Glimpse: On-demand Light Client with Constant-size Storage for DeFi

Giulia Scaffino[1,2], Lukas Aumayr[1], Zeta Avarikioti[1], and Matteo Maffei[1,2]

[1]TU Wien, {giulia.scaffino, lukas.aumayr, georgia.avarikioti, matteo.maffei}@tuwien.ac.at
[2]Christian Doppler Laboratory Blockchain Technologies for the Internet of Things

Abstract

Cross-chain communication is instrumental in unleashing the full potential of blockchain technologies, as it allows users and developers to exploit the unique design features and the profit opportunities of different existing blockchains. The majority of interoperability solutions are provided by centralized exchanges and bridge protocols based on a trusted majority, both introducing undesirable trust assumptions compared to native blockchain assets. Hence, increasing attention has been given to decentralized solutions: Light and super-light clients paved the way to chain relays, which allow verifying on a blockchain the state of another blockchain by respectively verifying and storing a linear and logarithmic amount of data. Unfortunately, relays turn out to be inefficient in terms of computational costs, storage, or compatibility.

We introduce *Glimpse*, an *on-demand bridge* that leverages a novel *on-demand light client* construction with only *constant* on-chain storage, cost, and computational overhead. Glimpse is *expressive*, enabling a plethora of DeFi and off-chain applications such as lending, pegs, proofs of oracle attestations, and betting hubs. Glimpse also remains *compatible* with blockchains featuring a limited scripting language (e.g., Bitcoin-based chains like Liquid), for which we present a concrete instantiation. We prove Glimpse security in the Universal Composability (UC) framework and further conduct an economic analysis. We evaluate the cost of Glimpse for Bitcoin-like chains: verifying a simple transaction has at most 700 bytes of on-chain overhead, resulting in a *one-time* fee of $3, only twice as much as a standard Bitcoin transaction.

1 Introduction

The blockchain landscape is fragmented into a plethora of blockchains featuring different technical ingredients (scripting languages, consensus mechanisms, etc.) and attracting users for their unique properties (e.g., Bitcoin for its robust design, Monero and ZCash for their privacy, Ethereum for the support of DeFi applications, Algorand for its high throughput). Blockchain platforms already hold an impressive amount of investments, users, and developers, who are often reluctant to migrate their assets and contracts to other chains. By providing interoperability solutions, centralized exchanges have enabled an appealing ecosystem of financial applications, such as trading different cryptocurrencies, collateral-based lending, and more. Unfortunately, centralized exchanges need to be trusted; additionally, they can be hacked, go bankrupt, or be fraudulent, in which case the users' money is at risk. The same holds true for solutions assuming a trusted majority of validators: e.g., an attacker managed to acquire five of the nine validation keys used in the Ronin bridge [1], stealing $624M; attacks on Wormhole ($320M), Nomad ($200M) or Harmony ($100M) and more, totaling in over $1.3B of stolen funds in the first 8 months of 2022 alone [2]. For these reasons, the design of decentralized interoperability solutions is crucial to unleashing the full potential of blockchain technologies.

Two of the fundamental challenges in blockchain interoperability are: (i) designing an efficient light client to allow a user of a destination chain $\mathcal{L}_D$ to verify that a transaction Tx_S has been included in a source blockchain $\mathcal{L}_S$ without participating in $\mathcal{L}_S$'s consensus protocol (cross-chain verification) and (ii) synchronizing transactions across different chains (cross-chain atomicity), e.g., in an atomic swap, a transaction Tx_D on $\mathcal{L}_D$ succeeds if and only if Tx_S was posted on $\mathcal{L}_S$.

Related Work. The core idea of *light clients*, illustrated for the first time in Nakamoto's original paper [3] for Simplified Payment Verification (SPV), is to store and verify the block headers alone, as opposed to the whole block, and to verify which chain carries the most Proof-of-Work (PoW). The assumption underlying the security of light clients is that the majority of miners follow the consensus rules; therefore, the chain with the most PoW represents the honest chain. SPV-based light clients save storage (a Bitcoin block header is about 80B in size, whereas a block is about 1MB) but still require the download and processing of a *linear* amount of information, with an overhead of 60MB for Bitcoin and 4GB for Ethereum. Light clients and SPVs are the basis of chain relays [4–6], an expressive but expensive solution to the cross-chain verification problem. They verify and store every block header of a source ledger $\mathcal{L}_S$ within a smart contract on a

destination ledger $\mathcal{L}_D$, thereby acting as light clients. The inefficiency of this construction, associated with the lack of incentives, is arguably one of the reasons why relays are not used in practice.[1] Later on, *super-light clients* with logarithmic complexity were proposed, but they require constant PoW difficulty [7] or require a hard fork in Bitcoin [8], and are thus not backward compatible. More recently, Xie et al. [9] have introduced an Ethereum-compatible bridge that requires, for the first time, constant size storage of a zk-SNARK proof guaranteeing that the blockchain has undergone a state update (single block or a bunch of them). Each verified state update is then stored (in a non-constant manner) within a helper contract and made available to other applications for SPV verifications.

All these solutions, however, require a quasi Turing-complete scripting language in the destination chain and are thus not compatible with blockchains with limited scripting capabilities, such as Bitcoin-based chains. The expressiveness of the scripting language is indeed one of the features setting apart different blockchains, with some (e.g., Ethereum) favoring the support for DeFi applications (albeit some smart contracts can be encoded in the Bitcoin scripting language too [10]) and others (e.g., Bitcoin) more conservatively arguing for a reduced trust base and easier script verification. For this reason, supporting blockchains with limited scripting capabilities is not only theoretically challenging but also a practically relevant research goal.

A different approach to the realization of bridges with constant size storage is *stateless SPV*, recently emerged within the Ethereum research community [11] and currently being integrated into existing systems: for instance, Barbára et al. [12] recently implemented, and for the first time formalized, stateless SPV within the BxTB cross-chain exchange. Instead of verifying all blockchain headers, the idea is to perform a proof of inclusion *on-demand* for a specific transaction: for that, one needs to verify the headers of a sufficiently long subchain whose first block contains the transaction of interest. The authors conduct an economical security analysis, showing that stateless SPV suffices to discourage attacks on the system (i.e., to construct sufficiently long invalid chains), as it would be economically more profitable to invest the mining power to honestly mine blocks. In this work, we introduce an attack against stateless SPV, which we call *upfront mining attack*: A malicious prover may produce upfront a forged proof, leveraging the fact that users on $\mathcal{L}_D$ have no way to ensure that the proof corresponds to the suffix of a valid chain. Since there is no backward time constraint on performing an upfront mining attack, the attacker will eventually succeed in finding a forged block regardless of her mining power and without needing to bribe or convince any miners. This attack is not considered in stateless SPV and BxTB and gives them a strictly weaker security notion than, e.g., SPV-based light clients where this cannot happen due to the honest majority assumption.

Finally, *lock contracts* such as Hashed TimeLock Contracts (HTLCs) and adaptor signatures [13] constitute an appealing secret-based cryptographic technique for cross-chain atomic synchronization, which use a statement S that ties the authorization of a transaction Tx_D on $\mathcal{L}_D$ to the leakage of a secret witness s of some hard relation within a transaction Tx_S posted on-chain $\mathcal{L}_S$. Lock contracts, however, require (e.g., in the context of atomic swaps) both parties to monitor and actively participate in both chains. Furthermore, they have limited expressiveness as they can only encode a class of asymmetric problems, where the proof of inclusion, i.e., the secret s, is only known to the party posting in $\mathcal{L}_D$ for security reasons. The former limitation renders lock contracts inefficient while the latter hinders their use in several applications such as Proofs-of-Burn and wrapping/unwrapping of tokens.

To summarize, the present research landscape leaves the following research question open: *"Is it possible to design a secure solution for cross-chain verification which guarantees cross-chain atomicity, requires constant size storage, and makes use of limited scripting language on the target chain?"*.

Our Contributions. In this work, we positively answer the above question by presenting Glimpse, the first secure on-demand bridge that achieves *atomicity* and *constant size storage* by leveraging a novel on-demand light client, while retaining *compatibility with Bitcoin-like target chains*. To achieve only constant-size overhead, our light client assumes the knowledge of the current PoW target and is acting on-demand, verifying only current transactions. In particular, Glimpse allows a *prover* and a *verifier* to establish a contract enforcing that if a specific set of transactions Tx_S are confirmed on a PoW $\mathcal{L}_S$ within a given time after the contract is settled (we call this time frame *contract validity*), then another set of transactions Tx_D can be published on $\mathcal{L}_D$. Technically, Glimpse is a contract living on $\mathcal{L}_D$, which receives from the prover a *proof* that Tx_S was included on $\mathcal{L}_S$ with the desired number of confirmations and enables Tx_D to appear on $\mathcal{L}_D$. Glimpse reconciles the *low on-chain costs and simple design* of lock contracts with the *expressiveness* of chain relays.

Glimpse builds on the notion of stateless SPV, but it refines and generalizes it in a number of ways. First, we propose a generic technique to *prevent upfront mining*, imposing prover and verifier to agree on a random value to be inserted in the to-be-verified transaction itself, without requiring any of them to verify $\mathcal{L}_S$. Second, we generalize stateless SPV to *support applications in which part of the transaction* Tx_S *is not known a priori* but is instead determined at run-time (e.g., lending, where for the loan payback transaction, the input it is not known beforehand, as the lent money can be used arbitrarily), as well as *applications requiring the synchronization of combinations of transactions on* $\mathcal{L}_S$. In particular, this allows to encode that if any set of transactions satisfying a logical formula expressed as Disjunctive Normal Form (DNF) (e.g., $\mathsf{Tx}_S \vee \mathsf{Tx}_S{}'$) is published on $\mathcal{L}_S$, then Tx_D can be published

[1]For instance, the most popular Bitcoin relay on Ethereum [4] stopped its development in 2017 and the last transaction is from about 4 years ago.

on $\mathcal{L}_D$. These generalizations allow us to encode a variety of DeFi applications, such as lending, pegs, wrapping/unwrapping of tokens, Proof-of-Burn, verification of multiple oracle attestations, and layer-2 applications such as cross-chain virtual channels, payments, and betting hubs. Third, we provide a construction that, for the first time, does not require quasi Turing-complete scripting languages on the destination chain and is thus compatible with Bitcoin-derived blockchains.

Our further contributions can be summarized as follows:

- We demonstrate the expressiveness of Glimpse by encoding a variety of DeFi and off-chain applications (Section 4);
- We formally analyze Glimpse in the UC framework, where we prove its atomicity (Section 5).
- We conduct an economic security analysis to quantify the costs of forgery (affecting any light client) and censorship attacks (harming any timelock-based protocol). Specifically, Glimpse is secure against proof forgery as long as the value simultaneously locked on all valid Glimpse contracts does not exceed a certain threshold (for concrete numbers, \$230M), which is equivalent to the value currently locked on popular bridges. We further impose an upper bound on the value held by each Glimpse contract to be secure against censorship on the destination chain, e.g., \$1.1M for Glimpse deployed on Ethereum (Section 6).
- We demonstrate the practicality of Glimpse by evaluating its on-chain costs in Ethereum- and Bitcoin-like chains showing that, e.g., in Bitcoin the overall cost is at most \$3, around twice as much as ordinary transactions. We also further optimize it with Taproot [14, 15] (Section 7).

2 Background

The UTXO Transaction Model. Each user U is identified by a pair of digital keys $(\mathsf{pk}_U, \mathsf{sk}_U)$ that are used to prove ownership over coins. A transaction $\mathsf{Tx} = (\mathsf{cntr}_{in}, \mathsf{inputs}, \mathsf{cntr}_{out}, \mathsf{outputs}, \mathsf{witnesses})$ is an atomic update of the blockchain state and is associated to a unique identifier $\mathsf{txid} \in \{0,1\}^{256}$ defined as the *hash* $\mathcal{H}([\mathsf{Tx}])$ *of the transaction*, where $[\mathsf{Tx}] := (\mathsf{cntr}_{in}, \mathsf{inputs}, \mathsf{cntr}_{out}, \mathsf{outputs})$is the *body of the transaction*. Intuitively, a transaction maps a non-empty list of inputs to a non-empty list of newly created outputs, describing a redistribution of funds from the users identified in the inputs to those identified in the outputs.

$\mathsf{cntr}_{in}, \mathsf{cntr}_{out} \in \mathbb{N}_{>0}$ represent the number of elements in the inputs and outputs lists. Any input ζ in the list of inputs is an unspent output from an older transaction, defined by the tuple $\zeta := (\mathsf{txid}, \mathsf{outid})$, with $\mathsf{txid} \in \{0,1\}^{256}$ representing the hash of the old transaction containing the to-be-spent output, and $\mathsf{outid} \in \mathbb{R}_{\geq 0}$ the index of such an output within the output list of the old transaction. These two fields uniquely identify the to-be-spent output. $\mathsf{witnesses} \in \{0,1\}^*$, also known as *scriptSig* or *unlocking script*, is a list of witnesses ω, i.e., the data that only the entity entitled to spend the output can provide, thereby authenticating and validating the transaction. Any output θ in the list of outputs is a pair $\theta := (\mathsf{coins}, \phi)$ and can be consumed by at most one transaction (i.e., no double-spend). The amount of coins in an output θ is denoted by $\mathsf{coins} \in \mathbb{R}_{\geq 0}$, whereas the spendability of θ is restricted by the conditions in ϕ, also known as the *scriptPubKey* or *locking script*. Such conditions are modeled in the native scripting language of the blockchain and can vary from single-user $\mathsf{OneSig}(\mathsf{pk}_U)$ and multi-user $\mathsf{MuSig}(\mathsf{pk}_{U1}, \mathsf{pk}_{U2})$ ownership, to time locks, hash locks, and more complex scripts.

Proof-of-Work Consensus. In a PoW blockchain, the probability that a node is selected as block proposer is proportional to its computational power. This is meant to hinder Sybil attacks since computational power is assumed hard to monopolize. Specifically, incentivized to win the reward in native assets, the nodes compete with each other to create, validate, and append new blocks to the ledger by solving a cryptographic puzzle that is hard to compute and easy to verify. The content of a block is summarized within a unique and cryptographically secured string that grants immutability to the blockchain: the *block header* $\mathsf{header}(\mathsf{B}) := (\mathsf{ParentHash}, \mathsf{MR}, \mathsf{Timestamp}, \mathsf{nBits}, \mathsf{Nonce})$, where $\mathsf{ParentHash}$ is the hash of the previous block, MR is the root of the Merkle tree whose leaves are the transactions in B, $\mathsf{Timestamp}$ is the creation time of the block, nBits is a parameter for the target space, and Nonce a value that can be arbitrarily iterated to reach the PoW.

In particular, the nodes, called *miners*, repeatedly change the Nonce field of the block header until the hash of the header lies within a *target space* that is smaller (by several orders of magnitude) than the output space of the hash function. This is a necessary condition for the block to be *valid*. The size of the target space is parameterized by the total computational power of the network and is periodically adjusted to keep the expected *block time*, i.e., the time it takes to find a valid block, almost constant. We refer to the *target* as $\mathcal{T}$, and we say that a block B is valid when $\mathcal{H}(\mathsf{header}(\mathsf{B})) < \mathcal{T}$. A miner is selected to propose the next block with probability proportional to the fraction of the network's hashing power he controls. PoW blockchains periodically adjust the network difficulty to maintain an (almost) constant average block creation time, preventing uncontrolled inflation and network congestion.

3 Glimpse

We introduce Glimpse, a new *primitive for cross-chain communication* that allows participants to obtain *on demand* the desired information about the state of a PoW source ledger $\mathcal{L}_S$ on a destination ledger $\mathcal{L}_D$. Glimpse achieves this without executing a light client, with only a *constant amount of data*, and assuming the *PoW target is known*.

