Clementine: A Collateral-Efficient, Trust-Minimized, and Scalable Bitcoin Bridge

Ekrem Bal[1] Lukas Aumayr[2,3] Atacan İyidoğan[1] Giulia Scaffino[2,4]
Hakan Karakuş[1] Cengiz Eray Aslan[1] Orfeas Stefanos Thyfronitis Litos[2,5]

[1] Citrea
[2] Common Prefix
[3] University of Edinburgh
[4] TU Wien
[5] Imperial College London

Abstract. This whitepaper introduces Clementine, a secure, collateral-efficient, trust-minimized, and scalable Bitcoin bridge based on BitVM2 that enables withdrawals from rollups or other side systems to Bitcoin. Clementine proposes a new Bitcoin light client that remains secure against adversaries controlling less than 50% of Bitcoin's hash rate, assuming at least one honest Watchtower in a permissioned set. The protocol is collateral-efficient, reusing locked funds over time and reducing unnecessary dust outputs through the strategic use of 0-value outputs, and scalable, enabling a single challenge per Operator to slash multiple misbehaviors. This increases throughput and reduces on-chain load without compromising security. Clementine enables trust-minimized and efficient peg-outs from Citrea to Bitcoin, making zk-rollups on Bitcoin practical and unlocking new paths for native scalability and interoperability.

1 Introduction

Since the inception of Bitcoin [29], hundreds of new blockchains have been deployed, each one with its own native coins and features, including programmability, privacy, throughput, or latency. Today's blockchain ecosystem is very diverse. Interestingly, Bitcoin has remained the most successful chain, dominating the market in many important metrics such as total value locked (TVL) [12] and security stemming from Proof-of-Work (PoW) [15]. Bitcoin's importance and inherent value have also been recently acknowledged by economists, banks, and governments [8, 11, 26]. The Bitcoin protocol, introduced by Satoshi Nakamoto, enables a peer-to-peer payment system, but by design, it only supports a very limited scripting language. While this design choice works perfectly for rather simple money transfers, it has limited Bitcoin's use cases and hindered its interoperability and scalability. Specifically, while protocols like bridges, DeFi, and rollups thrive in the EVM ecosystems, the restricted language of Bitcoin has prevented such applications from flourishing.

Related Work. Recently, BitVM1 [13] has been a major breakthrough for Bitcoin. BitVM enables parties to run, for the first time, practical execution of arbitrary computations without modifying Bitcoin's scripting language or relying on additional trust assumptions, such as trusted hardware, trusted third parties, or committees with some threshold honesty assumption. It uses a custom virtual machine to execute Turing-complete programs off-chain, producing a verifiable execution trace. If both parties agree on the result, the protocol is completed in just three on-chain transactions. In case of dispute, a bisection game narrows down the disagreement, and a single computational step is verified on-chain within Bitcoin Script. BitVM1's main limitations are (i) its reliance on one dedicated party, the Verifier, to check the computation and (ii) a worst-case overhead of up to 79 on-chain transactions, which can result in long delays due to per-transaction challenge windows (for instance, 79 days if the window is 1 day).

To overcome these limitations, a subsequent work introduces BitVM2 [14], where the program is split into sequential chunks and executed off-chain. The Prover commits to the inputs, the final outcome of the computation, and, if challenged, to all intermediate results. *Anyone* can then penalize the Prover for incorrect computation by triggering on-chain execution of the specific chunk that first led to a faulty result. BitVM2 requires fewer on-chain transactions and makes fewer assumptions about the participants, as it does not rely on a specific party for verification; anyone can challenge. The paper presents *BitVM Bridge*, utilizing BitVM2 to build a bridge.

Finally, BitVMX [28] improves over BitVM1 in terms of efficiency: instead of Merkle trees, it uses hash chains of program traces and other optimizations. BitVMX makes running any user-defined program efficient and practical but still requires dedicated parties for verification.

Citrea Rollup and Clementine Bridge. These advancements finally pave the way to deploy a variety of new applications in Bitcoin, including interoperability and scalability protocols. In the evolving Bitcoin ecosystem, Citrea [1] emerges as the *first rollup protocol built on top of Bitcoin.* It enhances Bitcoin's scalability by executing transactions off-chain while periodically checkpointing its state onto the Bitcoin base layer, which guarantees both data availability and settlement (a.k.a. finality). Citrea is fully *EVM-compatible*: it features a deterministic, stack-based virtual machine, equivalent to the Ethereum Virtual Machine (EVM), that operates on Bitcoin. Citrea uses cBTC as its native asset, which represents BTC bridged to Citrea. A key aspect of Citrea is its custom zkEVM, a scalable and trustless proof system based on zk-STARKs, which makes the full virtual machine's execution provable. Securely transferring funds from Bitcoin to Citrea (*peg in*) is straightforward thanks to the *quasi-Turing complete*[6] scripting capabilities of Citrea that allow it to run a Bitcoin light client in the rollup. Still, trustlessly and securely moving funds back from Citrea to Bitcoin (*peg out*) is non-trivial due to the limited scripting language supported by Bitcoin [18, 27]. A fundamental component of Citrea is the Clementine bridge, which allows users to exit the rollup (*peg out*) and get back their coins into Bitcoin in a secure, efficient, scalable, and trust-minimized manner.

Notably, Clementine features crucial improvements when compared to state-of-the-art protocols such as BitVM Bridge [14]. First, Clementine runs the first provably secure Bitcoin light client on Bitcoin, achieving *resilience against an adversary with less than* 50% *of the computational power* in the presence of a permissioned set of watchtowers, of which at least one is assumed to be honest. Then, Clementine is *collateral efficient*: while BitVM Bridge requires Operators to lock separate collateral for each deposit, Clementine minimizes the collateral that an Operator needs to lock by allowing collateral re-use across *all deposits.* Finally, Clementine is more *scalable*: it increases the *throughput* by allowing a single challenge to slash any malicious activity an adversarial Operator attempts. This contrasts with BitVM Bridge, which requires the challenge of every incorrect reimbursement claim of the Operator to slash. This vastly decreases the on-chain overhead and addresses some security concerns with BitVM Bridge, namely, that an adversarial Operator can cause digestions by forcing multiple, potentially block-filling challenges.

After giving a high-level overview of the Clementine protocol in Section 2 and presenting the necessary building blocks in Section 3, in this whitepaper, we make the following contributions:

– We define the models and assumptions that we use for Clementine security (Section 4) and we describe the design of Clementine (Section 5), the scalable, collateral-efficient, trust-minimized bridge of Citrea built on top of Bitcoin. We proceed by gradually introducing the different components of the protocol, e.g., the parties and their roles, the light client, and the payoff rounds.
– We formally define all the transactions (Section 6) of the Clementine protocol by showing their inputs, outputs, and witnesses in a compact table. We further expand on the logic of the zkSNARK proofs used by the protocol.
– We put forth the pseudocodes of Clementine (Section 7) protocol divided into the different phases: *Setup*, *Peg In*, and *Peg Out* of the bridge.
– We discuss the security of Clementine through a case analysis (Section 9). We prove that an honest Operator can always enforce the correct outcome, while a malicious Operator cannot enforce an incorrect outcome. We further prove the security holds for an adversary controlling less than 50% of the hash rate. We finally compute the failure probabilities for some concrete parameter settings.

2 Clementine: Protocol Overview

Clementine is a trust-minimized bridge used by the Citrea rollup on Bitcoin. Clementine relies on 5 sets of parties: Operators, Watchtowers, Challengers, Signers, and, of course, Users. Clementine allows

[6]This term is adopted in the blockchain community and indicates Turing-complete languages that bound the execution (e.g., Ethereum's gas consumption) and thus guarantee termination [30].

users to move their funds from Bitcoin to Citrea and back to Bitcoin again, in a seamless and trust-minimized way. Consider a user, Alice, that wants to move her 2 BTC to Citrea: she locks 2 BTC in a deposit transaction and she posts it onto Bitcoin. Upon seeing the deposit of Alice, the bridge Operators (or Alice herself) relay the request to a light client smart contract on the Citrea side which mints 2 cBTC and sends them to Alice. This mechanism, known as *peg in*, is rather simple thanks to the quasi-Turing complete scripting capabilities of Citrea. On the contrary, the opposite direction, known as *peg out*, is not trivial to achieve due to the limited scripting capabilities of Bitcoin, which does not permit deploying a light client contract of Bitcoin itself on the Bitcoin chain.

To allow users to securely peg out of the Citrea rollup and get back their money in Bitcoin, Clementine needs to guarantee users that, upon burning some BTC on Citrea, they get an equivalent amount of BTC back on Bitcoin. We achieve this by incentivizing Operators to front the money to users and then get reliably reimbursed thanks to the following components of Clementine:

– An *optimistic Bitcoin light client* which checks whether the Citrea rollup has checkpointed a state to Bitcoin that includes the users' burn transaction, as well as whether the *payout* transaction, in which Operator fronts the money to the user, is on-chain. This is done optimistically, with the Operator committing to a Bitcoin chain that it claims is the *canonical* one. A timer then starts, within which any Watchtower can dispute the claim. If the timeout expires without a dispute, then the Operator is reimbursed. Otherwise, if Watchtowers challenge the Operators' commitment, the light client logic must be executed to determine which is the canonical chain.
– A *BitVM program* executes the light client logic, with Challengers ensuring the correct outcome of the computation is enforced on-chain. The client logic consists of a recursive zkSNARK verifier: First, when disputing the canonical chain of the Operator, a Watchtower posts a transaction and reveals a signed zkSNARK proof showing that the Watchtower knows a valid chain with certain restrictions. Then, the Operator takes this proof and feeds it into another zkSNARK verifier which, given the Watchtower's proof, the payout transaction the Operator published on Bitcoin, the Operator's chain and necessary components of Citrea, outputs a proof proving that the Operator has the canonical chain that includes the correct payout transaction, and it is therefore entitled to be reimbursed.
– A *payoff round* mechanism which minimizes the Operator's collateral and makes it reusable, while retaining economic safety of the bridge.

3 Preliminaries and Key Building Blocks

3.1 The UTXO Model

On a blockchain, each user U is identified by the public key of a digital signature key pair $(\mathsf{pk}_U, \mathsf{sk}_U)$, proving ownership over a coin. In the Unspent Transaction Output (UTXO) model, a UTXO object holds some units of currency, *coins*, and a set of instructions that specify the requirements in order to spend those coins, e.g., the signature corresponding to which public key can spend the coins. A transaction tx is an atomic update of the state of the blockchain and maps a non-empty list of inputs, i.e., unspent outputs, to a non-empty list of newly created outputs. In other words, a transaction consumes some UTXOs and creates new UTXOs. In Bitcoin, a transaction includes (i) inputs, which uniquely identify unspent outputs, (ii) outputs, which specify the amount of currency held by the output and the conditions under which the coins can be spent, and (iii) witnesses, which store the data fulfilling the spending conditions of the inputs.

3.2 Emulating Covenants Using Presigned Transactions

In this work, we explore Bitcoin *covenants* [10, 16], a *proposed* class of script operators that enable a transaction to impose constraints on how its funds can be utilized by subsequent spending transactions. Covenants can also be recursive, requiring the script of the spending transaction to include the same covenant as the spent one. At the time of writing, covenants have not yet been integrated in Bitcoin. If added to Bitcoin Script, covenants would provide the ability to restrict the outputs of future transactions and would enable the storage and execution of a state machine across different transactions. This would equip Bitcoin with more expressive smart contract capabilities and it would enable more complex applications on Bitcoin.

In Clementine, the way some UTXOs can be spent must be restricted so that operators spending from Clementine transactions can be challenged. Without Bitcoin-native covenants, we closely follow [14] and emulate the covenant functionality using a committee of n signers $S_1, \dots S_n$, where at least one is honest (existential honesty assumption). In practice, anyone is allowed to join the committee as long as they are active (idle signers are kicked out of the committee to avoid denial of service); honest users could convince themselves of the existential honesty by joining the committee [21].

Specifically, during the setup phase of the Clementine bridge, we introduce the following steps:

– Each signer generates a fresh key pair.
– For each transaction output that should be spendable only under specific conditions, we add an n-of-n multi-signature spending condition, requiring all signers to collaboratively produce the signature in order to spend the UTXO.
– The signers pre-sign the specific transactions intended for spending the output. Each operator is given a dedicated set of pre-signed transactions, which they can spend under certain conditions.
– Finally, the signers erase their keys.

This mechanism guarantees that as long as at least one signer remains honest and deletes their key, the UTXOs can only be spent using one of the pre-signed transactions, i.e., only in the pre-defined way. One can leverage signature aggregation schemes to reduce the on-chain footprint. For readability and to highlight that enabling covenants can eliminate the need for the committee, we abstract this emulation and henceforth use $\mathsf{CheckCovenant}$ to refer to a spending condition on outputs.

3.3 Winternitz One-Time Signatures

A Winternitz one-time signature scheme [19] is a hash-based signature scheme. Let n be a security parameter, w the compression level, and m the message to be signed. A Winternitz one-time signature scheme consists of three algorithms:

– $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{WintKeyGen}(n, l)$ is a Probabilistic Polynomial Time (PPT) algorithm that takes as input a positive integer n and returns a key pair $(\mathsf{pk}, \mathsf{sk})$, consisting of a secret key $\mathsf{sk} = (\mathsf{sk}_1, \dots, \mathsf{sk}_l)$ of l n-bit strings chosen uniformly at random and a public key $\mathsf{pk} = (\mathsf{pk}_1, \dots, \mathsf{pk}_l)$ obtained by hashing w times the l strings of the secret key to obtain another set of l n-bit numbers. Usually, $l = 32$ and $n = 256$.
– $\sigma \leftarrow \mathsf{WintSig}(\mathsf{sk}, m)$ is a Deterministic Polynomial Time (DPT) algorithm parameterized by a secret key sk, that takes as input a message m and outputs the signature $\sigma = (\sigma_1, \dots, \sigma_l)$, which we also call (Winternitz) commitment. To generate σ, m is hashed to produce a n-bit digest.
– $\{\mathsf{True}, \mathsf{False}\} \leftarrow \mathsf{VerifyWintSig}(\mathsf{pk}, m, \sigma)$ is a DPT algorithm parameterized by a public key pk that takes as input a message m, a signature σ, and outputs True iff σ is a valid signature for m generated by the secret key sk, corresponding to pk, i.e., $(\mathsf{pk}, \mathsf{sk})$ is a key pair generated by $\mathsf{WintKeyGen}$.

The Winternitz signature scheme admits several optimizations. For example, Citrea derives all secret keys from a single master secret by appending a unique index to it and then applying a cryptographic hash function [2, 24]. This technique enables deterministic generation of all required secret keys from a single seed, eliminating the need to store multiple keys.

A crucial property of a Winternitz signature is that it is *one time*: When the message m is signed and σ is created, the key pair becomes bound to m. If the same key pair is then used to sign a message $m' \neq m$ the unforgeability property of the scheme is lost, and so is its security.

Importantly, Winternitz signature verification can be implemented with Bitcoin script, i.e., we can verify a Winternitz commitment provided in the witness of a Bitcoin transaction.

3.4 Succinct Non-Interactive Arguments (SNARGs)

Similar to [14], we make use of SNARGs and base our definitions on [23, 25]. We denote $R \leftarrow \mathcal{R}(\lambda)$ a relation generator that takes as input a security parameter λ and returns a polynomial time decidable binary relation R. We call ϕ the statement and w the witness for the pairs $(\phi, w) \in R$. An efficient publicly verifiable non-interactive argument for $\mathcal{R}$ consists of three PPT algorithms:

- $crs \leftarrow \mathsf{SNARG.Setup}(R)$: Given as input a relation R, it generates a common reference string crs.
- $\pi \leftarrow \mathsf{SNARG.Prove}(R, crs, \phi, w)$: Given as input a relation R, a common reference string crs, and $(\phi, w) \in R$, it generates an argument π.
- $\{0, 1\} \leftarrow \mathsf{SNARG.Vrfy}(R, crs, \phi, \pi)$: Given as input a relation R, a common reference string crs, a statement ϕ, and an argument π, it returns 1 if π is a valid argument, and 0 otherwise.

The SNARG tuple $(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Vrfy})$ is a non-interactive argument for $\mathcal{R}$ if it has *perfect completeness* and *computational soundness*, as defined in [23, 25]. On a high level, these two properties guarantee that: (i) For a true statement ϕ, an honest prover can convince an honest verifier except with negligible probability (completeness), and (ii) for a false statement ϕ, a prover cannot convince an honest verifier except with negligible probability (soundness). If the verifier runs in polynomial time in $\lambda + |\phi|$ and the proof size is polynomial in λ, we denote the scheme as a succinct non-interactive argument (SNARG). Later, we make use of succinct non-interactive arguments of knowledge (SNARKs) implementations, such as [25]. However, we neither care nor use the zero-knowledge property of this scheme.

3.5 BitVM Core

We make use of the BitVM core construction, as explained in [14]. This protocol allows us for some quasi-Turing complete function (or program) f, to assert on Bitcoin that $f(x) = y$. Observe that we can represent any quasi-Turing complete program in Bitcoin Script. The main problem is the resulting program size, which is likely to be too large to fit into a Bitcoin transaction, which can be at most of 4MB.

