivalently, one can see that bridges without ownership preservation can potentially become tools for money laundering.

Ownership preservation and tracing. It is worth noting that whether a protocol is ownership preserving or not is orthogonal to coin tracing, which is possible irrespectively of ownership preservation. Specifically, in the context of cross-chain bridges, ownership preservation is orthogonal to "intra-chain" misuses. Namely, once the asset has been bridged over, the recipient may dispose of it or use it for illegitimate purposes before selling it to another (potentially naive) user. What ownership preservation ensures here is that bridge operations are entirely *neutral* from the perspective of chain of custody: a user may end up with tainted coins, if they are not careful, but such an event will never be due to the user's coins crossing a bridge.

Our contributions. We formalize the concept of ownership-preserving blockchain bridges and design a secure Bitcoin bridge that is ownership-preserving. We choose Cardano as a target system, which also follows the UTxO model, albeit enables smart contracts; furthermore, it is easy to adapt our techniques for other systems such as Ethereum or Solana. Beyond the benefits of ownership preservation in the context of preventing tainting of Bitcoin and money laundering, another useful application of our Bridge design is that it enables Ordinal NFTs from Bitcoin to transfer to the target system and back—a feature that was, prior to our work, impossible with previous trust-minimized bridges.

In more detail, we base our formal chain pegging framework on the one proposed in [23] and we

Cloudflare `https://developers.cloudflare.com/waf/tools/lists/managed-lists`, or Google Cloud Armor `cloud.google.com/armor/docs/threat-intelligence` (last access, 2025/11/11).

[5] `https://www.ecfr.gov/current/title-31/subtitle-B/chapter-V/part-594/subpart-B/section-594.201` (last access, 2025/11/11).

[6] See, for instance `https://ofac.treasury.gov/recent-actions/20220506` (last access 2025/11/11).

[7] `https://eur-lex.europa.eu/legal-content/EN/TXT/HTML/?uri=OJ:L:2023:150:FULL` (last access, 2025/11/11).

extend it with ownership-preserving validity languages. We also introduce ChainVM, an abstraction of BitVM, that may be of independent interest as a general abstraction for BitVM-based designs that makes it feasible to describe our construction in a way that is compatible with different BitVM instantiations. Our design leverages Hashed TimeLock Contracts (HTLCs), the Transfer of Ownership Protocol (TOOP) [21], and ChainVM to avoid reliance on both fronted tokens and introspection. Our bridge requires all n operators to be online during the peg-in setup for *liveness*. This is a light assumption because the setup can simply be restarted with a fresh operator set if anyone is unresponsive. In contrast, *safety* for both peg-in and peg-out holds under a minimal 1-out-of-n honesty assumption. This makes our construction close to a fully trustless design and well-suited for Bitcoin's scripting limits.

Outline. The remainder of the paper is organized as follows. In Section 2, we start by comparing the Cardinal bridge with other similar cross-chain bridges. We then give an introduction to our modeling framework in Section 3, including the abstractions of distributed ledgers and sidechain systems we build on, as well as Cardinal's cryptographic building blocks. In Section 4 we formally define our ChainVM notion as a generalization of BitVM, and formalize the ownership preservation property. In Section 5, we first frame our cross-chain protocol under the proof-of-stake sidechains model, and then give a detailed definition of the Cardinal protocol. In Section 6 we state the main theorems proving security of Cardinal – in particular, we give a full proof of Cardinal's ownership preservation property, and give intuition for the remaining proofs, which are deferred to Section A and Section B. In Section 7, we first give some intuition on potential improvements to Cardinal, and then argue in more detail why other bridges built on BitVM are not ownership preserving. We conclude in Section 8 with some final remarks. Section A and Section B contain full proofs deferred for brevity, and Section C gives details on a concrete instantiation of ChainVM using BitVMX [26].

2 Related Work

A wide range of work has been done on cross-chain bridges. The first such bridges were custodial, i.e., users had to relinquish custody of the assets. These exchanges existed as early as 2010, but there have been countless incidents of hacks, asset seizures, or shutdowns, e.g., Mt Gox [5].

Threshold bridges. In the spirit of blockchain and the removal of trusted third parties, a first approach was to split custody across multiple parties, often called *operators*, in what is known as a multisignature or threshold bridge, e.g., [9, 37, 16, 32, 7]. In such a bridge, users deposit their source-chain assets in a k-of-n multisignature address owned by the operators. These operators are responsible for releasing funds on the destination chain and, similarly, for rereleasing the initial coins on the way out. To succeed, at least k operators must sign such transactions, which means we have a k-of-n honesty assumption on the operators for liveness. Similarly, we require at least $n - k + 1$ honest operators for safety, to ensure there are not more than $n - (n - k + 1) = k - 1$ malicious operators, which means that they cannot steal from the bridge.

Atomic swaps. A different approach surfaced in 2013, when atomic swaps were proposed [34]. This family of 2-party protocols relies on Hashed TimeLock Contracts (HTLCs), adaptor signatures, or similar to ensure that the swap is atomic. An HTLC is a simple contract that can be spent by one party knowing a pre-image, or the other party, after some timeout. In essence, both parties lock up assets in an HTLC on their respective chains conditioned on the same preimage. The timelocks are chosen to ensure that there is enough time for either party to refund the assets if the swap fails. Many different approaches exist, e.g., [38, 15, 30]. Since it is a 2-party protocol, it does

not require operators or external trust assumptions. However, this approach is very limited in its expressiveness; it requires users to find market-maker counterparties to swap with, and it enables only simple asset swaps, not more sophisticated bridge applications, such as wrapped assets.

Collateralized bridges. Operator-based bridges can remove or minimize trust assumptions by requiring operators to post collateral, which can be used to slash misbehaving operators and compensate victims. However, this comes at a cost: the operators need to lock up significant amounts of capital, up to the full volume of the bridge, which makes designs such as [40] prohibitively expensive in practice.

Light-client and SNARK-based bridges. These bridges allow trustless designs [24, 18, 31, 39], at least for safety. For liveness, they still require existential honesty for relaying messages between chains. As already mentioned, these rely on expressive smart contracts and are believed to be infeasible on Bitcoin.

BitVM bridges. Recently, the introduction of BitVM [6] has attracted attention in both research and industry. BitVM is a primitive that allows executing programs on Bitcoin, such as SNARK verifiers, optimistically. This allows us to build SNARK-based bridges on Bitcoin, which are trust-minimized, i.e., they require a single honest operator to be safe and live. This existential honesty, rather than full trustlessness, is an artifact of Bitcoin's lack of covenant support.

Several bridge proposals exist, such as BitVM1 bridge [6], BitVM2 bridge [28], and Clementine [8], which have different trade-offs: they differ in deposit requirements and use different BitVM constructions ([6] and [28]), for their SNARK verifiers, which again have different trade-offs. All of these designs require operators to front capital to facilitate peg-outs and then reimburse themselves, which leads to high opportunity costs, making the bridge more expensive and additionally breaking ownership preservation.

This work. Our Cardinal bridge is the first Bitcoin-compatible, trust-minimized bridge that is ownership-preserving and does not require operators to front capital, thus saving on opportunity costs. It is formally proven secure and allows advanced applications such as wrapped assets. Due to our ChainVM abstraction, our bridge is compatible and can be instantiated with any BitVM primitive. We call this being BitVM-agnostic. Bridge operators can thus choose whichever BitVM primitive they wish to use, e.g., [6, 28, 26, 27, 19], based on their own preferences on trade-offs associated with different BitVM primitives.

Table 1 summarizes the Bitcoin-compatible approaches listed in this section, comparing across the following dimensions: peg-out trust assumptions, ownership preservation, capital requirements for deposits and or capital fronting, being BitVM agnostic, applications (simple asset swaps or more sophisticated bridge applications), and whether there exist security proofs for these bridges. A different taxonomy of Bitcoin cross-chain bridges can be found in [36]. We highlight some clarifying points regarding Table 1 below.

Peg-in liveness. All BitVM bridges, including ours, nominally require n-out-of-n peg-in liveness for a single attempt. This is not problematic: a failed peg-in produces no state change and locks no funds. More importantly, the peg-in can be re-run instantly, dropping non-responsive operators. Hence, this single attempt peg-in liveness does not introduce any operational limitation.

Ownership preservation. For both threshold and XClaim bridges, we state that they are not ownership-preserving. While one could, in principle, modify these constructions to forward specific UTxOs rather than other pooled funds, existing designs do not do this, nor do they provide a mechanism that guarantees that peg-outs return the same coins originally locked. In these bridges, operators ultimately decide which coins are released during peg-out.

Table 1: Comparison of current Bitcoin-compatible bridges. For peg-out trust, "a-of-n / b-of-n" denotes a bridge imposing an "a out of n" trust assumption for safety, and "b out of n" liveness; n denotes the number of operators involved in the bridge. * These constructions do not support ownership preservation when implemented naively, but can likely easily be extended to support it. † Clementine requires both operator deposits and capital fronting. Since they optimize deposits for reuse across operations, we list only fronting. ‡ While we are not aware of formal security proofs for such bridges, proving them would be straightforward.

Bridge	Peg-out Trust (Safety / Liveness)	Ownership Preservation	Deposits / Fronting	BitVM Agnostic	Applications	Proven Security
Atomic Swaps (e.g., [34])	None	Yes	None	n/a	Simple	Yes
Threshold (e.g., [7, 32]	$(n-k+1)$-of-n / k-of-n	No*	Deposits	n/a	Advanced	Yes‡
XClaim [40]	None	No*	Both	n/a	Advanced	Informal
BitVM bridge [6]	1-of-n / 1-of-n	No	Both	No (uses [6])	Advanced	Informal
BitVM2 bridge [28]	1-of-n / 1-of-n	No	Both	No (uses [28])	Advanced	Yes
Clementine [8]	1-of-n / 1-of-n	No	Fronting†	No (uses [28])	Advanced	Informal
Cardinal (this work)	1-of-n / 1-of-n	**Yes**	**Deposits**	**Yes**	**Advanced**	**Yes**

Bitcoin agnosticity. Making constructions that rely on BitVM be BitVM-agnostic is not trivial, as it requires a significant effort of abstraction if security is to be formally proved.

3 Preliminaries

3.1 Modeling Framework

We build on the modeling approach of [22, 23], and follow a UC-style formulation of real-world executions [12]. Time advances in discrete rounds (or slots), with communication (semi-)synchronously via a diffusion of messages, where clock drift is absorbed into the round length to ensure approximate synchronization.

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

3.2 Distributed Ledgers, Decentralized Apps and Pegged Ledgers

A blockchain is a sequence of blocks, each referencing its predecessor via a hash pointer, forming a tamper-evident chain. A distributed ledger lets parties maintain a consistent global state by exchanging transactions recorded in these structures. Parties storing the full history with read/write access are called full nodes.

Definition 1 (Distributed ledger full node)**.** *A distributed ledger $\mathcal{L}$ is maintained by full nodes. Each (honest) full node P exposes the following methods:*

- $\mathsf{read}()$*: At round r, returns a finite sequence $\mathcal{L}_r^P$ of transactions.*
- $\mathsf{submit}(\mathtt{tx})$*: Takes as input a transaction $\mathtt{tx}$ at round r and attempts to include it in the ledger.*

Let $\preceq$ denote the prefix relation, that is, if $\mathcal{L}_1$ and $\mathcal{L}_2$ are two ledgers, we write $\mathcal{L}_1 \preceq \mathcal{L}_2$ if the blocks of $\mathcal{L}_1$ appear in order at the beginning of $\mathcal{L}_2$. Given a ledger $\mathcal{L}$, a party P and a round r, we denote by $\mathcal{L}_r^P$ the local view of $\mathcal{L}$ as observed by P at round r. For a ledger $\mathcal{L}$ that satisfies persistence at time t, we write $\mathcal{L}^{\cup}[t]$ (resp. $\mathcal{L}^{\cap}[t]$) for the sequence of transactions that are considered included in $\mathcal{L}$ by at least one (respectively, all) honest parties. A robust distributed ledger has the following properties:

Definition 2 (Ledger Safety (Persistence))**.** *A distributed ledger protocol is* safe *if for all honest parties P_1, P_2 and rounds $r_1 < r_2$, we have $\mathcal{L}_{r_1}^{P_1} \preceq \mathcal{L}_{r_2}^{P_2}$.*

Definition 3 (Ledger Liveness)**.** *A distributed ledger protocol protocol is* live *with parameter u if, whenever an honest party P submits a transaction $\mathtt{tx}$ to the ledger at round r, then $\mathtt{tx} \in \mathcal{L}_{r+u}^P$.*

Definition 4 (Decentralized Protocol)**.** *A* decentralized protocol $\Pi_{\mathcal{L}}$ *is parameterized by a distributed ledger $\mathcal{L}$ and is executed by a full node P of $\mathcal{L}$. P interacts with both:*

- *the* environment $\mathcal{Z}$*, which provides external inputs and receives outputs, and*
- *the* ledger interface*,* $\mathsf{read}()$ *and* $\mathsf{submit}(\cdot)$*, exposed by $\mathcal{L}$'s full nodes.*

Definition 5 (Decentralized Application (dApp))**.** *A* dApp *is a set of* distributed ledger protocols*.*

We closely adhere to the framework of [23]. For clarity, we first informally introduce several relevant notions. A formal treatment of the informally presented concepts is deferred to Section 3.5.

A system of pegged ledgers consists of a collection of ledgers together with auxiliary protocols that enable the transfer of assets across them. For a ledger $\mathcal{L}$, we write $\mathsf{Assets}(\mathcal{L})$ to denote the class of assets natively supported by $\mathcal{L}$. The valid operations on a given asset $A \in \mathsf{Assets}(\mathcal{L})$ are specified by its associated validity language $\mathbb{V}_A$. When analyzing transactions pertaining to a particular asset A, we employ projections of ledgers (or sets of ledgers) onto the subset of transactions relevant to A, denoted $\pi_A(\mathcal{L})$. Since a system of pegged ledgers is composed of a family of ledgers $\mathbf{L}$, we define a mapping $\mathsf{merge}(\mathbf{L})$ that yields a consolidated or "joint" ledger view over $\mathbf{L}$. We can now informally recall the notion of pegging security, formally defined in Section 3.5, Definition 14.

Definition 6 (Pegging security, informal)**.** *Informally, a system of pegged ledgers $\{\mathcal{L}_i\}_{i\in[n]}$, given a merge mapping and a validity language, is pegging-secure if, against all PPT adversaries and as long as the assumptions underlying each individual ledger hold, every ledger is persistent and live. Moreover, the ledger obtained by applying the merge mapping must adhere to the validity language, and if some ledgers fail, the* firewall *property ensures that the transactions observed on the remaining (secure) ledgers admit a plausible explanation.*

3.3 Cryptographic Building Blocks

Multisignatures. When multiple parties need to authenticate a common message, a typical approach is to use multisignatures schemes, or MS. A very popular MS scheme on Bitcoin is MuSig2 [33]. Formally, an MS scheme can be seen as a tuple $(\mathsf{Setup}, \mathsf{KG}, \mathsf{KA}, \mathsf{Sign}, \mathsf{SignA}, \mathsf{Verify})$, as follows:

- $\mathsf{param} \leftarrow \mathsf{MS.Setup}(1^\kappa)$ generates system-wide parameters.
- $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{MS.KG}(\mathsf{param})$, run by every participant to generate its key pair.
- $\sigma \leftarrow \mathsf{MS.Sign}(\mathsf{sk}, \mathsf{msg})$, run by every participant to produce an individual signature.
- $\Sigma \leftarrow \mathsf{MS.SignA}(\mathsf{msg}, \mathbf{pk}, \{\sigma_i\}_{i \in [|\mathbf{pk}|]}$ aggregates a set of individual signatures.
- $\mathsf{pk}' \leftarrow \mathsf{MS.KA}(\mathbf{pk})$ aggregates the keys in public key set $\mathbf{pk}$ into pk'.
- $b \leftarrow \mathsf{MS.Verify}(\Sigma, \mathsf{msg}, \mathsf{pk}')$ verifies an aggregated signature.

Very informally, a MS scheme is correct if an aggregated signature σ – produced by an honest set of signers running the previous functionalities honestly – is accepted by an honestly run $\mathsf{MS.Verify}$ algorithm. Also informally, a MS scheme is existentially unforgeable against chosen message attacks (EU-CMA) if no adversary can produce a forged signature σ' over any chosen message msg such that $\mathsf{MS.Verify}(\sigma', \mathsf{msg}, \mathsf{pk}') = 1$ on behalf of a set of n participants with public keys $\mathbf{pk} = \{\mathsf{pk}_1, \ldots, \mathsf{pk}_n\}$ and where $\mathsf{pk}' = \mathsf{MS.KA}(\mathbf{pk})$ as long as at least one of the participants is honest. We refer to [33] for a formal definition of multisignatures.

Asymmetric encryption. An asymmetric encryption scheme AE is a tuple of algorithms $(\mathsf{KG}, \mathsf{Enc}, \mathsf{Dec})$, as follows:

- $(\mathsf{ek}, \mathsf{dk}) \leftarrow \mathsf{AE.KG}(1^\kappa)$ computes encryption (ek) and decryption (dk) keys.
- $c \leftarrow \mathsf{AE.Enc}(\mathsf{ek}, m)$ encrypts message m under ek, producing ciphertext c.
- $m \leftarrow \mathsf{AE.Dec}(\mathsf{dk}, c)$ uses dk to decrypt ciphertext c into plaintext m.

Informally, an asymmetric encryption scheme is correct if, given $(\mathsf{ek}, \mathsf{dk}) \leftarrow \mathsf{AE.KG}(1^\kappa)$, $m = \mathsf{AE.Dec}(\mathsf{dk}, \mathsf{AE.Enc}(\mathsf{ek}, m))$, and it is IND-CPA secure if, when given two plaintexts and one ciphertext encrypting one of the two ciphertexts, no adversary –without access to a decryption oracle– can guess which plaintext was encrypted better than guessing at random. We refer to [10] for formal definitions.

SNARKs. A Succinct Non-interactive ARgument of Knowledge is a proving system that produces succinct proofs over NP statements, which are also efficient to verify. A SNARK scheme is a tuple $(\mathtt{Setup}, \mathtt{Prove}, \mathtt{Vrfy})$, defined as follows.

- $\mathsf{crs} \leftarrow \mathtt{SNARK.Setup}(\mathcal{R})$ takes as input a relation $\mathcal{R}$, and outputs a common reference string crs.
- $\pi \leftarrow \mathtt{SNARK.Prove}(\mathcal{R}, \mathsf{crs}, \phi, w)$ takes as input a common reference string crs along with $(\phi, w) \in \mathcal{R}$, and returns an argument π.

- $\{1,0\} \leftarrow \texttt{SNARK.Vrfy}(\mathcal{R}, \mathsf{crs}, \phi, \pi)$ takes as input a common reference string crs, a statement ϕ, and an argument π and returns 1 or 0, (informally) depending on whether or not π is a valid argument.

In a nutshell, a SNARK system is complete if all valid statements are provable by an honest prover and accepted by an honest verifier. It is knowledge-sound if, given a proof accepted by `Vrfy`, potentially produced by a malicious prover, a valid witness can be extracted from it. It is succinct if the length of the produced proofs is independent on the statement being proven, and can be verified in polynomial time on the security parameter. We refer to [20] for formal definitions of these properties.

Hash functions. Finally, we also require collision-resistant hash functions, where Hash receives arbitrarily long strings, and outputs a fixed-length string [17].