In particular, Glimpse resembles challenge-response protocols: On $\mathcal{L}_D$, a *prover* P and *verifier* V argue about the inclusion on $\mathcal{L}_S$ of a *specific* set of transactions Tx_S (challenge). Depending on the outcome, they want to publish on

$\mathcal{L}_D$ different transactions. To solve the argument, P and V first agree on the Glimpse specifics and some consensus parameters of $\mathcal{L}_S$, then deploy a *Glimpse contract* on $\mathcal{L}_D$. On ledger $\mathcal{L}_S$, an *issuer* I publishes the transaction set Tx_S, and an *off-chain untrusted relayer* R provides P with the necessary data to construct a *proof* $\mathcal{P}$ to prove the occurrence of Tx_S on $\mathcal{L}_S$. If P submits a valid proof (response) to the Glimpse contract on $\mathcal{L}_D$, he can post a pre-defined Tx_P on $\mathcal{L}_D$. Else, V can post a pre-defined Tx_V after time T has elapsed.

3.1 Assumptions and Models

System Model. We assume a source ledger $\mathcal{L}_S$ operating a PoW consensus. Glimpse relies on four parties: an *issuer* I that publishes transaction(s) Tx_S on $\mathcal{L}_S$, a prover P that proves the occurrence of Tx_S on $\mathcal{L}_D$, a relayer R (e.g., blockchain explorers, full nodes) that provides P with the necessary information to construct the proof $\mathcal{P}$, and a verifier V that guarantees contractual fairness.

We require P and V to have a key pair $(\mathsf{pk},\mathsf{sk})$ on $\mathcal{L}_D$, and I to have a key pair on $\mathcal{L}_S$. The Glimpse contract is deployed on $\mathcal{L}_D$ and holds coins either coming from P, V or from any other user of $\mathcal{L}_D$. We assume $\mathcal{L}_D$ to support the same hash function used by $\mathcal{L}_S$, and both $\mathcal{L}_S$ and $\mathcal{L}_D$ to allocate the same domain for the hash function, to avoid oversize preimage attacks [16,17]. Finally, $\mathcal{L}_D$ needs to support the following functionalities: (i) Merkle proof verification, (ii) hash comparison, and (iii) block header and transaction body reconstruction. While (ii) is supported by default in most chains, (i) and (iii) can also be supported by Bitcoin-based chains by enabling a concatenation opcode (e.g., already discussed in the context of Speedy Covenants [18]).

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

Communication Model. We assume there exist authenticated communication channels between the Glimpse parties, where all messages are delivered within a fixed time delay.

Cross-chain Communication (CCC) Model. Closely following [16], CCC protocols are usually articulated in three main phases: *Setup*, *Commit on* $\mathcal{L}_S$, and *Verify & Commit on* $\mathcal{L}_D$. The *Setup* phase parameterizes the involved blockchains, identifies the protocol participants, and specifies the transactions Tx_S and Tx_D to be synchronized. After a successful setup, in the *Commit on* $\mathcal{L}_S$ phase, a publicly verifiable commitment to execute the CCC protocol, i.e., Tx_S, is posted on $\mathcal{L}_S$. In the *Verify & Commit on* $\mathcal{L}_D$ phase, $\mathcal{L}_D$ verifies the commitment on $\mathcal{L}_S$ and, upon successful verification, a publicly verifiable commitment, i.e., Tx_D, is posted on $\mathcal{L}_D$. An optional *abort* phase reverts transaction Tx_S on $\mathcal{L}_S$ in case the verification of the commitment failed or the commitment on $\mathcal{L}_D$ is not executed.

A CCC protocol has to give some atomicity guarantees, which, for Glimpse, we articulate in a weak and strong variant. Simplified, weak atomicity ensures that Tx_D can only appear on $\mathcal{L}_D$ if Tx_S was already confirmed on $\mathcal{L}_S$. On the other hand, strong atomicity additionally ensures that Tx_D will appear on $\mathcal{L}_D$ after Tx_S has been confirmed on $\mathcal{L}_S$. Let $\Delta_D \in \mathbb{N}$ be the *wait time parameter* of $\mathcal{L}_D$, i.e., the upper bound of time it takes for a valid transactions to be included on $\mathcal{L}_D$. We consider n the number of confirmation blocks that need to be mined on top of a block containing a transaction Tx for Tx to be stable [19] on a PoW ledger.

Definition 1 (*Weak Atomicity*)**.** Let Tx_S and Tx_D be (sets of) transactions for $\mathcal{L}_S$ and $\mathcal{L}_D$, respectively. If honest player of $\mathcal{L}_D$ report a valid Tx_D as stable at time t, then honest players of $\mathcal{L}_S$ have reported Tx_S with at least n confirmations at time $t' \leq t - \Delta_D$: $\mathsf{Tx}_D \in \mathcal{L}_D \implies \mathsf{Tx}_S \in \mathcal{L}_S$.

Definition 2 (*Strong Atomicity*)**.** Let Tx_S and Tx_D be (sets of) transactions for $\mathcal{L}_S$ and $\mathcal{L}_D$, respectively. Tx_D is reported stable by honest players on $\mathcal{L}_D$ at time t if and only if honest players of $\mathcal{L}_S$ have reported Tx_S with at least n confirmations at time $t' \leq t - \Delta_D$: $\mathsf{Tx}_D \in \mathcal{L}_D \iff \mathsf{Tx}_S \in \mathcal{L}_S$. If either Tx_S or Tx_D is invalid and provided to honest players, then neither Tx_S nor Tx_D is reported stable on $\mathcal{L}_S$ and $\mathcal{L}_D$, respectively.

Adversarial Model. I, R, P and V are *mutually distrustful*, with *at least one of* P *and* V *being honest*. We assume a majority γ of honest miners, where γ depends on the underlying consensus. For this model, we formally prove in the UC framework that the Glimpse protocol UC-realizes an ideal functionality $\mathcal{F}_{W-Glimpse}$, and we show that atomicity holds (Section 5). We further extend our model to incorporate rational participants and miners and we provide an economic analysis in Section 6.

3.2 Protocol Overview

In the *Setup* phase, P and V cooperate in the creation of the Glimpse contract Tx_G, which hard-codes the target $\mathcal{T}_S$ of $\mathcal{L}_S$ (*PoW consensus parameter*), as well as the following *Glimpse specifics*: the hashes of the to-be-verified transactionTx_S, the contract lifetime T, the number of confirmation blocks in the proofs, and the funds' spending conditions in the contract.

P and V prepare transactions Tx_P and Tx_V, both spending the same funds in the contract Tx_G but in different ways; these two transactions are meant to be published by P and V respectively and are commitments to how the coins are distributed in case P provides a valid proof as a witness for Tx_P, or V reacts to the lack of such proof by publishing Tx_V after T. P signs Tx_V and sends the signature to V, whereas V signs Tx_P and gives the signature to P. They also exchange all necessary signatures over Tx_G and publish Tx_G on $\mathcal{L}_D$. Finally, they pass the Tx_S to I. In the *Commit on* $\mathcal{L}_S$ phase, I publishes Tx_S on $\mathcal{L}_S$.

In the *Verify & Commit on* $\mathcal{L}_D$ phase, P queries R about the inclusion of Tx_S on $\mathcal{L}_S$ and asks for the necessary data to

Figure 1: High level Glimpse flow: (1) Upon P and V agreeing on the Glimpse specifics and PoW consensus parameters, they construct Tx_G, Tx_P, and Tx_V. P publishes Tx_G on $\mathcal{L}_D$. (2) I publishes Tx_R on $\mathcal{L}_S$. (3) P gets the information to prove Tx_R has been included on $\mathcal{L}_S$ via R. (4) P constructs $\mathcal{P}^n$ and publishes Tx_P with $\mathcal{P}^n$ as part of the witness. Else, if after T the funds in Tx_G are unspent, V can publish Tx_V.

construct a proof $\mathcal{P}^n$. This proof can convince Tx_G that Tx_S appeared on $\mathcal{L}_S$ and we detail how it looks like in Section 3.3. Having constructed the proof, P publishes Tx_P on $\mathcal{L}_D$ with $\mathcal{P}^n$, V's signature, and his own signature as witnesses. After time T, if the funds are still unspent, V publishes Tx_V on $\mathcal{L}_D$ with P's signature as well as his own as witnesses. The Glimpse instance is now closed and the funds are distributed as agreed in the *Setup* phase.

We will see that this approach can be generalized so that Tx_S represents a set of transactions, logically expressed as a disjunctive normal form over transactions, that are considered valid by the Glimpse contract. Figure 1 depicts the Glimpse protocol flow. In the next section, we describe how the proof $\mathcal{P}^n$ is constructed and verified.

3.3 Designing the Proof

We show how our proof is constructed by starting with stateless SPV [12] as a straw man proposal. We outline its vulnerabilities to *upfront mining attacks* and iteratively improve the construction until we achieve our desired security properties and enhance expressiveness and compatibility.

Stateless SPVs. In stateless SPVs, as outlined in [11, 12], users convince a quasi Turing-complete smart contract hosted on a destination chain $\mathcal{L}_D$ that a transaction Tx has appeared on a source chain $\mathcal{L}_S$, i.e., Bitcoin. The inclusion proof $\mathcal{P}^n$ consists of n subsequent confirmation block headers and a Merkle proof for the transaction Tx that shows Tx is included in one of the n blocks. The smart contract verifies (a) the Merkle proof for Tx by checking it with respect to the Merkle root of the corresponding block header, (b) that each one of the $n+1$ headers is a valid child of its parent (they are chained together), and (c) that all headers hold enough PoW, i.e., their hash is small enough according to a pre-defined target.

Various proposals exist for stateless SPVs, each with its own requirements: some demand to check some fields of the to-be-verified transaction within the contract so that the transaction intended behavior is guaranteed; others require a timelock T to limit the time window for the submission of the proof. We show that both requirements are needed: the first to avoid verification of a random fake transaction, the second to prevent late submission of the proof and funds being locked indefinitely. For security reasons, we also show that the number of confirmation blocks in the proof must increase proportionally with the timelock, leading to short lived contracts for practicality.

Upfront Mining Attack. The current stateless SPV design imposes no restriction on Tx. In particular, a prover P wishing to convince the smart contract that Tx has been included on $\mathcal{L}_S$ can start mining well in advance without any time restrictions to find n block headers that hash to a low enough target. To illustrate this problem, we look at Bitcoin, whose blockchain is extended by approximately 53k blocks yearly. An attacker with 0.05% mining power is expected to find around 6 blocks in less than three months. This means that such an attacker can forge such a 6-block proof before setting up the stateless SPV. The attacker foregoes any potential reward from using the mining power to create blocks in the main chain but can claim all the money the contract holds with certainty, regardless of her mining power and without needing to bribe or convince any miners. To make matters worse, the attacker could set up multiple stateless SPV contracts with different users based on the same transaction Tx (e.g., a betting application) and simply replay the same, upfront-mined fake proof to all contracts. Thereby, the attacker cheats multiple users out of their funds at once.

Checkpoints. A suggested countermeasure to upfront mining is using checkpoints, i.e., the parties occasionally update their contract with blocks agreed to belong to the longest valid chain of $\mathcal{L}_S$. Valid proofs of inclusion can only start from the latest agreed checkpoint.

To determine the next checkpoint in [12], the parties update the contract with the last block (e.g., 6th block) of the previous valid inclusion proof. Upfront mining, however, is still possible but within time limitations imposed by the frequency of inclusion proofs. To limit the possibility of upfront mining even further, periodical checkpoints may be used. However, the verifier should either run a light client on $\mathcal{L}_S$ or trust a third party, e.g., `blockchain.com/explorer`; both of which defeat the point of a stateless SPV construction altogether.

Our goal is to design a solution without any trust assumptions where the secure verification of a transaction inclusion is possible on-demand with constant-size data (w.r.t. $\mathcal{L}_S$'s length), meaning that both security and the amount of data should be independent of the usage of the contract.

Randomized Tx. We prevent P from launching an upfront mining attack by asking V to *randomize the transac-*

tion Tx. Specifically, V samples a uniformly random string $r \leftarrow \{0,1\}^\lambda$ (λ is the security parameter) and plugs it into the body of Tx, producing $\mathsf{Tx_R}$. This can be done by adding an output of value 0 with spending condition `OP_RETURN`[2] followed by the random value r. The Glimpse transaction $\mathsf{Tx_G}$ must hardcode the hash of the randomized transaction $\mathcal{H}([\mathsf{Tx_R}])$. Since P cannot anticipate r, he cannot start forging $\mathcal{P}^n$ upfront, so his computational efforts are restricted to the timeout T. Additionally, the random value serves as *unique identifier*, preventing proof replay attacks.

We highlight that randomizing the transaction does not impose trust assumptions and verification remains constant-sized, therefore achieving our design goals stated above. An interesting observation is that security against upfront mining seems to restrict verification to transactions that take place in the future. Transactions in the past cannot be randomized anymore, and are thus vulnerable to upfront mining. This is a fundamental difference between stateless SPV solutions and light clients, which can instead verify past transactions.

Improving compatibility. Besides eliminating upfront mining attacks, our construction also forgoes the need of stateful smart contracts, opposed to [11, 12] which requires quasi Turing-complete contracts. Indeed, due to its simplicity, Glimpse can also be deployed on *Bitcoin-based chains* by hardcoding the necessary transaction fields as well as the verification logic in a transaction locking script, which is spendable with a witness as proof $\mathcal{P}^n$. Necessary scripting capabilities and compatible chains are discussed in Section 3.5.

3.4 Enhancing Expressiveness

Glimpse cannot yet encode sophisticated applications due to two shortcomings in terms of expressiveness: First, inputs and outputs of $\mathsf{Tx_R}$ must be entirely known a priori, which hinders applications like cross-chain lending. Second, only single transaction verification is supported, which prohibits applications like multiple oracle attestations. To cater to such use cases, we augment Glimpse (Figure 2) to verify transactions which are *not fully known* during the initial Setup phase and to capture *arbitrary combinations of transactions* in $\mathcal{L}_S$ expressed as *Disjunctive Normal Forms*. The core idea is that we encode in the Setup phase the abstract expression of a transaction's spending condition (e.g., a signature) and not the exact parameters (e.g., "whose" signature).

Recall our definition transaction bodies $[\mathsf{Tx}] := (\mathsf{cntr}_{in}, \mathsf{inputs}, \mathsf{cntr}_{out}, \mathsf{outputs})$, where inputs and outputs are tuples $(\mathsf{txid}, \mathsf{outid})$ and (coins, ϕ), respectively, from Section 2. To verify transactions that are not fully known in the Setup phase, we introduce the *Glimpse description* Desc of a transaction (see Figure 2). In such a description, we allow parameterized inputs and outputs. More concretely, $\mathsf{txid}, \mathsf{outid}$, and coins can either be static data or *variables* x_i.

[2] `OP_RETURN` is a Bitcoin script opcode that marks a transaction output as invalid and can be used to embed up to 80 bytes in a transaction.

Similarly, to avoid fixing a priori a specific script, we say that ϕ in a parameterized input or output can be a function f which encodes a family of scripts. f takes a fixed number of arguments for a well-defined spending condition and returns the desired locking script for the to-be-verified transaction. In other words, this is a parameterized locking script that can be filled with concrete values, e.g., public key, script hash, etc. The script which f encodes must be defined in the *Setup* phase: For instance, if the parties agree on $f(z)$ encoding any P2PKH (Pay-To-Public-Key-Hash), then it would accept any public key hash as parameter z and return the script.