The main idea of BitVM core is to split such a program f into k sub-programs $f_1, \ldots, f_k$, such that each f_i is small enough to fit into a Bitcoin transaction. The prover can then execute these off-chain:

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

The prover then posts a Winternitz commitment of every z_i for $i \in \{0, \ldots, k\}$ on-chain. The verifier can check these results off-chain, and should there be an incorrect result $z_i \neq f_i(z_{i-1})$ for some $i \in \{1, \ldots, k\}$, the verifier can execute $f_i(z_{i-1})$ on-chain and disprove the claim of the prover, that $f(x) = y$. One can use this construction to build a SNARG verifier (cf. Section 3.4 $\mathsf{SNARG.Vrfy}$), i.e., a Groth16 verifier [25].

4 Model and Assumptions

Time and Network Model. We model time to proceed in discrete rounds. Further, we assume a synchronous network, i.e., messages sent from one honest party to another are guaranteed to be delivered after a delay with a known upper bound δ. We assume there are authenticated communication channels between all parties of the protocol; in Section 8 we will discuss how to optimize communication.

Cryptographic Assumptions. We consider hash functions modeled as random oracles and digital signature schemes having Existential Unforgeability under Chosen Message Attack (EUF-CMA) security.

Ledger Model. The ledger is the total order of transaction output by a Proof-of-Work consensus protocol that is run in the dynamic population model, i.e., the dynamic difficulty model. We consider a *secure* ledger, i.e., a ledger that is *safe and live.* Safety and liveness are defined as follows:

Definition 1 (Safety). *A distributed ledger protocol is* safe *if it fulfills the following properties:*

Self-consistency *For any correct party P and any rounds $r_1 \leq r_2$, it holds that $\mathcal{L}^P_{r_1} \preceq \mathcal{L}^P_{r_2}$.*
View-consistency *For any correct parties P_1, P_2 and any round r, it holds that either $\mathcal{L}^{P_1}_r \preceq \mathcal{L}^{P_2}_r$ or $\mathcal{L}^{P_2}_r \preceq \mathcal{L}^{P_1}_r$.*

Definition 2 (Liveness). *A distributed ledger protocol is* live *with liveness parameter u if all transactions written by any correct party at round r, appear in the ledgers of all correct parties by round $r + u$.*

Clementine Model and Parties. Clementine is designed as an on-chain multi-party protocol executed by mutually distrusted participants: Operators, Watchtowers, Challengers, Signers, and users. Operators, Watchtowers, and Signers are known, fixed sets of parties (*permissioned* set). Although these sets comprise a well-known number of identified parties, they can evolve over time under certain circumstances: we discuss this in the adversarial model and in Section 8. Anyone can be a Challenger or user of the protocol (*permissionless* set). We assume that Operators, Watchtowers, and Challengers run a full node of Bitcoin. Operators are assumed to run a full node of the Citrea rollup and both Operators and Challengers are required to have an account (key pair) in Citrea. The protocol involves two ledgers, which are assumed to be *safe* and *live.* More specifically, the protocol involves Bitcoin and Citrea.

Adversarial Model. The protocol is executed in the presence of a single, overarching, rushing adversary, who can dynamically corrupt parties.[7] The adversary is constrained in the following ways: it controls strictly less than 50% of the total hashrate of Bitcoin. We assume existential honesty for Operators, Watchtowers, Challengers, and Signers, i.e., at any time, there is at least one honest Operator, one honest Watchtower, one honest Challenger, and one honest Signer. We observe that the existential honesty is a realistic assumption, since parties are economically motivated to follow the protocol.

While the Signers' existential honesty is required for *safety* of the bridge, for the bridge *liveness* during the peg in (i.e., when a user transfers funds from Bitcoin to Citrea) we need *all* Signers to sign. In practice, this does not hinder liveness, which is eventually achieved: if a peg in request does not go through within a certain time, the Signers that withhold are removed from the committee after a timelock. The timelock gives time to the users to withdraw their funds from the bridge and protect them in case the adversary is controlling the n-of-n of the Signer committee.

Bridge. A bridge protocol operates on top of two ledger protocols, e.g., $\mathcal{L}_A$ and $\mathcal{L}_B$. A bridge aims to condition the execution of a transaction in $\mathcal{L}_B$ on the execution of another transaction on $\mathcal{L}_A$. In a bidirectional bridge, $\mathcal{L}_A$ and $\mathcal{L}_B$ are interchangeable. For a formal definition and an extensive discussion of bridges and their properties, we refer to [31, 34].

5 Clementine: Protocol Description

Our goal is to design a bridge protocol that allows users to peg out of the Citrea rollup in a trust-minimized way. Towards this, we rely on the following components: the Citrea rollup, the Bitcoin ledger, a Bitcoin light client, a BitVM instance, and a distributed protocol that is run by the entities that maintain the bridge. As mentioned in Section 3 and Section 4, we assume a Bitcoin ledger that is safe and live, and we assume the Citrea rollup is correctly operating over Bitcoin.

The format and specification on the transactions introduced in the following will be given in Section 6.

[7]A single, overarching adversary is an adversary that can spawn nodes that are acting on her behalf. A rushing adversary is an adversary that, in any given round, gets to see all honest parties' messages before deciding her strategy [22]. The treatment in which the adversary is considered to be a single party with an overarching goal in mind gives the adversary more power than an adversary who is fighting against another [35].

5.1 Simplified Protocol Description

As a first step, for simplicity reasons, we will treat the light client as a black box, assuming that it exists and that it is secure. We also consider the BitVM program as a black box. We will expand on these two components in later subsections: this allows us to introduce the complex Clementine protocol one step at a time.

Pegging In. Consider a user, say Alice, that wants to peg in, i.e., move funds from Bitcoin to the Citrea rollup. Alice posts a **Deposit** transaction on Bitcoin that has an output holding the coins she wants to tranfer, e.g., 10 BTC, and that can be spent by Alice herself after $\mathsf{t_1}$ blocks, or by a transaction that reveals Alice's address on Citrea. This last spending condition is a covenant: In practice, to emulate covenants, Alice must share her Citrea address (along with the corresponding Tapscript) with the Signers, so that they can pre-sign a **MoveToVault** transaction. The **MoveToVault** transaction moves the user's funds to a new UTXO, making them redeemable in the *peg out* phase by the Operator; indeed, when pegging out a user, the Operator fronts the money to the user and, only after the correct execution has been checked by the bridge, it can be reimbursed with the money the user has locked during the peg in. The **MoveToVault** transaction, when posted, reveals the user's address on Citrea, where Citrea runs a Bitcoin light client. Anyone, including Alice herself, can send a valid proof to the light client that proves the **MoveToVault** transaction is on Bitcoin; this triggers a **Mint** transaction on Citrea which mints 10 cBTC. For user experience, the Operator takes care of relaying this proof.

Pegging Out. Now consider Alice that wants to *peg out*, i.e., move her 10 BTC out of Citrea and back to Bitcoin. Alice posts a **Burn** transaction in Citrea, burning her 10 cBTC and she waits for the rollup to checkpoint the new state onto Bitcoin that finalizes her transaction. The **Burn** transaction specifies an incomplete **Payout** transaction, which will be completed by the Operator when fronting the money to the user. Then, Alice waits for the rollup to checkpoint the new state onto Bitcoin, finalizing the **Burn** transaction. Upon seeing a **Burn** transaction on Citrea, the Operator posts the **Payout** transaction on Bitcoin but first, it completes it by adding, e.g., one input in which it fronts the 10 BTC to Alice. The Operator is now at loss of 10 BTC and it needs to have those coins reimbursed.

To initiate the reimbursement, the Operator posts a **KickOff** transaction on Bitcoin. The **KickOff** transaction allows the Clementine protocol to decide if the Operator has the right or not to get reimbursed - recall, an Operator is an untrusted party and, therefore, it can act maliciously. The **KickOff** transaction is fundamental in our construction, as it includes: (i) outputs that are part of the light client logic, (ii) an auxiliary output that, in the form of an OP_RETURN, identifies the **Payout** transaction the Operator wants a reimburse for, (iii) an output called *finalizer connector* that, depending on which transaction spends it, it gives information on whether the Operator has the right to get reimbursed and (iv) a covenant output that connects the **KickOff** to the **Reimburse** transaction in question. Let us consider the light client as a black box: we will describe how it works in the next subsection.

Let us focus on the output (iii). In the *optimistic case*, the Operator is honest and its reimbursement request is legit: the **NoChallenge** transaction spends the finalizer connector and the Operator gets 10 BTC back by posting the **Reimburse** transaction. The *pessimistic case*, the Operator is adversarial and tries to unfairly extract money from the bridge: To explore this case, we need to first understand how a light client can be emulated in Bitcoin.

Figure 1 depicts the transactions involved in the peg in and peg out phases. In this and the following figures, the blue arrow represent a pending path that needs to be pre-signed by covenant Signing committee, whereas orange arrows come out from an output that is spendable by the Operator.

5.2 The Light Client

The restrictions imposed by the Bitcoin scripting language prevent from implementing a light client in a programmatic way. Nonetheless, there are ways to securely emulate a light client protocol in Bitcoin. Before diving into the technical part, we highlight that we design an *optimistic* light client whose goal is to prove that: (1) Alice has burnt 10 cBTC in Citrea (recall that Citrea checkpoints its state into

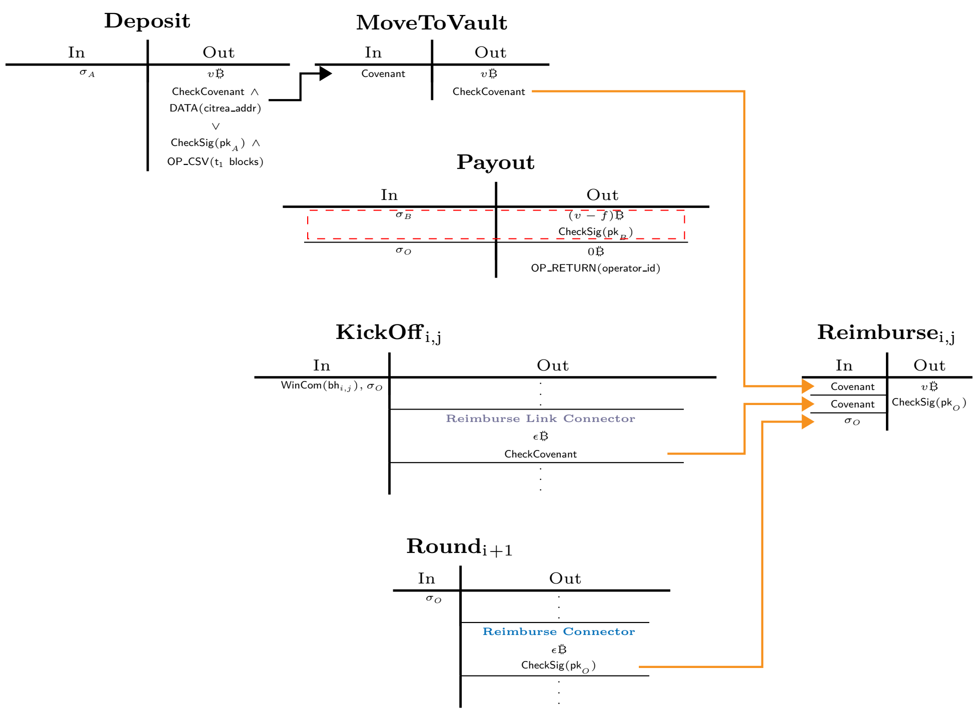


Fig. 1: **Deposit**, **Payout**, **MoveToVault** and **Reimburse** transactions.

Bitcoin) and the **Burn** transaction specifies an incomplete **Payout** transaction, and (2) the Operator has published on Bitcoin a complete **Payout** transaction that gives Alice 10 BTC.

The light client comprises two sets of parties: Watchtowers and Challengers. Watchtowers provide the light client with the necessary data to identify the canonical chain, while Challengers look after the security of the bridge. We introduce the notion of *payoff round*: payoff rounds are discrete moments in time in which the Operator batches together reimbursement requests. For simplicity, for now, we consider a single Watchtower, a single Challenger, and a single reimbursement.

From Section 5.1, we recall that the **KickOff** transaction includes outputs that serve to the light client logic. These outputs can be divided into two sets: an output that allows the Watchtower to provide a commitment (i.e., total accumulated work) to the canonical chain, and some outputs for the zkSNARK verifier. Let us focus on the first set: when the Operator publishes the **KickOff** transaction, it reveals a Winternitz commitment to the tip of the chain. If before the timeout expires no Challenger has posted a **Challenge** transaction (optimistic case), the Challenger legitimizes the Operator's right to get reimbursed; after the timeout, the Operator posts the **NoChallenge** transaction and, in the next payoff round it posts the **Reimburse** transaction, finally getting the money back.

Should the Challenger disagree with the Operator's commitment (pessimistic case), it posts a **Challenge** transaction on-chain, initiating the *challenge phase*. The Watchtower reacts within a (rather long) timeout by posting the **WatchtowerChallenge** transaction that spends from the designated **KickOff** transaction output, and reveals a different commitment to the chain. This commitment is done as a ZK proof, theproving the existence of a chain with a certain amount of total work that does not contain the blockhash signed by Operator as the unlocking Witness of the **KickOff** transaction. While this prevents any private mining attacks that can be performed by Watchtower prior to the commitment of this blockhash, since the Watchtower had a significant amount of time to provide a different commitment, we give the Operator the chance to counteract by posting the **BlockHash** transaction and revealing a fresh commitment to the chain.

The protocol now has two different commitments to the Bitcoin chain, one revealed by the Operator and one by the Watchtower. Here is where the light client comes into play: On input different commitments to the state of the chain, the light client outputs the correct one. After identifying the canonical chain, any statement about the chain can be proven. We use the BitVM protocol to encode on-chain a zkSNARK verifier for the bridge, which includes proving statements about an off-chain execution of a light client. The bridge program is locally run by the Operator and the Challengers and it outputs a proof that verifies the following: (i) The Operator has committed to a chain that has more Proof-of-Work compared to the one of the Watchtower's chain; (ii) Operator's chain contains the blockhash signed by Operator as the unlocking Witness of the **KickOff** transaction; (iii) This blockhash is from a block that contains the **Payout** transaction; (iv) Alice burned 10 cBTC on Citrea and **Payout** transaction is created from its endpoint; The protocol is ultimately designed to allow multiple Watchtowers to participate, and it is crucial that the client program receives on input all the commitments disclosed by these Watchtowers. No commitment should be omitted or censored. On a high-level, this is achieved by asking the Operator to post the **OperatorChallengeACK** transaction, acknowledging the challenge of the Watchtower. If the Operator does not post the **OperatorChallengeACK** transaction, the Challenger can post the **OperatorChallengeNACK** transaction, burning the finalizer connector and taking away the Operator's collateral and its possibility to ever facilitate a pegout again.

After running the light client off-chain, the Operator posts the **Assert** transaction[8] which reveals the output of the program and the output of some pre-defined intermediary steps; if the Operator does not post the **Assert** transaction, the Challenger posts the **AssertTimeout** after a timeout, spending the finalizer connector and taking the Operator's collateral. The Challenger runs the BitVM program as well and, if its local output differs from the one revealed by the Operator, it posts the **Disprove** transaction pinpointing the step of the program that has a different output. The **Disprove** transaction runs the questioned computational step *on-chain* and, if the outcome highlights the Operator's incorrect execution, the Challenger takes the Operator's collateral.

If the **Challenge** transaction is posted and no Watchtower posts the **WatchtowerChallenge** transaction, the Operator can post the **WatchtowerChallengeTimeout**, preventing Watchtowers to post a late commitment. If the **Challenge** transaction is posted and no Challenger posts a **Disprove** transaction, after some time the Operator posts the **DisproveTimeout** transaction. In this way, the Operator spends the kickoff finalizer connector, finalizing the kickoff transaction and enabling himself to post the **Reimburse** transaction. After the challenge phase is over, the Operator can get a reimbursement in the *next* payoff round by posting the **Reimburse** transaction.

The light client is, therefore, *optimistic*: if the Operator is honest, no light client program is run. If the Operator tries to cheat, the correct execution is enforced via the intervention of Watchtowers and Challengers.

Figure 2 depicts the **KickOff** transaction and all transactions spending its outputs.

5.3 Payoff Rounds and Multiple Reimbursement Requests

In the previous subsections, for the sake of simplicity, we have only dealt with a single payout and, in turn, a single reimbursement request. In the real world, the Operator must be able to deal with many, different peg outs. For different peg out requests, the Operator needs to post a dedicated **KickOff** transaction, each spending an output of what we call **Round**$_\text{i}$ transaction.