3.4 (E)UTxO transaction syntax

To specify (E)UTxO transactions, we follow the abstraction in [28]. A transaction in a UTxO-based blockchain is a tuple $(\mathsf{txn_id}, \mathsf{in}, \mathsf{out}, \mathsf{wit})$, where in and out are the sets of inputs consumed and outputs created, wit is a set of witnesses, and $\mathsf{txn_id}$ is a unique identifier. An output is a pair $(a_{\mathsf{coin}}, \mathsf{scr})$, with a the amount of assets of type coin and scr the conditions to spend it. In the EUTxO model [14], scr can be a smart contract, and outputs become $(a_{\mathsf{coin}}, \mathsf{scr}, \delta)$, where δ is on-chain state accessible during validation. We use both UTxO and EUTxO, but write δ only as needed. An input is a tuple $(\mathsf{prevtx}, \mathsf{index})$, uniquely identifying the prior output being spent. Finally, wit is the set of witnesses ensuring that the inputs can be spent, and their internals are as varied as the scr scripts in outputs. In this work, to specify spending conditions, we use the following scripts:

- $\mathsf{SigScr}_{\mathbf{pk}}(m)$. Requiring a signature by the owner of pk.
- $\mathsf{HTLCScr}(t, h, \mathsf{scr}_1, \mathsf{scr}_0)$. Before time t, anyone providing $\mathsf{Hash}^{-1}(h)$ and satisfying scr_1 can spend the output; after t, anyone satisfying scr_0 can spend.
- $\mathsf{MDataScr}(d)$. A script for storing a piece d of metadata inside UTxOs.
- $\mathsf{MSScr}_{\mathbf{pk}}(m)$. Requiring a multisignature by all owners of keys in $\mathbf{pk}$.

3.5 Definitions for Sidechains Systems

In this section, we recall the definitions from [23].

Definition 7 (Assets, syntactically valid transactions)**.** *For an asset A, we denote by $\mathcal{T}_A$ the valid transaction set of A, i.e., the set of all syntactically valid transactions involving A. For a ledger $\mathcal{L}$ we denote by $\mathcal{T}_{\mathcal{L}}$ the set of transactions that can be included into $\mathcal{L}$. For notational convenience, we define $\mathcal{T}_{A,\mathcal{L}} \triangleq \mathcal{T}_A \cap \mathcal{T}_{\mathcal{L}}$.*

Let $\mathsf{Assets}(\mathcal{L})$ denote the set of all assets that are supported by $\mathcal{L}$. Formally,

$$\mathsf{Assets}(\mathcal{L}) \triangleq \{A : \mathcal{T}_{A,\mathcal{L}} \neq \emptyset\}\,.$$

Definition 8 (Asset validity language)**.** *For an asset A, the asset validity language of A is any language $\mathbb{V}_A \subseteq \mathcal{T}_A^*$ that satisfies the following properties:*

1. ***Base.*** $\varepsilon \in \mathbb{V}_A$.

2. ***Monotonicity.*** *For any* $w, w' \in \mathcal{T}_A^*$, *we have*

$$w \notin \mathbb{V}_A \Rightarrow w \| w' \notin \mathbb{V}_A.$$

3. ***Uniqueness of transactions.*** *Words from* $\mathbb{V}_A$ *never contain the same transaction twice: for any* $\mathtt{tx} \in \mathcal{T}_A$ *and any* $w_1, w_2, w_3 \in \mathcal{T}_A^*$, *we have*

$$w_1 \| \mathtt{tx} \| w_2 \| \mathtt{tx} \| w_3 \notin \mathbb{V}_A.$$

Definition 9 (Ledger state projection)**.** *Given a ledger state* $\mathcal{L}$, *we call a projection of* $\mathcal{L}$ *with respect to a set* $\mathcal{X}$ *of transactions (and denote by* $\pi_{\mathcal{X}}(\mathcal{L})$*) the ledger state that is obtained from* $\mathcal{L}$ *by removing all transactions not in* $\mathcal{X}$.

To simplify notation, we will use π_A *and* $\pi_{\mathcal{I}}$ *as a shorthand for* $\pi_{\mathcal{T}_A}$ *and* $\pi_{\bigcup_{i \in \mathcal{I}} \mathcal{T}_{\mathcal{L}_i}}$, *denoting the projection of the transactions of a ledger state with respect to particular asset* A *or a particular set of individual ledger indices.*

Naturally, for a language $\mathbb{V}$, *we define the projected language* $\pi_{\mathcal{X}}(\mathbb{V}) := \{\pi_{\mathcal{X}}(w) : w \in \mathbb{V}\}$, *which contains all the sequences of transactions from the original language, each of them projected with respect to* $\mathcal{X}$.

Definition 10 (Effect Transactions)**.** *For two ledgers* $\mathcal{L}$ *and* $\mathcal{L}'$, *an effect mapping is a mapping of the form*

$$\mathsf{effect}_{\mathcal{L} \to \mathcal{L}'} : \mathcal{T}_{\mathcal{L}} \to (\mathcal{T}_{\mathcal{L}'} \cup \{\bot\}).$$

A transaction $\mathtt{tx}' = \mathsf{effect}_{\mathcal{L} \to \mathcal{L}'}(\mathtt{tx}) \neq \bot$ *is called the* effect transaction *of the transaction* $\mathtt{tx}$.

Definition 11 (Cross-Ledger Transfers)**.** *For two ledgers* $\mathcal{L}$ *and* $\mathcal{L}'$ *and an effect mapping* $\mathsf{effect}_{\mathcal{L} \to \mathcal{L}'}(\cdot)$, *we refer to a transaction in* $\mathcal{T}_{\mathcal{L}}$ *that requires some effect on* $\mathcal{L}'$ *as a* $(\mathcal{L}, \mathcal{L}')$*-cross-ledger transfer transaction (or* cross-ledger transfer *for short).*

The set of all cross-ledger transfers is denoted by $\mathcal{T}^{\mathsf{cl}}_{\mathcal{L},\mathcal{L}'} \subseteq \mathcal{T}_{\mathcal{L}}$, *formally:*

$$\mathcal{T}^{\mathsf{cl}}_{\mathcal{L},\mathcal{L}'} \triangleq \{\, \mathtt{tx} \in \mathcal{T}_{\mathcal{L}} \;:\; \mathsf{effect}_{\mathcal{L} \to \mathcal{L}'}(\mathtt{tx}) \neq \bot \}\,.$$

Definition 12 (Merging ledger states)**.** *The merge function, denoted* $\mathsf{merge}(\cdot)$, *is any mapping that takes a subset of ledger states* $\mathcal{L} \subseteq \{\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_n\}$ *and produces a ledger state* $\mathsf{merge}(\mathcal{L})$ *such that:*

1. ***Partitioning:*** *The ledger states in* $\mathcal{L}$ *are disjoint subsequences of* $\mathsf{merge}(\mathcal{L})$ *that together cover the entire sequence* $\mathsf{merge}(\mathcal{L})$.

2. ***Topological soundness:*** *For any* $i \neq j$ *such that* $\mathcal{L}_i, \mathcal{L}_j \in \mathcal{L}$ *and any two transactions* $\mathtt{tx} \in \mathcal{L}_i$ *and* $\mathtt{tx}' \in \mathcal{L}_j$, *if* $\mathtt{tx}' = \mathsf{effect}_{\mathcal{L}_i \to \mathcal{L}_j}(\mathtt{tx})$, *then* $\mathtt{tx}$ *precedes* $\mathtt{tx}'$ *in* $\mathsf{merge}(\mathcal{L})$.

Definition 13 (Correctness of $\mathbb{V}_A$)**.** *A validity language* $\mathbb{V}_A$ *is correct with respect to a mapping* $\mathsf{merge}(\cdot)$, *if for any ledger states* $\mathcal{L} \triangleq (\mathcal{L}_1, \ldots, \mathcal{L}_n)$ *such that*

$$\pi_A(\mathsf{merge}(\mathcal{L})) \in \mathbb{V}_A,$$

indices $i \neq j$, *and any cross-ledger transfer* $\mathtt{tx} \in \mathcal{L}_i \cap \mathcal{T}^{\mathsf{cl}}_{\mathcal{L}_i,\mathcal{L}_j}$ *such that* $\mathsf{effect}_{\mathcal{L}_i \to \mathcal{L}_j}(\mathtt{tx}) = \mathtt{tx}' \neq \bot$ *is not in* $\mathcal{L}_j$, *we have:*

$$\pi_A\left(\mathsf{merge}(\mathcal{L}_1, \ldots, \mathcal{L}_i, \ldots, \mathcal{L}_j \| \mathtt{tx}', \ldots, \mathcal{L}_n)\right) \in \mathbb{V}_A.$$

Definition 14 (Pegging security)**.** *A system-of-ledgers protocol* Π *for* $\{\mathcal{L}_i\}_{i\in[n]}$ *is* pegging-secure *with liveness parameter* $u \in \mathbb{N}$ *with respect to:*

- *a set of assumptions* $\mathbb{A}_i$ *for ledgers* $\{\mathcal{L}_i\}_{i\in[n]}$*,*
- *a merge mapping* $\mathsf{merge}(\cdot)$*,*
- *validity languages* $\mathbb{V}_A$ *for each* $A \in \bigcup_{i\in[n]} \mathsf{Assets}(\mathcal{L}_i)$*,*

if for all PPT adversaries, all rounds $t \in \mathbb{N}$*, and for* $S_t \triangleq \{i \in [n] : \mathbb{A}_i[t]$ *holds*$\}$*, we have that, except with negligible probability in the security parameter:*

1. ***Ledger persistence:*** *For each* $i \in S_t$*, ledger* $\mathcal{L}_i$ *satisfies the persistence property.*
2. ***Ledger liveness:*** *For each* $i \in S_t$*, ledger* $\mathcal{L}_i$ *satisfies the liveness property parametrized by* u*.*
3. ***Firewall:*** *For all* $A \in \bigcup_{i\in S_t} \mathsf{Assets}(\mathcal{L}_i)$*,* $\pi_A\left(\mathsf{merge}\left(\{\mathcal{L}_i^{\cup}[t] : i \in S_t\}\right)\right) \in \pi_{S_t}(\mathbb{V}_A)$
   Where $\mathcal{L}_i^{\cup}[t]$*, for a ledger satisfying persistence at round* t*, denotes the sequence of transactions included in* $\mathcal{L}$ *by at least one honest party.*

4 Definitions

4.1 ChainVM

The following definition, adapted from [28], captures the core functionality of ChainVM and abstracts the various existing BitVM paradigms [6, 28, 26, 27, 19]. Intuitively, a ChainVM proof system enables optimistic, arbitrary computation off-chain while allowing on-chain verification of claimed results via one or more transactions. Although ChainVM supports arbitrary computations, each requires a fresh instance, implicitly tied to a unique session identifier disambiguating concurrent executions; for clarity, we omit this identifier from the notation.

Definition 15 (ChainVM proof system)**.** *A* ChainVM proof system *is a* dApp *parameterized by a distributed ledger* $\mathcal{L}$*, a parameter* v*, and is executed by a prover* P *and multiple verifiers* $\mathcal{V}$*, who are full nodes in* $\mathcal{L}$*.* ChainVM *comprises of a tuple of* decentralized protocols $(\mathsf{Setup}, \mathsf{Execute})$ *defined as follows:*

- $\mathsf{ChainVM.Setup}(1^\kappa, Q) \to (\mathsf{sid}, S, \mathsf{aux})$: The prover, given a security parameter 1^κ and a predicate $Q\colon \{0,1\}^* \to \{0,1\}$, outputs a session id sid and predicate S over the ledger $\mathcal{L}$ of $\Pi_{\mathcal{L}}$ and some auxilliary data aux.
- $\mathsf{ChainVM.Execute}(\mathsf{sid}, \pi, \mathsf{aux}) \to r$: The prover, given a string π outputs $r \in \mathbb{N} \cup \{\perp\}$, a round or failure.

We write $X(\ldots) \downarrow_{\mathsf{view}} y$ if the invocation of a decentralized protocol $X(\ldots) \to y$ appears in view; i.e., view contains a record of $\mathcal{Z}$ invoking X with those arguments and later receiving output y. $\mu(\kappa)$ is a negligible function in κ.

Definition 16 (Completeness)**.** *A* ChainVM proof system *is* complete*, if for all PPT* $\mathcal{A}, \mathcal{Z}$ *and any* $z \in \{0,1\}^*$*,* $\Pr[\mathsf{Complete}(\mathsf{view}_{\Pi,\mathcal{A},\mathcal{Z}}(z)) = 0] \leq \mu(\kappa)$.

Algorithm 1 Complete(view)

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

Definition 17 (Soundness)**.** *A* ChainVM proof system *is* sound*, if there exists an efficient algorithm* Extract()*, such that for all PPT* $\mathcal{A}, \mathcal{Z}$ *and any* $z \in \{0,1\}^*$, $\Pr[\mathsf{Sound}(\mathsf{view}_{\Pi,\mathcal{A},\mathcal{Z}}(z)) = 0] \leq \mu(\kappa)$.

Algorithm 2 Sound(view)

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

Informally, soundness guarantees that whenever a ChainVM contract completes successfully, there exists a string π, which can be efficiently extracted from any honest verifier's view of the execution and satisfies predicate Q. We exemplify how this translates into our context in Section 5.4.

Note that on expressive blockchains such as Ethereum, a trivial way to realize ChainVM is to encode Q directly as a smart contract function. However, non-trivial realizations of ChainVM can be useful for such blockchains, by providing an *optimistic computation layer*, where the efficiency gain comes from comparing the cost of evaluating S to that of evaluating Q.

4.2 Ownership-Preserving Validity Language

Given $l \in \mathbb{N}$, let A be an asset. We assume that each token of A has an identifier $\mathsf{token_id} \in \mathsf{TokenID}_A = \{0,1\}^l$. Let $\mathcal{T}_A$ denote the set of syntactically valid transactions (cf. Definition 7) for asset A. For a validity language $\mathbb{V} \subseteq \mathcal{T}^*_A$, let $\mathsf{owner} : \mathbb{V} \times \mathsf{TokenID}_A \to \mathsf{Addresses} \cup \{\perp\}$ be a function that assigns to each transaction trace and token identifier either the current owner or $\perp$ if unassigned.

Definition 18 (Chain of Custody)**.** *A tuple* $(\mathbb{V}, \mathsf{owner})$ *consisting of a validity language* $\mathbb{V} \subseteq \mathcal{T}^*_A$ *and an* owner *function satisfies the* chain of custody *property if for all traces* $w = \mathtt{tx}_1 \| \cdots \| \mathtt{tx}_n \in \mathbb{V}$ *and all* $\mathsf{token_id} \in \mathsf{TokenID}_A$*, it holds that for each index* $i \in \{0, \ldots, n-1\}$*, if* $a := \mathsf{owner}(w[:i], \mathsf{token_id})$*,* $a' := \mathsf{owner}(w[:i+1], \mathsf{token_id})$ *are distinct, then transaction* $w[i]$ *contains a witness that satisfies the spending condition of address* a*.*

Definition 19 (Ownership Preservation)**.** *Let* $\{\mathcal{L}_i\}_{i\in[n]}$ *be the ledgers with native ownership functions* $\{\mathsf{owner}_i\}_{i\in[n]}$*, and let* ownerCombined *be a combined ownership function on the global merged trace* $W_{\mathrm{all}} = \mathsf{merge}(\{\mathcal{L}^{\cup}_i[t] : i \in [n]\})$*. Assume each* owner_i *and* ownerCombined *satisfies the chain-of-custody property. Then, for any any* $\mathsf{token_id} \in \mathsf{TokenID}_A$*, and any* $W_{\mathrm{all}}[:k]$*, we require for every* $i \in [n]$*:* $\Big(\ \mathsf{ownerCombined}\big(W_{\mathrm{all}}[:k], \mathsf{token_id}\big)$ *is an address in* $\mathcal{L}_i$ *and* $W_{\mathrm{all}}[k] \in \mathcal{L}_i\ \Big) \implies \mathsf{ownerCombined}\big(W_{\mathrm{all}}[:k], \mathsf{token_id}\big) = \mathsf{owner}_i\big(\pi_i(W_{\mathrm{all}}[:k]), \mathsf{token_id}\big)$.

Intuitively, *chain of custody* ensures every transfer is authorized by the previous owner, and *ownership preservation* requires that the global view of ownership is always consistent with each individual ledger.

5 Our Pegged Ledgers Construction

We restrict to the case of two pegged ledgers. The source ledger, $\mathcal{L}_S$, follows the UTxO model with restricted scripting capabilities, and the destination ledger, $\mathcal{L}_D$, follows the EUTxO model. Both ledgers are pre-existing and independent, hence we cannot apply *direct observation*. We give a bridge construction in this setting that satisfies the pegging security property of [23], plus the ownership preservation notion of Section 4.2. For brevity, we state the theorems that capture our construction under this framework here and defer proofs to Section A.

5.1 Defining the Asset

We define our asset language $\mathbb{V}_{\mathsf{UTxO}}$ using the generic specification of transactions given in Section 3.4. While we define concrete transactions used in our system, we do not require any changes to the transaction validation rules of the involved ledgers. Our pegged ledgers construction is defined in terms of the transaction set $\mathcal{T}_{\mathbb{V}_{\mathsf{UTxO}}} := \{\texttt{LockReqTX}, \texttt{MintTX}, \texttt{ReleaseTX}, \texttt{LockTX}, \texttt{CommitBurnTX}, \texttt{BurnTX}, \texttt{SpendTX}, \texttt{AbortTX}\}$. We specify these transactions in Section 5.3. Each of these transactions will be univocally associated with a token identifier $\mathsf{token_id}$. We assume that, given a transaction $\mathtt{tx}$, it is possible to extract the identifier of the underlying token via a function $\mathsf{token_id}/\bot \leftarrow \mathsf{get_token_id}(\mathsf{tx})$, which returns $\bot$ if tx is not associated to a (valid) token identifier.

Instantiating the asset validity language, $\mathbb{V}_{\mathsf{UTxO}}$. Algorithm 3 checks sequences of transactions, each related to a token identified by $\mathsf{get_token_id}$. Different tokens may move concurrently between the two ledgers, but validity can be verified independently per token. The algorithm partitions the trace into subsequences per $\mathsf{token_id}$, preserving order, and checks each subsequence separately. The ValidSeqTokenId procedure enforces that each subsequence follows the correct lifecycle ($\texttt{LockReqTX} \to \texttt{MintTX} \to \texttt{ReleaseTX} \to \texttt{LockTX} \to \texttt{CommitBurnTX} \to \texttt{BurnTX} \to \texttt{SpendTX}$), we define it in Algorithm 4.

Algorithm 3 Membership deciding algorithm for $\mathbb{V}_{\mathsf{UTxO}}$, ValidSeq($\mathbf{tx}$)

```
1:  function ValidSeq(tx)
2:      tx_token_id ← ε for all possible token_id; b ← true        ▷ Initial state (base property)
3:      for tx ∈ tx do                                             ▷ Create sub-sequences per token_id
4:          token_id ← get_token_id(tx); tid ← token_id ∪ {token_id}
5:          if token_id = ⊥ then return false
6:          else then tx_token_id ← tx_token_id||tx
7:      end for
8:      for token_id ∈ tid do                                      ▷ Validate each subsequence independently
9:          b ← b ∧ ValidSeqTokenId(tx_token_id)
10:     end for
11:     return b
12: end function
```

Example 1. *The Ordinals NFT system tracks each Satoshi by minting order, recording their creation and subsequent movements (FIFO within a block). It maintains a map from Satoshis to controlling UTxOs and vice versa [35]. Our bridge uses the latter:* $\mathsf{get_token_id}(\mathtt{tx})$ *returns the Satoshis linked to a transaction. On Bitcoin, this is the NFT-related UTxO in* $\mathtt{tx}$ *(first input/output in our*

Algorithm 4 Token subsequence validator algorithm

function ValidSeqTokenId(**tx**)
  $st \leftarrow \varepsilon$
  for tx $\in$ **tx do**
    if tx is of `LockReqTX` type **then**
      if $st = \varepsilon \vee st = spend \vee st = abort$ **then** $st \leftarrow lockreq$
      else then return false
    else if tx is of `MintTX` type **then**
      $\mathsf{tx}' \leftarrow \mathsf{effect}^{-1}_{\mathcal{L}_S \to \mathcal{L}_D}(\mathsf{tx})$ ▷ Mint on $\mathcal{L}_D$ requires lock request on $\mathcal{L}_S$
      if $st \neq lockreq \vee \mathsf{tx}'$ has not appeared before **then return false**
      $st \leftarrow mint$
    else if tx is of `ReleaseTX` type **then**
      if $st \neq mint$ **then return false**
      $st \leftarrow release$
    else if tx is of `LockTX` type **then**
      $\mathsf{tx}' \leftarrow \mathsf{effect}^{-1}_{\mathcal{L}_D \to \mathcal{L}_S}(\mathsf{tx})$ ▷ Lock on $\mathcal{L}_S$ requires release on $\mathcal{L}_D$
      if $st \neq release \vee \mathsf{tx}'$ has not appeared before **then return false**
      $st \leftarrow lock$
    else if tx is of `CommitBurnTX` type **then**
      if $st \neq lock$ **then return false** ▷ Initiate burn only if NFT was locked
      $st \leftarrow commitBurn$
    else if tx is of `BurnTX` type **then**
      if $st \neq commitBurn$ **then return false**
      $st \leftarrow burn$
    else if tx is of `SpendTX` type **then**
      $\mathsf{tx}' \leftarrow \mathsf{effect}^{-1}_{\mathcal{L}_D \to \mathcal{L}_S}(\mathsf{tx})$ ▷ Unlock on $\mathcal{L}_S$ requires burn on $\mathcal{L}_D$
      if $st \neq burn \vee \mathsf{tx}'$ has not appeared before **then return false**
      $st \leftarrow spend$
    else if tx is of `AbortTX` type **then**
      if $st \neq lockreq$ **then return false** ▷ Abort only if NFT was not locked
      $st \leftarrow$ abort
    end if
  end for
  return true
end function

design); on Cardano, the corresponding Satoshi range is stored in the asset's datum and returned by get_token_id.