Following Figure 2, inputs and $\mathsf{outputs}$ of descriptions are lists of such parameterized inputs and outputs, and cntr_{in} and cntr_{out} are the number of overall inputs and outputs, respectively. The latter must be known from the beginning to avoid miners interpreting transactions in an unintended way (see Appendix E). In the *Setup* phase, the parties agree on and commit to a description Desc, a target $\mathcal{T}_S$, a time T, and a proof size n. By replacing $\mathcal{H}(\mathsf{Tx_R})$ with Desc, we can now verify any $\mathsf{Tx_R}$ in the set of transactions that share the same description, i.e., any $\mathsf{Tx_R}$ whose body has the same static data in Desc and an arbitrary realization for the variable ones. For example, any value can be in the place of x_i and any parameter can be given to f, e.g., any public key hash in the previous example. We denote this as $[\mathsf{Tx_R}] \hookleftarrow \mathsf{Desc}$ or a concrete transaction $[\mathsf{Tx_R}]$ *fulfilling a description* Desc.

We note that the random string sampled by V must always be included in Desc. The variable realizations are included in the proof $\mathcal{P}^n$. Given $\mathcal{P}^n$ and Desc, the full transaction body can be reconstructed and hashed in the logic of $\mathsf{Tx_G}$.

We can efficiently verify any DNF formula $\mathcal{F}_S$ over k transactions or descriptions (see Figure 2), also known as literals L_i, as follows: the P and V replace $\mathsf{Tx_P}$ with as many sets of transactions ($\mathsf{Tx_T}$, $\mathsf{Tx_F}$, $\mathsf{Tx_P}$) as the number of (conjunctive) terms in the formula (see Appendix D). When I publishes on $\mathcal{L}_S$ a valid combination of transactions for the DNF formula, P queries R, constructs the set of proofs for the transactions published, and posts on $\mathcal{L}_D$ the corresponding set $\mathsf{Tx}_D := (\mathsf{Tx_T}, \mathsf{Tx_F}, \mathsf{Tx_P})$ of transactions. If P cheats by publishing an invalid set, i.e., P falsely claims a transaction was not published on $\mathcal{L}_S$, V can query R, disprove P, and publish $\mathsf{Tx_V}$.

3.5 Compatibility

In Table 1, we provide a non-exhaustive list of popular Bitcoin-based chains that can be used as $\mathcal{L}_S$ or $\mathcal{L}_D$ for Glimpse. Most PoW chains can be used as the source chain $\mathcal{L}_S$ for Glimpse, e.g., Bitcoin, Litecoin, Bitcoin Cash, Bitcoin SV, Rootstock, and Ethereum PoW. However, we need to make some distinctions for which chains can be supported by Glimpse as destination chains.

In particular, Glimpse requires the destination chain to support the hash function used for PoW in the source chain. Thus, the more restrictive the destination chain, the fewer source chains may be compatible. This strict requirement al-

$$
\begin{array}{rcl}
\mathsf{txid} & := & \{0,1\}^{256} \mid x_1, \\
\mathsf{outid} & := & \{0,1\}^{32} \mid x_2 \\
\mathsf{coins} & := & \{0,1\}^{64} \mid x_3 \\
\phi & := & f(z_1,\dots,z_n) \\
\mathsf{inputs} & := & [(\mathsf{txid},\mathsf{outid})] \mid \mathsf{inputs} \cup [(\mathsf{txid},\mathsf{outid})] \\
\mathsf{outputs} & := & [(\mathsf{coins},\phi)] \mid \mathsf{outputs} \cup [(\mathsf{coins},\phi)] \\
\mathsf{cntr}_{in}, \mathsf{cntr}_{out} & := & \{0,1\}^m \\
\mathsf{Desc} & := & (\mathsf{cntr}_{in}, \mathsf{inputs}, \mathsf{cntr}_{out}, \mathsf{outputs}) \\
L_i & := & \mathsf{Desc}_i \mid \neg\mathsf{Desc}_i \\
\mathcal{F}_S & := & (L_1 \wedge \dots \wedge L_k) \vee \dots \vee (L_1 \wedge \dots \wedge L_k)
\end{array}
$$

$$\forall(x_1,\dots,x_3,z_1\dots,z_n).(\mathcal{F}_S \iff \mathsf{Tx}_D)$$

Figure 2: Synchronization Patterns expressed in DNFs.

Source chain $\mathcal{L}_S$	Destination chain $\mathcal{L}_D$
Bitcoin	Bitcoin†
Bitcoin Cash	Liquid
Bitcoin Satoshi Vision	Bitcoin Cash*
Bitcoin Rootstock	Bitcoin SV*
Litecoin††	Litecoin†
Ethereum PoW/Classic††	Any quasi-Turing complete chain

Table 1: A non-exhaustive list of the most popular Bitcoin-based source ($\mathcal{L}_S$) and destination ($\mathcal{L}_D$) chains compatible with Glimpse. †: lack of string opcodes. ††: currently not supported by Bitcoin-based destination chains. *: lack of Taproot.

ready rules out some combinations in Table 1. For instance, Bitcoin-based chains do not have an opcode for computing Keccack or Scrypt hashes, used in Ethereum PoW and Litecoin, respectively. As a result, Ethereum PoW and Litecoin cannot be the source chains on Glimpse contracts deployed on Bitcoin-based chains.

Next, we discuss how the particular design of Glimpse makes it compatible with prominent Bitcoin-based blockchains, such as Liquid, and how it could similarly be supported by Bitcoin, Bitcoin Cash, or Bitcoin Satoshi Vision (SV), e.g., by enabling string opcodes or Taproot. For detailed discussion and examples, we refer to Appendix E.

Liquid (Fully Compatible). Liquid is a Bitcoin sidechain that has been operating since 2018. It inherits its design from Bitcoin but provides more expressiveness in its scripting language. In particular, the following opcodes crucial to Glimpse are disabled in Bitcoin but enabled on Liquid: (i) string concatenation (`OP_CAT`) for the Merkle Proof verification and block header/transaction reconstruction, and (ii) `OP_SUBSTR` for splitting numbers larger than 4 bytes for comparison (which only allows comparison of 4-byte numbers), i.e., the block header hash and PoW target. Moreover, Liquid adopts Taproot [14], which can significantly reduce the Glimpse script size and complexity with the use of the Merkelized Abstract Syntax Tree (MAST), as discussed in Section 7 and exemplified in Appendix E.

Bitcoin Cash (Missing Taproot). Bitcoin Cash is a blockchain resulting from a Bitcoin hard fork that took place in 2017 after Bitcoin moved to SegWit. It presents a larger block size and a wider variety of supported opcodes. Similarly to Liquid, Bitcoin Cash has `OP_CAT` and `OP_SPLIT` (the same as `OP_SUBSTR`). However, Taproot is not enabled; thus, we cannot encode the Glimpse script in a single output. While it is possible to unroll the script leaves of the MAST, the script is too large (around 300kB, see Section 7), even though it would be within the transaction size limits. In Bitcoin SV we have the same limitations as in Bitcoin Cash. It would need either Taproot or larger script sizes to support Bitcoin Cash or Bitcoin SV as a destination chain.

Bitcoin and Litecoin (Missing String Opcodes). Bitcoin and Litecoin adopted Taproot, but disabled the opcodes mentioned above in 2010. With Taproot at their disposal, they could efficiently support Glimpse if they had opcodes for Merkle root reconstruction and hash comparison available, e.g., if `OP_CAT` and `OP_SUBSTR` were available or if `OP_LESSTHAN` could compare 32-byte values. Interestingly, the Bitcoin community has been recently discussing the inclusion of the string concatenation opcode and is considering enabling it back with Speedy Covenants [18]. We hope this work provides additional motivation for such opcodes to be enabled in the future.

4 Glimpse for Lending and Cross-Chain DeFi

DeFi applications, such as lending, thrive on blockchains supporting quasi Turing-complete smart contracts, but do not exist on Bitcoin-based blockchains. To fill this gap, we now show how to use Glimpse for designing a lending protocol for Bitcoin-based chains.

Intuition. We consider a borrower P and a lender V. P has α coins on $\mathcal{L}_D$ and wants to take a loan of α' coins on $\mathcal{L}_S$. We assume the loan is over-collateralized to compensate for price drops of asset α. Having a surplus of α' coins in $\mathcal{L}_S$, V lends the α' coins to P. Intuitively, P locks α coins in a Glimpse transaction Tx_G, and V sets up a transaction Tx_{Loan} giving a loan of α' coins to P. Via an atomic swap conditioned to secret s, P publishes Tx_{Loan} on $\mathcal{L}_S$ revealing s to V, and V publishes Tx_G on $\mathcal{L}_D$ using the same secret s. Tx_G guarantees that if P gives back to V the α' coins on $\mathcal{L}_S$ within time T, P gets back his α coins on $\mathcal{L}_D$. Otherwise, V retains the α coin collateral after time T. Figure 3 depicts such a lending construction, which we detail below and later extended to support a liquidation mechanism.

Setup. P sends $\mathsf{Desc} := (1,[(x_1)],1,[(\alpha,\mathsf{OneSig}(\mathsf{pk}_V))])$ to V. V samples $r \leftarrow \{0,1\}^\lambda$ uniformly at random and generates $\mathsf{Desc} := (1,[(x_1)],2,[(\alpha,\mathsf{OneSig}(\mathsf{pk}_V)),(0,\texttt{OP_RETURN}(r))])$. V sends Desc to P.

We let θ_P be an unspent output of P holding α coins, and ζ_P be an input pointing to θ_P. Then, P constructs $[\mathsf{Tx}_G] := (1,[(\zeta_P)],1,[(\alpha,\mathsf{scriptG}(\mathsf{Desc},T,\mathcal{T}_S,n,(P,V)))])$. The locking script generated by scriptG can be spent as follows: (i) P can get back the α coins by submitting a valid $\mathcal{P}^n$ (witness); else, V can get the α coins after time T. T is strictly

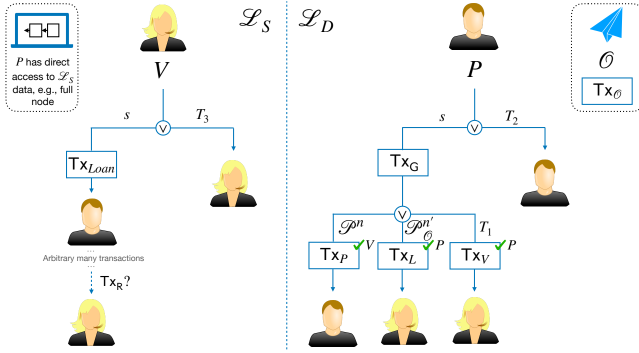

Figure 3: Illustration of Glimpse-based lending for Bitcoin-based blockchains. We have $T_1 < T_2 < T_3$.

bigger than one block time of $\mathcal{L}_S$ and $\mathcal{L}_D$. (ii) $\mathcal{P}^n$ proves inclusion for a transaction Tx compliant with Desc, i.e., whose $[\mathsf{Tx}] \hookleftarrow \mathsf{Desc}$, and (iii) $\mathcal{P}^n$ is verified against the PoW consensus parameter $\mathcal{T}_S$. Figure 4 shows the pseudocode for $\mathsf{scriptG}$ (sG), and Appendix E describes a concrete example.

After setting up $[\mathsf{Tx}_G]$, P constructs transaction $[\mathsf{Tx}_\mathsf{P}] = (1, [\zeta_G], 1, [(\alpha, \mathsf{OneSig}(\mathsf{pk}_P))])$ and $[\mathsf{Tx}_\mathsf{V}] := (1, [\zeta_G], 1, [(\alpha, \mathsf{OneSig}(\mathsf{pk}_V))])$. In other words, Tx_P (Tx_V) spends the output of Tx_G and creates a new unspent output that only P (V) can spend. Then, P signs $[\mathsf{Tx}_\mathsf{V}]$ producing $\sigma_P([\mathsf{Tx}_\mathsf{V}])$ and sends to V the following message with the Glimpse specifics: $(\zeta_P, \mathsf{Desc}, T, \mathcal{T}_S, n, \alpha, \mathsf{sG}, [\mathsf{Tx}_G], [\mathsf{Tx}_\mathsf{P}], [\mathsf{Tx}_\mathsf{V}], \sigma_P([\mathsf{Tx}_\mathsf{V}]))$.

Upon receiving the message from P, V verifies whether $\mathsf{scriptG}$ returns the intended locking script for Tx_G and $\sigma_P([\mathsf{Tx}_\mathsf{V}])$ is a valid signature of P over $[\mathsf{Tx}_\mathsf{V}]$. Upon successful verification, V computes the signatures $\sigma_V([\mathsf{Tx}_\mathsf{P}])$ and $\sigma_V([\mathsf{Tx}_\mathsf{G}])$, and sends them to P.

Upon receiving $(\sigma_V([\mathsf{Tx}_\mathsf{P}]), \sigma_V([\mathsf{Tx}_\mathsf{G}]))$ from V, P checks whether V's signatures are valid signatures, and if this is the case, P signs $[\mathsf{Tx}_\mathsf{G}]$ and publishes Tx_G on $\mathcal{L}_D$ with witness $\sigma_P([\mathsf{Tx}_\mathsf{G}])$.

Commit on $\mathcal{L}_S$. At this point the lending is set up and P can use the loan in any way he wants. Once P is done and wants to pay it back, P posts Tx on $\mathcal{L}_S$ such that $[\mathsf{Tx}] \hookleftarrow \mathsf{Desc}$: concretely, $[\mathsf{Tx}]$ is equal to Desc apart form x_1 being replaced by an arbitrary input of P.

Verify & Commit on $\mathcal{L}_D$. P monitors $\mathcal{L}_S$ checking for Tx being included. If Tx is included within T and has n confirmations, P constructs $\mathcal{P}^n$ by taking the concrete value of x_1 within Tx ($[\mathsf{Tx}]$), computing the Merkle proof (MP) for Tx, retrieving from $\mathcal{L}_S$ the block header without Merkle root for the block B including Tx, and fetching n confirmation block headers (without $\mathsf{ParentHash}$) after B. Concretely, P constructs $\mathcal{P}^n := ([\mathsf{Tx}] \setminus \mathsf{Desc}, \mathsf{MP}, \mathsf{headerWOMR}(\mathsf{B}), \mathsf{confHeaders}_\mathsf{n})$. We provide pseudocode for constructing $\mathcal{P}^n$ in Figure 4. Upon computing $\mathcal{P}^n$, P signs $[\mathsf{Tx}_\mathsf{P}]$ and publishes Tx_P on $\mathcal{L}_D$ with witness $\omega := (\mathcal{P}^n, \sigma_P([\mathsf{Tx}_\mathsf{P}]), \sigma_V([\mathsf{Tx}_\mathsf{P}]))$, thus having back his α coins staked in Glimpse.

After T, if the output of Tx_G is still unspent, V signs $[\mathsf{Tx}_\mathsf{V}]$ and publishes Tx_V with witness $\omega := (\sigma_P([\mathsf{Tx}_\mathsf{V}]), \sigma_V([\mathsf{Tx}_\mathsf{V}]))$, redeeming the α coins in Glimpse. We note that if V does not publish Tx_V right after time T, P can maliciously claim the funds by publishing Tx_P with a belated proof: this case is, however, ruled out by assuming rational parties. We also note that the time lock T prevents Tx_V from being valid before T.

The *Setup*, *Commit on $\mathcal{L}_S$*, and *Verify & Commit on $\mathcal{L}_D$* phases are the core of Glimpse construction, recurring (with minor application-specific variations) regardless of the specific use case. We now discuss the liquidation mechanism exclusively for lending.

Liquidation. Should the asset price on $\mathcal{L}_S$ shrink below a predefined liquidity threshold, V must be able to redeem the collateral before T. For this, we assume there exists a trusted oracle O on $\mathcal{L}_D$ that regularly publishes a transaction Tx_O with the current price of the asset on $\mathcal{L}_S$; for instance, O can be a Discreet Log Contract-based [20] or a voting-based [21] oracle. If O is not trusted, we can leverage a set of N different independent oracles, with the promise that if a large enough number of oracles agree on the same price, the liquidation is granted using Glimpse ability to verify DNFs over descriptions. The oracles do not need to coordinate with each other, nor have a common transaction structure. For simplicity, we discuss the case of a single trusted O.