We introduce the notion of *payoff round*, a mechanism that allows the Operator to minimize the collateral and reuse it over time. Payoff rounds divide the time into consecutive time windows in which the Operator handles a bunch of peg out requests. The duration of these rounds can vary, and it is decided by the Operator; for instance, when all the outputs of a **Round**$_\text{i}$ transaction are consumed, the Operator can move to the next round to get its money back – recall, an Operator fronts the money to a user in a round and it gets reimbursed in the next round. Alternatively, when no peg out requests appear for some time, the Operator might want to move to the next round to get the money back without waiting for all the outputs of the **Round**$_\text{i}$ transaction to be consumed. The duration of a round also depends on the load of the bridge: if the number of peg out requests is high, then rounds do not last long.

A **Round**$_\text{i}$ transaction takes on input the Operator's collateral and it has different output types. One of its outputs is the *burn connector*, which holds the collateral. The Operator can safely spend the burn connector and reuse the collateral in the next round only after all **KickOff** transactions for that round have been finalized so that the collateral can be reused in the next round: specifically,

[8]Due to practical limitations of Bitcoin, this transaction is broken down into many "mini-assert" transactions.

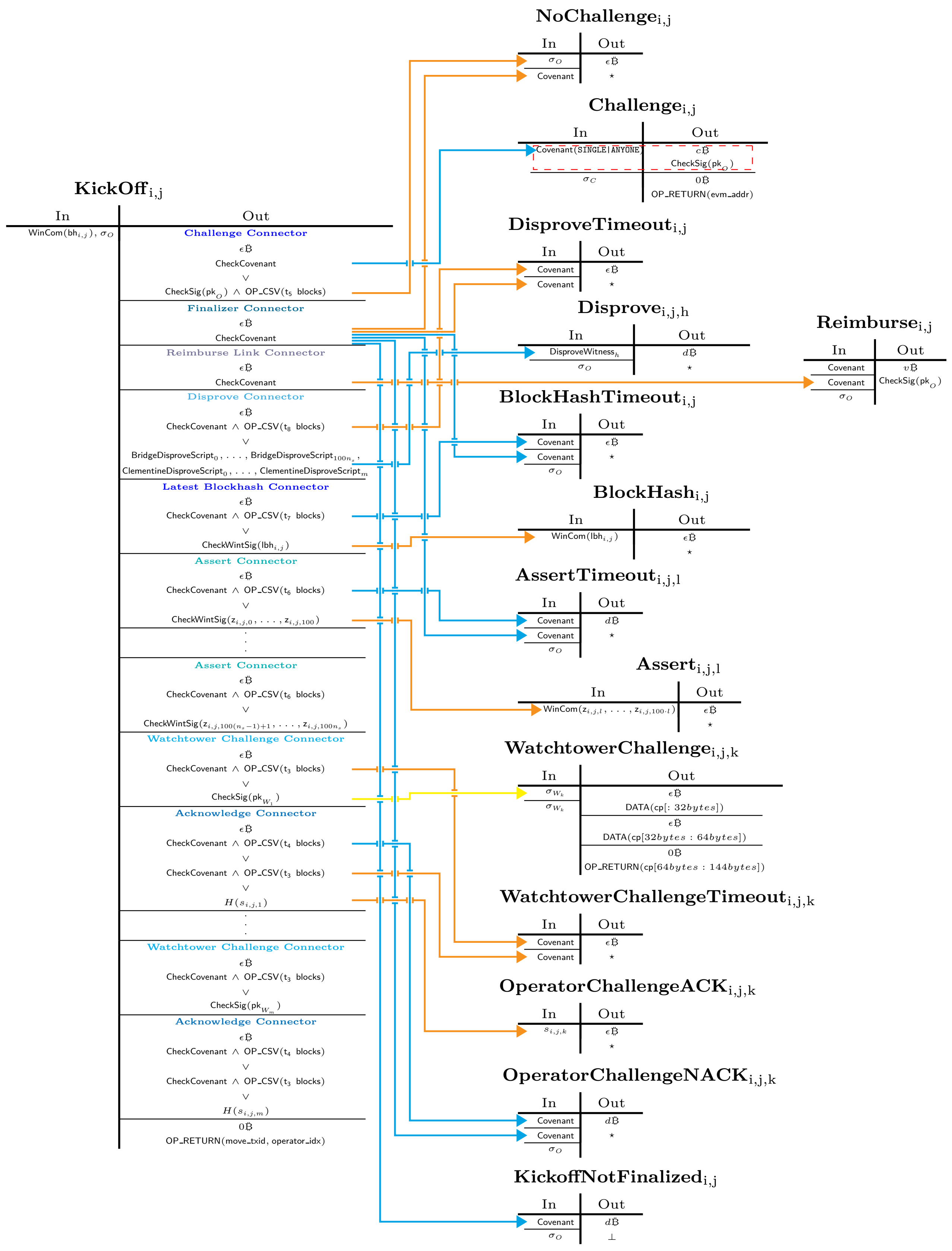


Fig. 2: **KickOff** transaction graph.

after all **KickOff** transactions of the round are finalized, the Operator spends the burn connector by posting the **ReadyToReimburse** transaction, locking the collateral into an output. This output is then spent by the next **Round**$_i$ transaction, transferring the collateral from one round to the next. Then, a **Round**$_i$ transaction has a set of outputs called *kickoff connectors* spendable by the Operator by posting a **KickOff** transaction and revealing a Winternitz commitment to the chain. Finally, the last set of outputs of a **Round**$_i$ transaction are the *reimburse connectors*, which are used as inputs to the **Reimburse** and ensure the Operator can only get a round's reimbursements in the next round, after having correctly concluded the previous one.

We observe that if the Operator posts a **ReadyToReimburse** transaction before all **KickOff** transactions have been finalized, a Challenger can punish it by posting a **KickoffNotFinalized** transaction that spends the finalizer connector and the output of the **ReadyToReimburse**, taking the Operator's collateral and thus preventing the Operator from getting reimbursed. We further observe that before moving to the next round, all unused kickoff connectors must be burned by the Operator by posting the **BurnUnusedKickOff** transaction: this contributes to economic safety of the bridge by preventing a malicious Operator from moving the collateral to a subsequent round while still being able to post **KickOff** transactions spending from the previous, no longer collateralized round. Similarly, should the Operator post a **ReadyToReimburse** before all the kickoff connectors of the **Round**$_i$ have been burnt, the Challenger can post a **UnspentKickOff** transaction and burn a *kickoff connector* as well as the Operator's collateral.[9]

Figure 3 depicts the payoff round transactions and the transactions spending their outputs, in the presence of a single **KickOff** transaction. Figure 4 depicts the case with multiple (in this case, two) **KickOff** transactions processed in a single round, showing how Clementine can handle high throughput.

5.4 Multiple Operators, Watchtowers, and Challengers

In the previous subsections, we have introduced all the key building blocks of the Clementine protocol, and we have introduced all the key participants of the protocol: an Operator, a Watchtower, and a Challenger. To fully enhance Clementine security and avoid single points of failure, we now consider multiple Operators, Watchtowers, and Challengers taking part to the protocol. We assume that, at least one Operator, one Watchtower, and one Challenger in the sets of Operators, Watchtowers, and Challengers are honest (existential honesty assumption).

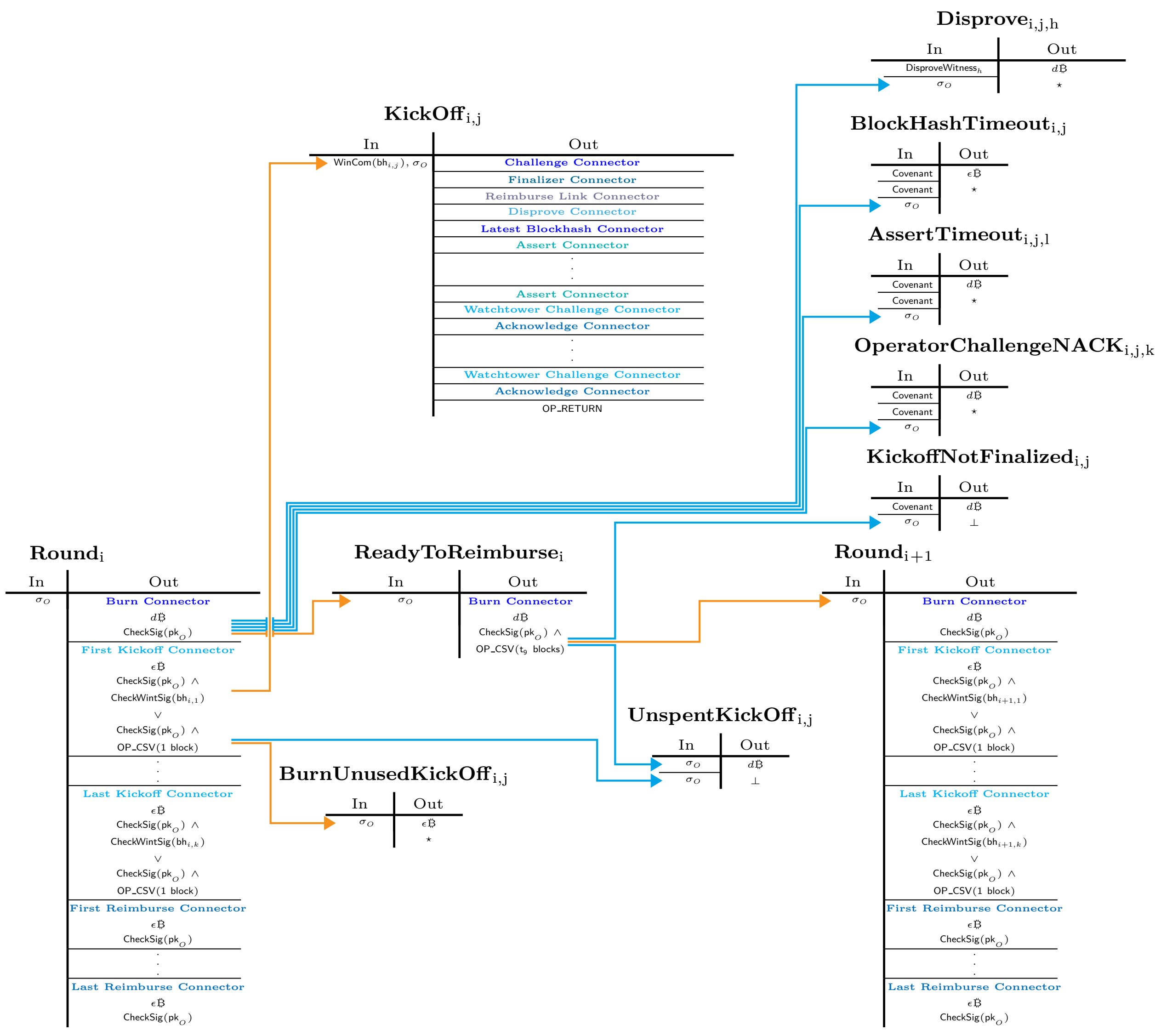
This new setting only requires minimal adjustments to the protocol described above. With multiple Operators supporting the bridge, all **KickOff** transactions need to have an OP_RETURN output that includes an identifier for the Operator that posted it, and an identifier for the **MoveToVault** transaction(s) the reimbursement takes the money from. We observe that for each peg out, only one reimbursement (to one Operator) can be issued, thanks to the way the **Payout** transaction is constructed. Recall that the **Payout** transaction is created by the user, Alice, and revealed in Citrea when posting the **Burn** transaction; its first input and first output are signed by Alice under the SIGHASH_SINGLE|ANYONECANPAY, which allows anyone to attach new inputs or outputs to the transaction without invalidating the signature. In this way, Operators compete to complete the transaction, and only one of them will end up posting on Bitcoin.

The **KickOff** transaction needs to include two outputs for each Watchtower: one that is spent by the Watchtower by revealing a commitment to the canonical chain that is different from the one presented by the Operator, and that is spent by the Operator when acknowledging the challenge.

Introducing multiple Challengers does not have any effect on the way transactions are constructed, nor on the way the protocol works: it only adds more people watching over the behavior of the Operator, guaranteeing that, should the Operator try to cheat, someone will react.

Figure 5 depicts the complete transaction graph of Clementine.

[9]After this, if an Operator posts a malicious kickoff, a Challenger needs to be careful to not send unnecessary challenges: without the collateral Operator cannot get reimbursed anyway.



Disprove$_{i,j,h}$
In
Out
DisproveWitness$_h$
σ_O
dɃ
$\star$
KickOff$_{i,j}$
WinCom(bh$_{i,j}$), σ_O
Challenge Connector
Finalizer Connector
Reimburse Link Connector
Disprove Connector
Latest Blockhash Connector
Assert Connector
Watchtower Challenge Connector
Acknowledge Connector
OP_RETURN
BlockHashTimeout$_{i,j}$
Covenant
ϵɃ
AssertTimeout$_{i,j,l}$
OperatorChallengeNACK$_{i,j,k}$
KickoffNotFinalized$_{i,j}$
$\perp$
Round$_i$
Burn Connector
CheckSig(pk$_O$)
First Kickoff Connector
CheckSig(pk$_O$) $\wedge$ CheckWintSig(bh$_{i,1}$)
$\vee$
CheckSig(pk$_O$) $\wedge$ OP_CSV(1 block)
Last Kickoff Connector
CheckSig(pk$_O$) $\wedge$ CheckWintSig(bh$_{i,k}$)
First Reimburse Connector
Last Reimburse Connector
ReadyToReimburse$_i$
CheckSig(pk$_O$) $\wedge$ OP_CSV(t_9 blocks)
UnspentKickOff$_{i,j}$
BurnUnusedKickOff$_{i,j}$
Round$_{i+1}$
CheckSig(pk$_O$) $\wedge$ CheckWintSig(bh$_{i+1,1}$)
CheckSig(pk$_O$) $\wedge$ CheckWintSig(bh$_{i+1,k}$)

Fig. 3: Round transaction graph of Clementine in the case of a single **KickOff** transaction.

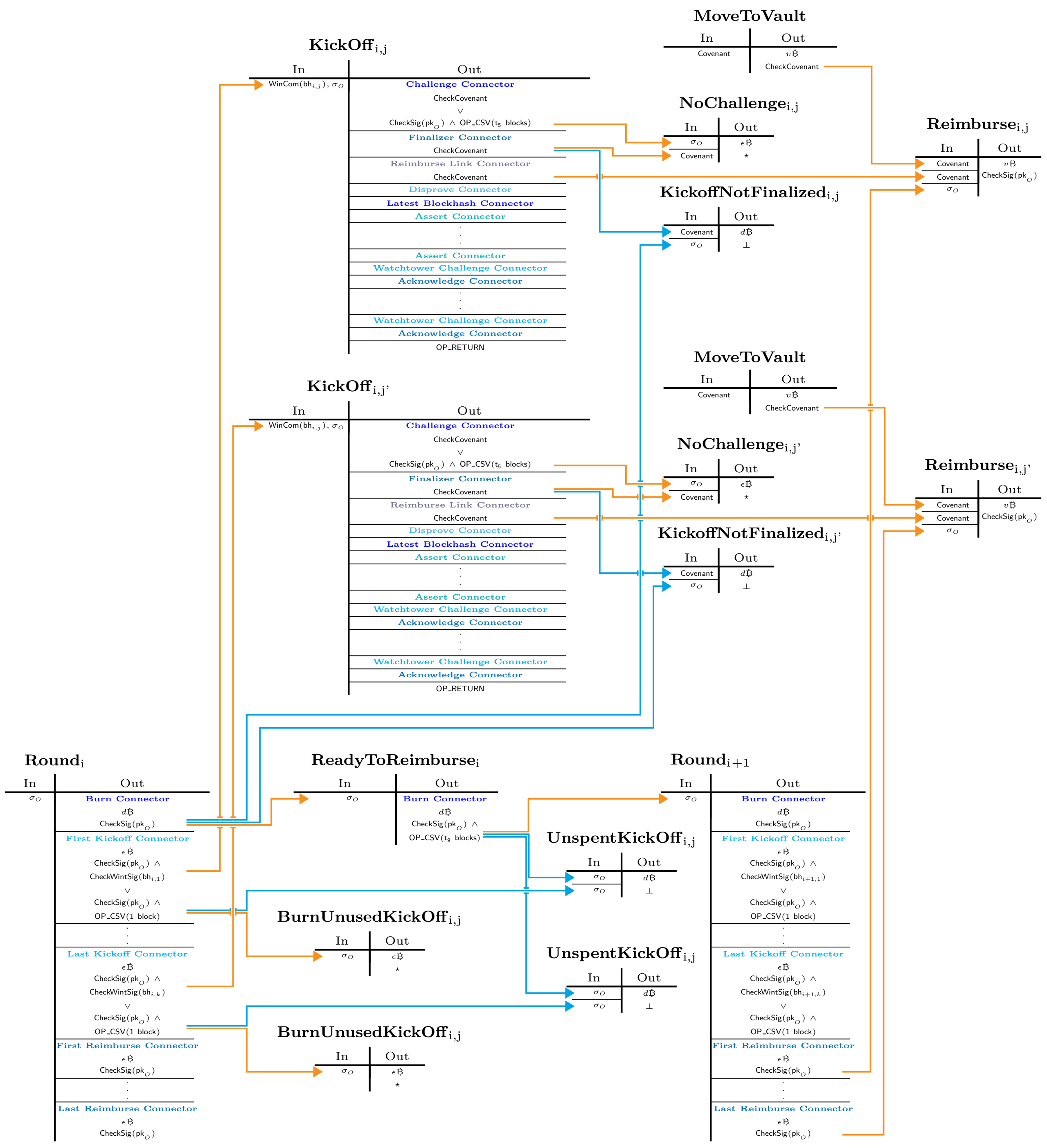


Fig. 4: Round transaction graph in the presence of multiple (in this case, due to space constraints, two) different **KickOff** transactions.