Theorem 1. $\mathbb{V}_{\mathsf{UTxO}}$ *satisfies the* base, monotonicity, *and* uniqueness of transactions *properties as defined in Section 3.5, Definition 8.*

Instantiating the effect maps, effect-UTxO**.** For any given token_id, we define $\mathsf{effect\text{-}UTxO}_{\mathcal{L}_S \to \mathcal{L}_D}$ to map `LockReqTX` for token_id to `MintTX` of the same token_id, and any other `tx` is mapped to $\perp$. Similarly, $\mathsf{effect\text{-}UTxO}_{\mathcal{L}_D \to \mathcal{L}_S}$ maps `ReleaseTX` to `LockTX`, `BurnTX` to `SpendTX`, and any other `tx` is mapped to $\perp$, while respecting the token_id, as before. We also use $\mathsf{effect\text{-}UTxO}^{-1}_{A \to B}(\mathtt{tx})$ to denote the preimage of `tx` under the $\mathsf{effect\text{-}UTxO}_{A \to B}$ map. That is, transaction $\mathtt{tx}'$ such that

$\texttt{tx} \leftarrow \mathsf{effect\text{-}UTxO}_{A\to B}(tx')$, or $\perp$ if there is no such $\texttt{tx}'$.

Instantiating the merge map, merge-UTxO. Given the two ledgers composing our system, $\mathcal{L}_S$ and $\mathcal{L}_D$, when merging them, transactions in $\mathcal{L}_S$ precede transactions in $\mathcal{L}_D$ (lexicographic minimality; in our case, peg operations begin in $\mathcal{L}_S$), except when comparing $\texttt{tx} \in \mathcal{L}_S$ and $\texttt{tx}' \in \mathcal{L}_D$ such that $\texttt{tx} \leftarrow \mathsf{effect\text{-}UTxO}_{\mathcal{L}_D\to\mathcal{L}_S}(\texttt{tx}')$. We define merge-UTxO in Algorithm 5.

Algorithm 5 Merge mapping for $\mathbb{V}_{\mathsf{UTxO}}$, merge-UTxO

```
function merge-UTxO(tx_{L_S}, tx_{L_D})
    merged ← ε; i_{L_D} ← 1
    for i ∈ [tx_{L_S}] do
        tx ← tx_{L_S}[i]
        if ∃tx' ∈ tx_{L_D}[i_A :] st tx = effect-UTxO_{L_D→L_{L_S}}(tx') then
            Let j_{L_D} be s.t. tx' = L_D[j_{L_D}]
            merged ← merged||tx_{L_D}[i_{L_D} : j_{L_D}]
            i_{L_D} ← j_{L_D} + 1
        else
            merged ← merged||tx
        end if
    end for
    merged ← merged||tx_{L_D}[i_{L_D} :]
    return merged
end function
```

Theorem 2. merge-UTxO *satisfies* partitioning *and* topological soundness *as defined in Section 3.5, Definition 12.*

The next theorem states that, given ledger states which, when merged and restricted to $\mathbb{V}_{\mathsf{UTxO}}$, yield a valid sequence, appending the effect of any transaction that is not already in the sequence produces a new valid sequence in $\mathbb{V}_{\mathsf{UTxO}}$.

Theorem 3. *The validity language* $\mathbb{V}_{\mathsf{UTxO}}$ *is correct with respect to* merge-UTxO *as defined in Section 3.5, Definition 13.*

Ownership preservation. Algorithm 6 presents an owner function for $\mathbb{V}_{\mathsf{UTxO}}$ over a merged ledger (cf. Section 4.2), constructed from any two ownership preserving owner functions, owner_S and owner_D that are applied respectively to $\mathcal{L}_S$ and $\mathcal{L}_D$.

5.2 The Pegging Protocol

We now instantiate our pegging $\mathbb{V}_{\mathsf{UTxO}}$ protocol for Bitcoin as $\mathcal{L}_S$, and Cardano as $\mathcal{L}_D$. For the Bitcoin side, we require covenants, which are not natively supported yet, and thus need to be emulated via a committee of signers [4, 28]. The protocol participants are as follows:

- **Bridge Operators** $\mathcal{O} := \{O_1, \ldots, O_n\}$: A permissioned set of operators that is responsible for facilitating the bridge operation. We assume that all operators are full nodes of both $\mathcal{L}_S$ and $\mathcal{L}_D$.

Algorithm 6 OwnerCombined($\mathcal{L}$, token_id)

```
1: function OwnerCombined(L, token_id)
2:     if L = π_S(L) then return Owner_S(L, token_id)
3:     if L = π_D(L) then return Owner_D(L, token_id)
4:     o_S ← Owner_S(π_S(L), token_id)
5:     if o_S is of LockTX type then
6:         o_D ← Owner_D(π_D(L), token_id)
7:         if o_D = ⊥ then                                   ▷ no owner on destination
8:             return o_S.out[1]                             ▷ owner specified in LockTX
9:         else if o_D is of BurnTX type then
10:            if L.ind(o_D) > L.ind(o_S) then return o_D.out[1]   ▷ owner in BurnTX
11:            else return o_S.out[1]
12:        else return o_D                  ▷ token not yet burned in destination ledger
13:        end if
14:    else return o_S          ▷ token in source ledger, before lock or after unlock
15:    end if
16: end function
```

- **Signing (Covenant) Committee $\mathcal{S}$**: A permissioned set of signers that is responsible for pre-signing the transactions to emulate covenants and thus prevent the spending UTxOs in an unintended way. We denote a spending condition requiring an emulated covenant signature (on a transaction `tx`) from $\mathcal{S}$ as $\mathsf{Cov}(\mathcal{S}, \mathtt{tx})$.
- **ChainVM operators, $\mathcal{CO} := \{CO_1, \ldots, CO_m\}$**: A (potentially permissioned) set of operators that is responsible for proving and/or challenging the computations run inside ChainVM.
- **Bridge users**: The users who hold and transfer the tokens on Bitcoin or Cardano. During peg-in, we denote with A the original party owning the token in Bitcoin, and with B the owner in Cardano; for the peg-out, C is the owner of the wrapped token in Cardano, who wants to bridge it back to Bitcoin

Protocol overview. In a nutshell, A initiates the protocol by posting a `LockReqTX` transaction on Bitcoin, which signals the token and bridge operator set $\mathcal{O}$, and the recipient in Cardano, B. One of the operators posts `MintTX` on Cardano, creating the wrapped token (wTKN), which is locked until all operators and A cooperate to post `ReleaseTX` on Cardano as well. `ReleaseTX` includes all the information needed to post `LockTX` on Bitcoin, which locks the token indefinitely. Eventually, the wTKN ends up owned by C, who wants to bridge it back to Bitcoin. He posts `CommitBurnTX` on Cardano, rendering the wTKN useless (but still, not burned; this prevents race conditions), and waits for at least one operator in $\mathcal{O}$ to cooperate. When that happens, C irreversibly burns the wTKN by posting `BurnTX` on Cardano. When the inclusion of `BurnTX` on Cardano is proven on Bitcoin (for which we rely on ChainVM), C can post `SpendTX` on Bitcoin, releasing the token locked before with `LockTX`. The pegging protocols are sketched in Figure 1.

Latency. Cardinal inherits the latency characteristics depending on which BitVM variant is used to instantiate ChainVM: in the optimistic case, peg-ins and peg-outs require no on-chain dispute steps, while challenged executions complete in logarithmic ([6, 26]) or constant ([28, 27, 19]) rounds. This matches the latency profile of prior BitVM bridges.

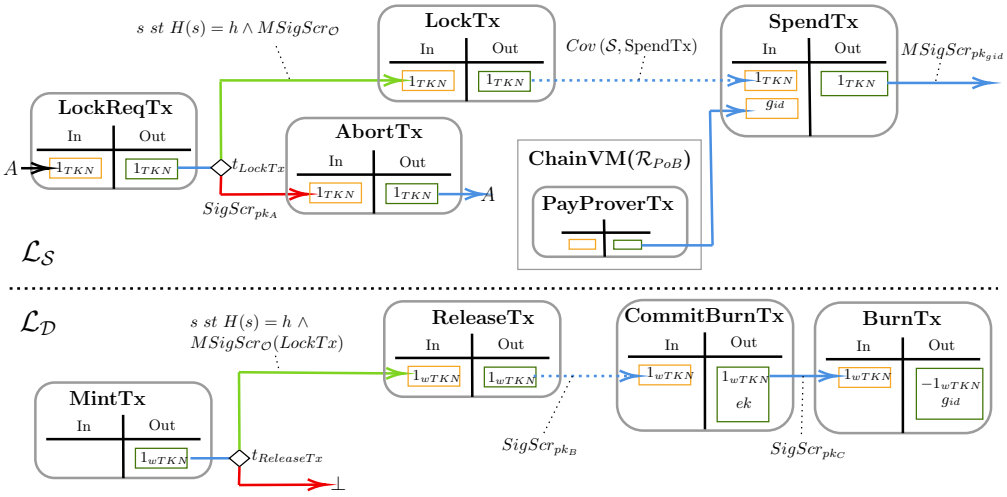

Figure 1: Simplified pegging protocols. Fee-related inputs/outputs are omitted.

5.3 Protocol Transactions

Let $\mathsf{param}_{\mathsf{MS}} \leftarrow \mathsf{MS.Setup}(1^\kappa)$ be the pre-agreed parameter for multisignature schemes, which is Musig2 [33] for Bitcoin. Let also $(\mathsf{txn_id}_{\mathsf{TKN}}, 0)$ be the UTxO containing the token to be bridged to Cardano. We assume that all transactions include a CPFP output [1], or a similar fee-bumping technique, and/or the necessary inputs and witnesses for paying the required fees. When specifying the peg-out, we assume that the wrapped token to be returned from Cardano to Bitcoin lives in an EUTxO represented by $(\mathsf{txn_id}_{\mathsf{wTKN}}, 0)$, spendable by providing witness $\mathsf{wit}_{\mathsf{wTKN}}$. Before defining the peg-in and peg-out protocols in Section 5.5, we specify the involved transactions. All transactions have an identifying field, $\mathsf{txn_id}$, computed as a hash of the transaction's inputs and outputs.

Peg-in transactions. The peg-in transactions, which move the token from Bitcoin to Cardano in an atomic way, are defined as follows.

$\texttt{LockReqTX}(h)$**.** Specifies the operators by locking the token into a Musig2 address controlled by them, with an HTLC script to get the token back if the protocol stalls. h is the value used for the HTLC script.

$\mathsf{in} \leftarrow [(\mathsf{txn_id}_{\mathsf{TKN}}, 0)]$. The only input is the UTxO containing the token to bridge

out. The outputs are as follows (we use d to denote "dust" values):

- $(a_{\mathsf{TKN}}, \mathsf{HTLCScr}(t_{\texttt{LockTX}}, h, \mathsf{MSScr}_{\mathcal{O}}, \mathsf{SigScr}_{pk_A}))$ - locks the token into a HTLC contract with timeout $t_{\texttt{LockTX}}$. Before the time-lock expires, the UTxO can be spent by providing a multisignature by all operators; after $t_{\texttt{LockTX}}$, A can recover the token by using her private key.
- $(d, \mathsf{MDataScr}(\mathsf{pk}_B, t_{\texttt{ReleaseTX}}, \texttt{LockTX}))$ - The second output just includes some dust value, and the metadata needed on Cardano side: the recipient's public key, the chosen expiration time for the release transactions described below, and an encoding of $\texttt{LockTX}$.

$\mathsf{wit} \leftarrow [\mathsf{wit}_{\mathsf{TKN}}]$. The witnesses required to spend the token UTxO.

`AbortTX`. Reclaims the token locked by `LockReqTX`(h) when there is no response.

in $\leftarrow$ [(`LockReqTX`, 0)]. The token locked in `LockReqTX`.

out $\leftarrow$ [($a_{\mathsf{TKN}}, \mathsf{SigScr}_{pk_A}$)]. Giving the token back to A.

wit $\leftarrow$ [$\sigma_{A,\texttt{AbortTX}}$]. A signature of `AbortTX`, under A's public key.

`MintTX`. Mints the wrapped token but puts it in a locked state.

in $\leftarrow$ []. Only fee-related inputs, omitted for simplicity.

out $\leftarrow$ [(1_{wTKN}, $\mathsf{HTLCScr}(t_{\texttt{ReleaseTX}}, h, \mathsf{MSScr}_{\mathcal{O}}(\texttt{LockTX}) \wedge \mathsf{NextRecipient}(\mathsf{pk}_B)$, **false**), $\delta = (t_{\texttt{ReleaseTX}}, \mathsf{pk}_B, \texttt{LockTX})$)]. A single EUTxO that mints exactly one native asset representing the wrapped token, and locks it under an HTLC requiring the pre-image of hash h, a MuSig2 multisignature of `LockTX` by all operators, and adds a condition NextRecipient[8] restricting the spending transaction to resend the wrapped token to the new owner's public key, pk_B. Note that ($t_{\texttt{ReleaseTX}}$, `LockTX`, and pk_B) are specified as part of the datum passed to the smart contract layer of Cardano, and can be extracted by operators from the `LockReqTX`, where $t_{\texttt{ReleaseTX}}$ denotes the timelock to use for the subsequent transaction.

wit $\leftarrow$ []. Only fee-related inputs, omitted for simplicity – although we require that all operators contribute to the witness computation.

`ReleaseTX`. Unlocks the wTKN, and enables final locking of the token TKN.

in $\leftarrow$ [(`MintTX`, 0)]. The EUTxO from `MintTX` containing the wTKN.

out $\leftarrow$ [($1_{\mathsf{wTKN}}, \mathsf{SigScr}_{\mathsf{pk}_B}, \cdot$)]. Sends 1 wrapped token to the owner of pk_B. Note that this satisfies the $\mathsf{NextRecipient}(\mathsf{pk}_B)$ requirement imposed in (`MintTX`, 0)

wit $\leftarrow$ [($s, \sigma_{\mathcal{O},\texttt{LockTX}}$)]. h's preimage, s, and $\sigma_{\mathcal{O},\texttt{LockTX}}$ a MuSig2 by all $\mathcal{O}$ of `LockTX`.

`LockTX`. Puts the token in a indeterminately-long lock.

in $\leftarrow$ [(`LockReqTX`, 0)]. The token being locked, originating from `LockReqTX`.

out $\leftarrow$ [$((1_{NFT}, \mathsf{Cov}(\mathcal{S}, \texttt{SpendTX}_{i,gid}))_{i\in|\mathcal{O}|,gid\in[2^{|\mathcal{O}|}]}$)]. Namely, the `LockTX` transaction has one UTxO that can be spent only by an (emulated) covenant transaction of the format `SpendTX`. The latter transactions are quantified over i (the index of the operator acting as prover in the underlying ChainVM instance) and gid (which, as defined in [21], is a member of the powerset of $\mathcal{O}$ and has thus cardinality $2^{|\mathcal{O}|}$). Full detail is given in Section 5.5.

wit $\leftarrow$ [$(s, \sigma_{\mathcal{O}}), \mathsf{wit}^{\mathcal{L}_S}_{fee,1}$], where the first witness includes the preimage s of the hash h used in the timelock of the first output of `LockReqTX`, and a Musig2 signature by all $\mathcal{O}$. The second output is just whatever witness is needed for spending the second output of the Bitcoin fees transaction.

[8] NextRecipient can be enforced via a Cardano smart contract.

Peg-out transactions. The peg-out transactions, for atomically returning the wrapped token from Cardano to Bitcoin, are defined as follows:

`CommitBurnTX`(pk)**.** Here, pk is an encryption public key whose private key is owned by C. Locks wTKN, only allowing it to burn in a subsequent `BurnTX`.

in $\leftarrow [(\mathsf{txn_id}_{\mathsf{wTKN}}, 0)]$. The EUTxO containing 1_{wTKN}.

out $\leftarrow [(1_{\mathsf{wTKN}}, \mathsf{SigScr}_{\mathsf{pk}_C} \wedge \mathsf{OnlyBurn}, \mathsf{pk})]$. `CommitBurnTX` only has one output that sends the wTKN to C, and imposes the condition OnlyBurn which restricts the next transaction by allowing it to only burn the token[9]. In addition, the EUTxO specifies as datum an encryption public key pk that bridge operators will use to send off-chain some data to the wrapped token owner.

wit $\leftarrow [\mathsf{wit}_{\mathsf{wTKN}}]$. The witness to spend the EUTxO containing the wTKN.

`BurnTX`(id)**.** Where id is some binary string of length n. Burns the wTKN.

in $\leftarrow [(\mathsf{txn_id}_{\texttt{CommitBurnTX}}, 0)]$. The EUTxO from `CommitBurnTX` containing the wrapped token.

out $\leftarrow [(-1_{\mathsf{wTKN}}, \bot, g_{\mathsf{id}})]$. The only EUTxO burns the wTKN, and specifies in the datum the chosen gid that will be used for unlocking the token on Bitcoin.

wit $\leftarrow [\sigma_{\mathsf{pk}_C}]$. A valid signature under pk_C. The OnlyBurn condition imposed in the smart contract of `CommitBurnTX`'s EUTxO is met by specifying a negative amount of wTKN, which, in Cardano, translates to burning a native asset.

$\texttt{SpendTX}_{i,\mathsf{gid}}$**.** There is one `SpendTX` transaction per operator in the underlying ChainVM instance, and each proving operator needs to be able to send the token to every possible group identifier gid that can take any value between 1 and $2^n - 1$, where n is the number of bridge operators as specified in [21].

in $\leftarrow [(\texttt{LockTX}, 0), (\mathsf{txn_id}_{\mathsf{PayProverTx}_i}, 2)]$. The first input points to the first UTxO of the `LockTX` transaction, which contains the locked token. The second input connects with the outcome of the ChainVM execution run during peg-out upon successful verification (cf. Section 5.4).

out $\leftarrow [(a_{\mathsf{TKN}}, \mathsf{MSScr}_{\mathsf{pk}_{\mathsf{gid}}}), (\mathsf{Payout}(\mathsf{TKN}, n, F), \mathsf{SigScr}_{\mathsf{pk}_i})]$. The first UTxO enables spending the token with a multi-signature by the group of operators identified by gid. The second output allows operator i to receive the reward for enabling the peg-out—defined via some pre-agreed function Payout of the TKN, number of operators, and the initial fees F paid to run the protocol.

wit $\leftarrow [\sigma_{\texttt{SpendTX}_{i,\mathsf{gid}}}, \sigma_{\mathsf{pk}_i}]$. The first witness is an emulated covenant signature of the current transaction. The second witness is a signature by operator i.

[9] OnlyBurn can be enforced via a Cardano smart contract.

5.4 Computation Run Inside ChainVM

As mentioned, the SpendTX transaction requires the presence of a transaction attesting to the validity of a computation run inside a ChainVM instance. We specify this computation next. In a nutshell, in order to allow unlocking some token, we require that a matching BurnTX exists on Cardano. We attest this via a SNARK proof for an NP relation $\mathcal{R}_{\mathsf{PoB}}$ ensuring that:

- There exists a BurnTX transaction, binding to a given gid, that burns the same wTKN that was minted in the MintTX transaction.
- The BurnTX transaction burning the wTKN has been included in Cardano.