When setting up Glimpse, we assume Tx_O is described by, e.g., $\mathsf{Desc}_O := (1, [\zeta_i], 2, [\theta_r, (0, \mathtt{OP_RETURN}(\mathsf{price}))])$, where $\theta_r := (0, \mathtt{OP_RETURN}(r))$ includes the Glimpse randomness, and the other output reports the price update. In this case, O must take the randomness from $\mathcal{L}_D$ itself so that Glimpse participants can, to some extent, anticipate it and be able to include it in θ_r: for example, r can be the hash of the transaction/block of the last price update published by O. It is the verifier's responsibility to ensure Glimpse embeds the most recent random string.

To include liquidation, $\mathsf{scriptG}$ has to be tweaked to encode the following: (i) if P repays his debt publishing Tx, s.t. $[\mathsf{Tx}] \hookleftarrow \mathsf{Desc}$ on $\mathcal{L}_S$ within time T, he can publish Tx_P with witness $\mathcal{P}^n$ and have his collateral back on $\mathcal{L}_D$, (ii) if P defaults the loan, V can publish Tx_V redeeming the collateral after T, and (iii) if before T O attests the collateral price on $\mathcal{L}_S$ below the predefined liquidity threshold, V can redeem the collateral publishing the liquidation transaction $\mathsf{Tx}_\mathsf{L} := (1, [\zeta_G], 1, [(\alpha, \mathsf{OneSig}(\mathsf{pk}_V))])$ with witness $\mathcal{P}^{n'}_O$ and sell the funds at a discount. The liquidation transaction has to be constructed and signed by P in the *Setup* phase.

Glimpse-based lending enables the first form of trustless peer-to-peer lending on Bitcoin-based chains. Moreover, publishing Tx_G and Tx_{Loan} via atomic swap only requires a single Glimpse instance on $\mathcal{L}_D$. On the other hand, without extensive programmability, funds cannot be pooled, leading to peer-to-peer lending where potential borrowers have to find would-be lenders and agree on the loan's amount and interest rate –

Setup$(\mathsf{Desc}, T, \mathcal{T}_S, n)$:

1. P sends a transaction description $\mathsf{Desc}' = (\mathsf{cntr}_{in}, \mathsf{inputs}, \mathsf{cntr}_{out}, \mathsf{outputs})$ to V.
2. Upon receving Desc', V samples a random string $r \xleftarrow{\$} \{0,1\}^\lambda$, creates $\mathsf{Desc} = (\mathsf{cntr}_{in}, \mathsf{inputs}, \mathsf{cntr}_{out}+1, \mathsf{outputs} \cup (0, \texttt{OP_RETURN}(r)))$, and sends Desc to P.
3. Let $\alpha := \theta_P.\mathsf{coins}$; let ζ_P point to an unspent output of P.
4. Let $[\mathsf{Tx}_G] := (1, [\zeta_P], 1, [\theta_G := (\alpha, \mathsf{scriptG}(\mathsf{Desc}, T, \mathcal{T}_S, n, (P,V)))])$. Let ζ_G point to θ_G.
5. Let $[\mathsf{Tx_P}] := (1, [\zeta_G], 2, [(\alpha \cdot \mathsf{outcomeP}, \mathsf{OneSig}(\mathsf{pk}_P)), (\alpha \cdot (1-\mathsf{outcomeP}), \mathsf{OneSig}(\mathsf{pk}_V))])$.
6. Let $[\mathsf{Tx_V}] := (1, [\zeta_G], 2, [(\alpha \cdot \mathsf{outcomeV}, \mathsf{OneSig}(\mathsf{pk}_V)), (\alpha \cdot (1-\mathsf{outcomeV}), \mathsf{OneSig}(\mathsf{pk}_P))])$.
7. P computes $\sigma_P([\mathsf{Tx_V}])$ and sends $(\zeta_P, \mathsf{Desc}, T, \mathcal{T}_S, n, \alpha, \mathsf{scriptG}, [\mathsf{Tx}_G], [\mathsf{Tx_P}], [\mathsf{Tx_V}], \sigma_P([\mathsf{Tx_V}]))$ to V.
8. Upon receiving $(\zeta_P, \mathsf{Desc}, T, \mathcal{T}_S, n, \alpha, \mathsf{scriptG}, [\mathsf{Tx}_G], [\mathsf{Tx_P}], [\mathsf{Tx_V}], \sigma_P([\mathsf{Tx_V}]))$, if V is interested in opening a Glimpse instance with P at the given specifics, V signs $\mathsf{Tx_P}$ and sends $(\sigma_V([\mathsf{Tx_P}]))$ and sends it to P.
9. Upon receiving $(\sigma_V([\mathsf{Tx_P}]))$, P verifies if $\sigma_V([\mathsf{Tx_P}])$ is a valid signatures and, upon successful verification, P signs $[\mathsf{Tx_G}]$ and posts $\mathsf{Tx_G}$ with witness $\omega := \sigma_P([\mathsf{Tx_G}])$.

Commit on $\mathcal{L}_S$ (Desc): P posts Tx such that $[\mathsf{Tx}] \hookleftarrow \mathsf{Desc}$

Verify & Commit on $\mathcal{L}_D$ (Desc, T, n):

1. Upon Tx s.t. $[\mathsf{Tx}] \hookleftarrow \mathsf{Desc}$ being included on $\mathcal{L}_S$ with n confirmations before T, P constructs $\mathcal{P}^n$ as in Figure 4 and signs $\mathsf{Tx_P}$. P posts $\mathsf{Tx_P}$ with witness $\omega := (\mathcal{P}^n, \sigma_P([\mathsf{Tx_P}]), \sigma_V([\mathsf{Tx_P}]))$.
2. If time T has elapsed and θ_G is still unspent, V signs $\mathsf{Tx_V}$ and posts $\mathsf{Tx_V}$ with witness $\omega := (\sigma_P([\mathsf{Tx_V}]), \sigma_V([\mathsf{Tx_V}]))$.

$\mathsf{scriptG}(\mathsf{Desc}, T, cp, n, (P,V))$, gen. the following locking script:

- Upon receiving $\omega = (\mathcal{P}^n, \sigma_P([\mathsf{Tx_P}]), \sigma_V([\mathsf{Tx_P}]))$, where $\mathcal{P}^n = ([\mathsf{Tx}] \setminus \mathsf{Desc}, \mathsf{MP}, \mathsf{headerWOMR}(\mathsf{B}), \mathsf{confHeaders}_n)$:
  1. If $[\mathsf{Tx}] \setminus \mathsf{Desc}$ matches the expected number of strings and their expected length, reconstruct $[\mathsf{Tx}]$. Else, return $\perp$.
  2. Compute $\mathcal{H}([\mathsf{Tx}]) := \mathsf{txid}$.
  3. Given txid and MP, compute the Merkle root MR for B. Given MR and $\mathsf{headerWOMR}(\mathsf{B})$, reconstruct header of B.
  4. Compute the hash of B's header and check if it is smaller than $cp := \mathcal{T}_S$. Else, return $\perp$.
  5. If $n > 0$, for each of the n confirmation blocks, reconstruct the header using $\mathsf{headerWOPH(i)}$ and $\mathcal{H}(\mathsf{B}_{i-1})$. Hash the header and check if it is smaller than $\mathcal{T}_S$. Else, $\perp$.
  6. If $(\sigma_P([\mathsf{Tx_P}]), \sigma_V([\mathsf{Tx_P}]))$ are valid signatures of P and V, unlock the coins. Else, return $\perp$.
- If $t > T$, upon receiving $\omega = (\sigma_P([\mathsf{Tx_V}]), \sigma_V([\mathsf{Tx_V}]))$, unlock the coins. Else, return $\perp$.

Construct $\mathcal{P}^n(\mathsf{Tx}, \mathsf{Desc}, n)$ for Bitcoin-like $\mathcal{L}_S$:

1. Upon $\mathsf{Tx} \in \mathsf{B} \in \mathcal{L}_S$ s.t. $[\mathsf{Tx}] \hookleftarrow \mathsf{Desc}$, fetch $[\mathsf{Tx}]$.
2. Compute the transaction Merkle proof MP for Tx.
3. $\mathsf{headerWOMR}(\mathsf{B}) := \mathsf{Header}(\mathsf{B}) \setminus \mathsf{MR}$.
4. Let $\mathsf{headerWOPH}(\mathsf{B}) := \mathsf{Header}(\mathsf{B}) \setminus \mathsf{ParentHash}$. If $n > 0$, retrieve $\mathsf{confHeaders}_n := \{\mathsf{headerWOPH}(\mathsf{B}+1), \ldots, \mathsf{headerWOPH}(\mathsf{B}+n)\}$.
5. Return $\mathcal{P}^n := ([\mathsf{Tx}] \setminus \mathsf{Desc}, \mathsf{MP}, \mathsf{headerWOMR}(\mathsf{B}), \mathsf{confHeaders}_n)$.

Figure 4: Pseudocode for the Glimpse-based lending.

reflected in the fund distribution in $\mathsf{Tx_P}$. To facilitate matching the demand and supply, we suggest setting up dedicated communication channels or platforms.

4.1 Other Applications

In this section, we give an intuition about how Glimpse can be leveraged in different use cases.

Backed Assets. We refer to *backed assets* as assets issued on a ledger $\mathcal{L}_D$ that are backed by a cryptocurrency or an asset on a ledger $\mathcal{L}_S$. In this category, we have, for instance, assets issued on sidechains and backed on parent chains, and native tokens on ledger $\mathcal{L}_S$ backing wrapped tokens on $\mathcal{L}_D$.

Sidechains are blockchains tightly bound to a pre-existing parent blockchain with the purpose of enabling or extending some features. Users can easily move funds from the parent chain to the sidechain and vice versa through verifiable two-way pegs: assets are locked in an address of the parent chain (sidechain) and are then released on the sidechain (parent chain), ready to be used. Starting from the lending protocol on Section 4, we show how to set up pegs with Glimpse. Let us remove the liquidation mechanism from the lending protocol and assume V *can create assets on* $\mathcal{L}_D$: with these two caveats, the same Glimpse-based construction can be used to encode trustless sidechain pegs, where V issues new assets on the sidechain (rather than giving a loan), and P is able to get back the funds on the parent chain by proving he returned the coins to an a priori well-defined peg address on the sidechain.

Similar to pegs, Glimpse can be used to wrap and unwrap tokens. Wrapped tokens allow representing assets native to chain $\mathcal{L}_S$, on another chain $\mathcal{L}_D$. They can be issued on $\mathcal{L}_D$ when a corresponding amount of native tokens are locked on $\mathcal{L}_S$, and they can then be released (unwrapped) on the native chain when the user locks up the wrapped ones on $\mathcal{L}_D$.

Proofs-of-Burn. Proof-of-Burn is a bootstrapping mechanism allows users to prove on a destination ledger $\mathcal{L}_D$ they burnt some coins on the source ledger $\mathcal{L}_S$, meaning they sent coins to a verifiable unspendable address. By verifying such proof, the user can obtain the correspondent amount of native assets on $\mathcal{L}_D$. The construction for a Proof-of-Burn is very similar to the one for backed assets, with the only difference being that funds are moved unidirectionally.

Proofs of Oracle Attestations. A noteworthy application enabled by Glimpse for Bitcoin-based chains is the following: let us assume on $\mathcal{L}_S$ there exist k oracles posting information about real-world events, such as real-time prices for currencies, and on $\mathcal{L}_D$ one wants to efficiently verify k oracle attestations for a specific event. In case of two oracles, $O1$ and $O2$, $\mathsf{Tx_G}$ verifies $\mathcal{F}_S = (\mathsf{Desc}_1 \wedge \mathsf{Desc}_2)$, with Desc_1 being the description for $O1$ and Desc_2 the one for $O2$. We note that $O1$ and $O2$ do not have to cooperate nor operate on the same chain; their chains need to run PoW consensus.

Off-chain Glimpse Applications. State channels [22–24] and generalized channels [13] enable Payment Channel Net-

works (PCNs) that offer off-chain the same functionality as the underlying chain offers. We can thus host Glimpse on PCNs, thereby improving its scalability and enabling a new range of applications on layer-2. Instead of posting a Glimpse contract on $\mathcal{L}_D$, it can be kept off-chain as an update of a channel and only posted on-chain to resolve disputes.

For instance, payment channel hubs may employ off-chain Glimpse contracts to set up a betting hub, where users connected to the hub net bet on a transaction(s) Tx_S published on a PoW source chain within an absolute time T'. If the bet is won, the correct outcome has to be reflected in the channel update by time $T < T'$. If the hub misbehaves, users' can post the Glimpse contract on-chain (thereby closing the channel) and enforce the correct outcome by submitting a valid proof within the absolute time T.

Using Glimpse in such an off-chain fashion would also provide an out-of-the-box solution for enabling off-chain constructions synchronized by an on-chain event in a cross-chain setting. Examples include cross-chain payments based on [22, 25] or cross-chain virtual channels based on [26].

5 Security in the UC Framework

We model Glimpse in the synchronous Global Universal Composability (GUC) framework [27], closely following prior work [13, 23, 24]. First, we state that according to basic assumptions, Glimpse achieves weak atomicity. Next, we show that by assuming the liveness of parties and access to $\mathcal{L}_S$, Glimpse achieves strong atomicity.

We use a global clock $\mathcal{G}_{Clock}$ [27] and authenticated channels with guaranteed delivery $\mathcal{G}_{GDC}$ [24] to model time and communication. We assume static corruption. We use the functionality $\mathcal{G}_{Ledger}$ defined in [28] to model a ledger $\mathcal{L}$. We use a concrete instantiation as specified in [28], where the parameters are chosen such that the ledger achieves both *liveness* and *consistency* as defined in [19]. We define two very similar ideal functionalities $\mathcal{F}_{W-Glimpse}$ and $\mathcal{F}_{S-Glimpse}$ (see Appendix B.1), formalizing our desired properties of *weak atomicity* and *strong atomicity* in the general case, respectively. More concretely, the ideal functionality is parameterized over two ledgers $\mathcal{L}_A$ or $\mathcal{L}_B$. After two parties have registered to the functionality, it will ensure that the respective transactions are posted on $\mathcal{L}_A$ or $\mathcal{L}_B$ such that *weak atomicity* or *strong atomicity* holds.

We then formally model our Glimpse protocol Π in the UC framework (see Appendix B.2) and prove that Π realizes $\mathcal{F}_{W-Glimpse}$ or $\mathcal{F}_{S-Glimpse}$ depending on the underlying assumptions. In a nutshell, this is done by designing an ideal world adversary (or simulator) $\mathcal{S}$ and showing that no PPT *environment* can computationally distinguish between interacting with the real world protocol Π in the presence of an adversary $\mathcal{A}$ and the ideal functionality in the presence of a simulator $\mathcal{S}$. In other words, $\mathcal{S}$ translates any attack on the protocol into an attack on the ideal functionality, which intuitively means that Π is "as secure", i.e., has the same properties as $\mathcal{F}_{W-Glimpse}$ or $\mathcal{F}_{S-Glimpse}$. This is formalized in Appendix B. In Appendix C we formally prove Theorems 1 and 2, which make use of Definitions 4 to 8. The definitions underlined in the theorems can be found in Appendix B.2.