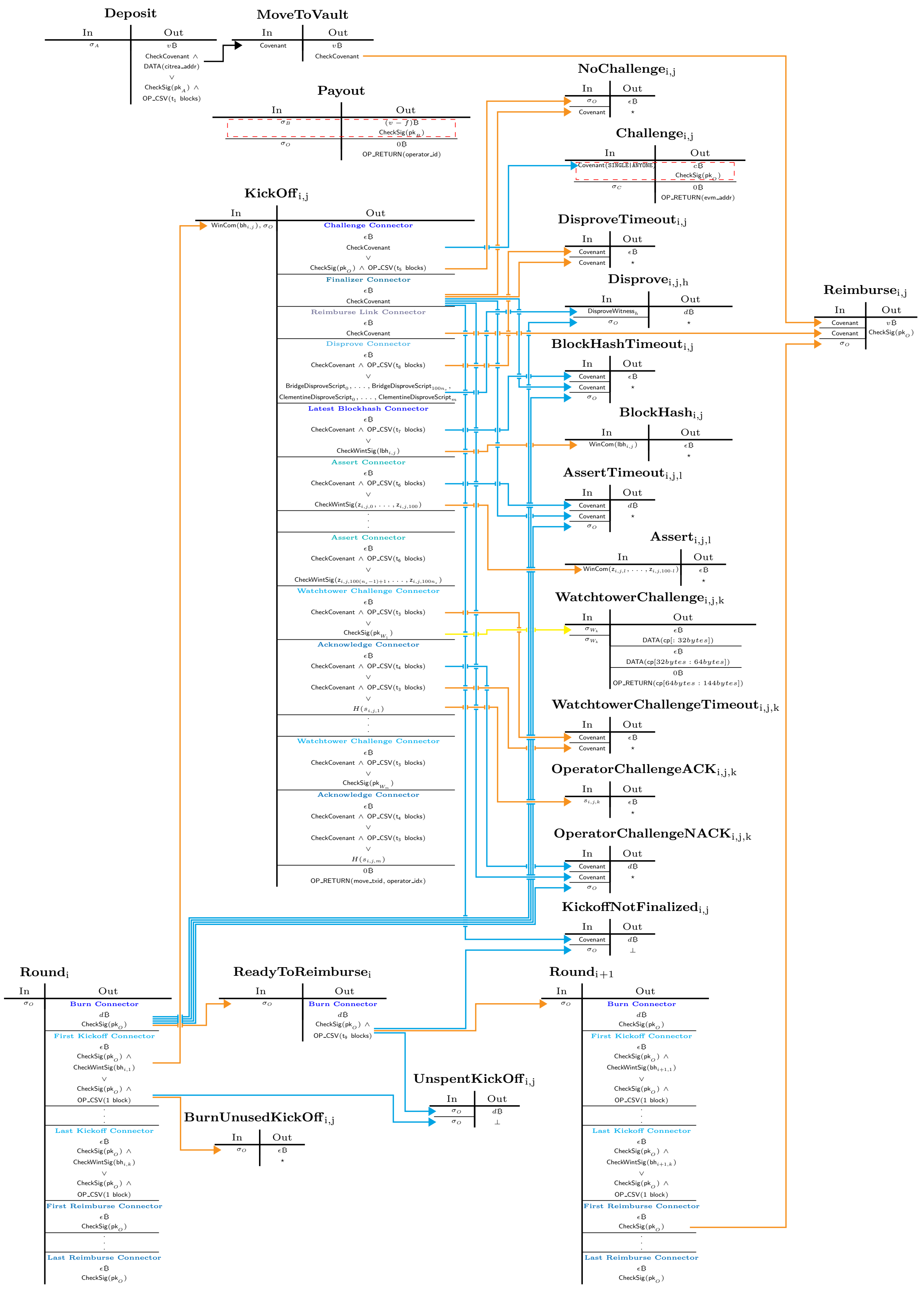

Deposit
MoveToVault
Payout
NoChallenge$_{i,j}$
Challenge$_{i,j}$
KickOff$_{i,j}$
DisproveTimeout$_{i,j}$
Disprove$_{i,j,h}$
Reimburse$_{i,j}$
BlockHashTimeout$_{i,j}$
BlockHash$_{i,j}$
AssertTimeout$_{i,j,l}$
Assert$_{i,j,l}$
WatchtowerChallenge$_{i,j,k}$
WatchtowerChallengeTimeout$_{i,j,k}$
OperatorChallengeACK$_{i,j,k}$
OperatorChallengeNACK$_{i,j,k}$
KickoffNotFinalized$_{i,j}$
Round$_i$
ReadyToReimburse$_i$
Round$_{i+1}$
UnspentKickOff$_{i,j}$
BurnUnusedKickOff$_{i,j}$
In
Out
Challenge Connector
Finalizer Connector
Reimburse Link Connector
Disprove Connector
Latest Blockhash Connector
Assert Connector
Watchtower Challenge Connector
Acknowledge Connector
Burn Connector
First Kickoff Connector
Last Kickoff Connector
First Reimburse Connector
Last Reimburse Connector
CheckCovenant
Covenant
Covenant(SINGLE|ANYONE)
CheckSig(pk_O)
CheckSig(pk_A) ∧ OP_CSV(t_1 blocks)
CheckCovenant ∧ DATA(citrea_addr)
OP_RETURN(operator_id)
OP_RETURN(evm_addr)
OP_RETURN(move_txid, operator_idx)
WinCom($bh_{i,j}$), σ_O
CheckSig(pk_O) ∧ OP_CSV(t_5 blocks)
CheckCovenant ∧ OP_CSV(t_8 blocks)
BridgeDisproveScript$_0$, ..., BridgeDisproveScript$_{100n_s}$,
ClementineDisproveScript$_0$, ..., ClementineDisproveScript$_m$
CheckCovenant ∧ OP_CSV(t_7 blocks)
CheckWintSig($lbh_{i,j}$)
CheckCovenant ∧ OP_CSV(t_6 blocks)
CheckWintSig($z_{i,j,0}$, ..., $z_{i,j,100}$)
CheckWintSig($z_{i,j,100(n_s-1)+1}$, ..., $z_{i,j,100n_s}$)
CheckCovenant ∧ OP_CSV(t_3 blocks)
CheckSig(pk_{W_1})
CheckSig(pk_{W_m})
CheckCovenant ∧ OP_CSV(t_4 blocks)
$H(s_{i,j,1})$
$H(s_{i,j,m})$
DisproveWitness$_h$
WinCom($lbh_{i,j}$)
WinCom($z_{i,j,l}$, ..., $z_{i,j,100\cdot l}$)
DATA(cp[: 32bytes])
DATA(cp[32bytes : 64bytes])
OP_RETURN(cp[64bytes : 144bytes])
$s_{i,j,k}$
CheckSig(pk_O) ∧ OP_CSV(t_9 blocks)
CheckSig(pk_O) ∧ CheckWintSig($bh_{i,1}$)
CheckSig(pk_O) ∧ CheckWintSig($bh_{i,k}$)
CheckSig(pk_O) ∧ CheckWintSig($bh_{i+1,1}$)
CheckSig(pk_O) ∧ CheckWintSig($bh_{i+1,k}$)
CheckSig(pk_O) ∧ OP_CSV(1 block)

Fig. 5: Complete transaction graph of Clementine.

6 Transactions and Disprove Scripts

6.1 Transaction Description

In this section, we formalize all the transactions encompassed by Clementine. We specify transactions in the form of tables, color-coding the cells as follows: In orange we outline the inputs, in blue the witness, and in green the information about the outputs. With $\star$ we denote a transaction input, witness, or output that can be anything (valid according to Bitcoin consensus rules), but is irrelevant to our protocol. With $\perp$ we refer to a script that burns the funds.

In the transaction tables and in the pseudocodes, we will use various indexes: with i we denote the index running through the number of rounds, with j the index running through number of outputs of a **Round** transaction, with k the index running through the number of Watchtowers, and with l the index running through the number of program splits.

The **Deposit** transaction is posted to Bitcoin by the user when pegging in. The output of this transaction holds the the amount v the user wants to transfer from Bitcoin to Citrea, and it can be spent either by the Operator in the **MoveToVault** transaction (enforced via covenants), or by the user itself after a (relative) timeout of t_1 blocks. This last condition allows the user to get back the money in case malicious Operators do not move forward with the peg in request. After posting the **Deposit** transaction, the user shares its own Citrea address with the Operators and Signers and, finally, an Operator issues a **Mint** transaction in Citrea that gives v coins to the user.

The **MoveToVault** transaction is posted by an Operator and makes the v coins of the user spendable by the **Reimburse** transaction. When posting **MoveToVault**, the Operator reveals the Citrea address of the user: this is checked by the Bitcoin light client running on Citrea, ensuring no cBTC are minted out of thin air.

Deposit				
Input	*Tx*	*Index*	*Tapleaf*	*Witness*
0	$\star$	$\star$	N/A	σ_A
Output	*Value*	*Script*		
0	v₿	$(\mathsf{CheckCovenant} \wedge \mathsf{DATA}(\mathsf{citrea\_addr})) \vee (\mathsf{CheckSig}(\mathsf{pk}_A) \wedge \mathsf{OP\_CSV}(\mathsf{t}_1\ \mathsf{blocks}))$		

MoveToVault				
Input	*Tx*	*Index*	*Tapleaf*	*Witness*
0	**Deposit**	0	0	Covenant
Output	*Value*	*Script*		
0	v₿	CheckCovenant		

When a user pegs out, it creates a 0-value output and it posts a **Burn** transaction that calls the `withdraw` function in Citrea. The **Burn** transaction is EVM-compatible and burns v coins; in its data, this transaction includes an *incomplete* **Payout** transaction that has the 0-value as input. The 0-value input guarantees that only one **Payout** transaction goes on-chain and that the correct user has been pegged out. Later, **Payout** is signed with $v - f$ coins as output spendable by the user. The 0-value input is signed with $\mathsf{SIGHASH\_SINGLE|ANYONECANPAY}$ and the signature along with output details is sent to the Operators off-chain. When Operators see an incomplete transaction on Citrea, they compete to complete it, adding one input in which it fronts money to the user (or more than one depending on the unspent outputs that the Operator has at his disposal) and one output that holds the Operator identifier (more outputs if the operator needs to send back to itself some coins for change). The complete **Payout** transaction is then posted on-chain by an Operator.

The **Reimburse** transaction is posted by an Operator to get reimbursed. We recall that an Operator can be reimbursed of a payout, only in the next payoff round.

A **Round** transaction is posted by Operators at the beginning of an Operator's new payoff round. The first round transaction, **Round**$_1$, created by an Operator is simpler than the following ones: it has a burn connector holding the d coins of the Operator's collateral and n_o kickoff connectors each one holding ϵ of coins.[10] The burn connector allows to propagate the collateral to the next round and it can be spent by the Operator after all the **KickOff** transactions of the round have been finalized. The Operator can also spend the burn connector to exit from the Operators' set. The kickoff

[10] Non-standard transactions [3, 5, 7], i.e., transactions s.t. $\epsilon = 0$ coins can be included in a block by a miner but, by default, these transactions are not propagated through the network.

Payout				
Input	*Tx*	*Index*	*Tapleaf*	*Witness*
0	$\star$	$\star$	N/A	σ_B
1	$\star$	$\star$	N/A	σ_O
Output	*Value*	*Script*		
0	$(v - f)$₿	CheckSig(pk$_B$)		
1	0	OP_RETURN(operator_id)		

Reimburse$_{i,j}$				
Input	*Tx*	*Index*	*Tapleaf*	*Witness*
0	**MoveToVault**	0	0	Covenant
1	**KickOff$_{i,j}$**	2	0	Covenant
2	**Round$_{i+1}$**	n_o+j	0	σ_O
Output	*Value*	*Script*		
0	v₿	CheckSig(pk$_O$)		

connectors of a **Round**$_1$ transaction can be spent by the Operator either by providing its signature and a Winternitz commitment to the chain, or after a relative timelock enforced by OP_CSV, aka. OP_CHECKSEQUENCEVERIFY. The opcode OP_CSV requires the spending transaction to set the nSequence number of a transaction field to a value that is equal or larger than the top stack value; we observe that an Operator cannot spend a kickoff connector with a **KickOff** transaction because all the transactions spending from the **KickOff** are bound by covenants to a **KickOff** with nSequence $= 0$. We recall that an Operator, before moving to the next round, must spend all the kickoff connectors: if there are not sufficient peg outs, it can issue a **BurnUnusedKickOff** transaction. Successive round transactions, **Round**$_i$, spend from a **ReadyToReimburse** and they include n_o additional outputs (reimburse connectors) that can be spent by the **Reimburse** transactions of previous round's peg outs.

Round$_1$				
Input	*Tx*	*Index*	*Tapleaf*	*Witness*
0	$\star$	$\star$	N/A	σ_O
Output	*Value*	*Script*		
0	d₿	**"Burn Connector"** CheckSig(pk$_O$)		
1	ϵ₿	**"First Kickoff Connector"** CheckSig(pk$_O$) $\wedge$ (CheckWintSig(bh$_{1,1}$) $\vee$ OP_CSV(1 block))		
...	...	...		
n_o	ϵ₿	**"Last Kickoff Connector"** CheckSig(pk$_O$) $\wedge$ (CheckWintSig(bh$_{1,n_o}$) $\vee$ OP_CSV(1 block))		

Round$_i$				
Input	*Tx*	*Index*	*Tapleaf*	*Witness*
0	ReadyToReimburse$_{i-1}$	0	0	σ_O
Output	*Value*	*Script*		
0	d₿	**"Burn Connector"** CheckSig(pk$_O$)		
1	ϵ₿	**"First Kickoff Connector"** CheckSig(pk$_O$) $\wedge$ (CheckWintSig(bh$_{i,1}$) $\vee$ OP_CSV(1 block))		
...	...	...		
n_o	ϵ₿	**"Last Kickoff Connector"** CheckSig(pk$_O$) $\wedge$ (CheckWintSig(bh$_{i,n_o}$) $\vee$ OP_CSV(1 block))		
n_o+1	ϵ₿	**"First Reimburse Connector"** CheckSig(pk$_O$)		
...	...	...		
n_o+n_o	ϵ₿	**"Last Reimburse Connector"** CheckSig(pk$_O$)		

A **ReadyToReimburse** transaction spends a burn connector and creates a new burn connector with the Operator's collateral. The output can either be spent by the Operator after a timeout or by a Challenger with the **UnspentKickOff** or the **KickoffNotFinalized** transactions. The former is posted if the Operator posts a **ReadyToReimburse** before all the kickoff connectors of the round have been burnt, the latter is posted if not every disprove process in the round was finished before the Operator sends **ReadyToReimburse**. The **ReadyToReimburse** transaction is necessary to safely move from one round to the next: if the Operator starts a new round before having burnt the kickoff connectors and before the timeout, it loses its collateral.