For the checks needed in $\mathcal{R}_{\mathsf{PoB}}$, we leverage Mithril certificates[10] [13]. In more detail, checking inclusion of BurnTX into a Cardano block – hereafter, BurnBLK – is done by verifying that it has been accumulated into the Merkle tree of transactions signed according to the Mithril scheme. However, that is not enough to attest whether the wrapped token is legitimate. For this, we have to check the full chain of blocks (or Mithril certificates) connecting the BurnTX transaction in BurnBLK with the MintTX transaction in MintBLK. Finally, BurnTX must contain a gid identifier for the subset of operators chosen by C – it is enough to check that this identifier appears in the datum of BurnTX.

We formalize the intuition in Section 5.4 into the relation shown in Figure 2.

$$\mathcal{R}_{\mathsf{PoB}} = \left\{ \begin{array}{c} (m, \mathtt{BurnTX}, \mathtt{BurnBLK}, \mathtt{MintTX}, \mathtt{MintBLK}, \mathsf{MithrilChain}), (\mathsf{wTKN}, \mathsf{gid}) : \\ \mathtt{BurnTX} = (\cdot, \cdot, [(-1_{\mathsf{wTKN}}, \cdot, \mathsf{gid})], \cdot)) \wedge \mathsf{gid} \in [2^n - 1] \wedge \\ (\mathtt{BurnTX} \in \mathtt{BurnBLK}) \wedge (\mathtt{BurnBLK} = \mathsf{MithrilChain}[-1]) \\ (\mathtt{MintTX} \in \mathtt{MintBLK}) \wedge (\mathtt{MintBLK} = \mathsf{MithrilChain}[0]) \wedge \\ \forall i \in [|\mathsf{MithrilChain}|], \mathsf{Mithril.Verify}(\mathsf{MithrilChain}[i]) \end{array} \right\}$$

Figure 2: NP relation proving a correct burn of wrapped tokens in Cardano. An NP relation is defined as $\{(x, w) : \textit{predicate}\}$, where w is the witness that is kept private, x is the public input, and *predicate* is a boolean predicate that x and w have to satisfy. MithrilChain[0] denotes the first block in a chain of Mithril blocks; MithrilChain[−1] denotes the last one.

ChainVM's predicate S for $\mathcal{R}_{\mathsf{PoB}}$. We verify proofs for $\mathcal{R}_{\mathsf{PoB}}$ inside a ChainVM instance. Connecting with the definition in Section 4.1, $Q(\pi)$ denotes the verification of a proof-of-inclusion of a burn transaction on Cardano (our $\mathcal{L}_D$), which implies that there is a ledger predicate S whose validity ensures that we can extract a valid proof for Q. In practice, the predicate S boils down to checking for the presence of a specific transaction in Bitcoin (our $\mathcal{L}_S$) that parametrizes the ChainVM instance. While any ChainVM construction is compatible with our bridge design, we give a full specification of the BitVMX case, deferred to Section C for brevity. Here, we simply observe that, in the case of BitVMX, when the computation is valid, a $\mathtt{PayProverTx}_i$ transaction appears in $\mathcal{L}_S$, where $i \leq m$ identifies a valid operator within the ChainVM operators set. That is, in a BitVMX-based construction of ChainVM, the predicate for our bridge becomes $S(\mathcal{L}_S) := \exists i \leq m \; s.t. \; \mathtt{PayProverTxn}_i \in \mathcal{L}_S$.

[10] Mithril is a protocol running on top of Cardano, by means of which blocks endorsed by a sufficient amount of stake-holders are certified. See https://mithril.network/ (last access, 2025/09/16.)

5.5 Peg-In and Peg-Out Protocols

We now describe our peg-in and peg-out protocols, instantiated from the transactions in Section 5.3 and the $\mathsf{ChainVM}$ computation of Section 5.4. For readability, we use shorthand notation: "Send x to A" means that message x is transmitted to A, while "Wait until receiving y from B" means execution halts until y is received. Protocol security guarantees that waiting periods and timelocks are such that there is enough time to exchange and post each transaction. Finally, we write preTx for a transaction Tx to denote the transaction body, i.e., $(\mathsf{txn_id}, \mathsf{in}, \mathsf{out})$ triple that is actually signed.

Peg-In

$A \in \mathcal{P}$ // Process for user A (initial token owner)

1. $(\mathsf{pk}_A, \mathsf{sk}_A) \leftarrow \mathsf{S.KG}(\mathsf{param_S})$, sample s randomly, let $h \leftarrow \mathsf{Hash}(s)$. // For HTLC
2. $\mathcal{L}_S.\mathsf{submit}(\texttt{LockReqTX}(h))$
3. Wait until $\texttt{MintTX} \in \mathcal{L}_D.\mathsf{read}()$, and some operator $O \in \mathcal{O}$ sends $\sigma_{\texttt{LockTX}}$ to A
4. Send $s, \sigma_{\texttt{LockTX}}$ to all $O \in \mathcal{O}$ // wTKN already minted $\implies$ honest O will release
5. If $\texttt{LockTX} \notin \mathcal{L}_S.\mathsf{read}()$ at round $t_{\texttt{LockTX}}$, submit $\mathcal{L}_S.\mathsf{submit}(\texttt{AbortTX})$

$O \in \mathcal{O}$ // Process for (honest) bridge operators

1. $(\mathsf{pk}_O, \mathsf{sk}_O) \leftarrow \mathsf{MS.KG}(\mathsf{param_{MS}})$
2. Wait until $\texttt{LockReqTX}(h) \in \mathcal{L}_S.\mathsf{read}()$
3. $\mathsf{crs} \leftarrow \texttt{SNARK.Setup}(\mathcal{R}_{\mathsf{PoB}})$ // $\mathcal{R}_{\mathsf{PoB}}$ as in Fig. 2
4. $(sid, S, aux) \leftarrow \mathsf{ChainVM.Setup}(1^k, \texttt{SNARK.Vrfy})$
5. Send $\{\texttt{SpendTX}_{i,g_{id}}\}_{i\in[m],id\in[2^n]}$ to $\mathcal{S}$ // Signing Committee presigns covenants
6. Wait until receiving $\{\sigma_{\texttt{SpendTX}_{i,g_{id}}}\}_{i\in[m],id\in[2^n]}$ from $\mathcal{S}$ and $\mathcal{L}_D.\mathsf{submit}(\texttt{MintTX})$.
7. $\sigma^O_{\texttt{LockTX}} \leftarrow \mathsf{MS.Sign}(\mathsf{sk}_O, \texttt{preLockTX})$ and send $\sigma^O_{\texttt{LockTX}}$ to all $O' \neq O \in \mathcal{O}$
8. Wait until receiving $\sigma^{O'}_{\texttt{LockTX}}$ from all $O' \neq O \in \mathcal{O}$
9. $\sigma_{\texttt{LockTX}} \leftarrow \mathsf{MS.SignA}(\{\sigma^O_{\texttt{LockTX}}\}_{O\in\mathcal{O}})$ and send $\sigma_{\texttt{LockTX}}$ to A // share multisig with A
10. Wait until receiving $s, \sigma_{\texttt{LockTX}}$ from A
11. **if** $h \neq \mathsf{Hash}(s) \vee \mathsf{MS.Verify}(\sigma_{\texttt{LockTX}}, \texttt{LockTX}, \{\mathsf{pk}_O\}_{O\in\mathcal{O}}) = 0$: **abort**
12. $\mathcal{L}_D.\mathsf{submit}(\texttt{ReleaseTX})$ // Honest op releases and locks
13. Wait until $\texttt{ReleaseTX} \in \mathcal{L}_D.\mathsf{read}()$
14. $\mathcal{L}_S.\mathsf{submit}(\texttt{LockTX})$.

$S \in \mathcal{S}$ // Process for emulated covenants Committee Signers

1. $(\mathsf{pk}_S, \mathsf{sk}_S) \leftarrow \mathsf{MS.KG}(\mathsf{param_{MS}})$
2. Receive $\{\texttt{SpendTX}_{i,g_{id}}\}_{i\in[n],id\in[m]}$ from $O \in \mathcal{O}$
3. **for all** $i \in [n], id \in [m]$
4. $\quad \sigma^S_{\texttt{SpendTX}_{i,g_{id}}} \leftarrow \mathsf{MS.Sign}(\mathsf{sk}_S, \texttt{preSpendTX}_{i,g_{id}})$

5. Send $\sigma^{S}_{\texttt{SpendTX}_{i,g_{id}}}$ to all $S' \neq S \in \mathcal{S}$
6. Wait until receiving $\sigma^{S'}_{\texttt{SpendTX}_{i,g_{id}}}$ from all $S' \neq S \in \mathcal{S}$
7. $\sigma_{\texttt{SpendTX}_{i,g_{id}}} \leftarrow \mathsf{MS.SignA}(\{\sigma^{S}_{\texttt{SpendTX}_{i,g_{id}}}\}_{S \in \mathcal{S}})$
8. Send $\sigma_{\texttt{SpendTX}_{i,g_{id}}}, i \in [n], id \in [m]$ to all $O \in \mathcal{O}$

Peg-Out

$C \in \mathcal{P}$ // Process for user C (final token owner)

1. $(\mathsf{ek}, \mathsf{dk}) \leftarrow \mathsf{AE.KG}(1^k)$
2. $\mathcal{L}_D.\mathsf{submit}(\texttt{CommitBurnTX}(\mathsf{ek}))$
3. Wait Δ rounds until receiving csk_{O_i} keys from $O_i \in \mathcal{O}$
4. $\mathsf{sk}_{O_i} \leftarrow \mathsf{AE.Dec}(\mathsf{dk}, csk_{O_i})$ // If dec fails, discard O_i
5. **for all** successful i: // Compute id from cooperating O_i
6. $id \leftarrow \sum 2^{i-1}$
7. $\mathcal{L}_D.\mathsf{submit}(\texttt{BurnTX}(id))$.

$O \in \mathcal{O}$ // Process for (honest) bridge operators

1. Wait until $\texttt{CommitBurnTX}(\mathsf{ek})$ appears on $\mathcal{L}_D$
2. $\mathsf{csk}_O \leftarrow \mathsf{AE.Enc}(\mathsf{ek}, \mathsf{sk}_O)$, send csk_O to B
3. Wait until $\texttt{BurnTX}(id)$ appears on $\mathcal{L}_D$
4. Let wTKN be the identifier of the wrapped token in $\mathcal{L}_D$
5. Let $\mathsf{BurnBLK}$ and $\mathsf{MintBLK}$ the $\mathcal{L}_D$blocks containing $\texttt{BurnTX}$ and $\texttt{MintTX}$
6. Let $\mathsf{MithrilChain}$ be the Mithril chain starting in $\mathsf{MintBLK}$ and ending in $\mathsf{BurnBLK}$
7. $x \leftarrow (n, \texttt{BurnTX}, \mathsf{BurnBLK}, \texttt{MintTX}, \mathsf{MintBLK}, \mathsf{MithrilChain})$
8. $\pi \leftarrow \mathsf{SNARK.Prove}(\mathcal{R}_{\mathsf{PoB}}, \mathsf{crs}, x, (\mathsf{wNFT}, id))$
9. $r \leftarrow \mathsf{ChainVM.Execute}(sid, x, \pi), aux)$
10. After round r, $\mathcal{L}_S.\mathsf{submit}(\texttt{SpendTX}_{p,g_{id}})$ // After r, $\texttt{SpendTX}$ inputs are in $\mathcal{L}_S$

Remark 1. *For the* $\texttt{SpendTX}$ *transactions, we leverage an optimization trick that prevents us from needing to precompute the emulated covenant signatures for* $n \cdot m$ *executions of the ChainVM (in our instantiation, BitVMX) instance. This results in very significant savings, mostly space-wise – in particular, we only need to precompute the emulated covenants for* m $\mathsf{ChainVM}$ *executions. We describe this trick, which preserves the described* $\mathsf{ChainVM}$ *interface, in Section C.*

6 Security

We prove that our construction satisfies pegging security. Our proof strategy follows that of [23], adapted to our setting. Persistence and liveness follow from [22] and [25] (Lemma 2), thus the main

focus will be on the firewall property.

We analyze the different cases, i.e., when both $\mathcal{L}_S$ and $\mathcal{L}_D$ are secure ($\mathbb{A}_S$ and $\mathbb{A}_D$ hold), and when only one of them is secure. The main idea is to assume that the merged ledger $L = \mathsf{merge\text{-}UTxO}(\mathcal{L}_S^{\cup}[t], \mathcal{L}_D^{\cup}[t])$ at some slot t violates the validity language $\mathbb{V}_{\mathsf{UTxO}}$ and argue about the transaction `tx` causing the violation. For each case, we consider the possible types of `tx` and show that acceptance of the transaction `tx` in $\mathcal{L}_S$ or $\mathcal{L}_D$ that breaks $\mathbb{V}_{\mathsf{UTxO}}$, can be used to break the underlying primitives: multi-signature EUF-CMA security, collision-resistant hash functions, or soundness of ChainVM.

Let $\mathbb{A}_S[t]$ be the honest-majority assumption of Bitcoin backbone [22], and $\mathbb{A}_D[t]$ be the honest-majority assumption of Ouroboros [25], for all slots $t' \leq t$. We now state the main theorems of the paper.

Theorem 4. *Consider the model described in Section 3.1. The protocol described in Section 5 using a sound* ChainVM*, collision-resistant hash function, an EUF-CMA multisignature scheme, IND-CPA asymmetric encryption scheme, and existential honesty of $\mathcal{O}$, is* pegging secure *(Section 3.5, Definition 14) with respect to $\mathbb{A}_S$ and $\mathbb{A}_D$,* merge-UTxO*,* effect-UTxO*, and $\mathbb{V}_{\mathsf{UTxO}}$.*

The proof of Theorem 4 is deferred to Section A.

Theorem 5. *The protocol described in Section 5 is* ownership preserving *(Definition 19) with respect to* ownerCombined *(Algorithm 6).*

Lemma 1 (Chain of custody of ownerCombined)**.** *If owner_S and owner_D both satisfy the chain-of-custody property, then the combined ownership function* ownerCombined *(Algorithm 6) also satisfies chain of custody.*

Proof. By Algorithm 6, ownerCombined changes value only in response to *send-side* transactions, namely `LockReqTX` on $\mathcal{L}_S$ and `BurnTX` on $\mathcal{L}_D$. Both types of transaction consume an output under the previous owner's address, i.e., they contain a valid witness for the prior custody. Hence any change of ownerCombined is justified by the spending condition of the previous owner, exactly as required by the chain-of-custody property. □

We now prove Theorem 5, i.e., that our protocol achieves *ownership preservation.*

Proof. Consider any merged trace $W = \mathsf{merge}(\mathcal{L}_S^{\cup}[t], \mathcal{L}_D^{\cup}[t])$, prefix $W[:k]$, and token token_id. If $\mathsf{ownerCombined}(W[:k], \mathsf{token_id})$ is an address on $\mathcal{L}_\ell \in \{\mathcal{L}_S, \mathcal{L}_D\}$ and $W[k] \in \mathcal{L}_\ell$, then by construction of Algorithm 6 the combined function delegates to the native function owner_ℓ applied to the projection $\pi_\ell(W[:k])$. That is, $\mathsf{ownerCombined}(W[:k], \mathsf{token_id}) = \mathsf{owner}_\ell\big(\pi_\ell(W[:k]), \mathsf{token_id}\big)$, so the condition of Definition 19 holds. Since the argument is independent of the choice of ℓ and prefix $W[:k]$, the protocol is ownership preserving. □

7 Discussion

7.1 Improvements for Cardinal

Covenant solution. If the source ledger (e.g., Bitcoin) supports covenants, the protocol can be greatly simplified. The covenant committee is no longer required, since covenants remove the need to emulate spending restrictions, and the exponential presigning of `SpendTX` transactions (cf.

Section 5.5) can be avoided. Instead, covenants enforce directly that outputs of ChainVM are only spendable by the designated g_{id} (operator subset).

Accountability of malicious users. In the peg-out phase, operators currently share ephemeral keys with the user C who issues the `BurnTX`. Although the protocol is secure under existential honesty, this mechanism leaves an accountability gap: a malicious C who successfully spends `SpendTX` could falsely claim that operators stole it, and outsiders cannot distinguish this from genuine operator misbehavior.

This limitation is orthogonal to ownership preservation and can be resolved by replacing key revelation with sending signature shares of the transaction that consumes `SpendTX` and gives funds directly to C.

Competition between operators. In the current design, only a single operator collects the fees. Splitting fees among all operators is undesirable, as it encourages free-riding. A fairer approach is to stagger ChainVM instances by adjusting the timelock of the happy path, so that the right to claim fees rotates among honest operators in a round-robin manner. This can help to balance incentives without undermining participation.

7.2 Vanilla BitVM bridges are not ownership preserving

Current bridges built on BitVM, e.g., [28, 8], do not achieve ownership preservation. At peg-in, assets are locked on $\mathcal{L}_S$ under an operator-controlled script, and users receive $\mathcal{L}_D$ tokens in return. At peg-out, users burn their $\mathcal{L}_D$ tokens, but the actual $\mathcal{L}_S$ tokens from the lock are spent by operators to their own address, with the user compensated from separate operator funds.

In vanilla BitVM, there exist peg-out traces where no combined ownership function ownerCombined can satisfy both chain of custody and ownership preservation (Definition 19). Intuitively, the burn transaction on $\mathcal{L}_D$ nominates a recipient address u on $\mathcal{L}_S$, but the subsequent spend of the locked coins delivers funds to an operator-controlled address $v \neq u$.

If ownerCombined follows the burn, it must assign u as owner. But then, once the spend on $\mathcal{L}_S$ pays v, preservation requires ownerCombined $=$ owner$_S$, which assigns v, which is a contradiction. If instead ownerCombined follows the spend and jumps directly to v, the change of owner from u to v has no witness from u's key, violating chain of custody. Thus, no such ownerCombined exists.

8 Conclusion

We introduced and formally analyzed Cardinal, the first provably secure bridge protocol between UTxO and EUTxO ledgers, instantiated for Bitcoin and Cardano. Our construction satisfies pegging security and a novel property, *ownership preservation*, which ensures users keep their original coins after a cross-chain activity where they do not spend their coins. Our bridge also facilitates the transmission of ordinal NFTs from Bitcoin and back.

A current limitation lies in accountability during peg-out: bridge operators reveal secret keys to unlock the tokens in Bitcoin, enabling a malicious operator to potentially front-run the user. While our 1-out-of-n assumption mitigates this, a stronger solution would use multisignature shares instead of keys, allowing users to prove misbehavior. We leave this for future work, along with benchmarking different ChainVM alternatives, where we expect interesting trade-offs.

Acknowledgements. The authors would like to thank Ramses Fernández, Ariel Futoransky, Emilio García, Hernán Pérez Rodal, and Diego Tiscornia from Fairgate Labs, and Nicolas Biri,

Lucas Escot, Romain Pellerin, Torben Poguntke, and Thomas Vellekoop from Input Output for their insightful contributions. This work was supported by Input Output (https://iohk.io) through their funding of the Edinburgh Blockchain Technology Lab.