Theorem 1. *Given functionalities $\mathcal{G}_{Clock}$, $\mathcal{G}_{GDC}$, the protocol Π is instantiated with two ledger instantiations $\mathcal{L}_A$ and $\mathcal{L}_B$ of $\mathcal{G}_{Ledger}$, a delay $\Delta_B \in \mathbb{N}$ and a proof generation function $gen\mathcal{P}$, where $gen\mathcal{P}$ is T-sound, and where Π has strictly randomized input, then the protocol Π UC-realizes the ideal functionality $\mathcal{F}_{W-Glimpse}$.*

Theorem 2. *Given functionalities $\mathcal{G}_{Clock}$, $\mathcal{G}_{GDC}$, the protocol Π is instantiated with two ledger instantiations $\mathcal{L}_A$ and $\mathcal{L}_B$ of $\mathcal{G}_{Ledger}$, a delay $\Delta_B \in \mathbb{N}$ and a proof generation function $gen\mathcal{P}$, where $gen\mathcal{P}$ is complete and T-sound, and where Π has strictly randomized input,* ***and where all parties have direct access to $\mathcal{L}_A$ and $\mathcal{L}_B$, and where parties exhibit liveness,*** *then the protocol Π UC-realizes the ideal functionality $\mathcal{F}_{S-Glimpse}$.*

6 Economic Security Analysis

We now extend our analysis to incorporate rational players. Glimpse security holds, similarly to light clients, assuming the underlying chains operate under a well-designed incentive mechanism that guarantees an honest majority and, by this, consistency and liveness. When the expected profit is confined to the blockchain the nodes operate on, there is no rational reason to violate the blockchain's security properties. However, introducing any cross-chain or cross-layer application introduces external profit opportunities beyond the original incentive design. In this case, if the value locked in the application exceeds the expected gain of nodes that follow the protocol, the incentive mechanism does not deter nodes from misbehaving. This is true for *all cross-chain and cross-layer applications and bridges* [29]: atomic swaps [30], chain relays [6], payment channels [31], bridges [9,32], etc. In particular, both Glimpse and chain relays equally suffer from forgery attacks, where miners use their computational power to forge a fake (sub)chain of blocks rather than using it to mine honestly. Furthermore, we study the cost of a bribing attack that breaks the security of Glimpse by compromising the liveness of the destination blockchain. For all these attacks, we define the *secure parameter space* specific to Glimpse.

6.1 Proof Forgery Attack

We proceed with an economic analysis under which conditions our assumptions hold. In particular, we describe a *proof forgery* attack, where a malicious prover bribes the miners of the source chain to forge a fake proof. Since this fake proof is not a valid extension of the longest chain, the attacker can fool the Glimpse contract and potentially steal the funds of the verifier. However, we strongly emphasize that the proof forgery attack is not limited to Glimpse, but applies to light

clients in general. Light clients operate under the assumption that the majority of the mining power, and thereby the longest chain, is honest. If this assumption is broken, a light client can also be fooled by a fake n-length suffix. This attack, both for Glimpse and light clients, becomes profitable if the cumulative value of the applications hosted on the Glimpse or light client contract, respectively, is larger than what miners receive in mining honestly.[3]

Attack Strategy. We consider the case an adversarial P bribes the miners of $\mathcal{L}_S$ to forge a proof for his Glimpse contract and promises to reward them with the coins held by Glimpse on $\mathcal{L}_D$. We further augment the adversary's power by assuming all provers P_i having *active Glimpse instances on top of the same $\mathcal{L}_S$ across m different destination chains* maliciously cooperate to launch a joint attack. In this case, bribed miners of $\mathcal{L}_S$ can optimize the computational overhead by forging a single proof for all the contracts: upon receiving from each P_i the transactions to include in the forgery, they forge a single block B^f including them all, and then mine n confirmation blocks on top of B^f in time T, where n and T are averaged over the Glimpse instances.

Total Value Simultaneously Locked in Glimpse. To analyze when this attack constitutes a threat in practice, we need to know α_c, i.e., the *cumulative economic value simultaneously held by all the active Glimpse contracts sharing the same source chain $\mathcal{L}_S$*. Indeed, the adversary controlling P_i bribes the miners with a *bribe value up to α_c*.

To compute the cumulative value α_c, an honest verifier would have to access and monitor each destination chain $\mathcal{L}_{Di}$ where, e.g., honest parties have marked a Glimpse contract $\mathsf{Tx_G}$ as such (e.g., by adding an output $(0, \texttt{OP_RETURN}(\text{"This is Glimpse contract: } T, n, \alpha\text{"})))$, to avoid opening a new Glimpse if that value exceeds the security threshold we discuss below. As this could be unpractical, we propose possible countermeasures: Honest parties may use a public bulletin board where they announce their Glimpse contracts or use some heuristics to estimate α_c, e.g., computing the cumulative value for the most used Glimpse destination chain and multiplying it by the number of compatible chains. For now, we assume honest parties can compute α_c. We leave a more rigorous analysis of how honest verifiers can compute α_c in the face of an adversary this powerful as future work.

Model. We bound the number n of required block confirmations by two constraints: (i) n should be equal to the minimum number of blocks for which the probability of an ordinary block reorganization (temporary fork) is negligible, and (ii) the probability of n being larger than the number of honest blocks mined for $\mathcal{L}_S$ in T is negligible. Constraint (i) protects V from the proof coming from an orphaned branch of a temporary fork, whereas constraint (ii) protects P from needing to provide more blocks than the ones honestly included on $\mathcal{L}_S$ over the time window T.

To study the economic cost of a proof forgery attack, we compare the expected gain of miners mining honestly to the expected gain of miners pulling the attack: If the latter is higher than the former, rational miners will launch the attack.

Expected Gain of Honest Miners. Let R be the number of coins given as block reward on $\mathcal{L}_S$, V_S the USD value of 1 coin of $\mathcal{L}_S$, $\mathbb{E}^{\mathsf{B}}_{\mathcal{L}_S}[T]$ the number of expected blocks on $\mathcal{L}_S$ in T, and μ_r the attacker's *relative* mining power on $\mathcal{L}_S$, i.e., the attacker's probability of finding a valid block. The expected gain for honest miners is given by:

$$\mathbb{E}[\mathrm{H}] = \mathsf{R} \cdot V_S \cdot \mathbb{E}^{\mathsf{B}}_{\mathcal{L}_S}[T] \cdot \mu_r. \tag{1}$$

We note that $0 \leq \mu_r \leq 1$ and $\mu_r \leq 1 - \gamma$, being $\gamma \in [0,1]$ the majority of honest miners.

Expected Gain of Miners for the Forgery Attack. The miners' expected gain when executing the attack depends on the number of confirmation blocks they need to forge for the proof, the mining power they hold, and the fluctuation (price drop) of the bribe value during the attack. Being μ the attacker's hashing power (hashes per second), the number of hashes computed in T is $\mathsf{N} := \mu \cdot T$. Considering N repeated, independent, and equally distributed hashes, and being $P_{v,\mathcal{T}}$ the probability to find a valid hash given a target $\mathcal{T}$, the binomial probability that the attacker finds at least n confirmation blocks in time T is:

$$P^T_{n,\mathcal{T}} = 1 - \sum_{k=0}^{n-1} \binom{\mathsf{N}}{k} P^k_{v,\mathcal{T}} (1 - P_{v,\mathcal{T}})^{\mathsf{N}\text{-}k}. \tag{2}$$

Let α_c be the bribe value, δ_i the percentage price drop of the bribe in the native asset of $\mathcal{L}_{D_i}$ (the i-th destination chain) over the duration of the attack, and V_{D_i} the USD value of 1 coin of $\mathcal{L}_{D_i}$. The miners' expected gain for pulling the forgery attack is thus given by:

$$\mathbb{E}[\mathrm{F}] = \alpha_c \cdot \sum_{i=1}^{m} ((1 - \delta_i) \cdot V_{D_i}) \cdot P^T_{n,\mathcal{T}}. \tag{3}$$

We note that if miners redirect a significant mining power to forge a proof, a drop in the network computational power can be observed, and the attack can be detected, thus undermining users' trust in the chain.

Secure Parameter Space. For Glimpse to be economically secure, it has to be more profitable to honestly mine blocks rather than launch a proof forgery attack: $\mathbb{E}[\mathrm{F}] < \mathbb{E}[\mathrm{H}]$. This yields the *upper bound* on the number of coins α_c held by all active Glimpse instances on the same $\mathcal{L}_S$:

$$\alpha_c < \frac{\mathsf{R} \cdot V_S \cdot \mathbb{E}^{\mathsf{B}}_{\mathcal{L}_S}[T] \cdot \mu_r}{\sum_{i=1}^{m} ((1 - \delta_i) \cdot V_{D_i}) \cdot P^T_{n,\mathcal{T}}}. \tag{4}$$

We recall that similarly to any other bridge solution [29], any honest user willing to open a new Glimpse instance of value

[3] One might think that fooling a light client harms the miners because it necessarily creates a hard fork in contrast to faking a Glimpse proof. However, miners can create a fake light client suffix that will not be included in the blockchain because it contains invalid transactions rejected by full nodes.

α must first compute the total value locked α'_c in Glimpse (for a given $\mathcal{L}_S$), and verify that $\alpha+\alpha'_c<\alpha_c$. However, this bound is not very restrictive as α_c is the *upper bound at each point in time* and the lifetime of Glimpse contracts is constrained to rather short time windows. Glimpse is, therefore, a dynamic and fast protocol that, over time, moves large amounts of money capped by α_c at every moment in time. The value α_c may vary considerably depending on the source chain and the market. In particular, the coins that can be securely locked in Glimpse proportionally increase as the block reward and the gap between the involved currencies' values increases.

Example. Let us assume all the active Glimpse instances are between Bitcoin ($\mathcal{L}_S$) and Liquid ($\mathcal{L}_D$) with proofs having, on average, $n=5$ and $T\sim 1$ hour. We consider the Bitcoin target $\mathcal{T}$ to have 19 leading zeros (the largest over 2022), an attacker controlling 23% of the total hashing power (largest mining pool in 2022), and the average prices of BTC and Liquid on November 2022. With minimal price drop over the attack time window, all the active Glimpse instances can hold ~ 100k bitcoin, i.e., 230 million USD in January 2023. This cap is similar to the total value locked on other bridges, e.g., Gravity [32, 33], but while in Glimpse funds are locked for a short time T, and α_c is the maximum at each point in time, other bridges (e.g., Gravity) have worst performances, as the funds are locked for long periods.

6.2 Censorship Attack

Similarly to every other construction relying on timelocks [30, 31, 34], Glimpse inherits a security vulnerability when the liveness of the destination chain ($\mathcal{L}_D$) is violated. To provide a complete economic analysis, we investigate how a liveness attack can be carried out in the specific context of Glimpse and estimate its economic cost/benefit.

To violate the liveness of $\mathcal{L}_D$, the verifier V may launch a censorship attack by *bribing the validators* (e.g., miners in PoW) not to include the transaction(s). Rational parties, however, will only censor $\mathcal{L}_D$ for economic gain. That being so, the economic security of Glimpse depends on the exact conditions making this attack profitable to V and the validators. We stress that censorship attacks are not specific to Glimpse but apply to all applications building on top of a blockchain that use timelocks to ensure safety, be it cross-chain bridges or simply layer-2 solutions [30, 31, 34, 35].

Closely following [30], we define the *bribing game* as a Markov game running in $T+1$ sequential stages, a stage being the period between two blocks. In each stage, the block proposer can choose between censoring the transaction suggested by V, actively enabling the attack, or including the transaction in her block, refusing to play the game. We define the bribing game as *safe* if, after eliminating the strictly dominated strategies, the only action for each block proposer in stage one is to *refuse* the bribery and include the transaction(s).

In our analysis, we make the following assumptions as in [30]: (i) validators are rational, i.e., they always try to maximize their profit and, if they can choose, they always follow dominant strategies; (ii) validators do not create forks; (iii) the probability μ_r of a validator to be selected as block proposer is publicly known and is constant during the attack; (iv) the attacker and the victim of the bribing attack are not validators; (v) all validators can see time-locked transactions that will be valid in the future; (vi) the Glimpse lifetime T is a time lock expressed in the number of blocks, and each block generation is equivalent to a tick of the clock; finally, (vii) block rewards and fees generated outside the Glimpse protocol are constant and do not impact the attack.

We refer to *weak validator* as a validator whose probability to be selected as the next block proposer is $\mu_r<\frac{f}{\alpha}$, and we let μ_w be the sum of the probabilities of all weak validators in the system. We refer to f as the fee of the to-be-censored transaction and α as *the bribe, which is bounded by the economic value of the single Glimpse contract*. We reasonably consider $\alpha>f$. The following theorem holds, proven in [30]:

Theorem 3. *The bribing game is safe if there is at least one validator such that $\mu_r<\frac{f}{\alpha}$ (weak validator) and*

$$T>\frac{\log\frac{f}{\alpha}}{\log(1-\mu_w)}. \tag{5}$$

Therefore, to secure Glimpse from censorship attacks, we (at least) require $\alpha<\frac{f}{\mu_r}$ and T defined as in Theorem 3. For instance, considering a \$2 transaction fee in Ethereum and the lowest probability to be selected as block proposer being $1.8\cdot 10^{-6}$ [36], each Glimpse contract in Ethereum can hold at most $\alpha<1.1$ million USD. With $\mu_w=1.5\%$ and a quite high fee-to-bribe ratio we have: $T>25$ with $\frac{f}{\alpha}=0.7$, $T>15$ with $\frac{f}{\alpha}=0.8$, and $T>7$ with $\frac{f}{\alpha}=0.9$.

7 Evaluation

Overhead in Ethereum-like Chains. We now consider the Ethereum main chain, but the same discussion also applies to any Ethereum-based chain, e.g., Ethereum Classic and Ethereum PoW. In Ethereum, the cost of a transaction is measured in *gas*, which together with a gas-price specified by the issuer of the transaction results in the fees expressed in the native currency. Every computation consumes an amount of gas proportional to its complexity, and every data that is stored on-chain consumes an amount of gas proportional to its length. The computational costs of Glimpse come from the proof $\mathcal{P}^n$ verification, which, for a given target $\mathcal{T}$, consists of a Merkle proof verification for a specific transaction (or description), the transaction body and block header reconstruction, and final hash comparison, as shown in Figure 4. A Merkle tree with k leaves has a Merkle proof of size $O(\log_2 k)$. This leads to Merkle proof verification cost scaling logarithmically in the number of transactions in a block. Each of the n confirmation blocks in $\mathcal{P}^n$ yields an overhead of 36k gas.

Besides these computational costs, the Glimpse contract has to initially store the source chain target $\mathcal{T}$ as well as either the to-be-published transaction hash or a transaction description. In Ethereum the data are stored in 32-bytes slots and for each slot 20k gas are consumed: this leads to 40k gas storage cost for the target and the transaction hash, or to 188k in the case of target and ~300-bytes description. We have implemented an open-source cost evaluation which can be found in an anonymized Github repository [37].

Glimpse has lower on-chain costs compared to the overall costs of some optimized and naive relay solutions, such as Ethrelay [6] and zkRelay [5] presented in Appendix A. For Ethrelay, each block header submission results in an average cost of 280k gas, whereas the inclusion of a transaction is verified via SPV combined with an advanced search algorithm that checks for main chain membership. For relatively recent blocks, this leads to a gas consumption of 110k gas. Using zkRelay, the submission of a batch of blocks of arbitrary size costs 522k gas, including the validation of the zero-knowledge proof and the storage costs. The proof validation alone results in 351k gas. To verify that a transaction has been included in a block on the source blockchain, users have to provide the relay contract with a Merkle proof for verifying the block inclusion in the batch and a Merkle proof for the SPV.

While the relay costs for verifying the inclusion of a transaction are somewhat similar to the ones of Glimpse, the crucial difference lies in the operating costs: Relays incur high ongoing costs for relaying, verifying and storing the full list of block headers, Glimpse has none due to its on-demand nature. For Glimpse with a single well-defined transaction and $n = 5$, we have an upper bound of 330k gas: we note that this is a *one-time* fee, compared to the continuous 280k gas for each block header submission of Ethrelay and 522k per batch submission of zkRelay.