ReadyToReimburse$_i$				
Input	*Tx*	*Index*	*Tapleaf*	*Witness*
0	**Round**$_i$	0	0	σ_O
Output	*Value*	*Script*		
0	d₿	**"Burn Connector"** CheckSig(pk$_O$) $\wedge$ OP_CSV(t$_9$ blocks)		

BurnUnusedKickOff$_{i,j}$				
Input	*Tx*	*Index*	*Tapleaf*	*Witness*
0	**Round**$_i$	j	1	σ_O
Output	*Value*	*Script*		
0	ϵ₿	$\star$		

KickOffNotFinalized$_{i,j}$				
Input	*Tx*	*Index*	*Tapleaf*	*Witness*
0	**KickOff**$_{i,j}$	1	0	Covenant
1	**ReadyToReimburse**$_i$	0	0	σ_O
Output	*Value*	*Script*		
0	d₿	$\perp$		

UnspentKickOff$_{i,j}$				
Input	*Tx*	*Index*	*Tapleaf*	*Witness*
0	**ReadyToReimburse**$_i$	0	0	σ_O
1	**Round**$_i$	j	1	σ_O
Output	*Value*	*Script*		
0	d₿	$\perp$		

A **KickOff** transaction is published by the Operator plays a pivotal role in Clementine. By posting it, the Operator reveals a commitment to the chain. A **KickOff** transaction has the outputs that Watchtowers, Challengers, and the Operator himself use to determine whether a reimbursement is legit or not. The outputs of the **KickOff** transaction include:

– *Challenge Connector*: It can be spent by a Challenger, by posting the **Challenge** transaction. The **Challenge** transaction allows to challenge the Operator's commitment to the chain: an *incomplete* version of it is created during *peg in*, and then any Challenger can complete it by adding one c-value input that the Operator can spend to fund the (expensive) **Assert** transaction – this protects the Operator from griefing. In the **Challenge** transaction, the Challenger also includes an output with the its own address on Citrea: if the challenge was correct, the Challenger can have the c coins back on Citrea (by proving the outcome of the challenge through a light client proof). If no **Challenge** transactions is posted, then the challenge connector is spent by the Operator via the **NoChallenge** transaction after a relative timelock t$_5$ blocks.
– *Finalizer Connector*: When this connector is spent, the **KickOff** is finalized. Depending on how this connector is spent, we have information on whether the Operator can post the **Reimburse** transaction or if the Operator's collateral has been taken. In particular, if the finalizer connector is spent by the **KickOffNotFinalized**, **OperatorChallengeNACK**, **BlockHashTimeout**, or **AssertTimeout** transactions, then the (malicious) Operator loses its collateral and the possibility of acting as Operator again. On the other hand, if the finalizer connector is spent by the **NoChallenge** and **DisproveTimeout**, the Operator can get reimbursed, as no challenge took place or the challenge has been won by the Operator. We observe that, by design, if the Operator attempts to ask invalid reimbursement requests, it is sufficient for Challengers to contest a single **KickOff** transaction. In fact, a single successful challenge is enough to seize the Operator's collateral and prevent them from claiming any reimbursement.
– *Reimburse Link Connector*: Links the **KickOff** to its respective **Reimburse** transaction.
– *Disprove Connector*: This outputs allows to execute on-chain one step of the bridge program. There are two different types of disprove script: the Groth16 and the Clementine. The bridge program is split into n_s steps, while the Clementine program is split into as many steps as the number of Watchtowers, i.e., m. The Clementine scripts ensure that the inputs given to the bridge zkSNARK contain the contributions of *all* the Watchtowers that provided a commitment to the chain and that these commitments are correct, i.e., the signature of the Watchtower over the commitment is verified. The disprove connector can be spent by a Challenger posting the **Disprove**

transaction: this happens when a Challenger does not agree with the output of (one of the steps of) the Operator's off-chain execution of the bridge or Clementine programs. Otherwise, it can be spent by the Operator by posting the **DisproveTimeout** transaction, thus finalizing the **KickOff** transaction. We expand on the disprove scripts in Section 6.2.

– *Latest Blockhash Connector*: This connector, if spent by the **BlockHash** transaction, allows the Operator to post a *recent* commitment to the chain. This is necessary because 2 weeks pass by between the Operator's and the Watchtowers' commitments, and a malicious Watchtower with $< 50\%$ computational power can leverage this advantage and win an invalid challenge. With the latest blockchain connector, the Operator can post a new commitment 2.1 weeks after its last one, being therefore safe as long as the blockchain has an honest computational power. If the Operator does not post **BlockHash** after 2.1 weeks, a Challenger can post **BlockHashTimeout** transaction.
– *Assert Connectors*: This set of outputs are spent by the Operator via an **Assert** transaction. In practice, due to limits in the Bitcoin stack [4] and limits of Child-Pays-For-Parent (CPFP) transactions [9], we use n_s mini-**Assert** transactions. When posting these transactions, the Operator reveals (Winternitz commitments to) the final result of the execution of the off-chain program, as well as the intermediary results thereof. If an Operator does not post the **Assert** transaction within the available time window, a Challenger can post the **AssertTimeout** transaction, which spends the finalizer and burn connectors, finalizing the **KickOff** and taking the Operator's collateral.
– *Watchtower Challenge Connectors*: Each **KickOff** transaction has m of these connectors, one for each Watchtower. These outputs are spent by Watchtowers by posting a **WatchtowerChallenge** transaction and revealing a commitment to the chain. A total of 144 bytes are committed, split into 3 outputs, one OP_RETURN with 80 bytes of data and two taproot outputs with their script pubkey as the 32 byte data, making these outputs unspendable. Script pubkey of a taproot output is used to write arbitrary data because of the Bitcoin's limit of only one OP_RETURN allowed per standard transaction and 80 bytes limit of OP_RETURN data. 128 of these bytes represent the Groth16 proof required to verify the exclusion of the blockhash signed by Operator as the unlocking Witness of the **KickOff** transaction and validity of chain in the Bridge Circuit, while the remaining 16 bytes, used for the public input of the proof, represent the total work of the chain. When a challenge is raised and no Watchtower posts the **WatchtowerChallenge** transaction, the Operator can post the **WatchtowerChallengeTimeout** transaction after some time, preventing Watchtowers to post a late commitment.
– *Acknowledge Connectors*: Each **KickOff** transaction has m of these connectors, one for each Watchtower. When the Operator sees a **WatchtowerChallenge** transaction on-chain, it posts a **OperatorChallengeACK** transaction, revealing the secret preimage $s_{i,j,k}$ that is sampled uniformly during peg in. If the Operator fails to do so, a Challenger can take the Operator's collateral by posting the **OperatorChallengeNACK** transaction. The secret $s_{i,j,k}$ must be given on input to the Clementine disprove script, ensuring that the contributions of all the watchtowers have been considered (no censorship).
– *Output n_s+2m+5*: This last output of the **KickOff** transaction includes the Operator's identifier as well as the identifier of the **MoveToVault** transaction the **KickOff** refers to.

We note that the **KickOff** transaction cannot be used to spend a *reimburse connector* because its `nSequence` field is set to 0; should the Operator set it to a value other than 0, it would be unable to post the subsequent transactions since, due to covenants, they spend from a **KickOff** with `nSequence` = 0.

KickOff$_{i,j}$

Input	*Tx*	*Index*	*Tapleaf*	*Witness*
0	**Round**$_i$	j	0	WinCom($bh_{i,j}$), σ_O
Output	*Value*	*Script*		
0	ϵ₿	**“Challenge Connector”** CheckCovenant ∨ (CheckSig(pk_O) ∧ OP_CSV(t_5 blocks))		
1	ϵ₿	**“Finalizer Connector”** CheckCovenant		
2	ϵ₿	**“Reimburse Link Connector”** CheckCovenant		
3	ϵ₿	**“Disprove Connector”** (CheckSig(pk_O) ∧ OP_CSV(t_8 blocks))∨ BridgeDisproveScript$_1$ ∨ BridgeDisproveScript$_2$ ∨ . . . BridgeDisproveScript$_{n_s}$ ∨ ClementineDisproveScript$_1$ ∨ ClementineDisproveScript$_2$ ∨ . . . , ClementineDisproveScript$_m$		
4	ϵ₿	**“Latest Blockhash Connector”** (CheckCovenant ∧ OP_CSV(t_7 blocks)) ∨ CheckWintSig($lbh_{i,j}$)		
5	ϵ₿	**“Assert Connector”** (CheckCovenant ∧ OP_CSV(t_6 blocks)) ∨ CheckWintSig($z_{i,j,1}, \ldots, z_{i,j,100}$)		
...	...	...		
n_s+4	ϵ₿	**“Assert Connector”** (CheckCovenant ∧ OP_CSV(t_6 blocks)) ∨ CheckWintSig($z_{i,j,100(n_s-1)+1}, \ldots, z_{i,j,100n_s}$)		
n_s+5	ϵ₿	**“Watchtower Challenge Connector”** (CheckCovenant ∧ OP_CSV(t_3 blocks)) ∨ (CheckSig(pk_{W_1}))		
n_s+6	ϵ₿	**“Acknowledge Connector”** (CheckCovenant ∧ OP_CSV(t_4 blocks)) ∨ (CheckCovenant ∧ OP_CSV(t_3 blocks)) ∨ ($H(s_{i,j,1})$)		
...	...	...		
n_s+2m+3	ϵ₿	**“Watchtower Challenge Connector”** (CheckCovenant ∧ OP_CSV(t_3 blocks)) ∨ (CheckSig(pk_{W_m}))		
n_s+2m+4	ϵ₿	**“Acknowledge Connector”** (CheckCovenant ∧ OP_CSV(t_4 blocks)) ∨ (CheckCovenant ∧ OP_CSV(t_3 blocks)) ∨ ($H(s_{i,j,m})$)		
n_s+2m+5	0	OP_RETURN(move_to_vault_txid, operator_idx)		

Challenge$_{i,j}$

Input	*Tx*	*Index*	*Tapleaf*	*Witness*
0	**KickOff**$_{i,j}$	0	0	Covenant(SINGLE|ANYONE)
1	⋆	⋆	N/A	σ_C
Output	*Value*	*Script*		
0	c₿	CheckSig(pk_O)		
1	0	OP_RETURN(evm_addr)		

NoChallenge$_{i,j}$

Input	*Tx*	*Index*	*Tapleaf*	*Witness*
0	**KickOff**$_{i,j}$	0	1	σ_O
1	**KickOff**$_{i,j}$	1	0	Covenant
Output	*Value*	*Script*		
0	ϵ₿	⋆		

Disprove$_{i,j,h}$

Input	*Tx*	*Index*	*Tapleaf*	*Witness*
0	**KickOff**$_{i,j}$	3	h	DisproveWitness$_h$
1	**Round**$_i$	0	0	σ_O
Output	*Value*	*Script*		
0	d₿	⋆		

DisproveTimeout$_{i,j}$

Input	*Tx*	*Index*	*Tapleaf*	*Witness*
0	**KickOff**$_{i,j}$	3	0	Covenant
1	**KickOff**$_{i,j}$	1	0	Covenant
Output	*Value*	*Script*		
0	ϵ₿	⋆		

6.2 Disprove Scripts

Notation. Let O be an Operator in the set $\mathcal{O}$ of Operators, and W be a Watchtower in the set $\mathcal{W}$ of Watchtowers. Within the disprove script connector of the **KickOff** transaction, two types of disprove scripts are used. These scripts aim to synchronize the bridge with the canonical chain (i.e. the one with the most Proof-of-Work) and to verify that it meets the necessary conditions.

$\mathbf{BlockHash_{i,j}}$

Input	*Tx*	*Index*	*Tapleaf*	*Witness*
0	**KickOff**$_{i,j}$	4	1	$\mathsf{WinCom}(\mathsf{lbh}_{i,j})$
Output	*Value*	*Script*		
0	ϵ₿	$\star$		

$\mathbf{BlockHashTimeout_{i,j}}$

Input	*Tx*	*Index*	*Tapleaf*	*Witness*
0	**KickOff**$_{i,j}$	4	0	Covenant
1	**KickOff**$_{i,j}$	1	0	Covenant
2	**Round**$_i$	0	0	σ_O
Output	*Value*	*Script*		
0	ϵ₿	$\star$		

$\mathbf{Assert_{i,j,l}}$

Input	*Tx*	*Index*	*Tapleaf*	*Witness*
0	**KickOff**$_{i,j}$	$4+l$	1	$\mathsf{WinCom}(\mathsf{z}_{i,j,100\cdot(l-1)+1}, .., \mathsf{z}_{i,j,100\cdot l})$
Output	*Value*	*Script*		
0	ϵ₿	$\star$		

$\mathbf{AssertTimeout_{i,j,l}}$

Input	*Tx*	*Index*	*Tapleaf*	*Witness*
0	**KickOff**$_{i,j}$	$4+l$	0	Covenant
1	**KickOff**$_{i,j}$	1	0	Covenant
2	**Round**$_i$	0	0	σ_O
Output	*Value*	*Script*		
0	d₿	$\star$		

To obtain the canonical chain, Watchtowers and the Operator are asked to commit to a chain. While the Operator's chain is handled discretely, Watchtowers provide a Groth16 zero-knowledge proof [25], generated using the Watchtower Circuit (Figure 6). More formally, we define our SNARG instance (via $\mathsf{SNARG.Setup}$) over the relation $R_W := \{(\phi, w) : \Pi_W(\phi, w) = 1\}$, where Π_W, w, and the public input ϕ are defined in Figure 6. It proves that they have a valid chain with a certain amount of total work (given as ϕ) and that it does not contain the blockhash signed by the Operator as the unlocking Witness of the **KickOff** transaction.

The $\mathsf{BridgeDisproveScript}$ consists of a BitVM script split into chunks, verifying the Bridge Circuit (Figure 7) using a recursive Groth16 zero-knowledge proof [17, 25]. More formally, we define our SNARG instance (via $\mathsf{SNARG.Setup}$) over the relation $R_B := \{(\phi, w) : \Pi_B(\phi, w) = 1\}$, where Π_B, w, and the public input ϕ are defined in Figure 7. Intermediary steps between chunks are committed via the **Assert** transaction (cf. Section 3.5), and collectively, they verify that:

- The latest blockhash of the Operator's chain is equal to the blockhash signed by O as the input Witness of the **BlockHash** transaction.
- The Operator has a valid chain with the most Proof-of-Work compared to the Watchtowers' challenges with correct proofs.
- The blockhash signed by the Operator as the unlocking Witness of the **KickOff** transaction contains the **Payout** transaction.
- The Operator's chain includes the blockhash signed by the Operator as the unlocking Witness of the **KickOff** transaction.
- cBTC are burnt on Citrea through a withdrawal request, and the **Payout** transaction is derived from the endpoint of that request.

The $\mathsf{ClementineDisproveScript}$ (Figure 8) consists of m subroutines and verifies that the list of challenge-sending Watchtowers on-chain is consistent with the Bridge Circuit Groth16 inputs.

$\mathbf{WatchtowerChallenge_{i,j,k}}$

Input	*Tx*	*Index*	*Tapleaf*	*Witness*
0	**KickOff**$_{i,j}$	n_s+2k+5	1	σ_{W_k}
1	$\star$	$\star$	N/A	σ_{W_k}
Output	*Value*	*Script*		
0	ϵ₿	$\mathsf{DATA}(\mathsf{cp}[: 32\mathit{bytes}])$		
1	ϵ₿	$\mathsf{DATA}(\mathsf{cp}[32\mathit{bytes} : 64\mathit{bytes}])$		
2	0	$\mathsf{OP\_RETURN}(\mathsf{cp}[64\mathit{bytes} : 144\mathit{bytes}])$		

$\mathbf{WatchtowerChallengeTimeout_{i,j,k}}$

Input	*Tx*	*Index*	*Tapleaf*	*Witness*
0	**KickOff**$_{i,j}$	n_s+2k+5	0	Covenant
1	**KickOff**$_{i,j}$	n_s+2k+6	1	Covenant
Output	*Value*	*Script*		
0	ϵ₿	$\star$		

OperatorChallengeNACK$_{i,j,k}$				
Input	*Tx*	*Index*	*Tapleaf*	*Witness*
0	**KickOff**$_{i,j}$	n_s+2k+6	0	Covenant
1	**KickOff**$_{i,j}$	1	0	Covenant
2	**Round**$_i$	0	0	σ_O
Output	*Value*	*Script*		
0	d₿	$\star$		

OperatorChallengeACK$_{i,j,k}$				
Input	*Tx*	*Index*	*Tapleaf*	*Witness*
0	**KickOff**$_{i,j}$	n_s+2k+6	2	$s_{i,j,k}$
Output	*Value*	*Script*		
0	ϵ₿	$\star$		

m: Number of Watchtowers (we index this with k)
n_r: Number of **Round** transactions (we index this with i)
n_o: Max. number of peg-outs that can be served by a single **Round** transaction (we index this with j)
n_s: Number of chunks the BitVM program is split into (we index this with l)
f: Fee the Operator gets for the peg out
v: Denomination pegged into Citrea (10 BTC in Section 5.1)
d: Operator collateral to prevent cheating + fee for the disprove transaction
c: Challenge amount
ϵ: 0-value if **Round** and **KickOff** txs are non-standard, 330 sats if they are standard

Table 1: Protocol parameters.

7 Protocol Pseudocode

We now present the pseudocodes of the *setup*, *peg in*, and *peg out* phases. The transactions appearing in the pseudocodes are constructed as shown in Section 6.

Notation and Parameters. Let O be an Operator in the set $\mathcal{O}$ of Operators, S be a Signer in the set $\mathcal{S}$ of Signers, and W be a Watchtower in the set $\mathcal{W}$ of Watchtowers. When a message is sent to $\mathcal{P}$ (the set of all parties), we mean that it becomes public knowledge, and it is shared to all parties. We denote with $\mathcal{T}$ a set of transactions. We denote with u_c the liveness parameter of the Citrea rollup. Table 1 summarizes the protocol parameters.

7.1 Setup Phase

The *Setup* phase includes the actions that Operators and Signers need to perform at the beginning, when bootstrapping the bridge. We observe that the setup of the bridge is secure given a signer committee with existential honesty. In this phase, transactions such as **Round**, **ReadyToReimburse**, and **UnspentKickOff** need to be created and signed by the designated parties. Figure 9 shows the pseudocode of the *Setup* phase.

7.2 Peg In Phase

The *peg in* phase includes the actions that users, watchtowers, and signers need to perform when a user wants to move funds from Bitcoin to Citrea. During peg in, the **Deposit** and **MoveToVault** transactions are posted on Bitcoin, and the Mint transaction is posted on Citrea. Importantly, in this phase, the Operator creates transactions that will make the bridge secure at peg out, and it shares them with the Signers to establish the covenants.