References

[1] Child pays for parent (cpfp). https://bitcoinops.org/en/topics/cpfp/, accessed: 2025-09-19

[2] Coinbase prohibited & conditional use policy. https://www.coinbase.com/en-gb/legal/prohibited_use, accessed: 2025-09-19

[3] Coinmarketcap. https://coinmarketcap.com/, accessed: 2025-09-19

[4] Bitcoin optech: Covenants (2025), https://bitcoinops.org/en/topics/covenants/

[5] Mt. gox explained: History, 2014 collapse, and current status. https://www.investopedia.com/terms/m/mt-gox.asp (2025), accessed: 2025-09-19

[6] Aumayr, L., Avarikioti, Z., Linus, R., Maffei, M., Pelosi, A., Stefo, C., Zamyatin, A.: BitVM: Quasi-Turing Complete Computation on Bitcoin. Cryptology ePrint Archive, Paper 2024/1995 (2024), https://eprint.iacr.org/2024/1995

[7] Back, A., Corallo, M., Dashjr, L., Friedenbach, M., Maxwell, G., Miller, A., Poelstra, A., Timón, J., Wuille, P.: Enabling blockchain innovations with pegged sidechains (2014), https://blockstream.com/sidechains.pdf

[8] Bal, E., Aumayr, L., İyidoğan, A., Scaffino, G., Karakuş, H., Aslan, C.E., Litos, O.S.T.: Clementine: A collateral-efficient, trust-minimized, and scalable bitcoin bridge. Cryptology ePrint Archive, Paper 2025/776 (2025), https://eprint.iacr.org/2025/776

[9] Belenkov, N., Callens, V., Murashkin, A., Bak, K., Derka, M., Gorzny, J., Lee, S.: Sok: A review of cross-chain bridge hacks in 2023. CoRR **abs/2501.03423** (2025). https://doi.org/10.48550/ARXIV.2501.03423, https://doi.org/10.48550/arXiv.2501.03423

[10] Bellare, M., Desai, A., Pointcheval, D., Rogaway, P.: Relations among notions of security for public-key encryption schemes. In: Krawczyk, H. (ed.) Advances in Cryptology - CRYPTO '98, 18th Annual International Cryptology Conference, Santa Barbara, California, USA, August 23-27, 1998, Proceedings. Lecture Notes in Computer Science, vol. 1462, pp. 26–45. Springer (1998). https://doi.org/10.1007/BFB0055718, https://doi.org/10.1007/BFb0055718

[11] Buchmann, J., Dahmen, E., Ereth, S., Hülsing, A., Rückert, M.: On the security of the winternitz one-time signature scheme. Int. J. Appl. Cryptogr. **3**(1), 84–96 (2013). https://doi.org/10.1504/IJACT.2013.053435, https://doi.org/10.1504/IJACT.2013.053435

[12] Canetti, R.: Universally composable security: a new paradigm for cryptographic protocols. In: Proceedings 42nd IEEE Symposium on Foundations of Computer Science. pp. 136–145 (2001). https://doi.org/10.1109/SFCS.2001.959888

[13] Chaidos, P., Kiayias, A.: Mithril: Stake-based threshold multisignatures. In: Kohlweiss, M., Pietro, R.D., Beresford, A.R. (eds.) Cryptology and Network Security - 23rd International Conference, CANS 2024, Cambridge, UK, September 24-27, 2024, Proceedings, Part I. Lecture Notes in Computer Science, vol. 14905, pp. 239–263. Springer (2024). https://doi.org/10.1007/978-981-97-8013-6_11, https://doi.org/10.1007/978-981-97-8013-6_11

[14] Chakravarty, M.M.T., Chapman, J., MacKenzie, K., Melkonian, O., Jones, M.P., Wadler, P.: The extended UTXO model. In: Bernhard, M., Bracciali, A., Camp, L.J., Matsuo, S., Maurushat, A., Rønne, P.B., Sala, M. (eds.) Financial Cryptography and Data Security - FC 2020 International Workshops, AsiaUSEC, CoDeFi, VOTING, and WTSC, Kota Kinabalu, Malaysia, February 14, 2020, Revised Selected Papers. Lecture Notes in Computer Science, vol. 12063, pp. 525–539. Springer (2020). https://doi.org/10.1007/978-3-030-54455-3_37, https://doi.org/10.1007/978-3-030-54455-3_37

[15] Chung, H., Masserova, E., Shi, E., Thyagarajan, S.A.: Rapidash: Atomic Swaps Secure under User-Miner Collusion. In: Financial Cryptography (2025)

[16] CoinDesk, by Shaurya Malwa: North korean hacking group tied to $100m harmony hack moves 41,000 ether over weekend (2023), https://www.coindesk.com/tech/2023/01/16/north-korean-hacking-group-tied-to-100m-harmony-hack-moves-41000-ether-over-weekend

[17] Damgård, I.: A design principle for hash functions. In: Brassard, G. (ed.) Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings. Lecture Notes in Computer Science, vol. 435, pp. 416–427. Springer (1989). https://doi.org/10.1007/0-387-34805-0_39, https://doi.org/10.1007/0-387-34805-0_39

[18] Developers, Consensys: BTC Relay, http://btcrelay.org/

[19] Eagen, L.: Glock: Garbled locks for bitcoin. Cryptology ePrint Archive, Paper 2025/1485 (2025), https://eprint.iacr.org/2025/1485

[20] Fiore, D., Nitulescu, A.: On the (in)security of snarks in the presence of oracles. In: Hirt, M., Smith, A.D. (eds.) Theory of Cryptography - 14th International Conference, TCC 2016-B, Beijing, China, October 31 - November 3, 2016, Proceedings, Part I. Lecture Notes in Computer Science, vol. 9985, pp. 108–138 (2016). https://doi.org/10.1007/978-3-662-53641-4_5, https://doi.org/10.1007/978-3-662-53641-4_5

[21] Futoransky, A., Barbara, F., Fernandez, R., Larotonda, G., Lerner, S.D.: TOOP: A transfer of ownership protocol over bitcoin. Cryptology ePrint Archive, Paper 2025/964 (2025), https://eprint.iacr.org/2025/964

[22] Garay, J., Kiayias, A., Leonardos, N.: The bitcoin backbone protocol with chains of variable difficulty. In: Advances in Cryptology – CRYPTO 2017. Springer International Publishing (2017)

[23] Gaži, P., Kiayias, A., Zindros, D.: Proof-of-stake sidechains. In: 2019 IEEE Symposium on Security and Privacy (SP). pp. 139–156 (2019). https://doi.org/10.1109/SP.2019.00040

[24] Goes, C.: The interblockchain communication protocol: An overview (2020), https://arxiv.org/abs/2006.15918

[25] Kiayias, A., Russell, A., David, B., Oliynykov, R.: Ouroboros: A provably secure proof-of-stake blockchain protocol. In: Katz, J., Shacham, H. (eds.) Advances in Cryptology – CRYPTO 2017. pp. 357–388. Springer International Publishing, Cham (2017)

[26] Lerner, S.D., Amela, R., Mishra, S., Jonas, M., Álvarez Cid-Fuentes, J.: BitVMX: A CPU for Universal Computation on Bitcoin (2024), https://arxiv.org/abs/2405.06842

[27] Linus, R.: BitVM 3s – Garbled Circuits for Efficient Computation on Bitcoin (2025), https://bitvm.org/bitvm3.pdf

[28] Linus, R., Aumayr, L., Avarikioti, Z., Maffei, M., Pelosi, A., Litos, O.T., Stefo, C., Tse, D., Zamyatin, A.: Bridging bitcoin to second layers via BitVM2. Cryptology ePrint Archive, Paper 2025/1158 (2025), https://eprint.iacr.org/2025/1158

[29] Möser, M., Eyal, I., Sirer, E.G.: Bitcoin covenants. In: Clark, J., Meiklejohn, S., Ryan, P.Y.A., Wallach, D.S., Brenner, M., Rohloff, K. (eds.) Financial Cryptography and Data Security - FC 2016 International Workshops, BITCOIN, VOTING, and WAHC, Christ Church, Barbados, February 26, 2016, Revised Selected Papers. Lecture Notes in Computer Science, vol. 9604, pp. 126–141. Springer (2016). https://doi.org/10.1007/978-3-662-53357-4_9, https://doi.org/10.1007/978-3-662-53357-4_9

[30] Nadahalli, T., Khabbazian, M., Wattenhofer, R.: Grief-free atomic swaps. In: 2022 IEEE International Conference on Blockchain and Cryptocurrency (ICBC). pp. 1–9 (2022). https://doi.org/10.1109/ICBC54727.2022.9805490

[31] NEAR: Eth-near rainbow bridge. https://www.near.org/blog/eth-near-rainbow-bridge (2020), accessed: 2025-09-19

[32] Network, T.: tbtc: Threshold network documentation. https://docs.threshold.network/ (2020), accessed: 2025-09-19

[33] Nick, J., Ruffing, T., Seurin, Y.: Musig2: Simple two-round schnorr multi-signatures. In: Malkin, T., Peikert, C. (eds.) Advances in Cryptology - CRYPTO 2021 - 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16-20, 2021, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12825, pp. 189–221. Springer (2021). https://doi.org/10.1007/978-3-030-84242-0_8, https://doi.org/10.1007/978-3-030-84242-0_8

[34] Nolan, T.: Alt chains and atomic transfers. https://bitcointalk.org/index.php?topic=193281.msg2224949 (2013), accessed: 2025-11-19

[35] Ordinals: Bip — ordinals ord project. https://github.com/ordinals/ord/blob/master/bip.mediawiki (2025), accessed: 2025-09-19

[36] Tang, G., Chan, C., Nan, N., Yang, S., Zhou, J., Leung, H., Mamun, M., Drew, S.: Bitcoin cross-chain bridge: A taxonomy and its promise in artificial intelligence of things (2025), https://arxiv.org/abs/2509.10413

[37] Team, A.: Axelar: Connecting applications with blockchain ecosystems. https://axelar.network/whitepaper (2021), accessed: 2025-09-19

[38] Thyagarajan, S.A., Malavolta, G., Moreno-Sanchez, P.: Universal atomic swaps: Secure exchange of coins across all blockchains. In: 2022 IEEE Symposium on Security and Privacy (SP). pp. 1299–1316 (2022). https://doi.org/10.1109/SP46214.2022.9833731

[39] Xie, T., Zhang, J., Cheng, Z., Zhang, F., Zhang, Y., Jia, Y., Boneh, D., Song, D.: zkbridge: Trustless cross-chain bridges made practical. In: Yin, H., Stavrou, A., Cremers, C., Shi, E. (eds.) Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022. pp. 3003–3017. ACM (2022). https://doi.org/10.1145/3548606.3560652, https://doi.org/10.1145/3548606.3560652

[40] Zamyatin, A., Harz, D., Lind, J., Panayiotou, P., Gervais, A., Knottenbelt, W.J.: XCLAIM: trustless, interoperable, cryptocurrency-backed assets. In: 2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019. pp. 193–210. IEEE (2019). https://doi.org/10.1109/SP.2019.00085, https://doi.org/10.1109/SP.2019.00085

A Deferred proofs

A.1 Merge, effect, and asset validity proofs

Theorem 1. $\mathbb{V}_{\mathsf{UTxO}}$ *satisfies the base, monotonicity, and uniqueness of transactions properties as defined in [23].*

Proof. The base property is easy to check, by observing that, in Algorithm 3, ValidSeq and ValidSeqTid both return **true** when receiving ε as input.

For monotonicity, observe that ValidSeqTid in Algorithm 3 checks independently any subsequence. Assume that some subsequence w is rejected by ValidSeqTid in some arbitrary execution. If $w||w'$ is passed to ValidSeq, we have two cases: (1) w and w' relate to a different token_id, and (2) they relate to the same token_id. In (1), if w was rejected when passed independently to ValidSeqTid, it will be rejected as well – and this will cause ValidSeq to reject the $w||w'$ sequence because of the $\wedge$ composition in line 9. For (2), note that the separation done in ValidSeq preserves order. That is, if w came before w' in the full sequence, it will still be the case in the subsequences divided by *tid*. Thus, the verification of $w||w'$ in ValidSeqTid will still parse w first, and reject it for whatever reason it was rejected when checked independently. Therefore, monotonicity is preserved.

Finally, uniqueness of transactions is ensured by the rules of the underlying ledgers, which in our case are both assumed to follow the (E)UTxO design – thus, by assumption, a duplicate transaction would imply a double spend. □

Theorem 2. merge-UTxO *satisfies partitioning and topological soundness as defined in [23].*

Proof. The partitioning property is straightforward to see. Regarding topological soundness, by construction of merge-UTxO, every transaction $tx \in \mathcal{L}_S$ that is an effect of $tx' \in \mathcal{L}_D$ is preceded by tx' in the merged sequence (lines 5-8 in Algorithm 5). For the converse, we must have that for any transaction $tx' \in \mathcal{L}_D$ that is an effect of $tx \in \mathcal{L}_S$, tx' is preceded by tx in the merged sequence. Let $tx_{\mathcal{L}_S}$ be the transaction in $\mathcal{L}_S$ that is picked at line 4 of Algorithm 5. Let $\mathbf{tx}_A$ be $\mathcal{L}_D$'s tail sequence being checked against $tx_{\mathcal{L}_S}$ at line 5. If $\mathbf{tx}_A$ does not contain any preimage of

effect-UTxO for $tx_{\mathcal{L}_S}$, then $tx_{\mathcal{L}_S}$ is appended next to the merged map at line 10. On the other hand, if $\mathbf{tx}_A$ contains such a preimage, the full slice since the first transaction in $\mathbf{tx}_A$ and up to (and including) $tx_{\mathcal{L}_S}$'s preimage is appended next to the merge map. Let us denote this slice with $\mathbf{tx}_{\mathcal{L}_D}^{tx_{\mathcal{L}_S}}$, and $tx_{\mathcal{L}_S}$'s preimage in effect-UTxO with $tx_{\mathcal{L}_S}^{-1}$. In order to ensure topological soundness, we need to prove that $\mathbf{tx}_{\mathcal{L}_D}^{tx_{\mathcal{L}_S}}$ does not contain any transaction that is an effect of $tx_{\mathcal{L}_S}$, or of any transaction that follows $tx_{\mathcal{L}_S}$ in $\mathcal{L}_S$. Assume such a transaction $tx'_{\mathcal{L}_D}$ exists in $\mathbf{tx}_{\mathcal{L}_D}^{tx_{\mathcal{L}_S}}$. $tx'_{\mathcal{L}_D}$ cannot be an effect of $tx_{\mathcal{L}_S}$, as $tx_{\mathcal{L}_S}^{-1}$ ($tx_{\mathcal{L}_S}$'s effect preimage) is the last transaction in $\mathbf{tx}_{\mathcal{L}_D}^{tx_{\mathcal{L}_S}}$, and an effect of a transaction cannot happen before that same transaction's effect preimage. Somehow similarly, $tx'_{\mathcal{L}_D}$ can also not be an effect of any transaction $tx'_{\mathcal{L}_S}$ that follows $tx_{\mathcal{L}_S}$ in $\mathcal{L}_S$, as $tx_{\mathcal{L}_S}$ precedes $tx'_{\mathcal{L}_S}$ in $\mathcal{L}_S$, and $tx_{\mathcal{L}_S}$ cannot be written in $\mathcal{L}_S$ without $tx_{\mathcal{L}_S}^{-1}$ being written in $\mathcal{L}_D$. Hence, such a $tx'_{\mathcal{L}_D} = \mathsf{effect\text{-}UTxO}_{\mathcal{L}_S \to \mathcal{L}_D}(tx'_{\mathcal{L}_S})$ preceding $tx_{\mathcal{L}_D}^{-1}$ in $\mathcal{L}_D$ is a contradiction. □

Theorem 3. *The validity language* $\mathbb{V}_{\mathsf{UTxO}}$ *is correct with respect to* merge-UTxO.

Proof. Let $\mathcal{L}_0 := (\mathcal{L}_S, \mathcal{L}_D)$ be some initial ledger state such that $\mathsf{merge\text{-}UTxO}(\mathcal{L}_0)$ is in $\mathbb{V}_{\mathsf{UTxO}}$. Let $\mathcal{T}^{cl}_{\mathcal{L}_S,\mathcal{L}_D}$ be the set of cross-ledger transfers between $\mathcal{L}_S$ and $\mathcal{L}_D$, as defined in [23, Definition 5]:

- **Case** $tx \in \mathcal{L}_S \cap \mathcal{T}^{cl}_{\mathcal{L}_S,\mathcal{L}_D}$. Assume that $\mathcal{L}_S$ includes $tx \in \mathcal{L}_S \cap \mathcal{T}^{cl}_{\mathcal{L}_S,\mathcal{L}_D}$, but $\mathcal{L}_D$ does not include $tx' = \mathsf{effect\text{-}UTxO}_{\mathcal{L}_S \to \mathcal{L}_D}(tx)$. Let $\mathcal{L}_1 := \mathsf{merge\text{-}UTxO}(\mathcal{L}_S||\mathcal{L}_D||tx') = \mathsf{merge\text{-}UTxO}(\mathcal{L}_S||\mathcal{L}_D)||tx'$. Since merge-UTxO is topologically sound as per Theorem 2, then tx precedes tx' in the merged state. In particular, note that the definition of effect-UTxO only applies for transactions for the same token_id (i.e., some transaction with token identifier token_id cannot be an effect of another transaction with identifier token_id′). Thus, when parsing $\mathcal{L}_1$ with ValidSeq, both tx and tx' will be assigned to the same subsequence, and tx precedes tx' in that subsequence as well. In addition, the subsequence's last transaction must be tx, as by assumption $\mathcal{L}_0 \in \mathbb{V}_{\mathsf{UTxO}}$: by assumption, $tx' \notin \mathcal{L}_D$, and if a transaction other than tx' follows, $\mathcal{L}_0 \notin \mathbb{V}_{\mathsf{UTxO}}$. Now, tx must be of type `LockReqTX`, and tx' must be of type `MintTX`(as this is the only $\mathcal{L}_S \to \mathcal{L}_D$ effect in effect-UTxO. Thus, when parsing $\mathcal{L}_1$, the internal call to ValidSeqTid will accept (rather, not reject) the extended subsequence when running steps 7-10 in Algorithm 3, and return **true** in line 33, hence accepting the extended subsequence.
- **Case** $tx \in \mathcal{L}_S \cap \mathcal{T}^{cl}_{\mathcal{L}_S,\mathcal{L}_D}$. Assume that $\mathcal{L}_D$ includes $tx \in \mathcal{L}_D \cap \mathcal{T}^{cl}_{\mathcal{L}_D,\mathcal{L}_S}$, but $\mathcal{L}_S$ does not include $tx' = \mathsf{effect\text{-}UTxO}_{\mathcal{L}_D \to \mathcal{L}_S}(tx)$. Let $\mathcal{L}_2 := \mathsf{merge\text{-}UTxO}(\mathcal{L}_S||tx'||\mathcal{L}_D) = \mathsf{merge\text{-}UTxO}(\mathcal{L}_S||\mathcal{L}_D)||tx'$, where the last equality is consequence of tx' being an effect of $tx \in \mathcal{L}_D$ – and it gets pushed to the back in line 5 of Algorithm 5. Moreover, tx must be of type `ReleaseTX` or `BurnTX`, and tx' must be respectively of type `LockTX` or `SpendTX`. A similar analysis to the one done for the case "$tx \in \mathcal{L}_S \cap \mathcal{T}^{cl}_{\mathcal{L}_S,\mathcal{L}_D}$" applies here as well.

□

B Full proof of pegging security

The persistence and liveness guarantees of our construction follow from those of the Bitcoin backbone [22] and Ouroboros [25].

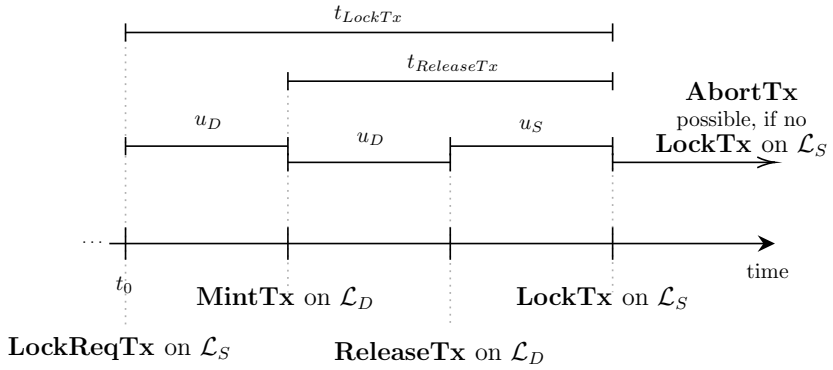

Figure 3: Illustration of the timelocks.