Overhead in Bitcoin-like Chains. The transaction cost in Bitcoin-based chains follows a different fee mechanism. Transaction fees are usually proportional to the size of the transaction and in Bitcoin in the order of a few satoshi per byte as of October 2022.

To cope with the limited scripting capabilities of Bitcoin-like chains, when Taproot is supported, the parties can use Merkelized Abstract Syntax Trees (MAST) [38]. A MAST can compress many scripts into a single root of a Merkle tree and we use it to efficiently encode our Glimpse contract. The MAST first needs to be constructed and then exchanged off-chain. The MAST size depends on (i) the number of variable inputs and outputs in Desc, as their well-formedness needs to be checked with dedicated opcodes, (ii) the number of confirmation blocks in $\mathcal{P}^n$, (iii) the number of transactions in the block determining the number of levels in the Merkle tree, and (iv) the size of the description, being the static data to be hard-coded in the script. For example, in a single to-be-verified transaction Tx of ~350 bytes, $n = 6$ confirmation blocks, one variable input or output, the upper bound for the MAST is 10 MB. For DNF formulas, parties need to compute and exchange the MAST for each literal. For a detailed discussion see Appendix E.

We provide a theoretical estimation for a Glimpse transaction size in Liquid, with Taproot and all the necessary string opcodes. Assuming two P2PKH inputs and one P2TR output for Tx_G, we have a transaction size of approximately 350 bytes. For Tx_P, assuming one P2TR input (Glimpse witness + selected script of the MAST) and two P2PKH outputs, the size is again roughly 350 bytes. Instead, considering one P2TR input and two P2PKH outputs, the size of Tx_V is about 200 bytes. Concretely, in November 2022, users' fees for Tx_G and Tx_P would amount to \$1.5 each, whereas for Tx_V to \$0.84. The total cost would be at most 3\$, in line with the costs for standard transactions. In the *Setup* phase, parties need to exchange Tx_G, Tx_P and Tx_V, as well as the description for the to-be-verified transaction. For verifying a DNF formula with m literals, the parties need to exchange $4 \cdot 2^m$ transactions, and V has to send to P $2 \cdot 2^m$ signatures.

For chains like Bitcoin Cash which do not support functionalities like those of Taproot, one could unroll the MAST tree, obtaining a large Glimpse script within Tx_G that is by far dominated by the opcodes for the Merkle proof verification. Let M be the maximum number of transactions in a block; then we have $\sum_{l=0}^{\log_2(M)} 2^l \cdot (3l+3)+1$ opcodes for the Merkle root reconstruction (see Appendix E). In this case, Glimpse can be supported by removing the limit for the maximum number of opcodes in a transaction (`MAX_OPS_PER_SCRIPT`). For instance, assuming $M = 3\text{k}$, one would have $l = \{0, \ldots, 12\}$, leading to ~300k opcodes in the locking script.

Computational Overhead. The computational overhead consists of the creation and verification of signatures and of the MAST, and the construction of the proofs, all of which can be performed using commodity hardware.

8 Conclusion

We present Glimpse, an *on-demand cross-chain synchronization primitive* that requires only constant on-chain storage, cost and computational overhead (*efficiency*), enables many applications such as lending, Proofs-of-Burn, proofs of oracle attestations, or off-chain applications (*expressiveness*), and features *compatibility* with chains having limited scripting capabilities. We demonstrate how Glimpse enables the design of sophisticated cross-chain applications, such as lending, backed assets, Proofs-of-Burn, multiple oracle attestations, and layer-2 applications. Security and atomic synchronization are proven in the UC framework, and we then extend the security analysis of Glimpse by including rational players in our model. Glimpse opens many exciting research directions, which we intend to explore. E.g., how can different consensus mechanisms (Proof-of-Stake, Proof-of-Space) be supported, or conducting a more exhaustive the economic analysis of cross-chain protocols, e.g., including feather fork attacks [39].

Acknowledgments

The work was partially supported by CoBloX Labs, by the European Research Council (ERC) under the European Union's Horizon 2020 research (grant agreement 771527-BROWSEC), by the Austrian Science Fund (FWF) through the projects PROFET (grant agreement P31621), the project CoRaF (grant agreement 2020388), and the project W1255-N23, by the Austrian Research Promotion Agency (FFG) through the COMET K1 SBA and COMET K1 ABC, by the Vienna Business Agency through the project Vienna Cybersecurity and Privacy Research Center (VISP), by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development and the Christian Doppler Research Association through the Christian Doppler Laboratory Blockchain Technologies for the Internet of Things (CDL-BOT).

References

[1] "Ronin attack shows cross-chain crypto is a 'bridge' too far." [Online]. Available: https://www.coindesk.com/layer2/2022/04/05/ronin-attack-shows-cross-chain-crypto-is-a-bridge-too-far/

[2] "Hackers have stolen $1.4 billion this year using crypto bridges. here's why it's happening," 2022.

[3] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2009, http://bitcoin.org/bitcoin.pdf.

[4] "Btc relay," https://github.com/ethereum/btcrelay,http://btcrelay.org/, 2016.

[5] M. Westerkamp and J. Eberhardt, "zkrelay: Facilitating sidechains using zksnark-based chain-relays," in *IEEE European Symposium on Security and Privacy Workshops*, 2020.

[6] P. Frauenthaler, M. Sigwart, C. Spanring, M. Sober, and S. Schulte, "ETH relay: A cost-efficient relay for ethereum-based blockchains," in *IEEE International Conference on Blockchain*. IEEE, 2020.

[7] A. Kiayias, A. Miller, and D. Zindros, "Non-interactive proofs of proof-of-work," in *Financial Cryptography and Data Security FC*, 2020. [Online]. Available: https://doi.org/10.1007/978-3-030-51280-4_27

[8] B. Bünz, L. Kiffer, L. Luu, and M. Zamani, "Flyclient: Super-light clients for cryptocurrencies," in *IEEE Symposium on Security and Privacy, SP*, 2020. [Online]. Available: https://doi.org/10.1109/SP40000.2020.00049

[9] T. Xie, J. Zhang, Z. Cheng, F. Zhang, Y. Zhang, Y. Jia, D. Boneh, and D. Song, "zkbridge: Trustless cross-chain bridges made practical," in *ACM SIGSAC Conference on Computer and Communications Security, CCS*, 2022. [Online]. Available: https://doi.org/10.1145/3548606.3560652

[10] M. Bartoletti and R. Zunino, "Bitml: A calculus for bitcoin smart contracts," in *ACM SIGSAC Conference on Computer and Communications Security, CCS*, 2018. [Online]. Available: https://doi.org/10.1145/3243734.3243795

[11] J. Prestwich, "Non-atomic swaps," 2019, https://ethresear.ch/t/stateless-spv-proofs-and-economic-security/5451.

[12] F. Barbàra and C. Schifanella, "Bxtb: cross-chain exchanges of bitcoins for all bitcoin wrapped tokens," in *Fourth International Conference on Blockchain Computing and Applications, BCCA*, 2022. [Online]. Available: https://doi.org/10.1109/BCCA55292.2022.9922019

[13] L. Aumayr, O. Ersoy, A. Erwig, S. Faust, K. Hostáková, M. Maffei, P. Moreno-Sanchez, and S. Riahi, "Generalized channels from limited blockchain scripts and adaptor signatures," in *Asiacrypt*, 2021.

[14] "Taproot: Segwit version 1 spending rules," https://github.com/bitcoin/bips/blob/master/bip-0341.mediawiki, 2020.

[15] "Validation of taproot scripts," https://github.com/bitcoin/bips/blob/master/bip-0342.mediawiki, 2020.

[16] A. Zamyatin, M. Al-Bassam, D. Zindros, E. Kokoris-Kogias, P. Moreno-Sanchez, A. Kiayias, and W. J. Knottenbelt, "Sok: Communication across distributed ledgers," in *Financial Cryptography and Data Security: 25th International Conference, FC 2021,*, 2021.

[17] "BSIP 64: Optional HTLC preimage length and add hash160 algorithm," https://github.com/bitshares/bsips/issues/163.

[18] "bitcoindev Speedy covenants (OP_CAT2)," https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2022-May/020434.html, 2022.

[19] J. A. Garay, A. Kiayias, and N. Leonardos, "The bitcoin backbone protocol: Analysis and applications," in *Advances in Cryptology - EUROCRYPT*. Springer, 2015.

[20] T. Dryja, "Discreet log contracts," https://adiabat.github.io/dlc.pdf.

[21] M. Sober, G. Scaffino, and C. S. S. Schulte, "A voting-based blockchain interoperability oracle," in *IEEE International Conference on Blockchain*, 2021.

[22] A. Miller, I. Bentov, R. Kumaresan, and P. McCorry, "Sprites and state channels: Payment networks that go faster than lightning," in *FC 2019: Financial Cryptography and Data Security*, 2019.

[23] S. Dziembowski, S. Faust, and K. Hostáková, "General State Channel Networks," in *Computer and Communications Security, CCS*, 2018.

[24] S. Dziembowski, L. Eckey, S. Faust, J. Hesse, and K. Hostáková, "Multi-party Virtual State Channels," in *Advances in Cryptology - EUROCRYPT*, 2019, pp. 625–656.

[25] L. Aumayr, P. Moreno-Sanchez, A. Kate, and M. Maffei, "Blitz: Secure Multi-Hop Payments Without Two-Phase Commits," in *USENIX Security Symposium*, 2021.

[26] L. Aumayr, P. M. Sanchez, A. Kate, and M. Maffei, "Breaking and Fixing Virtual Channels: Domino Attack and Donner," in *NDSS*, 2023.

[27] R. Canetti, Y. Dodis, R. Pass, and S. Walfish, "Universally composable security with global setup," in *Theory of Cryptography*, 2007.

[28] C. Badertscher, U. Maurer, D. Tschudi, and V. Zikas, "Bitcoin as a transaction ledger: A composable treatment," in *Advances in Cryptology – CRYPTO 2017*, 2017.

[29] "Vitalik butering blog post about cross-chain applications," https://old.reddit.com/r/ethereum/comments/rwojtk/ama_we_are_the_efs_research_team_pt_7_07_january/hrngyk8/, 2022.

[30] T. Nadahalli, M. Khabbazian, and R. Wattenhofer, "Timelocked bribing." Berlin, Heidelberg: Springer-Verlag, 2021. [Online]. Available: https://doi.org/10.1007/978-3-662-64322-8_3

[31] Z. Avarikioti, L. Thyfronitis, and S. Orfeas, "Suborn channels: Incentives against timelock bribes," in *Financial Cryptography and Data Security*, I. Eyal and J. Garay, Eds. Springer International Publishing, 2022.

[32] "Gravity bridge," https://github.com/Gravity-Bridge/Gravity-Docs, 2022.

[33] "Value locked in ethereum 11 bridges," 2023, https://www.theblock.co/data/scaling-solutions/scaling-overview/value-locked-of-ethereum-11-bridges.

[34] I. Tsabary, M. Yechieli, A. Manuskin, and I. Eyal, "MAD-HTLC: because HTLC is crazy-cheap to attack," in *IEEE Symposium on Security and Privacy, SP*, 2021. [Online]. Available: https://doi.org/10.1109/SP40001.2021.00080

[35] Z. Avarikioti, E. Kokoris-Kogias, R. Wattenhofer, and D. Zindros, "Brick: Asynchronous incentive-compatible payment channels," in *Financial Cryptography and Data Security FC*, 2021. [Online]. Available: https://doi.org/10.1007/978-3-662-64331-0_11

[36] "Ethereum beacon scan," https://beaconscan.com/statistics.

[37] "Glimpse," https://github.com/Glimpse-CrossChainPrimitive/Glimpse, 2022.

[38] "Merkelized Abstract Syntax Tree (MAST)," https://bitcoinops.org/en/topics/mast/.

[39] "Feather-forks: enforcing a blacklist with sub-50https://bitcointalk.org/index.php?topic=312668.msg3353004#msg3353004, 2013.

[40] A. Zamyatin, D. Harz, J. Lind, P. Panayiotou, A. Gervais, and W. Knottenbelt, "Xclaim: Trustless, interoperable, cryptocurrency-backed assets," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019.

[41] TierNolan, "Alt chains and atomic transfers," 2013.

[42] M. Herlihy, "Atomic cross-chain swaps," *CoRR*, vol. abs/1801.09515, 2018. [Online]. Available: http://arxiv.org/abs/1801.09515

[43] J. Xu, D. Ackerer, and A. Dubovitskaya, "A game-theoretic analysis of cross-chain atomic swaps with htlcs," *CoRR*, vol. abs/2011.11325, 2020. [Online]. Available: https://arxiv.org/abs/2011.11325

[44] J. Gugger, "Bitcoin-monero cross-chain atomic swap," Cryptology ePrint Archive, Paper 2020/1126, 2020, https://eprint.iacr.org/2020/1126. [Online]. Available: https://eprint.iacr.org/2020/1126

[45] S. Thyagarajan, G. Malavolta, and P. Moreno-Sanchez, "Universal atomic swaps: Secure exchange of coins across all blockchains," in *2022 IEEE Symposium on Security and Privacy (SP) (SP)*, 2022.

[46] P. Hoenisch, S. Mazumdar, P. Moreno-Sanchez, and S. Ruj, "Lightswap: An atomic swap does not require timeouts at both blockchains," Cryptology ePrint Archive, Paper 2022/1650, 2022, https://eprint.iacr.org/2022/1650. [Online]. Available: https://eprint.iacr.org/2022/1650

[47] "Submarine swap in lightning network," https://wiki.ion.radar.tech/tech/research/submarine-swap, 2021.

[48] "What is atomic swap and how to implement it," https://www.axiomadev.com/blog/what-is-atomic-swap-and-how-to-implement-it/.

[49] M. Westerkamp and M. Diez, “Verilay: A verifiable proof of stake chain relay,” in *IEEE International Conference on Blockchain and Cryptocurrency, ICBC*, 2022.

[50] T. Bugnet and A. Zamyatin, “XCC: Theft-resilient and collateral-optimized cryptocurrency-backed assets,” Cryptology ePrint Archive, Paper 2022/113, 2022.

[51] “Bitcoin wiki: Payment channels,” 2018, https://en.bitcoin.it/wiki/Payment_channels.

[52] G. Malavolta, P. Moreno-Sanchez, C. Schneidewind, A. Kate, and M. Maffei, “Anonymous multi-hop locks for blockchain scalability and interoperability,” in *Network and Distributed System Security Symposium, NDSS*, 2019.

[53] S. Dziembowski, L. Eckey, S. Faust, and D. Malinowski, “Perun: Virtual payment hubs over cryptocurrencies,” in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 106–123.

[54] L. Aumayr, O. Ersoy, A. Erwig, S. Faust, K. Hostáková, M. Maffei, P. Moreno-Sanchez, and S. Riahi, “Bitcoin-Compatible Virtual Channels,” in *IEEE Symposium on Security and Privacy*, 2021.

[55] J. Katz, U. Maurer, B. Tackmann, and V. Zikas, “Universally composable synchronous computation,” in *Theory of Cryptography TCC*, A. Sahai, Ed., vol. 7785, 2013, pp. 477–498.

[56] “The bitcoin script language,” https://betterprogramming.pub/the-bitcoin-script-language-e4379908448f, 2021.

[57] “Transactions,” Bitcoin developer: https://developer.bitcoin.org/reference/transactions.html#:~:text=Bitcoin%20transactions%20are%20broadcast%20between,part%20of%20the%20consensus%20rules..

[58] “Bitcoin core,” https://github.com/bitcoin/bitcoin/blob/master/src/script/script.h, 2022.