We observe that Signers hold two different types of keys: they have a generic key pair $(\mathsf{pk}_{S,\mathsf{pid}}, \mathsf{sk}_{S,\mathsf{pid}})$ that does not change over time and that is used to sign all **MoveToVault** transactions (of all peg ins), and they have a key pair $(\mathsf{pk}, \mathsf{sk})$ which, on the contrary, is created anew at every peg in and that they are used for the covenants. Figure 10 showcases the pseudocode of the peg in phase.

7.3 Peg Out Phase

The *peg out* phase includes the actions that users, Operators, Watchtowers, and Challengers need to perform to securely move funds from Citrea to Bitcoin. When a user issues a peg out request on

Citrea by posting a Burn transaction, an Operator fronts the money to the user and then asks for a reimbursement to the bridge. The bridge logic comprises of several transactions which can be posted by Watchtowers, Challengers, and the Operator to guarantee a secure execution. Figure 11 presents the pseudocode of the peg out mechanism.

8 Discussion

In this section, we discuss some features and optimizations of Clementine.

Optimistic Payout. The protocol we described above guarantees that any peg out is completed even if all Signers are offline and all but one are malicious. However, if all Signers are honest and online, they have some time (in Clementine, it is $\simeq$ 1 hour) to sign an issue a user's peg out by posting an **OptimisticPayout** transaction. This transaction resembles the **Payout** transaction, with only two differences: (i) it spends the output of the **MoveToVault** transaction, so that the funds given to the user do not come from the Operator, and (ii) there is no OP_RETURN output. If no **OptimisticPayout** transaction appears on-chain within some time, the peg out request is picked up by the Operator and the Clementine continue as described in Section 5. We note that to enable the optimistic payout, Signers do not have to erase their keys, making the protocol secure only against a *non-adaptive* adversary.

Minimizing Collateral. To enable Clementine to facilitate enough (e.g., 100k) withdrawals, we can set the protocol parameters as follows: Operator's collateral $d = 2$ BTC, number of peg outs served by a round transaction $n_o = 500$, weeks in 12 years[11] divided by 2 weeks $n_3 = 300$, number of splits of the BitVM program $n_s = 42$, $m = 100$ Watchtowers.

This means that the initial capital to be locked for the **Round** in the *setup* phase is given by $d + n_3 \cdot 2n_o(n_s + 2m + 5)\epsilon$. If $\epsilon = 330$ sats, then the initial capital requirement is extremely high, i.e., approximately 250 BTC. Instead, by setting $\epsilon = 0$, the initial capital requirement drops to $d = 2$ BTC—a 125$\times$ reduction. This significantly lowers Clementine's collateral demands. In this scenario, the **Round** and **KickOff** transactions become non-standard due to their 0-sat outputs. However, the remaining transactions (with the exception of **Disprove**, which is non-standard due to large BitVM scripts) remain standard. Thanks to Ephemeral Anchors [6], these transactions can carry a 0-fee and still enter the mempool as part of a package where their child transactions pay the fees. The **Round** and **KickOff** transactions are not time-critical, meaning that the Operators can ask a miner to include these transactions in a block.

Optimize Communication. The Clementine protocol illustrated in Section 5 assumes that each party communicates to all other parties (all-to-all communication). In practice, this can lead to high bandwidth requirements for participants and it can result in increased network latency. Communication can be optimized by introducing, e.g., an *untrusted Aggregator* that orchestrates the communication between the different parties of the protocol. Should the Aggregator be down, anyone can take over and act as an Aggregator.

Rotating Committees. First, we observe that the sets of Watchtowers, Signers, and Operators, albeit permissioned, can change and evolve over time. Citrea hosts a contract that defines the Signers in the committee and it has a mechanism in place that monitors the Signers' behavior: if a Signer does not respond, it is removed within a time window to avoid denial of service. Similarly, new Watchtowers and Operators can be add, if all Signers agree.

Optimizing the Round Transaction. For real use cases, the **Round** transaction described in Section 6 would have a high number of reimburse connector and kickoff connector outputs (e.g., 1000 for each connector). This results in large transactions and higher on-chain fees. To reduce the size of these transactions and minimize fees, one could use techniques that allow to add extra connectors on-demand, depending on the current load of the bridge. For instance, one could have **Round** transactions with just a few connectors for each type and then one extra, new output that can be spent by a transaction that adds more of these connectors whenever necessary.

[11] Planned lifetime of the bridge.

Watchtower Circuit

Watchtower $W \in \mathcal{W}$, Operator $O \in \mathcal{O}$

In order to challenge the O, W prepares a chain, encoded with Header Chain Proof as Groth16 input (w). We call it `watchtower_chain`.

Blockhash signed by O as the input Witness of **KickOff** transaction is assumed to be accessible by the Watchtower Circuit, given as the Groth16 public input (ϕ). We call it `payout_tx_blockhash`.

Necessary restrictions are checked:

```
1. Verify that the watchtower_chain is valid with Header Chain Proof
2. Verify that watchtower_chain does not contain the block with payout_tx_blockhash.
```

Fig. 6: Pseudocode of the Watchtower Circuit used to compute the proofs Π_W in the **WatchtowerChallenge** transactions. This circuit defines the relation R over which we define the SNARG.

Bridge Circuit

Operator $O \in \mathcal{O}$ with **Payout** transaction T

Note that in order to do equivalency checks over private inputs (w) with Winternitz signed values or to extract them for more complicated uses, we can turn them into public ones (ϕ) in specific conditions. This is possible due to the fact that the O is forced to commit to the public inputs with Winternitz signatures in the **Assert** transaction with the intermediary steps, which allows us to use same key pairs in other scripts for further verifications. We call it public input extraction. In order to make the explanations more concise and clear, instead of giving these additional scripts separate names, we add their functionality in the Bridge Circuit and specify the parts we use this technique, in addition to the inputs we maintain.

The following variables are accessible in the Bridge Circuit:

- Schnorr public keys of Watchtowers, used for signing **WatchtowerChallenge** transaction, since they are constants during the setup
- `payout_tx_blockhash`, blockhash signed by O as the input Witness of the **KickOff** transaction, possible due to public input extraction
- `latest_blockhash`, blockhash signed by O as the input Witness of the **BlockHash** transaction, possible due to public input extraction

After Watchtower challenges are revealed, O prepares the following variables as Groth16 inputs (w):

- A chain `operator_chain`, expected to represent the actual Bitcoin chain in the honest case, encoded with a Header Chain Proof
- Payout transaction T and a Bitcoin block `payout_block`, expected to contain T and to be equal to the block with blockhash committed in the **KickOff** transaction
- Citrea components `citrea_vault_tx` and `citrea_light_client_proof`, representing the state written on-chain to Bitcoin
- Schnorr signatures of Watchtowers `watchtower_signatures`, each containing a Groth16 proof with the Watchtower Circuit, providing a chain with a certain amount of total work
- A boolean array `watchtower_sent_challenge[]` indicating whether each corresponding Watchtower has sent a challenge, this value will be maintained with public input extraction, to be used in ClementineDisproveScript (Figure 8)

Necessary restrictions are checked with the algorithm:

```
1. Verify that $T$ has the operator ID of $O$
2. Verify that payout_block contains $T$
3. With public input extraction of payout_block's blockhash, verify that it is equal to payout_tx_blockhash
4. With public input extraction of operator_chain's latest blockhash, verify that it is equal to latest_blockhash
5. Verify that operator_chain is valid and contains payout_block with Header Chain Proof
6. Verify using citrea_light_client_proof that Citrea includes the deposit transaction of citrea_vault_tx and withdrawal endpoint of $T$
7. Let max_watchtower_total_work := 0
8. For each Watchtower with ID $i$ and signature $W$ from watchtower_signatures:
   (a) If watchtower_sent_challenge[$i$] is false, skip this iteration
   (b) Verify that the given signature $W$ is valid
   (c) If the proof provided by signature $W$ for the Watchtower Circuit (Figure 6) verifies that given chain with total work $w$ is valid and doesn't contain payout_tx_blockhash:
       i. Let max_watchtower_total_work := max{max_watchtower_total_work, $w$}
9. Verify that the operator_chain has greater total work compared to max_watchtower_total_work
```

Fig. 7: Pseudocode of the Bridge Circuit used to compute the BridgeDisproveScript proof Π_B. This circuit defines the relation R over which we define the (recursive) SNARG.

```
ClementineDisproveScript

Created for Operator $O \in \mathcal{O}$ with Watchtower hashes $H$

Preimages are known only by $O$ and are revealed upon a WatchtowerChallenge. The corresponding hashes are
publicly known at setup. Each script is created using the following variables:

  – Hashes $H$ as constants
  – watchtower_sent_challenge[] as an input, enforced to be the same as its counterpart in the Bridge Circuit
    through public input extraction

Each subroutine $i$ does the necessary verifications for Watchtower with ID $i$, expects a preimage $P$ and its script
executes the following code:

  1. If preimage $P$ corresponds to the hash of Watchtower $i$ and watchtower_sent_challenge[$i$] is false, make the
     script spendable
```

Fig. 8: Pseudocode of ClementineDisproveScript.

```
Setup

Operator $O \in \mathcal{O}$

Upon Setup being invoked with an input $(txid, idx)$ on Bitcoin owned by $O$, do the following:

  1. For $j \in \{0, \dots, n_o - 1\}$, create $(\mathsf{pk}_j, \mathsf{sk}_j)$ using WintKeyGen and broadcast $(\mathsf{pk}_j)$ to $\mathcal{P}$. Using these public keys,
     create $\mathbf{Round}_1$ transaction with $\star := (txid, idx)$.
  2. Let $\mathcal{T} := \emptyset$
  3. For $i \in \{2, \dots, n_r\}$:
     (a) Create $\mathbf{ReadyToReimburse}_{i-1}$ transaction.
     (b) For $j \in \{0, \dots, n_o - 1\}$, create $(\mathsf{pk}_j, \mathsf{sk}_j)$ using WintKeyGen and broadcast $(\mathsf{pk}_j)$ to $\mathcal{P}$. Using these public keys,
         create $\mathbf{Round}_i$ transaction.
     (c) For $j \in \{0, \dots, n_o - 1\}$:
          i. Create $\mathbf{UnspentKickOff}_{i-1,j}$ transaction
         ii. Let $\sigma_O := \Sigma.\mathsf{Sign}_O(\mathbf{UnspentKickOff}_{i-1,j})$
        iii. Let $\mathcal{T} := \mathcal{T} \cup \{(\mathbf{UnspentKickOff}_{i-1,j}, \sigma_O)\}$
  4. Broadcast $(\texttt{setupComplete}, (txid, idx), \mathcal{T})$ to $\mathcal{P}$
  5. Post $\mathbf{Round}_1$ to Bitcoin and wait $u$ rounds for it to appear.

Signer $S \in \mathcal{S}$

Upon receiving $(\texttt{setupComplete}, (txid, idx), \mathcal{T})$ from $O \in \mathcal{O}$, do the following:

  1. Using the Winternitz public keys $(\mathsf{pk})$ received from $O$, create the following transactions (mimicking the protocol
     steps of the Operator). If any public key is missing, go idle.
  2. Create $\mathbf{Round}_1$ transaction with $\star := (txid, idx)$.
  3. For $i \in \{2, \dots, n_r\}$:
     (a) Create $\mathbf{ReadyToReimburse}_{i-1}$ transaction.
     (b) Create $\mathbf{Round}_i$ transaction.
     (c) For $j \in \{0, \dots, n_o - 1\}$:
          i. Verify that $(\mathbf{tx}, \sigma_O) \in \mathcal{T}$, where $\mathbf{tx}$ is $O$'s $\mathbf{UnspentKickOff}_{i-1,j}$ transaction and $\Sigma.\mathsf{Vrfy}_O(\mathbf{tx}, \sigma_O)$
  4. If all checks pass, mark $O$ as "ok".
```

Fig. 9: Pseudocode of the *Setup* phase.

Peg In

Alice $A \in \mathcal{P}$

Upon *Peg In* being invoked with input $(txid, idx)$ on Bitcoin and a Citrea address $\mathsf{citrea\_addr}$, both owned by A, do:

1. Create **Deposit** transaction with $\star := (txid, idx)$ and $\mathsf{citrea\_addr}$ inscribed.
2. Post **Deposit** on Bitcoin, which appears after u rounds.
3. If **MoveToVault** appears on Bitcoin, wait for u_c rounds for **Mint** to appear on Citrea. If no **Mint** transaction appears on Citrea, create a light client proof for **MoveToVault** and **Mint**, and post **Mint** to Citrea.
4. If **Deposit** remains unspent after $\mathsf{t_1\ blocks}$ rounds it has appeared on-chain, create and publish a transaction that spends the output of **Deposit** to an address owned by A.

Operator $O \in \mathcal{O}$

After completing *Setup*, whenever a **Deposit** appears on Bitcoin, let $\mathsf{pid} := H(\mathbf{Deposit})$, and do the following:

1. Broadcast $(\mathtt{requestKey}, \mathsf{pid})$ to $\mathcal{S}$. Wait for 2δ rounds to receive $(\mathtt{replyKey}, \mathsf{pid}, \mathsf{pk}_{S,\mathsf{pid}})$ for every $S \in \mathcal{S}$. Combine all of these $|\mathcal{S}|$ public keys $\mathsf{pk}_{S,\mathsf{pid}}$ to create the n-of-n condition $\mathsf{CheckCovenant}$ used to generate the transactions below. If not all messages are received, go idle.
2. Run $crs \leftarrow \mathsf{SNARG.Setup}(R_W)$ with each Watchtower $W \in \mathcal{W}$ and $crs \leftarrow \mathsf{SNARG.Setup}(R_B)$ (cf. Section 3.4), where R_W and R_B are defined in Section 6.2. This is used to encode $\mathsf{SNARG.Vrfy}$ in the "Disprove Connectors" of $\mathbf{KickOff}_{i,j}$
3. Let $\mathcal{T}_2 := \emptyset$. Create **MoveToVault** and add it to the set $\mathcal{T}_2$.
4. Let c be the maximum index, such that $\mathbf{Round}_c$ is on Bitcoin.
5. For $i \in \{c, \dots, n_r\}$:
   (a) Choose $J \subseteq \{1, \dots, n_o\}$ uniformly at random, such that $|J| = n_u$.
   (b) For $j \in J$:
      i. Let $\{s_{i,j,1}, \dots, s_{i,j,m}\}$ be a set of m numbers sampled uniformly at random.
      ii. Create $\mathbf{KickOff}_{i,j}$ using $\{H(s_{i,j,1}), \dots, H(s_{i,j,m})\}$, and add the transaction to the set $\mathcal{T}_2$.
      iii. For $k \in \{1, \dots, m\}$:
         A. Create $\mathbf{WatchtowerChallengeTimeout}_{i,j,k}$ and add it to the set $\mathcal{T}_2$.
         B. Create the transaction $\mathbf{OperatorChallengeNACK}_{i,j,k}$ and add it to the set $\mathcal{T}_2$. Let $\sigma := \Sigma.\mathsf{Sign}_O(\mathbf{OperatorChallengeNACK}_{i,j,k})$ and broadcast (σ) to $\mathcal{P}$.
      iv. Create $\mathbf{Challenge}_{i,j}$ and add it to the set $\mathcal{T}_2$. Create $\mathbf{NoChallenge}_{i,j}$ and add it to the set $\mathcal{T}_2$.
      v. Create $\mathbf{BlockHashTimeout}_{i,j}$ and add it to $\mathcal{T}_2$. Let $\sigma := \Sigma.\mathsf{Sign}_O(\mathbf{BlockHashTimeout}_{i,j})$, broadcast (σ) to $\mathcal{P}$.
      vi. For $l \in \{1, \dots, n_s\}$:
         A. Create $\mathbf{AssertTimeout}_{i,j,l}$ and add it to the set $\mathcal{T}_2$. Let $\sigma := \Sigma.\mathsf{Sign}_O(\mathbf{AssertTimeout}_{i,j,l})$ and broadcast (σ) to $\mathcal{P}$.
      vii. Create $\mathbf{Disprove}_{i,j,h}$ and add it to the set $\mathcal{T}_2$. Let $\sigma := \Sigma.\mathsf{Sign}_O(\mathbf{Disprove}_{i,j,h})$ and broadcast (σ) to $\mathcal{P}$.
      viii. Create $\mathbf{DisproveTimeout}_{i,j}$, $\mathbf{KickOffNotFinalized}_{i,j}$, and $\mathbf{Reimburse}_{i,j}$, then add them to $\mathcal{T}_2$. Let $\sigma := \Sigma.\mathsf{Sign}_O(\mathbf{KickOffNotFinalized}_{i,j})$ and broadcast (σ) to $\mathcal{P}$.
6. Broadcast $(\mathtt{sendTxSet}, \mathcal{T}_2)$ to $\mathcal{P}$ (and in particular, to $\mathcal{S}$).
7. Wait for 2δ rounds to receive $(\mathtt{replySig}, \mathcal{V}_S)$ for every $S \in \mathcal{S}$. If not all messages are received, go idle.
8. For $S \in \mathcal{S}$:
   (a) If $\mathcal{V}_S$ does not contain a signature σ, such that $\Sigma.\mathsf{Vrfy}_{\mathsf{pk}_{S,\mathsf{pid}}}(\mathbf{tx}, \sigma)$ for every transaction (regardless of indicies) $\mathbf{tx} \in \{$**WatchtowerChallengeTimeout**, **OperatorChallengeNACK**, **Challenge**, **NoChallenge**, **BlockHashTimeout**, **AssertTimeout**, **DisproveTimeout**, **KickOffNotFinalized**, **Reimburse**$\}$, go idle. $\mathsf{pk}_{S,\mathsf{pid}}$ denotes the key received at step 1.
   (b) If $\mathcal{V}_S$ does not contain a signature σ, such that $\Sigma.\mathsf{Vrfy}_{\mathsf{pk}_S}(\mathbf{MoveToVault}, \sigma)$, where pk_S denotes S's generic public key, go idle.
9. Combine the signatures for $S \in \mathcal{S}$ in the sets $\mathcal{V}_S$ to form the $\mathsf{Covenant}$s for the transactions in $\mathcal{T}_2$. Store all these covenants in the set $\mathcal{V}$.
10. Post **MoveToVault** to Bitcoin, which appears after u rounds. Create a light client proof for **MoveToVault** and **Mint**, and post **Mint** to Citrea. Store $\mathcal{M}_T(\mathsf{pid}) := \mathcal{T}_2$, $\mathcal{M}_V(\mathsf{pid}) := \mathcal{V}$

Signer $S \in \mathcal{S}$

Upon receiving $(\mathtt{requestKey}, \mathsf{pid})$ from an operator $O \in \mathcal{O}$, do: (1) Let $(\mathsf{pk}, \mathsf{sk}) := \Sigma.\mathsf{Gen}(1^\kappa)$. (2) Store sk for pid. (3) Send $(\mathtt{replyKey}, \mathsf{pid}, \mathsf{pk})$ to O.