Lemma 2 (Persistence and Liveness)**.** *Consider the construction of Section 5 with the assumptions $\mathbb{A}_S$ and $\mathbb{A}_D$. For all slots t, if $\mathbb{A}_S[t]$ (respectively $\mathbb{A}_D[t]$) holds, then $\mathcal{L}_S$ (respectively $\mathcal{L}_D$) satisfies persistence and liveness up to slot t with overwhelming probability in k.*

Timelocks. From the liveness property of $\mathcal{L}_S$ and $\mathcal{L}_D$, with liveness parameters u_S and u_D, respectively, we can define lower bounds for the timelocks $t_{\texttt{LockTX}}$ and $t_{\texttt{ReleaseTX}}$, such that the transactions intended in the protocol's flow have enough time to be included in their respective ledgers. We illustrate this in Fig. 3.

In particular, in order to post `AbortTX`, $t_{\texttt{LockTX}}$ needs to expire after `LockReqTX` is included on $\mathcal{L}_S$. $t_{\texttt{LockTX}}$ needs to be large enough, to allow for `MintTX` and `ReleaseTX` to be included on $\mathcal{L}_D$ (takes up to u_D time slots each), and then `LockTX` to be included (takes up to u_S time slots). Therefore, $t_{\texttt{LockTX}} \geq 2u_D + u_S + x$, where $x \ll u_S + u_D$ is the network delay for exchanging the corresponding protocol messages (Section 5).

Similarly, $t_{\texttt{ReleaseTX}}$ needs to be large enough to allow for `ReleaseTX` to be included on $\mathcal{L}_D$ (takes up to u_D time slots), and then `LockTX` to be included (takes up to u_S time slots). Therefore, $t_{\texttt{ReleaseTX}} \geq u_D + u_S + y$, where $y \ll u_S + u_D$ is the network delay for exchanging the corresponding protocol messages (Section 5).

Peg-in and peg-out liveness follows from Lemma 2 and these timelocks, under the stated honesty assumptions. In particular, in our protocol (Section 5) the *witness exchange* needed to advance the flow requires *all* operators for peg-in and *at least one* operator for peg-out. Under synchrony, these witnesses are propagated within a known upper-bounded network delay, and the resulting transactions are included on $\mathcal{L}_S$ and $\mathcal{L}_D$ within their liveness bounds u_S and u_D, respectively.

Native verifiers. The functions $\text{VERIFYTX}_{\mathcal{L}_S}$ and $\text{VERIFYTX}_{\mathcal{L}_D}$ represent the native transaction validation rules of the respective chains. In our instantiation, $\mathcal{L}_S$ is Bitcoin and $\mathcal{L}_D$ is Cardano, so these validators enforce the usual checks of each ledger: inputs must be unspent, signatures or witnesses must match the spending condition, and transaction syntax and consensus rules must be satisfied.

We define these functions to remove any invalid transactions from the sequence. Therefore, once a ledger $\mathcal{L}_S^{\cup}$ (resp. $\mathcal{L}_D^{\cup}$) was adopted by some party, it must have passed the local verifier, and thus is a fixpoint under the corresponding verifier, i.e., applying $\text{VERIFYTX}_{\mathcal{L}_S}$ (resp. $\text{VERIFYTX}_{\mathcal{L}_D}$) will yield the same state back.

B.1 Firewall Property During $\mathcal{L}_D$ Failure

We now analyze the case, where $\mathcal{L}_D$ has suffered a failure, i.e., $S_t = \{\mathcal{L}_S\}$. Such a failure will not violate the firewall property, and we show this by defining an algorithm that, given any valid transaction sequence of $\mathcal{L}_S$, can fill the gaps with valid, "plausible" $\mathcal{L}_D$ transactions, such that the resulting sequence **txs** is a valid merged sequence with respect to $\mathbb{V}_{\mathsf{UTxO}}$ (cf. Algorithm 3). We prove this in Lemma 3, and describe the aforementioned algorithm in Algorithm 7. For a given transaction sequence L, the algorithm maps each $\mathtt{tx} \in L$ to one or more transactions, yielding **txs**. Importantly, note that we are only interested in showing the existence of **txs**, therefore Algorithm 7 does not need to be efficient, and therefore saying a private key exists is sufficient.

Remark. For readability, we write $\mathsf{get_token_id}(\mathtt{tx}) = \mathsf{get_token_id}(\mathtt{tx}')$ to denote that two transactions handle the same token. Strictly speaking, transactions may include fee or auxiliary inputs/outputs, so the correct condition is $\mathsf{get_token_id}(tx) \cap \mathsf{get_token_id}(tx') \neq \emptyset$.

Lemma 3. *For all $L \in \mathcal{T}_{\mathsf{UTxO}}$, if $\textsc{verifyTx}_{\mathcal{L}_S}(L) = L$ then* **txs** $\leftarrow \textsc{plausible}_{\mathcal{L}_D}$ *will satisfy* **txs** $\in \mathbb{V}_{\mathsf{UTxO}}$.

Proof. Suppose towards contradiction, that **txs** $\notin \mathbb{V}_{\mathsf{UTxO}}$, and let **txs**$'$ be the minimum prefix of **txs** such that **txs**$' \notin \mathbb{V}_{\mathsf{UTxO}}$. From the base property of $\mathbb{V}_{\mathsf{UTxO}}$, we know that **txs**$' \neq \epsilon$. We set $\mathtt{tx} :=$ **txs**$'[-1]$, and further, $\mathtt{tx}_L \in L$ be the input to $\textsc{plausibilityMap}_{\mathcal{L}_D}$ that caused $\mathtt{tx}$ to be included in **txs**. Since Algorithm 3 processes transactions sequentially (via (Algorithm 4), it must return **false** when $\mathtt{tx}$ is processed. Looking at Algorithm 7, we need to distinguish the following two cases for $\mathtt{tx}_L$, since for all other cases, no transactions are added.

Case 1: $\mathtt{tx}_L$ is of `LockReqTX` type. In this case, we add $\mathtt{tx}_1$ and $\mathtt{tx}_2$ after $\mathtt{tx}_L$, where $\mathtt{tx}_1 = \mathsf{effect\text{-}UTxO}_{\mathcal{L}_S \to \mathcal{L}_D}(\mathtt{tx}_L)$ is of `MintTX` type and $\mathtt{tx}_2 = \mathsf{effect\text{-}UTxO}_{\mathcal{L}_D \to \mathcal{L}_D}(\mathtt{tx}_1)$ of `ReleaseTX` type. We know that $\mathtt{tx}_L \in \mathcal{L}_S$, otherwise $\textsc{verifyTx}_{\mathcal{L}_S}(L) \neq L$. Since $\mathtt{tx}_L$ is of `LockReqTX` type, it is either preceded by a `SpendTX` on $\mathcal{L}_S$, or by local transactions, which means that Algorithm 4 has a state $st \in \{\epsilon, \mathit{spend}, \mathit{abort}\}$. In either case, $\mathtt{tx}_L$ does not cause *Algorithm* 3 and assigns state $st \leftarrow \mathit{lockreq}$ (Algorithm 4). However, this means that $\mathtt{tx}_1$ cannot fail, since that only happens when $st \neq \mathit{lockreq}$ or $\mathtt{tx}_L$ has not appeared before (Algorithm 4). Similarly, since $\mathtt{tx}_1$ did not cause failure, Algorithm 4 has set the state $st \leftarrow \mathit{mint}$. However, this means that $\mathtt{tx}_2$ cannot fail, since that only happens when $st \neq \mathit{mint}$ (Algorithm 4).

Case 2: $\mathtt{tx}_L$ is of `SpendTX` type. In this case, we add $\mathtt{tx}_3$, $\mathtt{tx}_2$ and $\mathtt{tx}_1$ before $\mathtt{tx}_L$. $\mathtt{tx}_1 = \mathsf{effect\text{-}UTxO}^{-1}_{\mathcal{L}_D \to \mathcal{L}_S}(\mathtt{tx}_L)$ is of `CommitBurnTX` type, and $\mathtt{tx}_2 = \mathsf{effect\text{-}UTxO}^{-1}_{\mathcal{L}_D \to \mathcal{L}_S}(\mathtt{tx}_1)$ is of `CommitBurnTX` type. and $\mathtt{tx}_3$ is a local, auxiliary transaction on $\mathcal{L}_D$. We know that $\mathtt{tx}_L \in \mathcal{L}_S$, otherwise $\textsc{verifyTx}_{\mathcal{L}_S}(L) \neq L$. Further, since $\mathtt{tx}_L$ is of `SpendTX` type, it has an input $\mathtt{SpendTX.in}[0]$ referencing a transaction $\mathtt{tx}'$, which must be of `LockTX` type. As $\mathtt{tx}'$ comes before $\mathtt{tx}$, we know that $\mathtt{tx}'$ did not cause Algorithm 4 to fail, and thus put it in state $st \leftarrow \mathit{lock}$. Further, as `LockTX` is spending a `LockReqTX` transaction, we know from *Case 1* that a `MintTX` was added for this $\mathsf{token_id}$, which is the largest index in **txs**. The auxiliary transaction $\mathtt{tx}_3$ is spending from this `MintTX` to $\mathtt{tx}_2$. It is local to $\mathcal{L}_D$ and valid on $\mathcal{L}_D$, thus it will not cause Algorithm 4 to fail. Similarly, $\mathtt{tx}_2$ will not cause failure, since Algorithm 4 previously assigned $st \leftarrow \mathit{lock}$, instead assigning $st \leftarrow \mathit{commitBurn}$ (Algorithm 4). Then, $\mathtt{tx}_1$ does not cause failure, and instead assigns $st \leftarrow \mathit{burn}$ (Algorithm 4). Finally, this means that $\mathtt{tx}_L$ cannot fail, since that only happens when $st \neq \mathit{burn}$ or $\mathtt{tx}_1$ has not appeared before (Algorithm 4).

Both cases result in contradiction, concluding the proof. □

Algorithm 7 Plausible transaction sequence generator for $\mathcal{L}_D$. $\mathtt{tx}_2 \leftarrow \mathsf{effect\text{-}UTxO}_{\mathcal{L}_D \to \mathcal{L}_D}(\mathtt{tx}_1)$ is a syntactic sugar, where $\mathtt{tx}_2$ is the transaction following $\mathtt{tx}_1$ in the state machine defined in Algorithm 3.

function $\textsc{plausible}_{\mathcal{L}_D}(L)$
  $\mathbf{txs} \leftarrow \varepsilon$
  for $\mathtt{tx} \in L$ **do**
    $\mathbf{txs} \leftarrow \mathbf{txs} || \textsc{plausibilityMap}_{\mathcal{L}_D}(\mathtt{tx}, \mathbf{txs})$
  end for
  return txs
end function
function $\textsc{plausibilityMap}_{\mathcal{L}_D}(\mathtt{tx}, \mathbf{txs})$
  if tx is of LockReqTX type **then**
    if $\exists \mathtt{tx}' \in \mathbf{txs}$ such that $\mathtt{tx} \in \mathtt{tx}'.\mathsf{in}$ and $\mathtt{tx}'$ is of LockTX type **then**
      $\mathtt{tx}_1 = \mathsf{effect\text{-}UTxO}_{\mathcal{L}_S \to \mathcal{L}_D}(\mathtt{tx})$ ▷ Create MintTX
      $\mathtt{tx}_2 = \mathsf{effect\text{-}UTxO}_{\mathcal{L}_D \to \mathcal{L}_D}(\mathtt{tx}_1)$ ▷ Create ReleaseTX
      Sign transactions with corresponding private keys
      return $\mathtt{tx} || \mathtt{tx}_1 || \mathtt{tx}_2$
    end if
  end if
  if tx is of SpendTX type **then**
    $\mathtt{tx}_1 = \mathsf{effect\text{-}UTxO}^{-1}_{\mathcal{L}_D \to \mathcal{L}_S}(\mathtt{tx})$ ▷ Create BurnTX
    $\mathtt{tx}_2 = \mathsf{effect\text{-}UTxO}^{-1}_{\mathcal{L}_D \to \mathcal{L}_D}(\mathtt{tx}_1)$ ▷ Create CommitBurnTX
    Let $\mathtt{tx}'$ be the transaction with the largest index in **txs**, such that $\mathsf{get_token_id}(\mathtt{tx}') = \mathsf{get_token_id}(\mathtt{tx})$
    Create a transaction tx_3 spending from tx' to $tx_2.\mathsf{in}[0]$
    Sign transactions with corresponding private keys
    return $\mathtt{tx}_3 || \mathtt{tx}_2 || \mathtt{tx}_1 || \mathtt{tx}$
  end if
  return tx
end function

Lemma 4 ($\mathcal{L}_D$ failure firewall (cf. Lemma 5 of [23]))**.** *Consider any execution of the protocol described in Section 5, where persistence holds for $\mathcal{L}_S$. For all slots t for which $S_t = \{\mathcal{L}_S\}$ we have*

$$\mathsf{merge\text{-}UTxO}(\mathcal{L}_S^{\cup}[t]) \in \pi_{\mathcal{L}_S}(\mathbb{V}_{\mathsf{UTxO}})$$

Proof. From the assumption that persistence holds, there exists some party P for which $\mathcal{L}_S^P[t] = \mathcal{L}_S^{\cup}[t]$. Further, $\mathsf{merge\text{-}UTxO}(\{\mathcal{L}_S^{\cup}[t]\}) = \mathcal{L}_S^{\cup}[t]$ due to the *partitioning* property. It suffices to show that there exists some $\mathbf{txs} \in \mathbb{V}_{\mathsf{UTxO}}$ such that $\pi_{\mathcal{L}_S}(\mathbf{txs}) = \mathcal{L}_S^P[t]$. Let $\mathbf{txs} = \textsc{plausible}_{\mathcal{L}_D}(\mathcal{L}_S^P[t])$. We have that $\textsc{verifyTx}_{\mathcal{L}_S}(\mathcal{L}_S^P[t]) = \mathbf{true}$, so apply Lemma 3 to obtain that $\mathbf{txs} \in \mathbb{V}_{\mathsf{UTxO}}$.

To see that $\pi_{\mathcal{L}_S}(\mathbf{txs}) = \mathcal{L}_S^P[t]$, note that Algorithm 7 for input L includes all $\mathtt{tx} \in L$ in the same order as in its input. Furthermore, all $\mathtt{tx} \in \mathbf{txs}$ such that $\mathtt{tx} \notin L$ are excluded from the projection. □

B.2 Firewall Property During $\mathcal{L}_S$ Failure

Analogous to Section B.1, we now analyze the case, where $\mathcal{L}_S$ has suffered a failure, i.e., $S_T = \{\mathcal{L}_D\}$. Such a failure will not violate the firewall property, and we show this by defining an algorithm that, given any valid transaction sequence of $\mathcal{L}_D$, can fill the gaps with valid, "plausible" $\mathcal{L}_S$ transactions, such that the resulting sequence **txs** is a valid merged sequence with respect to $\mathbb{V}_{\mathsf{UTxO}}$ (cf. Algorithm 3). We prove this in Lemma 5, and describe the aforementioned algorithm in Algorithm 8. For a given transaction sequence L, the algorithm maps each $\mathtt{tx} \in L$ to one or more transactions, yielding **txs**. Importantly, note that we are only interested in showing the existence of **txs**, therefore Algorithm 8 does not need to be efficient, and therefore saying a private key exists is sufficient.

Lemma 5. *For all* $L \in \mathcal{T}_{\mathsf{UTxO}}$*, if* $\textsc{verifyTx}_{\mathcal{L}_D}(L) = L$ *then* $\mathbf{txs} \leftarrow \textsc{plausible}_{\mathcal{L}_S}$ *will satisfy* $\mathbf{txs} \in \mathbb{V}_{\mathsf{UTxO}}$.

Proof. Suppose towards contradiction, that $\mathbf{txs} \notin \mathbb{V}_{\mathsf{UTxO}}$, and let $\mathbf{txs}'$ be the minimum prefix of **txs** such that $\mathbf{txs}' \notin \mathbb{V}_{\mathsf{UTxO}}$. From the base property of $\mathbb{V}_{\mathsf{UTxO}}$, we know that $\mathbf{txs}' \neq \epsilon$. We set $\mathtt{tx} := \mathbf{txs}'[-1]$, and further, $\mathtt{tx}_L \in L$ be the input to $\textsc{plausibilityMap}_{\mathcal{L}_D}$ that caused $\mathtt{tx}$ to be included in **txs**. Since Algorithm 3 processes transactions sequentially (via (Algorithm 4), it must return **false** when $\mathtt{tx}$ is processed. Looking at Algorithm 8, we need to distinguish the following three cases for $\mathtt{tx}_L$, since for all other cases, no transactions are added.

Case 1: $\mathtt{tx}_L$ is of `MintTX` type. In this case, we add $\mathtt{tx}_2$ and $\mathtt{tx}_1$ before $\mathtt{tx}_L$, where $\mathtt{tx}_1 = \mathsf{effect\text{-}UTxO}^{-1}_{\mathcal{L}_S \to \mathcal{L}_D}(\mathtt{tx}_L)$ is of `LockReqTX` type, and $\mathtt{tx}_2$ is a local, auxiliary transaction on $\mathcal{L}_S$. We know that $\mathtt{tx}_L \in \mathcal{L}_D$, otherwise $\textsc{verifyTx}_{\mathcal{L}_D}(L) \neq L$. There are two cases which we distinguish in Algorithm 8. Either (i) there exists a transaction $\mathtt{tx}' \in \mathbf{txs}$ for this token id $\mathsf{get_token_id}(\mathtt{tx}_L)$, in which we select the latest such transaction, or (ii) there does not exist such a transaction. In (ii), Algorithm 4 has $st = \epsilon$ before evaluating $\mathtt{tx}_2$ and $\mathtt{tx}_1$. $\mathtt{tx}_2$ is a local transaction, which does not cause failure nor change the state. $\mathtt{tx}_1$ does not cause failure since $st = \epsilon$, and assigns $st = lockreq$ (Algorithm 4). In (i), there was a previous transaction, which must be of `SpendTX` type, since we have existential honesty of the operator set $\mathcal{O}$, which only sign a `MintTX` if a token with this token id currently does not exist on $\mathcal{L}_D$, i.e., was burnt via `BurnTX`, for which Algorithm 8 adds a `SpendTX`, as we will see in *Case 2* below. Again, $\mathtt{tx}_2$ is a local transaction, which does not cause failure nor change the state. $\mathtt{tx}_1$ does not cause failure since $st = spend$, and assigns $st = lockreq$ (Algorithm 4). Therefore, also $\mathtt{tx}_L$ cannot cause failure (Algorithm 4).

Case 2: $\mathtt{tx}_L$ is of `ReleaseTX` type. In this case, we add $\mathtt{tx}_1$ after $\mathtt{tx}_L$, where $\mathtt{tx}_1 = \mathsf{effect\text{-}UTxO}_{\mathcal{L}_D \to \mathcal{L}_S}(\mathtt{tx}_L)$ is of `LockTX` type. We know that $\mathtt{tx}_L \in \mathcal{L}_D$, otherwise $\textsc{verifyTx}_{\mathcal{L}_D}(L) \neq L$. Since $\mathtt{tx}_L$ is of `ReleaseTX` type, it has an input $\mathtt{ReleaseTX}.\mathsf{in}[0]$ referencing a transaction $\mathtt{tx}'$, which must be of `MintTX` type. As $\mathtt{tx}'$ comes before $\mathtt{tx}$, we know that $\mathtt{tx}'$ did not cause Algorithm 4 to fail, and thus put it in state $st \leftarrow mint$. This means, that $\mathtt{tx}_L$ cannot cause failure, since that only happens when $st \neq mint$ (Algorithm 4). Finally, this will assign $st \leftarrow release$, which means that $\mathtt{tx}_1$ cannot fail, as this only happens when $st \neq release$ or tx_L has not appeared before (Algorithm 4).

Case 3: $\mathtt{tx}_L$ is of `BurnTX` type. In this case, we add $\mathtt{tx}_1$ after $\mathtt{tx}_L$, where $\mathtt{tx}_1 = \mathsf{effect\text{-}UTxO}_{\mathcal{L}_D \to \mathcal{L}_S}(\mathtt{tx}_L)$ is of `SpendTX` type. We know that $\mathtt{tx}_L \in \mathcal{L}_D$, otherwise $\textsc{verifyTx}_{\mathcal{L}_D}(L) \neq L$. Since $\mathtt{tx}_L$ is of `BurnTX` type, it has an input $\mathtt{ReleaseTX}.\mathsf{in}[0]$ referencing a transaction $\mathtt{tx}'$, which must be of `CommitBurnTX` type. As $\mathtt{tx}'$ comes before $\mathtt{tx}$, we know that $\mathtt{tx}'$ did not cause Algorithm 4 to fail, and thus put it in state $st \leftarrow commitBurn$. This means, that $\mathtt{tx}_L$ cannot cause failure, since that only happens when $st \neq commitBurn$ (Algorithm 4). Finally, this will assign $st \leftarrow burn$, which

means that $\mathtt{tx}_1$ cannot fail, as this only happens when $st \neq burn$ or tx_L has not appeared before (Algorithm 4).