A Further Related Work

The idea of chain relays first appeared with BTC Relay [4], realizing a Bitcoin relay on Ethereum. BTC Relay verifies and stores Bitcoin block headers; the costs the relayers had to bare for keeping the relay up-to-date are high (linear in the total number of blocks within the blockchain) and not compensated by user's fees.

Westerkamp et al. [5] introduced zkRelay which batches multiple headers. Their validity is verified off-chain and proven on-chain via zkSNARKs. zkRelay has constant verification costs and releases the target ledger from processing and storing every single block header of the source blockchain. Although the on-chain costs are lower than for BTC Relay, a maintenance overhead for the off-chain computation and for on-chain storage remain. Furthermore, the users' costs for transaction inclusion verification are doubled, as both the block inclusion in the batch and the transaction inclusion in the block have to be verified.

Efficient relay contracts for Ethereum on Ethereum Classic (and vice versa) are even more challenging to design, as Ethereum has a complex and ASIC-resistant consensus in place. To this end, Frauenthaler et al. [6] propose Ethrelay, a relay that employs an optimistic approach: Block headers are optimistically accepted as valid and only validated on-demand. The computational costs per header are cut out, but the storage costs persist.

FlyClient [8] is a super-light client with a logarithmic overhead for PoW chain. It leverages Non-Interactive Proofs of Proof of Work (NiPoPoW) proposed in [7], augmenting it to work for chains of variable difficulty. FlyClient only requires processing a logarithmic number of block headers while storing only a single block header between executions. Despite the remarkable achievement, by using NiPoPoW, FlyClient is not backward compatible with Bitcoin-base chains and its use in practice requires a hard fork.

Zamyatin et al. [40] propose XCLAIM, a framework for trustless and efficient cross-chain exchanges. XCLAIM exhibits functionalities for issuing, transferring, swapping and redeeming cryptocurrency-backed assets securely on existing blockchains. To make the protocol non-interactive, the XCLAIM implementation operating between Bitcoin and Ethereum makes use of a chain relay on Ethereum, specifically of the implementation of BTC Relay. The relay costs are shared among all users of XCLAIM, with decreasing costs for very active users.

Gravity [32] is a bidirectional bridge solution between Ethereum and the Cosmos ecosystem. The Gravity bridge has two main components: a Solidity smart contract deployed on Ethereum and a Cosmos SDK blockchain module. Users deposit assets on one side of the bridge (e.g., Cosmos) and a token representation is minted on the other side of the bridge (e.g., Ethereum), and vice versa. Gravity relies on 2/3 of a set of 140 validators to sign transactions attesting on Cosmos deposits on the Ethereum side and vice versa. To join as a validator, one has to stake assets, which are slashed upon detected misbehavior. In Gravity, as well as similar bridge solutions, users needs to assume an honest super majority of validators: this additional trust assumption inherently makes their cross-chain token less secure than the native ones.

Another conceptually and technically different solution for cross-chain communication is atomic swaps, which likely originated from a forum user TierNolan [41] and was later analyzed by, e.g., Herlihy [42] or Xu et al. [43]. Atomic swaps allow multiple parties to exchange assets across multiple blockchains in a distributed and coordinated manner. These atomic swaps are based on HTLCs, i.e., contracts storing a

Protocol	Commit on $\mathcal{L}_S$			Verify & Commit on $\mathcal{L}_D$		Expressiveness
	Ass.	Comp.	Consensus	Ass.	Comp.	
Universal Atomic Swap [45]	Sync	①	Any	Sync	①	Secret-based logic (Adapt. Sigs)
HTLC Atomic Swap [42, 47, 48]	Sync	①	Any	Sync	①	Secret-based logic (HTLC)
Glimpse (this work)	Sync	①	PoW	Sync	②	DNF formulas over transactions (or descriptions)
Bidirectional chain relays [4–6, 49]	Sync	③	PoW, PoS	Sync	③	Arbitrary logic
XCLAIM [40], XCC [50]	TTP	①	PoW, PoS	Sync	③	Arbitrary logic
Gravity Bridge [32]	TTP	③	Ethereum, BFT	TTP	③	Arbitrary logic

Table 2: Classification of state-of-the-art CCC protocols w.r.t.: (i) the assumption they make (TTP/Synchrony), (ii) the interoperability they achieve w.r.t. scripting requirements, and (iii) the consensus they operate on.

pair (h,t) and ensuring that if the contract receives the secret s such that $h = \mathcal{H}(s)$ before time t has elapsed, then the ownership of the asset locked in the contract are transferred to the counter party. This secret-based solution is cheap and elegant, but, contrarily to Glimpse, it can only be used in an asymmetric setting, where the party posting a transaction on $\mathcal{L}_S$ cannot be the same one posting a transaction on $\mathcal{L}_D$, besides being limited in expressiveness.

There exist numerous proposals for atomic swaps, e.g., Gugger proposed a construction compatible with Monero [44]. Thyagarajan et al. [45] further enhanced the compatibility by removing the need for hash locks or timelocks, thereby being usable in any blockchain that allows signature verification of transactions. Hoenisch et al. proposed atomic swaps that require no timelock (or even timelock puzzle) on one of the chains [46].

Table 2 compares Glimpse to other state-of-the-art cross-chain solutions: (i) Glimpse completely relies on synchrony assumptions, (ii) Glimpse makes use of a basic set of scripting operations, and (iii) Glimpse can be used to encode DNF-based synchronization patterns. With ①,②,③ we denote three classes of scripting languages: ① comprises hash locks, time locks, and signature locks, ② includes the operations in ① along with the following functionalities for string concatenation and hash comparison, and ③ finally represents any quasi-Turing complete language.

Lock Contract and Chain Relay Limitations. Existing cross-chain communication solutions not relying on a TTP fall into two main categories: lock contracts and chain relays. Lock contracts are an umbrella term for non-custodial locking mechanisms (e.g., Hashed-Timelocked-Contracts, adaptor signatures) that achieve security and atomicity from the hardness of some cryptographic assumptions. Hash locks and adaptor signatures are, for instance, lock contract schemes broadly used to encode blockchain applications such as atomic swaps, payment channels [22, 51], multi-hop payments [25, 52], virtual channels [24, 26, 53, 54], and discreet log contracts [20]. Lock contracts use a statement S that ties the authorization of a transaction Tx_2 to the leakage of a secret witness s of some hard relation (usually leaked within a transaction Tx_1 posted on-chain). Lock contracts can encode a class of *asymmetric problems: The party posting transaction* Tx_1 *cannot be the same posting transaction* Tx_2. Intuitively, the party who posts transaction Tx_2 has to gain knowledge of s only after transaction Tx_1 has been posted. Lock contracts are cheap and lightweight, and since they require minimal scripting capabilities, they can be leveraged on all existing chains. On the other hand, they enable a very limited number of (asymmetric) applications: They cannot be used, e.g., for Proofs-of-Burn, wrapping and unwrapping of tokens, etc.

Chain relays theoretically represent the ideal solution for interoperability, allowing *any* party to verify on $\mathcal{L}_D$ the inclusion of *any* transaction in $\mathcal{L}_S$. However, they are costly to operate and do not represent an easily viable solution for interoperability: To the best of our knowledge, there is a single relay currently operating, where relayers are heavily subsidized via ad-hoc incentive mechanisms [40]. A relay is essentially a light client operating within a smart contract. The block headers are constantly relayed from $\mathcal{L}_S$ to $\mathcal{L}_D$ by off-chain untrusted clients called *relayers*. Since malicious relayers might submit invalid block headers, the contract ensures correct functioning by (i) internally validating the headers by partially replicating the consensus mechanism of $\mathcal{L}_S$, and (ii) resolving temporary forks.

B Modeling Glimpse in the UC-Framework

Protocol, Adversarial Model, Time, Communication. We consider a *real world* protocol Π executed by a set of parties $\mathcal{U}$ and in the presence of an *adversary* $\mathcal{A}$. Π is parameterized by a security parameter $\lambda \in \mathbb{N}$ and an auxiliary input $z \in \{0,1\}^*$. $\mathcal{A}$ can *corrupt* any party $P \in \mathcal{U}$ prior to the protocol execution (static corruption) by taking full control of it and learning its internal state. A special entity $\mathcal{Z}$, the *environment*, provides parties and $\mathcal{A}$ with inputs and receives their outputs. $\mathcal{Z}$ represents anything external to the protocol.

We assume synchronous communication, i.e., the protocol execution proceeds in synchronized rounds. We follow [24, 55], formalizing these rounds with a global ideal functionality $\mathcal{G}_{Clock}$ which can be seen as a global clock. At a high level, this functionality proceeds to the next round only after all honest parties indicate that they are ready to do so. Every party knows what the current round is.

We model message exchange between parties via authenticated communication channels with guaranteed delivery after one round. This notion is formalized with the functionality $\mathcal{G}_{GDC}$ (e.g., [24]), and basically, this means that if a party P

sends a message to Q in round t, Q will receive this message exactly at the beginning of round $t+1$. The adversary $\mathcal{A}$ has the power to observe the content of messages between parties and can reorder any messages sent within the same round, but $\mathcal{A}$ cannot delay, modify or censor messages or insert new messages on an honest party's behalf. We assume that any computation made by entities and communication that does not involve two parties, but rather a special entity such as $\mathcal{A}$ or $\mathcal{Z}$, takes zero rounds.

Modeling the Ledger. For modeling the ledger we refer to the functionality $\mathcal{G}_{Ledger}$ as defined in [28]. Concretely, we use a concrete instantiation also as specified in [28], where the parameters are chosen such that the ledger achieves both *liveness* and *consistency* (or just consistency), which is defined in [19]. Concretely, we interact with the ledger mainly in two ways: posting transactions and reading the ledger (e.g., to see if a certain transaction appeared on it). A ledger has a delay parameter Δ which is an upper bound on the number of rounds which it takes for valid transactions to appear on the ledger after being posted.

The GUC Security Definition. Let Π be a hybrid protocol with access to the preliminary functionalities $\mathcal{F}_{prelim}$ consisting of $\mathcal{G}_{Clock}$, $\mathcal{G}_{GDC}$ and $\mathcal{G}_{Ledger}$. We define as $\texttt{EXEC}^{\mathcal{F}_{prelim}}_{\Pi,\mathcal{A},\mathcal{Z}}(\lambda,z)$ the output of $\mathcal{Z}$ interacting with Π and $\mathcal{A}$ on input a security parameter λ and an auxiliary input z. Further, we denote $\phi_{\mathcal{F}}$ as the ideal protocol of the ideal functionality $\mathcal{F}_{Glimpse}$ with access to the same preliminary functionalities $\mathcal{F}_{prelim}$. $\phi_{\mathcal{F}}$ is a trivial protocol where parties merely forward any input to $\mathcal{F}_{Glimpse}$. The output of an environment $\mathcal{Z}$ on input λ and an auxiliary input z interacting with $\phi_{\mathcal{F}}$ and an ideal world adversary $\mathcal{S}$ (also called *simulator)* is denoted as $\texttt{EXEC}^{\mathcal{F}_{prelim}}_{\phi_{\mathcal{F}},\mathcal{S},\mathcal{Z}}(\lambda,z)$.

We proceed with our main security definition. Informally, if a real world protocol Π GUC-realizes an ideal functionality $\mathcal{F}_{Glimpse}$, any attack carried out against Π can be carried out against $\phi_{\mathcal{F}}$.

Definition 3. A real world protocol Π GUC-realizes an ideal functionality $\mathcal{F}_{Glimpse}$ with respect to preliminary functionalities $\mathcal{F}_{prelim}$, if for any real world adversary $\mathcal{A}$ there exists an ideal world adversary $\mathcal{S}$ such that

$$\left\{\texttt{EXEC}^{\mathcal{F}_{prelim}}_{\Pi,\mathcal{A},\mathcal{Z}}(\lambda,z)\right\}_{\substack{\lambda\in\mathbb{N},\\ z\in\{0,1\}^*}} \overset{c}{\approx} \left\{\texttt{EXEC}^{\mathcal{F}_{prelim}}_{\phi_{\mathcal{F}},\mathcal{S},\mathcal{Z}}(\lambda,z)\right\}_{\substack{\lambda\in\mathbb{N},\\ z\in\{0,1\}^*}}$$

where $\approx^c$ denotes computational indistinguishability.

B.1 Ideal Functionality

To capture the desired functionality of our scheme, we model Glimpse as an ideal functionality. In fact, we provide two slightly different functionality definitions $\mathcal{F}_{W-Glimpse}$ and $\mathcal{F}_{S-Glimpse}$, the former achieving *weak atomicity* and the latter achieving *strong atomicity*, which are our desired properties. Note that this functionality (and subsequently also the protocol) considers only two parties per execution, P and V. In particular, we let the issuance of the transaction relevant to $\mathcal{F}_{Glimpse}$ (or the protocol) on $\mathcal{L}_S$ (the *issuer I*) be handled by $\mathcal{Z}$. This captures any conceivable setting, e.g., where I is adversarial, the same as or colluding with either of the parties P and V.

A Generic Functionality. The functionalities are parameterized by two ledgers $\mathcal{L}_S$ and $\mathcal{L}_D$, both of which are instances of $\mathcal{G}_{Ledger}$, and a delay parameter Δ_D, which for readability we write explicitly as a parameter, but which is also implicitly given by $\mathcal{L}_D$.

Both functionalities allow for (i) verifying DNF formulas over descriptions posted on $\mathcal{L}_S$ on $\mathcal{L}_D$ instead of single transactions and (ii) multiple different outcomes for the prover. In other words, an outcome on $\mathcal{L}_D$ can be tied to a specific combination of truth values of the variables in the formula $\mathcal{F}_S$ (see Figure 2). The truth variables inside these logical formulas are descriptions of transactions. On a high level, each description is set to *true* if a transaction Tx corresponding to this description, i.e., $[\mathsf{Tx}] \hookleftarrow \mathsf{Desc}$, appears on $\mathcal{L}_S$, and is set to *false* otherwise. The formula $\mathcal{F}_S$, in combination with this interpretation of descriptions as boolean variables, generates a truth table, which we say is the truth table associated with $\mathcal{F}_S$.

For each possible combination of truth values (i.e., each row of the truth table), which should benefit the prover, we can now assign a unique outcome, whereas for the other ones, the verifier gets all the money (see also Appendix D).

Functionality and Properties. Our functionality proceeds in two phases, *Setup* and *Verify & Commit on* $\mathcal{L}_D$: the former is the same in both functionalities, the latter changes depending on *weak* or *strong* atomicity.

- *Setup:* In this phase, the functionality gets the required inputs to set up a Glimpse contract from V, checks that they are well-formed, informs P. Finally, a transaction hosting the Glimpse contract has to appear on $\mathcal{L}_D$. This phase is the same for both functionalities. We do not constrain how two parties P and V agree on the parameters, which is why V already sends the setup message with all parameters specified.
- *Verify & Commit on $\mathcal{L}_D$ (weak atomicity):* The functionality $\mathcal{F}_{W-Glimpse}$ expects that if a transaction spending the Glimpse transaction appears on $\mathcal{L}_D$ corresponding to the outcome associated with one of the rows of the truth table for $\mathcal{F}_S$, then a transaction corresponding to each description that is set to *true* in that row must already be on $\mathcal{L}_S$. Otherwise, the functionality outputs `error`. For $\mathcal{F}_{W-Glimpse}$, $\mathcal{L}_S$ and $\mathcal{L}_D$ need to have have consistency, otherwise this notion of weak atomicity would be meaningless, as transactions are not stable and can be removed from the ledger again.
- *Verify & Commit on $\mathcal{L}_D$ (strong atomicity):* In addition to what happens for weak atomicity, the functionality $\mathcal{F}_{S-Glimpse}$ expects the "other way around". This means that if a set of transactions appears on $\mathcal{L}_S$ that correspond

to descriptions in $\mathcal{F}_S$ for some row of its truth table, then a transaction with the outcome corresponding to that row must appear on $\mathcal{L}_D$, spending from the transaction hosting the Glimpse contract. Otherwise, the functionality outputs `error`. For $\mathcal{F}_{S-Glimpse}$, in addition to $\mathcal{L}_S$ and $\mathcal{L}_D$ needing to have consistency, $\mathcal{L}_D$ needs to have liveness. This notion of strong atomicity would not be realizable without $\mathcal{L}_D$ having liveness, as the functionality always expects the corresponding transaction to appear on $\mathcal{L}_D$.