Upon receiving a message $(\mathtt{sendTxSet}, \mathcal{T}_2)$ from an operator $O \in \mathcal{O}$, who has been marked as "ok" in the *Setup* phase of $\mathcal{S}$, step 4, do the following:

1. Verify that $\mathcal{T}_2$ contains every transaction outlined in the Operator's PegIn protocol code from steps 3 to 5(b)viii, and only those transactions, correctly constructed, and building on some valid **Deposit** as input. If not, go idle.
2. Verify, that the correct $\mathsf{SNARG.Vrfy}$ (using R_B, R_W, and the corresponding crs) is encoded in the "Disprove Connectors" of $\mathbf{KickOff}_{i,j}$.
3. If having previously not received σ, such that $\Sigma.\mathsf{Vrfy}_{\mathsf{pk}_{S,\mathsf{pid}}}(\mathbf{tx}, \sigma)$ for every transaction (regardless of indicies) $\mathbf{tx} \in \{\mathbf{OperatorChallengeNACK}_{i,j,k}, \mathbf{BlockHashTimeout}_{i,j}, \mathbf{AssertTimeout}_{i,j,l}, \mathbf{KickOffNotFinalized}_{i,j}\}$, go idle. Else, sign every transaction, regardless of indices, **Challenge**, **WatchtowerChallengeTimeout**, **OperatorChallengeNACK**, **NoChallenge**, **BlockHashTimeout**, **AssertTimeout**, **DisproveTimeout**, **KickOffNotFinalized**, **Reimburse**, using sk for pid, and add the resulting signatures to $\mathcal{V}_S$.
4. Sign **MoveToVault** using S's generic secret key, and add the resulting signature to $\mathcal{V}_S$.
5. Broadcast $(\mathtt{replySig}, \mathcal{V}_S)$ to $\mathcal{P}$ (and in particular, send it to O). Delete sk for pid.

Fig. 10: Pseudocode of the *Peg In* phase.

Peg Out

Bob $B \in \mathcal{P}$

Upon *Peg Out* being invoked with an input $(txid, idx)$ holding v₿ on Citrea, owned by B, do the following:

1. Create a (partial) **Payout**. Let $\sigma := \Sigma.\mathsf{Sign}_B(\textbf{Payout})$ with flag `SIGHASH_SINGLE|ANYONECANPAY`.
2. Create a **Burn** transaction based on $(txid, idx)$, which specifies the first output of **Payout**, and post it on Citrea. It appears after at most u_c rounds. Send $(\texttt{sendPayout}, \textbf{Payout}, \sigma)$ to $O \in \mathcal{O}$.

Operator $O \in \mathcal{O}$

After completing *Setup*, upon receiving $(\texttt{sendPayout}, \textbf{Payout}, \sigma)$ from $B \in \mathcal{P}$, do the following:

1. If $\mathsf{Citrea.read}()$ does not contain a **Burn** in which v₿ are burnt by B, **Burn** specifies $\textbf{Payout}.\mathsf{output}[0]$ in its body, and $\Sigma.\mathsf{Vrfy}_B(\textbf{Payout}, \sigma)$ with flag `SIGHASH_SINGLE|ANYONECANPAY`, go idle.
2. Find the corresponding **Deposit**, let $\mathsf{pid} := H(\textbf{Deposit})$ and retrieve the Store $\mathcal{T}_2 := \mathcal{M}_T(\mathsf{pid})$, $\mathcal{V} := \mathcal{M}_V(\mathsf{pid})$. If either entry does not exist, go idle. The following transactions and witnesses come from $\mathcal{T}_2$ and $\mathcal{V}$ (or are created on-the-fly). Let i be the maximum index, such that $\textbf{Round}_i$ is on Bitcoin. Select the smallest j, for which $\textbf{KickOff}_{i,j} \in \mathcal{T}_2$ and $\textbf{Round}_i.\mathsf{output}[j]$ is unspent. If it does not exist, go idle.
3. Finish **Payout** using an output holding $v - f$ coins on Bitcoin owned by O.
4. Sign **Payout** and post it to Bitcoin. If it does not appear after u rounds, go idle.
5. Using a Winternitz commitment to $\mathsf{bh} := H(B)$ for $B := \mathsf{Bitcoin.read}()[-1]$, post $\textbf{KickOff}_{i,j}$ to Bitcoin and wait for it to appear after at most u rounds and denote the block height of the block in which it is included as h_1.
6. If, within t_3 blocks after h_1, a $\textbf{WatchtowerChallenge}_{i,j,k}$ transaction appears on Bitcoin, create transaction $\textbf{OperatorChallengeACK}_{i,j,k}$ using $s_{i,j,k}$ (the secret chosen in the Operator's *PegIn* procedure step 5(b)i). Post this transaction to Bitcoin. For every $k \in \{0, \dots, m-1\}$, for which $\textbf{KickOff}_{i,j}.\mathsf{output}[n_s + 2k + 5]$ is unspent at t_3 blocks after h_1, post $\textbf{WatchtowerChallengeTimeout}_{i,j,k}$
7. If, within t_5 blocks after h_1, a **Challenge** transaction appears on Bitcoin, do the following:
   (a) Using a Winternitz commitment to $\mathsf{lbh} := H(B)$ for $B := \mathsf{Bitcoin.read}$, create and post $\textbf{BlockHash}_{i,j}$.
   (b) Let $\pi \leftarrow \mathsf{SNARG.Prove}(R_B, crs, \phi, w)$, where R_B is defined in Section 6.2, crs is created in the Setup, Operator, step 2, ϕ contains all Watchtower proofs from $\textbf{WatchtowerChallenge}_{i,j,k}$, bh, all Watchtower's Schnorr public keys and a boolean array indicating which Watchtower has sent a challenge. Run $\mathsf{SNARG.Vrfy}(R_B, crs, \phi, \pi)$ yielding $z_0, \dots, z_{i,j,100n_s}$, create $\textbf{Assert}_{i,j,l}$ and post these transactions to Bitcoin.
   (c) If, at t_8 blocks after h_1, no $\textbf{Disprove}_{i,j}$ is on Bitcoin, post $\textbf{DisproveTimeout}_{i,j}$ to Bitcoin.
8. If, at t_5 blocks after h_1, no $\textbf{Challenge}_{i,j}$ is on Bitcoin, post $\textbf{NoChallenge}_{i,j}$ to Bitcoin.
9. If there are no other unfinished PegOut requests of O and O wants to proceed with the reimbursement (else wait until this is the case), spend all outputs in $\textbf{Round}_{i,j}$ via **BurnUnusedKickOff**, after they all appear on Bitcoin, post **ReadyToReimburse**, after it appears, wait for t_9 blocks, post $\textbf{Round}_{i+1}$, and then post $\textbf{Reimburse}_{i,j}$.

Watchtower $W_k \in \mathcal{W}$

Whenever a $\textbf{KickOff}_{i,j}$ appears on Bitcoin do the following:

1. Extract $\mathsf{WinCom}(\mathsf{bh})$ from $\textbf{KickOff}.\mathsf{witness}[0]$. Let $\mathfrak{C} := \mathsf{Bitcoin.read}()$
2. If $\nexists B \in \mathfrak{C} : H(B) = \mathsf{bh}$:
   (a) $\pi \leftarrow \mathsf{SNARG.Prove}(R_W, crs, \phi, w)$, where R_W is defined in Section 6.2, crs is created with the O in the Setup, Operator, step 2, ϕ is the blockhash committed to in the witness of $\textbf{KickOff}_{i,j}$, and $w := \mathfrak{C}$.
   (b) Create $\textbf{WatchtowerChallenge}_{i,j,k}$ using π, then sign $\textbf{WatchtowerChallenge}_{i,j,k}$ and post it to Bitcoin.

Challenger $C \in \mathcal{C}$

Challengers maintain a mapping from pid to a validated transaction/signature set $(\mathcal{T}_2, \mathcal{V})$ for each operator. These sets are gathered during the protocol execution (e.g., via broadcasts or local observation of $(\texttt{sendTxSet}, \mathcal{T}_2)$ and $(\texttt{replySig}, \mathcal{V}_S)$ messages from operators and signers).
Whenever a new $\textbf{KickOff}_{i,j}$ appears on Bitcoin, let h_1 denote the block in which it is included, do the following:

1. If $\nexists B \in \mathfrak{C} : H(B) = \mathsf{bh}$, where bh is the the committed blockhash in $\textbf{KickOff}_{i,j}$'s witness, post $\textbf{Challenge}_{i,j}$.
2. For every $k \in \{0, \dots, m-1\}$, for which $\textbf{KickOff}_{i,j}.\mathsf{output}[n_s + 2k + 6]$ is unspent at t_4 blocks after h_1, post $\textbf{OperatorChallengeNACK}_{i,j,k}$.
3. If there is a $\textbf{Challenge}_{i,j}$ on chain, and if $\textbf{KickOff}_{i,j}.\mathsf{output}[4]$ is unspent at t_7 blocks after h_1, post $\textbf{BlockHashTimeout}_{i,j}$.
4. If there is a $\textbf{Challenge}_{i,j}$ on chain, and if there is any $l \in \{1, \dots, n_s\}$, such that $\textbf{KickOff}_{i,j}.\mathsf{output}[5+l]$ is unspent at t_6 blocks after h_1, post $\textbf{AssertTimeout}_{i,j,l}$.
5. If there is a $\textbf{Challenge}_{i,j}$ on chain, and if there is no $l \in \{0, \dots, n_s\}$, such that $\textbf{KickOff}_{i,j}.\mathsf{output}[5+l]$ is unspent at t_6 blocks after h_1, Execute $\mathsf{SNARG.Vrfy}(R_B, crs, \phi, \pi)$where R_B is defined in Section 6.2, crs is created in the Setup, Operator, step 2, and ϕ, π are contained in $\textbf{Assert}_{i,j,0}$ (z_0'), yielding $z_0, \dots, z_{i,j,100n_s}$. If $z_0, \dots, z_{i,j,100n_s}$ does not match any $\textbf{Assert}_{i,j,l}$ or there is a $\textbf{WatchtowerChallenge}_{i,j,k}$ on-chain, but not set in the boolean array in ϕ, create and post $\textbf{Disprove}_{i,j}$ and post it to Bitcoin. In the latter case, using the secret $s_{i,j,k}$ of O, revealed in $\textbf{OperatorChallengeACK}_{i,j,k}$

Whenever a new $\textbf{ReadyToReimburse}_i$ (spending $\textbf{Round}_i.\mathsf{output}[0]$) appears on Bitcoin, do the following:

1. If there is a $\textbf{KickOff}_{i,j}$ on Bitcoin and $\textbf{KickOff}_{i,j}.\mathsf{output}[2]$ is unspent, post $\textbf{KickOffNotFinalized}_{i,j}$.
2. If there exists a $j \in \{1, \dots, n_o\}$, such that $\textbf{Round}_i.\mathsf{output}[j]$ is unspent, post $\textbf{UnspentKickOff}_{i,j}$.

Fig. 11: Pseudocode of the *Peg Out* phase.

Peg In and Peg Out Amounts. Clementine inherits the same limitation of the BitVM Bridge: the amount of the peg in must be equal to the amount of that is peg out. As discussed in [14], a simple mitigation is to spread the bridge deposits across multiple Clementine instances of different sizes. In practice, however, peg-in and peg-out transactions will likely be performed by professional users as a service.

9 Informal Security Analysis

Clementine is designed as an on-chain multi-party protocol, which is executed by mutually distrusting participants. The protocol ensures that any correct participant is guaranteed to not lose money. As a result of the protocol execution, BTC can be securely locked (or released) and cBTC can be securely minted (or burnt) between Bitcoin and Citrea, thus behaving as a blockchain bridge.

We now state Clementine's security guarantees, showing what it means for it to be safe, live, and balance secure. We will focus on the peg out mechanism, which is the one Clementine realizes. Recall that bridging out of Citrea is the most challenging part, due to the limitations of Bitcoin script that do not allow to deploy and run standard bridge protocols. On the other hand, the security of the peg in is trivially achieved via a standard light client-based bridge protocol running on Citrea. Consider the following informal properties.

Definition 3 (Clementine safety). *No incorrect peg out succeeds.*

In other words, Clementine's safety means the adversary cannot successfully get the corresponding BTC on Bitcoin, without first burning the cBTC on Citrea.

Definition 4 (Clementine liveness). *Any correct peg out will succeed, eventually.*

In other words, Clementine's liveness means that an honest user can always eventually peg out of Citrea (by burning their cBTC) and get the corresponding BTC back on Bitcoin.

Definition 5 (Clementine balance security). *No correct Clementine participant will lose its money.*

In the following, we informally analyze the security of Clementine, and argue why and how this properties are ensured given our model and assumptions Section 4. The goal is twofold: First, ensure that Clementine users can always peg-in and peg-out without putting their money at risk. Second, we ensure that no correct participant (users, Operators, Challengers, Signers, and Watchtowers) loses money during the protocol execution by arguing that the correct, intended outcome is always enforceable. Informally, Clementine achieves liveness by economically motivating Operators to act honestly, i.e., by pegging out users, else they will not collect fees. Similarly, safety is ensured by design via Bitcoin script and via economically motivating Operators to behave honestly, else they lose their collateral. We observe that from Section 5 and Section 6, the intended outcome of the protocol is defined such that the balance of correct parties is not a risk (balance security).

We do a case analysis for the protocol at each stage, investigating how an honest Operator can always enforce the correct outcome while a Challenger or a Watchtower can always successfully challenge a malicious Operator's incorrect claim. Finally, we analyze the hash rate resilience of our scheme against a Byzantine adversary trying to fool the light client by mining a private fork.

9.1 KickOff Transaction Not Finalized

Malicious Operator. An Operator posting a $\textbf{KickOff}_{i,j}$ transaction can be forced to respond to $\textbf{WatchtowerChallenge}_{i,j}$ (via the transaction $\textbf{OperatorChallengeACK}_{i,j}$), and to $\textbf{Challenge}_{i,j}$ (via $\textbf{BlockHash}_{i,j}$ and all $\textbf{Assert}_{i,j,l}$ transactions). Regardless of how these mechanisms work (cf. Sections 9.2 and 9.3 for details on that), if the Operator remains unresponsive to these challenges, the $\textbf{KickOff}_{i,j}$ transaction remains unfinalized, which is indicated by $\textbf{KickOff}_{i,j}.\mathsf{output}[1]$ ("Finalizer Connector") being unspent. Should the Operator try to get to reimbursement by posting the

$\mathbf{ReadyToReimburse}_i$ with the hope of also posting $\mathbf{Round}_{i+1}$ and $\mathbf{Reimburse}_{i,j}$, a Challenger has a window of t_9 blocks (timelock of $\mathbf{ReadyToReimburse}_i$'s only output) to post the transaction $\mathbf{KickOffNotFinalized}_{i,j}$, which will spend the output of $\mathbf{ReadyToReimburse}_i$, destroying any chance of this Operator to complete this or any future reimbursement.

Since this mechanism is only active for the largest i, such that $\mathbf{Round}_i$ is on-chain for that Operator (and not for previous i), we need to ensure that Operators do not use outputs of stale $\mathbf{Round}_i$ transactions (for any i that is not the largest one, such that $\mathbf{Round}_i$ is on-chain for that Operator). An Operator is thus supposed to post a $\mathbf{BurnUnusedKickOff}_{i,j}$ for every output with index j that is unspent before posting the $\mathbf{ReadyToReimburse}_i$. Otherwise, a Challenger has a window of t_9 blocks to post $\mathbf{UnspentKickOff}_{i,j}$, destroying any chance of this Operator completing this or any future reimbursement.