All of these cases result in contradiction, concluding the proof. □

Algorithm 8 Plausible transaction sequence generator for $\mathcal{L}_S$

function PLAUSIBLE$_{\mathcal{L}_S}(L)$
  txs $\leftarrow \varepsilon$
  for tx $\in L$ **do**
    txs $\leftarrow$ **txs**$||$PLAUSIBILITYMAP$_{\mathcal{L}_S}$(tx, **txs**)
  end for
  return txs
end function
function PLAUSIBILITYMAP$_{\mathcal{L}_S}$(tx, **txs**)
  if tx is of MintTX type **then**
    $\mathtt{tx}_1 = \mathsf{effect\text{-}UTxO}^{-1}_{\mathcal{L}_S \to \mathcal{L}_D}(\mathtt{tx})$ ▷ Create LockReqTX
    if $\exists \mathtt{tx}' \in$ **txs** such that **txs**.*index*(tx′) $<$ **txs**.*index*(tx) and get_token_id(tx′) = get_token_id(tx) and $\nexists \mathtt{tx}''$ such that **txs**.*index*(tx′) $<$ **txs**.*index*(tx″) $<$ **txs**.*index*(tx) and get_token_id(tx″) = get_token_id(tx) **then**
      Create a transaction tx_2 spending from tx' to tx_1.in[0]
      Sign transaction with corresponding private keys
      return $\mathtt{tx}_2||\mathtt{tx}_1||\mathtt{tx}$
    else
      Create a transaction tx_2 that mints token_id
      Sign transaction with corresponding private keys
      return $\mathtt{tx}_2||\mathtt{tx}_1||\mathtt{tx}$
    end if
  end if
  if tx is of ReleaseTX type **then**
    $\mathtt{tx}_1 = \mathsf{effect\text{-}UTxO}_{\mathcal{L}_D \to \mathcal{L}_S}(\mathtt{tx})$ ▷ Create LockTX
    Sign transaction with corresponding private keys
    return $\mathtt{tx}||\mathtt{tx}_1$
  end if
  if tx is of BurnTX type **then**
    $\mathtt{tx}_1 = \mathsf{effect\text{-}UTxO}_{\mathcal{L}_D \to \mathcal{L}_S}(\mathtt{tx})$ ▷ Create SpendTX
    Sign transaction with corresponding private keys ▷ $\mathtt{tx}_1$ can have other inputs which we can append along with $\mathtt{tx}_1$, but these do not affect Algorithm 3.
    return $\mathtt{tx}||\mathtt{tx}_1$
  end if
end function

Lemma 6 ($\mathcal{L}_S$ failure firewall)**.** *Consider any execution of the protocol described in Section 5, where persistence holds for $\mathcal{L}_D$. For all slots t for which $S_t = \{\mathcal{L}_D\}$ we have*

$$\mathsf{merge\text{-}UTxO}(\mathcal{L}_D^{\cup}[t]) \in \pi_{\mathcal{L}_D}(\mathbb{V}_{\mathsf{UTxO}})$$

Proof. This proof is analogous to Lemma 4. □

B.3 Firewall Property

Lemma 7 (Firewall)**.** *For all PPT adversaries A, the construction of Section 5 with a sound* ChainVM*, collision-resistant hash function, an EUF-CMA multisignature scheme, and existential honesty of $\mathcal{O}$ satisfies the firewall property with respect to assumptions $\mathbb{A}_S$, $\mathbb{A}_D$ with overwhelming probability in k.*

Proof. **Case 1: $S_t = \{\mathcal{L}_S, \mathcal{L}_D\}$.** In this case, we have Persistence holding for both $\mathcal{L}_S$ and $\mathcal{L}_D$ up to t, and π_{S_t} is the identity function.

Let L^* be the minimum prefix of L such that $L^* \notin \mathbb{V}_{\mathsf{UTxO}}$. By the base property of $\mathbb{V}_{\mathsf{UTxO}}$, we know that $L^* \neq \epsilon$, and let $\mathtt{tx} = L^*[-1]$.

Subcase 1a: `tx` is of `MintTX` type. Since $\mathtt{tx} \in \mathcal{L}_D^{\cup}$, there must be at least one honest party P, that has $\mathcal{L}_D^P[t] = \mathcal{L}_D^{\cup}[t]$, therefore `tx` is a valid transaction. By definition (Section 5.3), we know that `MintTX` requires an n-of-n aggregated signature $\sigma_{\mathcal{O}}$, signed by every $O \in \mathcal{O}$. Honest operators will not sign `tx`, if $\mathsf{effect}^{-1}_{\mathcal{L}_S \to \mathcal{L}_D}(\mathtt{tx}) \notin \mathcal{L}_S^{\cup}[t]$ (cf. Section 5.5). Since EX-HON holds until t, there must be one honest operator $O_h \in \mathcal{O}$, who did not contribute to $\sigma_{\mathcal{O}}$. However, since $\mathtt{tx} \in \mathcal{L}_D$ means that a valid $\sigma_{\mathcal{O}}$ is present, which we can use to win the EUF-CMA game of the multi-signature scheme.

Subcase 1b: `tx` is of `LockTX` type. Since $\mathtt{tx} \in \mathcal{L}_S^{\cup}$, there must be at least one honest party P, that has $\mathcal{L}_S^P[t] = \mathcal{L}_S^{\cup}[t]$, therefore `tx` is a valid transaction. By definition (Section 5.3), we know that `LockTX` requires an n-of-n aggregated signature $\sigma_{\mathcal{O}}$, signed by every $O \in \mathcal{O}$ and the pre-image r of the hashlock h. Since EX-HON holds until t, there must be one honest operator $O_h \in \mathcal{O}$ who did not divulge his partial signature, since honest operators do not divulge their partial signature if $\mathsf{effect}^{-1}_{\mathcal{L}_D \to \mathcal{L}_S}(\mathtt{tx}) \notin \mathcal{L}_D^{\cup}[t]$ (cf. Section 5.5). However, since $\mathtt{tx} \in \mathcal{L}_S$ means that a valid $\sigma_{\mathcal{O}}$ is present, which we can use to win the EUF-CMA game of the multi-signature scheme.

Subcase 1c: `tx` is of `SpendTX` type. Since $\mathtt{tx} \in \mathcal{L}_S^{\cup}$, there must be at least one honest party P, that has $\mathcal{L}_S^P[t] = \mathcal{L}_S^{\cup}[t]$, therefore `tx` is a valid transaction. By definition (Section 5.3), we know that `SpendTX` requires an n-of-n aggregated signature $\sigma_{\mathcal{O}}$, signed by every $O \in \mathcal{O}$, as well as the predicate $S(\mathcal{L}_S)$ holding. `SpendTX` should only happen if it was preceded by a corresponding `BurnTX`. ChainVM soundness ensures that $Q(\pi)$, where π is a proof that $\mathtt{BurnTX} \in \mathcal{L}_D$ and Q is the corresponding verifier, as instantiated in Section 5. Thus, if `tx` is of `SpendTX` and caused the failure of Algorithm 3, it means that ChainVM soundness is broken.

Subcase 1d: `tx` is of `LockReqTX` type. This transaction can only cause Algorithm 3, if the state machine of Algorithm 4 $st \notin \{\epsilon, spend, abort\}$. By definition of the transaction (Section 5.3), `LockReqTX` has a transaction local to $\mathcal{L}_S$ as input, that is not `LockTX`, or another `LockReqTX`. Thus, the state st cannot be *lockreq* or *lock*, as this would imply a `LockReqTX` or `LockTX` being the latest transaction for this token_id on $\mathcal{L}_S$. A *mint* state would imply a `LockReqTX` being the latest transaction for this token_id on $\mathcal{L}_S$. A *release* state would imply a `MintTX` being the latest transaction on $\mathcal{L}_D$, and `LockReqTX` being the latest transaction on $\mathcal{L}_S$. Similarly, *commitBurn* and *burn* states would imply `LockTX` being the latest transaction on $\mathcal{L}_S$.

Subcase 1e: `tx` is of `CommitBurnTX` type. This transaction can only cause Algorithm 3, if the state machine of Algorithm 4 $st \neq lock\}$. Similar to the previous case, by definition of the transaction (Section 5.3), `CommitBurnTX` has a transaction local to $\mathcal{L}_D$ as input, that is not mint, `BurnTX`, or another `CommitBurnTX`. Thus, the state st cannot be *mint*, *release*, *burn*, or *commitBurn*, as this would imply a `MintTX`, `ReleaseTX`, `BurnTX`, or `CommitBurnTX` being the latest transaction for this token_id on $\mathcal{L}_D$. A *spend* state would imply a `BurnTX` being the latest transaction on $\mathcal{L}_D$. A *lockreq* state would imply either no transaction or a `BurnTX` being the latest

transaction on $\mathcal{L}_D$.

Subcase 1f: `tx` is of `BurnTX`, `ReleaseTX`, or `AbortTX` type. These two transaction can only cause Algorithm 3, if the immediately preceding transaction $\texttt{tx}'$ is not of `CommitBurnTX`, `MintTX`, or `LockReqTX` type, respectively. By definition (Section 5.3), these transactions cannot be preceded on the same ledger by incorrect transactions. Similar to the previous cases, we can see that any other state st in the state machine Algorithm 4 would cause an incorrect transaction preceding `tx`, which would be rejected by the ledger verifier, $\text{VERIFYTX}_{\mathcal{L}_S}$ or $\text{VERIFYTX}_{\mathcal{L}_D}$.
Case 2: $S_t = \{\mathcal{L}_S\}$**.** By Lemma 4, we obtain $\mathsf{merge\text{-}UTxO}(\{\mathcal{L}_S^{\cup}[t]\}) \in \pi_{\mathcal{L}_S}(\mathbb{V}_{\mathsf{UTxO}})$.
Case 3: $S_t = \{\mathcal{L}_D\}$**.** By Lemma 6, we obtain $\mathsf{merge\text{-}UTxO}(\{\mathcal{L}_D^{\cup}[t]\}) \in \pi_{\mathcal{L}_D}(\mathbb{V}_{\mathsf{UTxO}})$.
Case 4: $S_t = \emptyset$**.** By the base property, we know that $\epsilon \in \mathbb{V}_{\mathsf{UTxO}}$. □

Theorem 4 follows directly from Lemmas 2 to 7.

C BitVMX instantiation of ChainVM

We now specify the internals of BitVMX, as a concrete instantiation of ChainVM as defined in Section 3. Before proceeding and for readability, we introduce some custom notation helpful for abstracting out some details.

$m \leftarrow \mathsf{MaxChRspRounds}(f)$**.** Given a function f to be (optimistically) computed, returns the maximum number of challenge-response rounds, including all possible spending paths, that might be necessary. This is used to compute as many one-time signature key pairs as needed, per operator. The possible spending paths for BitVMX are specified in [26].

$c \leftarrow \mathsf{Payout}(f, n, F)$**.** Given a function f to be (optimistically) computed, with n operators involved in the process, each providing a deposit of F coins, returns the amount c in coins that a winning prover (resp. challenger) will receive in case its computation (resp. challenge) is accepted as correct.

$c \leftarrow \mathsf{CPFP}(d)$**.** Given some fixed dust amount d used in a prior transaction, returns an amount c in coins to use as input in a subsequent Child-Pays-For-Parent transaction aimed at expediting the prior transaction to be written on-chain. Note that this is a well-known technique in Bitcoin[11].

$out \leftarrow \mathsf{Conn}(id, \mathbf{pk})$. Given some arbitrary identifier id, returns a connector UTxO out derived deterministically id, spendable via a SigScr verifiable under the public keys in $\mathbf{pk}$. Connector outputs are a well known technique[12] to ensure that only one out of several possible spending paths for a related UTxO are followed.

$rsp/\perp \leftarrow \mathsf{NextRsp}(\mathsf{f}, \mathsf{y}, \mathsf{m})$**.** Given a function f that has been optimistically computed producing a claimed result y, and given the last message m received from a challenger, returns a message rsp to include in the next response, or $\perp$ if no response is possible (the challenger wins). The possible challenge-response paths for BitVMX are introduced in [26].

$ch/\perp \leftarrow \mathsf{NextChal}(\mathsf{f}, \mathsf{y}, \mathsf{m})$**.** Given a function f that has been optimistically computed producing a claimed result y, and given the last message m received from the prover, returns a message ch to include in the next challenge, or $\perp$ if no challenge is possible (the prover wins). The possible challenge-response paths for BitVMX are introduced in [26].

[11] See, e.g., `https://bitcoinops.org/en/topics/cpfp/`. Last access, June 20th, 2025.
[12] See, e.g., `https://web.archive.org/web/20240328181345/https://www.arkpill.me/deep-dive`. Last access, June 20th, 2025

$\mathsf{LastTx}(sid)$**.** Returns the last transaction that has been posted on chain related to a BitVM execution with session identifier sid. If no transaction has been yet posted, it returns ε.

$\mathsf{Cov}(\mathcal{S})$ Some UTxOs in the execution graph require covenants [29]. Since, at the time of this writing, covenants are not natively supported in Bitcoin, we resort to emulating them. A UTxO that includes $\mathsf{Cov}(\mathcal{S}, pattern)$ requires that the spending transaction follows the pattern $pattern$ (which can be one among a set of predefined transaction types) and, additionally, a multisignature by every member of the set $\mathcal{S}$.

Winternitz OTS public keys opk_i**,** $opk^{prv}_{i,j,k}$**,** $opk^{ch}_{i,j,k}$**.** We use three sets of Winternitz key pairs. The first set is denoted with opk_i, and must be interpreted as the WOTS public key that prover i uses to claim a computation result. Since we restrict in BitVMX that each operator can only act once as prover, one such key per operator is enough. The second set is denoted with $opk^{prv}_{i,j,k}$, where $i \in [n]$, $j \in [n]$, with $i \neq j$, and $k \in [\mathsf{MaxChRspRounds}(f)]$. Keys in this set are used to encode responses in the on chain verification protocol run as part of $BitVM.Claim$. Concretely, $opk^{prv}_{i,j,k}$ is used by O_i acting as prover to encode a response to O_j acting as challenger, in round k of the challenge-response protocol. Similarly, $opk^{ch}_{i,j,k}$ is used by operator O_i acting as challenger to encode a challenge to O_j acting as prover, in round k of the challenge-response protocol.

Transaction types. A BitVMX (optimistic) computation may involve up to four types of transactions to be posted on-chain. Concretely:

- **Funding transaction,** $\mathsf{FundTx}(f, F, \mathcal{O})$. Used to collect one deposit per operator, the sum of which is eventually used to reward honest provers and challengers. We write it as $(txid, in, out, wit) \leftarrow \mathsf{FundTx}(f, F, \mathcal{O})$, where:

  $txid$**.** The transaction identifier, computed as usual (hash of in and out).

  $in \leftarrow [(dep_1, 0), \ldots, (dep_n, 0)]$ is an array of UTxOs, each with value F bitcoins, such that $(dep_i, 0)$ corresponds to operator $O_i \in \mathcal{O}$, and where $|\mathcal{O}| = n$. For simplicity, we assume they are simple P2PK UTxOs and, thus, their witnesses are just signatures.

  $out \leftarrow [(\mathsf{Cov}(\mathcal{S}, \mathsf{PayProverTx}_i))_{i\in[n]}, ((\mathsf{Cov}(\mathcal{S}, \mathsf{PayChalTx}_{i,j}))_{j\neq i\in[n]})_{i\in[n]}, (((n{+}1)d, WOTSScr_{opk_i}(x) \wedge \mathsf{Cov}(\mathcal{S}, \mathsf{StartTx}_i)))_{i\in[n]}]$. The first output, $(\mathsf{Cov}(\mathcal{S}, \mathsf{PayProverTx}_i))_{i\in[n]}$ allows $Payout(f, n, F)$ bitcoins to be given to any operator that succeeds in computing the function f, and is pre-signed in an emulated covenant; the next n outputs are locked with s script $(\mathsf{Cov}(\mathcal{S}, \mathsf{PayChalTx}_{i,j}))_{j\neq i\in[n]}$, which allows any challenger to receive $Payout(f, n, F)$ bitcoins if they successfully challenge a claim by operator i. Finally, the last n outputs just enable any of the n operators to initiate a claim as provers. The i-th output among these, requires the i-th operator to post a Winternitz OTS [11] of the claimed result, using its first registered key pair for that purpose, along with the corresponding precomputed covenant for $\mathsf{StartTx}_i$.

  $wit \leftarrow [\sigma_1, \ldots, \sigma_n]$ are the signatures witnessing correct spending of their corresponding input.

- **Start transactions,** $\mathsf{StartTx}_i(f, y)$**, for** $i \in [n]$. When operator $O_i \in \mathcal{O}$ wants to claim that y is the result of computing f (on some agreed input), posts the $(txid, in, out, wit) \leftarrow \mathsf{Start}_i(f, y)$ transaction, where:

  $txid$ The transaction identifier, computed as usual (hash of in and out).

  $in \leftarrow [(\mathsf{FundTx}(f, F, \mathcal{O}), 1 + n + i, ((n + 1)d), \cdot)]$, the i-th output within the last n outputs of $\mathsf{FundTx}(f, F, \mathcal{O})$.

out $\leftarrow [(d, \mathsf{TLockScr}(t, \bot, pk_i)), ((d, \mathsf{WOTSScr}_{opk_i}(x) \wedge \mathsf{WOTSScr}_{opk^{ch}_{j,i,1}}(x||y) \wedge \mathsf{Cov}(\mathcal{S}, \mathsf{ChTx}^1_{i,j})))_{j \neq i \in [n]}, \mathsf{Conn}((i, f, y), pk_{\mathcal{O}})]$. Here, the first output can only be spent by O_i after time t (and cannot be spent before that). The following $n-1$ outputs can each be spent by operator O_j for $j \neq i$ in order to challenge the claim posted by operator i, for which O_j needs to provide: the WOTS over x previously posted by operator O_i, a new WOTS under $opk^{ch}_{j,i,1}$ over $(x||y)$, where y is the concrete challenge posed by O_j, and the emulated covenant signature of the spending transaction (of $\mathsf{ChTx}^1_{i,j}$ type). Finally the last output is a connector output with identifier (i, f, y) spendable by any of the operators, which will be included as inputs in all emulated covenant signatures for the Payout transactions (see next), so that only one of them can be posted on chain.

wit $\leftarrow [\rho_i, \sigma_1]$ a WOTS ρ_i verifiable under opk_i, encoding the result that operator i is claiming to correspond to the agreed computation f, and the covenant precomputed signature σ_1 for $\mathsf{Cov}(\mathcal{S}, \mathsf{StartTx_i})$.