Honest Operator. An honest Operator can always make sure to post a $\mathbf{BurnUnusedKickOff}_{i,j}$ before posting a $\mathbf{ReadyToReimburse}_i$, thereby preventing any (malicious) Challenger from posting an $\mathbf{UnspentKickOff}_{i,j}$ transaction. Similarly, an honest Operator can always make sure that a $\mathbf{KickOff}_{i,j}$ is finalized, i.e., the "Finalizer Connector" is spent: Either via a $\mathbf{NoChallenge}_{i,j}$ or via a $\mathbf{DisproveTimeout}_{i,j}$ transaction.

9.2 Disputing the KickOff Transaction

Malicious Operator. A malicious Operator can try to post a $\mathbf{KickOff}_{i,j}$ with an incorrect blockhash commitment $\mathsf{WinCom}(\mathsf{bh}_{i,j})$ as witness, i.e., one that does not below to the honest, longest Bitcoin chain. A malicious Operator can try this, putting a blockhash commitment of a private fork that contains a **Payout** where the Operator is paying the user who is pegging out; however, the **Payout** is not part of the honest longest chain.

The Challengers have a window of t_5 blocks (after $\mathbf{KickOff}_{i,j}$ appears on-chain) to post the $\mathbf{Challenge}_{i,j}$ transaction before the Operator can post $\mathbf{NoChallenge}_{i,j}$. After posting the challenge, the Operator has a window of t_7 blocks (after $\mathbf{KickOff}_{i,j}$ appears on-chain) to post $\mathbf{BlockHash}_{i,j}$ and a window of t_6 blocks (after $\mathbf{KickOff}_{i,j}$ appears on-chain) to post all $\mathbf{Assert}_{i,j,l}$. If the Operator posts neither, a Challenger can come in and post $\mathbf{AssertTimeout}_{i,j,l}$. If these transactions appear on-chain, the Challengers check whether all the computations of the SNARK verifier were done correctly. If not, a Challenger can always post a $\mathbf{Disprove}_{i,j}$ using the corresponding $\mathsf{BridgeDisproveScript}$. Further, if the Operator was not using all the chain proofs provided by the Watchtowers (cf. Section 9.3), a Challenger can always post $\mathbf{Disprove}_{i,j}$ using the corresponding $\mathsf{ClementineDisproveScript}$.

Honest Operator. An honest Operator, on the other hand, will always have enough time to post $\mathbf{BlockHash}_{i,j}$ and all $\mathbf{Assert}_{i,j,l}$, in case of $\mathbf{Challenge}_{i,j}$ (or to post $\mathbf{NoChallenge}_{i,j}$ otherwise). Similarly, an honest Operator can always post the correct $\mathbf{Assert}_{i,j,l}$ corresponding to the correct SNARK verifier computation results. Thus, there is no $\mathbf{Disprove}_{i,j}$ possible. See Section 9.4 for more details.

9.3 Incorrect Longest Chain Proof

Malicious Operator. Suppose a malicious Operator tries to post a $\mathbf{KickOff}_{i,j}$ with an incorrect blockhash commitment $\mathsf{WinCom}(\mathsf{bh}_{i,j})$ as witness, and a challenge was started in response (as outlined in Section 9.2). In this case, Watchtowers can always post a $\mathbf{WatchtowerChallenge}_{i,j,k}$ within a time window of t_3 blocks. Should the Operator then fail to post $\mathbf{OperatorChallengeACK}_{i,j,k}$ a Challenger can post $\mathbf{OperatorChallengeNACK}_{i,j,k}$.

Honest Operator. Given $\mathsf{t}_4\ \mathsf{blocks} > \mathsf{t}_3\ \mathsf{blocks} + u$, an honest Operator always has enough time, in particular a window of $\mathsf{t}_4\ \mathsf{blocks} - \mathsf{t}_3\ \mathsf{blocks}$, to post a $\mathbf{OperatorChallengeACK}_{i,j,k}$ in response to a $\mathbf{WatchtowerChallenge}_{i,j,k}$.

9.4 Private Mining Attack

Let us consider the private mining attack in Bitcoin [29], aka. private double-spend attack, which has been proven to be the worst attack in [20]. In a private mining attack, the adversary mines a local, private chain of blocks, racing to take over the public longest chain and replace it with the private one once a block in the public chain becomes k-deep. Let λ_h and λ_α be the rate at which the honest nodes and the adversary mine blocks, proportional to their respective hashing powers. If $\lambda_\alpha < \lambda_h$, the probability that the adversary succeed in pulling off the attack decreases exponentially in k. We denote with T denote expected block time, which in Bitcoin is of 10 minutes per block, and with $\lambda = \frac{1}{T}$ the block production rate. We model the mining process as a Poisson process with rate λ, since block discoveries happen independently and randomly over time [33].

Let $h + \alpha = 1$ be the total hashrate in the network; we constrain the adversarial hashrate α such that $0 \leq \alpha < 0.5$. The adversarial mining rate is thus $\lambda_a = \alpha\lambda = \frac{\alpha}{T}$. From [20], we know that for a private attack on Nakamoto's Proof-of-Work protocol to *not* be successful, it must hold that $\lambda_\alpha < \frac{(1-\alpha)\lambda}{1+(1-\alpha)\lambda\delta}$: being δ the network delay between honest nodes [32] and $\lambda\delta$ the number of blocks mined per network delay, we have that the honest mining rate in the worst case is $\lambda_h \geq \frac{(1-\alpha)\lambda}{1+(1-\alpha)\lambda\delta}$. Note that if $\delta \ll T$, as it is the case in Bitcoin, then $\lambda_h \approx (1-\alpha)\lambda = \frac{1-\alpha}{T}$.

In the following we consider the worst-case forking [32] taking place in an ideal decentralized network, where the computational power is well distributed among many equidistant nodes. We also consider a powerful adversary whose valid blocks and transactions, when broadcast, are immediately included on chain; Honest miners, instead, can include a transaction or a block after a maximum delay of u, with u being the liveness parameter. In the following we will use time and blocks interchangeably to refer to time: the conversion from block to blocks can be approximated by the expected block rate of the chain.

Malicious Operator

Let r_1 be the time at which a block including the **KickOff** transaction is mined. The timelock of the **KickOff**'s latest blockhash connector expires at $r_1 + \mathsf{t_7\ blocks}$. A malicious Operator is able to choose his latest commitment to a block hash via the **BlockHash** transaction at most at round $r_1 + \mathsf{t_7\ blocks} + u$. Since $\mathsf{t_7\ blocks} = \mathsf{t_3\ blocks} + x$ (for some $x > 0$), the malicious Operator has a time window of $t_a = \mathsf{t_3\ blocks} + x + u$ to launch a private mining attack. An honest Watchtower, on the other hand, has to choose the longest chain at $r_1 + \mathsf{t_3\ blocks} - u$, thus the time window in which the honest chain grows is $t_h = \mathsf{t_3\ blocks} - u$. Note that the adversary has a slightly bigger window and, thus, a small advantage compared to a classical private mining attack scenario. We define the following random variables.

- $X_h \sim \text{Pois}(\lambda_h \cdot t_h)$ is the random variable counting the number of honest blocks that are found in the time window t_h.
- $X_a \sim \text{Pois}(\lambda_a \cdot t_a)$ is the random variable counting the number of adversarial blocks that are found in the time window t_a.

The probability that the adversary wins this race is thus $\Pr[X_a > X_h]$, which we can write as $\sum_{k=0}^{\infty} \left(Pr[X_h = k] \cdot \sum_{l=k+1}^{\infty} Pr[X_a = l]\right)$. Alternatively the difference of two independent, Poisson-distributed random variables follows a Skellam distribution.

Examples. We can compute the failure probability for some realistic numbers. Suppose $\delta = 8$ seconds, $\mathsf{t_3\ blocks} = 2$ weeks, $\mathsf{t_7\ blocks} = \mathsf{t_3\ blocks} + 1$ day, $u = 2$ hours. We thus have $t_a = 15$ days 2 hours, and $t_h = 13$ days 22 hours. The failure probability for a given α is as follows:

- $\alpha = 0.45$, $\Pr[X_a > X_h] \approx 0.005$
- $\alpha = 0.44$, $\Pr[X_a > X_h] \approx 0.0002$
- $\alpha = 0.43$, $\Pr[X_a > X_h] \approx$ 5e-06
- $\alpha = 0.42$, $\Pr[X_a > X_h] \approx$ 5e-08

Similarly, we can achieve a lower failure probability (for a higher resilience α) by increasing $\mathsf{t_3\ blocks}$. Choosing $\mathsf{t_3\ blocks}$ high enough, we can get an arbitrarily low failure probability for any $\alpha < 0.5$.

Honest Operator

A malicious Watchtower can choose a commitment to the chain at the latest at $r_1 + \mathsf{t}_3\ \mathsf{blocks} + u$, while an honest Operator can choose the latest commitment to a block hash at $r_1 + \mathsf{t}_7\ \mathsf{blocks} - u$. If $\mathsf{t}_7\ \mathsf{blocks} > \mathsf{t}_3\ \mathsf{blocks} + 2u$, this collapses to a standard Bitcoin private mining race (the adversary even has a disadvantage of $\mathsf{t}_7\ \mathsf{blocks} - \mathsf{t}_3\ \mathsf{blocks} - 2u$). Again, choosing $\mathsf{t}_3\ \mathsf{blocks}$ high enough, we can get an arbitrarily low failure probability for any $\alpha < 0.5$.

10 Conclusions

In this whitepaper we have introduced Clementine, a secure, collateral-efficient, trust-minimized, and scalable Bitcoin bridge. Clementine is based on BitVM2 and it enables secure withdrawals from rollups or other side systems to Bitcoin. The Bitcoin light client of Clementine is the first to remain secure against an adversary that controls strictly less than 50% of the network hash rate and as long as at least one honest Watchtower exists within a permissioned set. The protocol presents several advancements with respect to state-of-the-art bridge protocols: it is collateral-efficient, allowing collateral to be reused over time, and scalable, as a single challenge can slash multiple misbehaviors by an Operator.

We view Clementine as a crucial step towards having more practical and secure bridges on Bitcoin and we hope that our work encourages further research on the interoperability and scalability of Bitcoin.

Acknowledgments

We thank Robin Linus and Sergio Demian Lerner for their fruitful discussions and feedback. We are grateful to Ozan Kaymak, Ahmet Oğuz Engin, and Şevket Onur Yılmaz for their reviews. We are also deeply grateful to the Bitcoin developers and the broader Bitcoin community for their years of dedicated, passionate, and tireless work. Their collective efforts have significantly advanced Bitcoin's usability and scripting capabilities, laying the essential groundwork that made our contribution both possible and impactful. Finally, we extend our gratitude to everyone who has contributed — and will contribute — to BitVM and its extensions. Their work has been, and will continue to be, a source of inspiration for us as we strive to further enhance Citrea and contribute to the growth of the Bitcoin ecosystem.

References

1. Citrea. `https://citrea.xyz/`.
2. Lamport Signature - Short Private Key. `https://en.wikipedia.org/wiki/Lamport_signature?utm_source=chatgpt.com#Short_private_key`.
3. An overview of recent non-standard Bitcoin transactions, 2024. `https://b10c.me/observations/09-non-standard-transactions/`.
4. BIP: 431, Topology Restrictions for Pinning, 2024. `https://github.com/bitcoin/bips/blob/master/bip-0431.mediawiki#specification`.
5. Bitcoin Stack Exchange: What is meant by ”non-standard”?, 2024. `https://bitcoin.stackexchange.com/questions/122858/what-is-meant-by-non-standard`.
6. Ephemeral Anchors, 2024. `https://bitcoinops.org/en/topics/ephemeral-anchors/]`.
7. Ephemeral Dust, 2024. `https://github.com/bitcoin/bitcoin/pull/30239`.
8. iShares Bitcoin Trust ETF, 2024. `https://www.blackrock.com/us/individual/products/333011/ishares-bitcoin-trust`.
9. Mempool Limits, 2024. `https://github.com/bitcoin/bitcoin/blob/master/doc/policy/mempool-limits.md#cpfp-carve-out`.
10. Bitcoin optech: Covenants, 2025. `https://bitcoinops.org/en/topics/covenants/`.
11. Bitcoin stock-to-flow model, 2025. `https://www.bitcoinmagazinepro.com/charts/stock-to-flow-model/`.
12. Trading view, 2025. `https://www.tradingview.com/markets/cryptocurrencies/prices-all/`.
13. L. Aumayr, Z. Avarikioti, R. Linus, M. Maffei, A. Pelosi, C. Stefo, and A. Zamyatin. BitVM: Quasi-Turing Complete Computation on Bitcoin. Cryptology ePrint Archive, Paper 2024/1995, 2024.
14. L. Aumayr, Z. Avarikioti, R. Linus, M. Maffei, A. Pelosi, and A. Zamyatin. BitVM2: Bridging Bitcoin to Second Layers, 2024.
15. Z. Avarikioti. The Battle of Blockchains: PoW vs PoS Unveiled , 2024. `https://www.commonprefix.com/blog/battle-of-blockchains`.

16. M. Bartoletti, S. Lande, and R. Zunino. Bitcoin covenants unchained. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Applications*, pages 25–42. Springer International Publishing, 2020.
17. E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Scalable zero knowledge via cycles of elliptic curves. In J. A. Garay and R. Gennaro, editors, *Advances in Cryptology – CRYPTO 2014*, pages 276–294, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.
18. Bitcoin Wiki. Script, 2025. `https://en.bitcoin.it/wiki/Script`.
19. J. Buchmann, E. Dahmen, S. Ereth, A. Hülsing, and M. Rückert. On the security of the winternitz one-time signature scheme. In *Progress in Cryptology – AFRICACRYPT 2011*, pages 363–378, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.
20. A. Dembo, S. Kannan, E. N. Tas, D. Tse, P. Viswanath, X. Wang, and O. Zeitouni. Everything is a race and nakamoto always wins. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, page 859–878, New York, NY, USA, 2020. Association for Computing Machinery.
21. X. Dong, O. S. T. Litos, E. N. Tas, D. Tse, R. L. Woll, L. Yang, and M. Yu. Remote staking with economic safety, 2024.
22. J. Garay, A. Kiayias, and N. Leonardos. The bitcoin backbone protocol with chains of variable difficulty. In *Advances in Cryptology – CRYPTO 2017.* Springer International Publishing, 2017.
23. C. Gentry and D. Wichs. Separating succinct non-interactive arguments from all falsifiable assumptions. In *Proceedings of the Forty-Third Annual ACM Symposium on Theory of Computing*, STOC '11, page 99–108, New York, NY, USA, 2011. Association for Computing Machinery.
24. M. Green. Hash-based Signatures: An illustrated Primer, 2018. `https://blog.cryptographyengineering.com/2018/04/07/hash-based-signatures-an-illustrated-primer/?utm_source=chatgpt.com`.
25. J. Groth. On the size of pairing-based non-interactive arguments. In M. Fischlin and J.-S. Coron, editors, *Advances in Cryptology – EUROCRYPT 2016*, pages 305–326, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.
26. T. W. House. Establishment of the strategic bitcoin reserve and united states digital asset stockpile, 2025. `https://www.whitehouse.gov/presidential-actions/2025/03/establishment-of-the-strategic-bitcoin-reserve-and-united-states-digital-asset-stockpile/`.
27. Learn me a Bitcoin. Bitcoin script, 2025. `https://learnmeabitcoin.com/technical/script/`.
28. S. D. Lerner, R. Amela, S. Mishra, M. Jonas, and J. Álvarez Cid-Fuentes. BitVMX: A CPU for Universal Computation on Bitcoin, 2024. `https://arxiv.org/abs/2405.06842`.
29. S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System, 2009. `http://bitcoin.org/bitcoin.pdf`.
30. F. Plesu. Turing completeness in evm machines, 2023. Accessed: 2025-04.
31. G. Scaffino, L. Aumayr, M. Bastankhah, Z. Avarikioti, and M. Maffei. Alba: The dawn of scalable bridges for blockchains. In *32nd Network and Distributed System Security Symposium, NDSS 2025.* ISOC, 2025.
32. Y. Sompolinsky and A. Zohar. Secure high-rate transaction processing in bitcoin. In *Financial Cryptography*, 2015.
33. P. Viswanath. Safety of Bitcoin, 2024. `https://courses.grainger.illinois.edu/ece598pv/sp2021/lectureslides2021/ECE_598_PV_course_notes6.pdf`.
34. A. Zamyatin, M. Al-Bassam, D. Zindros, E. Kokoris-Kogias, P. Moreno-Sanchez, A. Kiayias, and W. J. Knottenbelt. SoK: Communication across distributed ledgers. Cryptology ePrint Archive, Paper 2019/1128, 2019.
35. D. Zindros. Blockchain Foundations. 2023. `https://ee374.stanford.edu/blockchain-foundations.pdf`.