- **Challenge transactions,** $\mathsf{ChTx}^k_{i,j}(m)$. The challenge transaction posted by operator O_j acting as challenger, in response to O_i acting as prover, where the last message posted in a transaction by O_i (as a WOTS inside the witnesses of a $\mathsf{StartTx}_i$ transaction or a $\mathsf{RspTx}^{k-1}_{i,j}$ transaction) is x. Concretely, $(txid, in, out, wit) \leftarrow \mathsf{ChTx}^k_{i,j}(x)$, where:

  txid. The transaction identifier, computed as usual (hash of *in* and *out*).

  in if $k = 1$, then $in \leftarrow [(\mathsf{StartTx}_i, c, (d, \cdot)), (\cdot, \cdot, (CPFP(d), \cdot))]$, with $c = j + 1$ if $j < i$ and $c = j$ otherwise, where the first input is the c-th output of the $\mathsf{StartTx}_i$ transaction – meaning that the j-th operator is challenging a result posted by operator i – and the second input is some UTxO owned by operator j with at least $CPFP(d)$ coins, and is used to expedite the posting of $\mathsf{StartTx}_i$ on chain. For the latter, for simplicity we assume that the UTxO contained a simple P2PK script. On the other hand, if $k > 1$, then $in \leftarrow [(\mathsf{RspTx}^{k-1}_i, 0, (d, \cdot)), (\cdot, \cdot, (CPFP(d), \cdot))]$, where the first input means that operator O_j is challenging the $(k-1)$ response posted by operator O_i, and the last input can be interpreted as before.

  out $\leftarrow [\mathsf{TLockScr}(t_c, WOTSScr_{opk^{ch}_{j,i,k}}(x||y) \wedge WOTSScr_{opk^{prv}_{i,j,k}}(y||z) \wedge \mathsf{Cov}(\mathcal{S}, \mathsf{RspTx}^k_{i,j}), \mathsf{Cov}(\mathcal{S}, \mathsf{PayChalTx}^k_{i,j}))]$. That is, before a timelock t_c expires, O_i (the prover) can respond to this challenge by replaying the WOTS signed by O_j with key $opk^{ch}_{j,i,k}$, and adding a new WOTS of the response z signed with O_i's key $opk^{prv}_{i,j,k}$; and, in addition, O_j needs to provide the emulated covenants signature for the $\mathsf{RspTx}^k_{i,j}$ transaction. After the t_c timelock expires, the challenger O_j can spend this UTxO by posting a $\mathsf{PayChalTx}^k_{i,j}$ transaction, which requires an emulated covenants signature.

  wit $\leftarrow [(\rho_{i,j,k-1}, \rho_{j,i,k}), \sigma_k]$, where $\rho_{i,j,k-1}$ is the WOTS encoding the last message from O_i (either from a $\mathsf{StarTx_i}$ transaction, or a $\mathsf{Rsp^{k-1}_{i,j}}$ transaction, $\rho_{j,i,k}$ is O_j's WOTS of the response to O_i's last message, and σ_k is the emulated covenants signature for the current $\mathsf{ChTx}^k_{i,j}(m)$ transaction.

- **Response transactions,** $\mathsf{RspTx}^k_{i,j}(y)$. The response transaction posted by operator O_i acting as prover, in response to O_j acting as challenger, where the last message posted in a transaction by O_j is y. Concretely, $(txid, in, out, wit) \leftarrow \mathsf{RspTx}_{i,j}(m)$, where:

txid The transaction identifier, computed as usual (hash of *in* and *out*).

in $\leftarrow [(\mathsf{ChalTx}^{k}_{i,j}(x), 0), (\cdot, \cdot, (d, \cdot)]$. That is, the k-th response transaction by O_i connects via its first input to the k-th challenge transaction by O_j, $\mathsf{ChalTx}^{k}_{i,j}(x)$. Additionally, O_i specifies a second input with some dust d amount of bitcoins. For simplicity, we assume that this comes from a P2PK UTxO under the control of O_i.

out $\leftarrow [(WOTSScr_{opk^{prv}_{i,j,k}}(y||z) \wedge WOTSScr_{opk^{ch}_{j,i,k+1}}(z||t) \wedge \mathsf{Cov}(\mathcal{S}, \mathsf{ChalTx}^{k+1}_{i,j}))]$. That is, in order to spend the only UTxO of the transaction, O_j needs to copy the WOTS provided by O_i which includes the last response z, and add a new WOTS verifiable under the next one time key pair owned by O_j ($opk^{ch}_{j,i,k+1}$) which encodes z concatenated with the new challenge t. Additionally, O_j needs to provide a valid emulated covenant signature of $\mathsf{ChalTx}^{k+1}_{i,j}$.

wit $\leftarrow [(\rho^{ch}_{i,j,k}, \rho^{prv}_{i,j,k}), \sigma_i]$, where $\rho^{ch}_{i,j,k}$ is the WOTS provided by O_j as witness to $\mathsf{ChalTx}^{k}_{i,j}$, and which encodes the challenge y posed by O_j, and $\rho^{prv}_{i,j,k}$ is O_i's WOTS, encoding the response to the challenge, and which must be verifiable under $opk^{prv}_{i,j,k}$. The second witness, σ_i, is the emulated covenant's signature over $\mathsf{RspTx}^{k}_{i,j}(y)$.

- **Prover payout transactions,** $\mathsf{PayProverTx}_i$. When no challenger succeeds in proving the prover wrong, the latter may use a PayProverTx transaction to receive the payout. Prover payout transactions must spend the first UTxO from the FundTx (via one of the precomputed PayProverTx emulated covenants), the first UTxO from the corresponding StartTx transaction (i.e., if this is prover i, it should spend from $\mathsf{StartTx}_i$), and the connector output transaction also from the corresponding StartTx. The first input to PayProverTx ensures that no other prover can receive the reward; the second ensures that prover i can only receive the reward if it has not been successfully challenged before the predefined timelock; and finally, the third input ensures that no challenger will be able to receive a challenge reward (see the definition of PayChalTx). More concretely, the prover payout transactions are composed as follows:

  txid The transaction identifier, computed as usual (hash of *in* and *out*).

  in $\leftarrow [(\mathsf{FundTx}, 0, (Payout(f, n, F), \cdot)), (\mathsf{StartTx}_i, 0, (d, \cdot)), (\mathsf{StartTx}_i, n+1, (d, \cdot))]$. Namely, the first input connects with the funding transaction previously submitted to pay for the costs of the (optimistic) computation; the second input connects with the prover's (the i-th operator) transaction claiming a result of the computation, enabling it to claim the reward; and the third input cancels out the right for other operators to claim a challenge reward.

  out $\leftarrow [(\mathsf{Payout}(f, n, F), \mathsf{SigScr}_{pk_i}), (d, \mathsf{SigScr}_{pk_i})]$. That is, there are two outputs. The first sends $\mathsf{Payout}(f, n, F)$ coins to a P2PK script, unlockable only by the owner of pk_i – the i-th operator. The second output sends some dust value d also to the owner of pk_i. The purpose of the second output is to facilitate binding the result of the BitVMX computation with higher-level logic – i.e., acceptance of conditional flows on related protocols in case of a BitVMX computation producing some predefined value. In settings such as the bridge protocol in the main body of the paper, where the posting of a PayProverTx transactions is required to automatically enable the posting of another subsequent transaction, we need to replace the second spending script by a covenant enabling only the follow up transaction – e.g., SpendTX in our case – $\mathsf{Cov}(\mathcal{S}, \mathtt{SpendTX}_{i,\mathsf{gid}})$.

wit $\leftarrow [\sigma_0, \sigma_1, \sigma_2]$, where the first witness is an emulated covenants signature of the spending transaction $\mathsf{PayProverTx}_i$, and the rest are signatures by the the prover.

- **Challenger payout transactions,** $\mathsf{PayChalTx}^k_{i,j}$. When a challenger succeeds in challenging a proof (i.e., the prover fails to provide a valid response in time), the challenger can claim a payout. The structure of challenger payout transactions is similar to prover payout transactions. Concretely:

  txid The transaction identifier, computed as usual (hash of *in* and *out*).

  in $\leftarrow [(\mathsf{ChalTx}^k_{i,j}, 0, (d, \cdot)), (\mathsf{FundTx}, c, (Payout(f, n, F), \cdot)), (\mathsf{StartTx}_i, n+1, (d, \cdot))]$. Namely, the first input connect the challenge transaction won by the challenger (operator j); the second input connects with the funding transaction via the c-th output, where $c = j + 1$ if $j < i$ and $c = j$ otherwise, enabling it to claim the reward; and the third input cancels out the right for the prover to claim a reward.

  out $\leftarrow [(\mathsf{Payout}(f, n, F), \mathsf{SigScr}_{pk_j})]$. That is, there is only one output, sending $\mathsf{Payout}(f, n, F)$ coins to a P2PK script, unlockable only by the owner of pk_j – the j-th operator.

  wit $\leftarrow [\sigma_0, \sigma_1, \sigma_2]$, where the first witness is an emulated covenants signature of the spending transaction $\mathsf{PayChalTx}^k_{i,j}$, and the rest are signatures by the the challenger.

C.1 BitVMX blockchain protocols

We now proceed to specify in full detail the $(Setup, Claim)$ blockchain protocols that compose the BitVMX instantiation of BitVM Core. For readability, we use the terms introduced in the previous paragraphs instead of explicitly stating the contents of each transaction, or concrete functions.

BitVMX (1/2)

$\mathsf{Setup}(1^\kappa, Q)$

1. // $\mathcal{S}$, $\mathcal{O}$, Sig, $WOTS$, n, $proc$, $[\mathsf{depTx}_i]_{i\in[|\mathcal{O}|]}$, F,
2. // and t are globally available
3. $m \leftarrow \mathsf{MaxChRspRounds}(proc)$
4. **for** $i \in [|\mathcal{O}|]$: // Generate WOTS key pairs (can be done externally)
5. $\quad pk_i \leftarrow Sig.KG(1^\kappa)$; $opk_i \leftarrow WOTS.KG(1^\kappa)$
6. $\quad$ **for** $j \in [|\mathcal{O}|], j \neq i \wedge k \in [m]$
7. $\quad\quad opk^{prv}_{i,j,k} \leftarrow WOTS.KG(1^\kappa)$; $opk^{ch}_{i,j,k} \leftarrow WOTS.KG(1^\kappa)$
8. Precompute $\mathsf{FundTx}(proc, F, \mathcal{O})$, $\mathsf{StartTx}_i(proc, y)$, $\mathsf{ChTx}^k_{i,j}(m)$, $\mathsf{Rsp}^k_{i,j}(y)$, $\mathsf{PayProverTx}_i$, and $\mathsf{PayChalTx}^k_{i,j}$ as specified above.
9. // Obtain emulated covenants signatures from $\mathcal{S}$
10. Send $\mathsf{StartTx}_i(proc, y)$, $\mathsf{ChTx}^k_{i,j}(m)$, $\mathsf{Rsp}^k_{i,j}(y)$, $\mathsf{PayProverTx}_i$, and $\mathsf{PayChalTx}^k_{i,j}$ to all $\mathcal{S}$
11. Wait until receiving $\sigma_{\mathsf{StartTx}_i}$, $\sigma_{\mathsf{ChTx}^k_{i,j}}$, $\sigma_{\mathsf{RspTx}^k_{i,j}}$, $\sigma_{\mathsf{PayProverTx}_i}$, and $\sigma_{\mathsf{PayChalTx}^k_{i,j}}$ back
12. // Partially fill the witnesses of all emulated
13. // covenant transactions

14. **for** $i \in [|\mathcal{O}|]$:
15. $\quad \mathsf{StartTx}_i.wit[1] \leftarrow \sigma_{\mathsf{StartTx}_i}$
16. $\quad \mathsf{PayProverTx}_i.wit[0] \leftarrow \sigma_{\mathsf{PayProverTx}_i}$
17. $\quad$ **for** $(j \neq i, k) \in [n] \times [m]$:
18. $\quad\quad \mathsf{ChTx}^k_{i,j}(m).wit[0] \leftarrow \sigma_{\mathsf{ChTx}^k_{i,j}}$
19. $\quad\quad \mathsf{RspTx}^k_{i,j}(m).wit[0] \leftarrow \sigma_{\mathsf{RspTx}^k_{i,j}}$
20. $\quad\quad \mathsf{PayChalTx}^k_{i,j}(m).wit[0] \leftarrow \sigma_{\mathsf{PayChalTx}^k_{i,j}}$
21. $S := \exists i \in [|\mathcal{O}|]\ s.t.\ \mathsf{PayProverTx}_i \in \Pi_{\mathcal{L}}.\mathsf{read}()$
22. $sid \leftarrow \mathsf{Hash}(\{opk_i\}_{i\in[|\mathcal{O}|]})$
23. $aux \leftarrow (\mathsf{FundTx}(proc, F, \mathcal{O}), (\mathsf{StartTx}_i, \mathsf{PayProverTx}_i)_{i\in[n]},$
24. $\quad\quad (\mathsf{ChTx}^k_{i,j}, \mathsf{RspTx}^k_{i,j}, \mathsf{PayChalTx}^k_{i,j})_{i,j\neq i,k\in[n]\times[n]\times[m]})$
25. $\Pi_{\mathcal{L}}.\mathsf{write}(\mathsf{FundTx}(proc, F, \mathcal{O})$
26. $return\ (sid, S, aux)$

BitVMX (2/2)

$\underline{\mathsf{Execute}(sid, \pi, aux)}$

1. // Let P's index be p, and C's be c. $p \neq c \in [n]$
2. $(\mathsf{FundTx}(proc, F, \mathcal{O}), (\mathsf{StartTx}_i, \mathsf{PayProverTx}_i)_{i\in[n]},$
3. $\quad (\mathsf{ChTx}^k_{i,j}, \mathsf{RspTx}^k_{i,j}, \mathsf{PayChalTx}^k_{i,j})_{i,j\neq i,k\in[n]\times[n]\times[m]}) \leftarrow aux$
4. **while true**:
5. $\quad$ **if** Prover:
6. $\quad\quad L^P \leftarrow \Pi_{\mathcal{L}}.\mathsf{read}()$
7. $\quad\quad$ **if** $\mathsf{LastTx}(sid) = \varepsilon$: **return** $\perp$ // Nothing to prove.
8. $\quad\quad$ **else if** $\mathsf{LastTx}(sid) = \mathsf{FundTx}(proc, F, \mathcal{O})$: // Start claim
9. $\quad\quad\quad \rho_p \leftarrow OTS.Sign(osk_p, \pi)$ // Note: Need access to osk_p!
10. $\quad\quad\quad \mathsf{StartTx}_p.wit[0] \leftarrow \rho_p$; $\Pi_{\mathcal{L}}.\mathsf{write}(\mathsf{StartTx}_p)$
11. $\quad\quad$ **else if** $\mathsf{LastTx}(sid) = \mathsf{StartTx}_i$ **or** $\mathsf{LastTx}(sid) = \mathsf{RspTx}^k_{p,\cdot}$ **and** t is expired: // Prover wins
12. $\quad\quad\quad \sigma_1 \leftarrow Sig.Sign(\mathsf{PayProverTx}_p, sk_p)$, $\sigma_2 \leftarrow Sig.Sign(\mathsf{PayProverTx}_p, sk_p)$
13. $\quad\quad\quad \mathsf{PayProverTx}_p.wit[1,2] \leftarrow [\sigma_1, \sigma_2]$; $\Pi_{\mathcal{L}}.\mathsf{write}(\mathsf{PayProverTx}_p)$
14. $\quad\quad\quad$ **return** $CurrentRound(\Pi_{\mathcal{L}})$ // Return current round of ledger
15. $\quad\quad$ **if** $\mathsf{LastTx}(sid) = \mathsf{ChTx}^k_{p,c}$: // Reply challenge
16. $\quad\quad\quad \rho^{ch}_{p,c,k} \leftarrow \mathsf{ChTx}^k_{p,c}.wit[0][1]$ // Challenger message
17. $\quad\quad\quad m \leftarrow \mathsf{NextRsp}(proc, \rho_p, \rho^{ch}_{p,c,k})$
18. $\quad\quad\quad$ **if** $m = \perp$: **return** $\perp$

19. $\rho^{prv}_{p,c,k} \leftarrow OTS.Sign(osk^{prv}_{p,c,k}, m)$ // Note: Need access to $osk^{prv}_{p,c,k}$!
20. $\mathsf{RspTx}^k_{p,c}.wit[0] \leftarrow (\rho^{ch}_{p,c,k}, \rho^{prv}_{p,c,k})$; $\Pi_{\mathcal{L}}.\mathsf{write}(\mathsf{RspTx}^k_{p,c})$
21. **else** Challenger:
22. $L^C \leftarrow \Pi_{\mathcal{L}}.\mathsf{read}()$
23. **if** $\mathsf{LastTx}(sid) = \mathsf{StartTx}_p$: // Last message is a claim: check if challengeable
24. $\rho_p \leftarrow \mathsf{StartTx}_p.wit[0]$ // Prover's claim
25. $m \leftarrow \mathsf{NextChal}(proc, \rho_p, \rho_p)$
26. **if** $m = \bot$: **continue**
27. $\rho^{ch}_{p,c,1} \leftarrow OTS.Sign(osk^{ch}_{p,c,1}, m)$ // Note: Need access to $osk^{ch}_{p,c,1}$!
28. $\mathsf{ChTx}^1_{p,c}.wit[0] \leftarrow (\rho_p, \rho^{ch}_{p,c,1})$; $\Pi_{\mathcal{L}}.\mathsf{write}(\mathsf{ChTx}^1_{p_c})$
29. **if** $\mathsf{LastTx}(sid) = \mathsf{RspTx}^k_{p,c}$: // Last message is a response: check if challengeable
30. $\rho^{prv}_{p,c,k} \leftarrow \mathsf{RspTx}^k_{p,c}.wit[1]$ // Prover's claim
31. $m \leftarrow \mathsf{NextChal}(proc, \rho_p, \rho^{prv}_{p,c,k})$
32. **if** $m = \bot$: **continue**
33. $\rho^{ch}_{p,c,k+1} \leftarrow OTS.Sign(osk^{ch}_{p,c,k+1}, m)$ // Note: Need access to $osk^{ch}_{p,c,1}$!
34. $\mathsf{ChTx}^k_{p,c}.wit[0] \leftarrow (\rho^{prv}_{p,c,k}, \rho^{ch}_{p,c,k+1})$; $\Pi_{\mathcal{L}}.\mathsf{write}(\mathsf{ChTx}^k_{p_c})$
35. **if** $\mathsf{LastTx}(sid) = \mathsf{ChalTx}^k_{p,c}$ **and** t_c is expired: // Challenge goes un-responded
36. $\sigma_1 \leftarrow Sig.Sign(\mathsf{PayChalTx}^k_{p,c}, sk_c)$, $\sigma_2 \leftarrow Sig.Sign(\mathsf{PayChalTx}^k_{p,c}, sk_c)$
37. $\mathsf{PayChalTx}^k_{p,c}.wit[1,2] \leftarrow [\sigma_1, \sigma_2]$; $\Pi_{\mathcal{L}}.\mathsf{write}(\mathsf{PayChalTx}^k_{p,c})$
38. **return** $\bot$

C.1.1 Reducing overhead blow-up via selector transactions

While the description we give above of BitVMX can be seen as a specification of its canonical behavior, there are many optimizations that can be applied depending on concrete needs. One such particular and very relevant optimization in our setting allows us to select one among many possible spending paths by the protocol invoking BitVMX. Simplifying, assume we want to prove computation $f(a, b)$, where $a \in [n]$, and depending on the value of a we want to only enable one concrete transaction spending the dust output of $\mathsf{PayProverTx}$, $\mathsf{PayProverTx}.out[1]$. A naive way to achieve this is creating n copies of the same BitVMX instance, where each of the n possible values of parameter a requires a signature by a prefixed WOTS key pair (enforced via covenants). While this works, this has a multiplicative overhead of factor n over the full BitVMX costs, as we need to create n copies of the same BitVMX instance.

Instead of doing this, an interesting optimization introduced by Fairgate for these cases is what we call "selector transactions". In a nutshell, lets call "selector parameters" these input parameters that select one among many spending paths. For such a selector parameter, we create a new dust-valued output – the selector output – in the $\mathsf{StartTx}$ transaction posted by the prover. The spending script for this selector output in $\mathsf{StartTx}$ enforces, in a covenant manner, that a WOTS is provided encoding the value chosen for the selector parameter, and that the transaction spending the selector output just forwards it into a new UTxO requiring the same WOTS to be provided as witness. In addition, the BitVMX computation layer is modified to parse the value for the selector parameter

from this new selector transaction, as opposed to extracting it from the witnesses provided in StartTx. Now, observe that, in the case of emulated covenants, we need n copies of the selector transaction – one per possible value that the selector parameter may take. But now we only need n copies of one transaction, as opposed to n copies of the full BitVMX execution graph – as in the naive approach. Yet another modification to the vanilla specification of BitVMX in this case is that we need to add one UTxO to the outputs of the BitVMX protocol. That is, in the case of a successful computation, the vanilla BitVMX protocol produces as output the two UTxOs specified as outputs in the PayProverTx transactions. But now, we also have to consider the forwarded output in the selector transaction as an output of the full BitVMX execution.