Errors, Staleness and Notation. Naturally, the ideal functionalities directly defines the desired properties. We note that our functionalities satisfy weak or strong atomicity if no `error` is output. If an error is output, then all guarantees are lost. Thus, *we are only interested in protocols realizing either functionality that never output* `error`.

The *Verify & Commit on $\mathcal{L}_D$* phase for both weak and strong atomicity are "executed in every round". This phrasing is used to ease readability. This can be achieved by marking the functionality as *stale*, if it does not receive the execution token from the environment in every round. Then, the next time the functionality receives the execution token and is *stale*, it outputs `error`.

Finally, to ease readability, we omit explicit calls to $\mathcal{G}_{Clock}$ and $\mathcal{G}_{GDC}$. Instead, we denote $(m) \overset{t}{\hookrightarrow} X$ as sending message (m) to party X in round t and denote $(m) \overset{t}{\hookleftarrow} X$ as receiving message (m) from X in round t. We abstain from explicitly mentioning session identifiers *sid* or sub-session *ssid* identifiers in every message. The formal definition of the functionality follows.

$\mathcal{F}_{Glimpse}(\mathcal{L}_S, \mathcal{L}_D, \Delta_D)$
consisting of $\mathcal{F}_{W-Glimpse}$ and $\mathcal{F}_{S-Glimpse}$

Parameters:
$\mathcal{L}_S$, $\mathcal{L}_D$... two instances of $\mathcal{G}_{Ledger}$, representing the source and destination blockchain
$\Delta_D \in \mathbb{N}$ ... the blockchain delay of $\mathcal{L}_D$, i.e., the upper bound on the time it takes from posting a valid transaction Tx to Tx appearing on the the ledger.

Variables:

Φ ... a set of tuples $(\mathtt{id}, \mathcal{F}_S, T_P, T_V, n, P, V, [\mathsf{outcome}_i], \mathsf{Tx_G})$, where $\mathtt{id} \in \{0,1\}^*$ is an identifier unique to the pair P and V. P and V are in turn both distinct elements of the set of all users $\mathcal{U}$. $\mathcal{F}_S$ is a logical formula as defined in Figure 2. Further, $T_P, T_V, n \in \mathbb{N}$, and $[\mathsf{outcome}_i]$ is a list of outcomes, which in turn are tuples $(\mathsf{outcome}.P, \mathsf{outcome}.V) \in \mathbb{N}^2$.

Setup

1. Upon $(\mathtt{SETUP}, \mathtt{id}, \mathcal{F}_S, P, [\mathsf{outcome}_i]_{i\in[1,r]}, T_P, T_V, n, \{inputP\}, \{inputV\}) \overset{\tau}{\hookleftarrow} V$, where the following holds:
   (a) $\mathcal{F}_S$ is a logical formula as defined in Figure 2.
   (b) $[\mathsf{outcome}_i]$ is a list of $r := 2^d$ outcomes, where d is the number of descriptions in $\mathcal{F}_S$ (in other words, the number of rows in the truth table when considering all descriptions in $\mathcal{F}_S$ as boolean variables).
   (c) For all rows i of the truth table generated by $\mathcal{F}_S$, where the result is *false*, it must hold that $\mathsf{outcome}_i := (0, \alpha)$.
   (d) For each $\mathsf{outcome}_i$ it must hold that $\mathsf{outcome}_i.P + \mathsf{outcome}_i.V = \alpha$ for some number α
   (e) $T_V > T_P$ are both times in the future
   (f) $n \in \mathbb{N}$
   (g) $\{inputV\}$ is a (potentially empty) set of inputs under control of V and $\{inputP\}$ is a (potentially empty) set of inputs under control of P
   (h) $|\{inputV\} \cup \{inputP\}| > 0$ and the sum of coins stored in $\{inputV\} \cup \{inputP\} \geq \alpha + d \cdot \varepsilon$
   (i) If these checks hold continue, else go idle.
2. Send $(\mathtt{id}, \mathcal{F}_S, [\mathsf{outcome}_i]_{i\in[1,r]}, T_P, T_V, n, \{inputP\}, \{inputV\}) \overset{\tau+1}{\hookrightarrow} P$, receive $(\mathtt{id}) \overset{\tau+1}{\hookleftarrow} P$.
3. At round $\tau_1 \leq \tau + 1 + \Delta_D$, if a transaction $\mathsf{Tx_G}$ appears on $\mathcal{L}_D$ which takes $\{inputP\}$ and $\{inputV\}$ as input and has at least one output θ_α holding α coins, add $(\mathtt{id}, \mathcal{F}_S, T_P, T_V, n, P, V, [\mathsf{outcome}_i]_{i\in[1,r]}, \mathsf{Tx_G})$ in Φ.

(a) Weak atomicity (Functionality $\mathcal{F}_{W-Glimpse}$)

Verify & Commit on $\mathcal{L}_D$ (in every round τ)

For every $(\mathtt{id}, \mathcal{F}_S, T_P, T_V, n, P, V, [\mathsf{outcome}_i]_{i\in[1,r]}, \mathsf{Tx_G})$ in Φ, if current round τ is smaller than T_P, do the following.

1. If there is a transaction Tx on $\mathcal{L}_D$, such that Tx spends output θ_α of $\mathsf{Tx_G}$ and has two outputs $\theta_P := (x, \mathsf{OneSig}(\mathsf{pk}_P))$ and $\theta_V := (y, \mathsf{OneSig}(\mathsf{pk}_P))$, s.t. $x \geq \mathsf{outcome}_i.P$ and $y \geq \mathsf{outcome}_i.V$ corresponds to the k-th element in the list $[\mathsf{outcome}_i]_{i\in[1,r]}$. Check the k-th row in the truth table corresponding to $\mathcal{F}_S$.
2. For each description $\mathsf{Desc} \in \mathcal{F}_S$ which is set to *true* in the k-th row in the truth table, a transaction Tx_i with n subsequent blocks, s.t. $[\mathsf{Tx}_i] \hookleftarrow \mathsf{Desc}$, must be on $\mathcal{L}_S$. Additionally, for each description $\mathsf{Desc} \in \mathcal{F}_S$ which is set to *true* in the k-th row in the truth table, there must not be a transaction Tx_j, s.t. $[\mathsf{Tx}_j] \hookleftarrow \mathsf{Desc}$ on $\mathcal{L}_S$. If this does not hold, output $(\mathtt{id}, \mathtt{error})$.

(b) Strong atomicity (Functionality $\mathcal{F}_{S-Glimpse}$)

Verify & Commit on $\mathcal{L}_D$ from (a) (in every round τ)

Verify & Commit on $\mathcal{L}_D$: P (in every round τ)

For every $(\mathtt{id}, \mathcal{F}_S, T_P, T_V, n, P, V, [\mathsf{outcome}_i]_{i\in[1,r]}, \mathsf{Tx_G})$ in Φ where P is honest and θ_α of $\mathsf{Tx_G}$ is unspent, do the following.

1. If current round τ is $T_P - \Delta_D$. Let $\{\mathsf{Tx}_i\}$ be the set of all transactions on $\mathcal{L}_S$ where the block in which each transaction is each has at least n subsequent blocks, and where for each $\mathsf{Tx}_i \in \{\mathsf{Tx}_i\}$ there exists $\mathsf{Desc} \in \mathcal{F}_S$ where $[\mathsf{Tx}_i] \hookleftarrow \mathsf{Desc}$. Evaluate the statement $\mathcal{F}_S$ by setting to *true* all descriptions for $[\mathsf{Tx}_i]$ whose corresponding transaction Tx_i is on $\mathcal{L}_S$. Set all other descriptions to *false*.
2. If the statement evaluates to *true*, do the following, let k be the row in the truth table corresponding to the evaluation of the statement of the previous step and proceed. Otherwise go idle.

3. Expect a transaction Tx to appear on $\mathcal{L}_D$ after at most $2 \cdot \Delta_D$ rounds, such that Tx spends output θ_α of Tx_G and has two outputs $\theta_P := (x, \mathsf{OneSig}(\mathsf{pk}_P))$ and $\theta_V := (y, \mathsf{OneSig}(\mathsf{pk}_P))$, s.t. $x \geq \mathsf{outcome}_i.P$ and $y \geq \mathsf{outcome}_i.V$ of the k-th element in the list $[\mathsf{outcome}_i]_{i\in[1,r]}$. If no such transaction appears within said time, output $(\mathtt{id}, \mathtt{error})$.

Verify & Commit on $\mathcal{L}_D$: V (in every round τ)

For every $(\mathtt{id}, \mathcal{F}_S, T_P, T_V, n, P, V, [\mathsf{outcome}_i]_{i\in[1,r]}, \mathsf{Tx}_\mathsf{G})$ in Φ where V is honest and θ_α of Tx_G is unspent, do the following.

1. If current round τ is $T_P - \Delta_D$. Let $\{\mathsf{Tx}_i\}$ be the set of all transactions on $\mathcal{L}_S$ where the block in which each transaction is each has at least n subsequent blocks, and where for each $\mathsf{Tx}_i \in \{\mathsf{Tx}_i\}$ there exists $\mathsf{Desc} \in \mathcal{F}_S$ where $[\mathsf{Tx}_i] \hookleftarrow \mathsf{Desc}$. Evaluate the statement $\mathcal{F}_S$ by setting to *true* all descriptions for $[\mathsf{Tx}_i]$ whose corresponding transaction Tx_i is on $\mathcal{L}_S$. Set all other descriptions to *false*.
2. If the statement evaluates to *false*, the following must happen.
3. At time T_V, a transaction Tx, that takes as input θ_α of Tx_G and as output $\theta := (\alpha, \mathsf{OneSig}(\mathsf{pk}_V))$, must appear on $\mathcal{L}_D$ within Δ_D rounds. If no such transaction appears within said time, output $(\mathtt{id}, \mathtt{error})$.

B.2 Glimpse Protocol

In this section we present the formal UC protocol Π of Glimpse. Π is a *hybrid* protocol with access to the functionalities $\mathcal{G}_{Clock}$, $\mathcal{G}_{GDC}$ and $\mathcal{G}_{Ledger}$. In contrast to the simplified pseudocode protocol shown in Section 4, this formal protocol includes communication with the environment and the notion of time, and it is more generic. Indeed, similar to the ideal functionality, the protocol allows for verifying logical formulas of descriptions instead of single transactions and there can be multiple different outcomes for the prover. To keep our protocol definition generic, we parameterize it over two ledgers $\mathcal{L}_S$, $\mathcal{L}_D$, Δ_D (which is explicitly stated for readability, even though it is implicitly given by $\mathcal{L}_D$), as well as over a function parameter $gen\mathcal{P}$, which should generate a proof $\mathcal{P}$ for $\mathcal{L}_D$ that a transaction has appeared on $\mathcal{L}_S$. The two ledger parameters $\mathcal{L}_S$ and $\mathcal{L}_D$ have to have the same properties as the ledger parameters of the functionality, which Π should realize. I.e., to realize $\mathcal{F}_{W-Glimpse}$, $\mathcal{L}_S$ and $\mathcal{L}_D$ have to have consistency, whereas to realize $\mathcal{F}_{S-Glimpse}$ $\mathcal{L}_D$ needs also to have liveness.

Properties of $gen\mathcal{P}$**.** The proof generation function is specific to $\mathcal{L}_S$ and $\mathcal{L}_D$ and is parameterized over a transaction Tx, a description Desc (as defined in Figure 2) and a consensus parameter cp that is specific to $\mathcal{L}_S$. The function generates a proof proving that a transaction Tx that matches description Desc, i.e., $[\mathsf{Tx}] \hookleftarrow \mathsf{Desc}$, is on $\mathcal{L}_S$, in a witness-like format that is readable by the scripting of $\mathcal{L}_D$. In our protocol instantiation, we use "Construct $\mathcal{P}_i^n$" defined in Figure 4, which uses n as consensus parameter cp. We require this function $gen\mathcal{P}$ to have the following properties: *complete* and *T-sound*.

Definition 4. A function $gen\mathcal{P}$ is *complete*, if for every transaction Tx that is on $\mathcal{L}_S$ and every description Desc, such that $[\mathsf{Tx}] \hookleftarrow \mathsf{Desc}$, it returns a proof $\mathcal{P}$ which is a witness that is accepted by $\mathcal{L}_D$.

Definition 5. A function $gen\mathcal{P}$ is *T-sound* (or T-unforgeable), if within a given time T, no proof $\mathcal{P}$ can be generated for Tx with non-negligible probability unless Tx is on $\mathcal{L}_S$.

Access to $\mathcal{L}_S$. In this protocol, if we want to achieve strong atomicity, we require both P and V to have access to $\mathcal{L}_S$ (and, of course, also $\mathcal{L}_D$). In the model, a party P having access to $\mathcal{L}_S$ means that P is an element of the set of registered parties of the functionality $\mathcal{L}_S$. In practice, it means, for example, P runs a full node. Indeed, in the pseudocode protocol in Section 4, V does not need access to $\mathcal{L}_S$. This requirement comes from the fact that we allow logical formulas (or DNFs) instead of single transactions. An intuitive example for this is $\mathcal{F}_S := \mathsf{Tx}_1 \oplus \mathsf{Tx}_2$ (xor), where V needs to prevent P from claiming the money from the Glimpse contract if both Tx_1 and Tx_2 are posted on $\mathcal{L}_S$. In a simplified case, e.g., where there is only a single transaction, this requirement can be dropped.

As explained in the main body (see Figure 1), we note that we can replace the requirement that P and V need access to $\mathcal{L}_S$ by an untrusted (i, weak atomicity) or trusted (ii, strong atomicity) relayer R, that provides the parties with the necessary data of $\mathcal{L}_S$. We model this by simply replacing the parameter $\mathcal{L}_S$ with a wrapper functionality, which can be seen as a relayer R, which provides the same interface as $\mathcal{L}_S$. R simply forwards any calls to $\mathcal{L}_S$. Similarly, the calls to $\mathcal{L}_S$ within the macro "Construct $\mathcal{P}^n$" defined in Figure 4 are replaced with calls to this functionality. The adversary $\mathcal{S}$ can replace modify responses of R to parties, that do not have access to $\mathcal{L}_S$. We allow (weak atomicity, untrusted relayer) or do not allow (strong atomicity, trusted relayer) the adversary $\mathcal{S}$ to modify responses made by this functionality. Note that the weak atomicity notion also holds when parties have no access to $\mathcal{L}_S$ at all. For the security proof, we introduce the definition of *direct access* to $\mathcal{L}$.

Definition 6. A party has *direct access to $\mathcal{L}$* if it is an element of the set of registered parties of $\mathcal{L}$ or it has access to a *trusted* relayer wrapper functionality (as defined above) of $\mathcal{L}$.

Restriction on the Environment. As we explain in Section 3.3, we need to introduce randomness to prevent upfront mining on the proof. In the general case, we need to have randomness in every description $\mathsf{Desc} \in \mathcal{F}_S$. Since $\mathcal{F}_S$ is part of the initial message to the ideal functionality and therefore also part of the initial message in Π, we put a restriction on the environment to only send an $\mathcal{F}_S$, where every $\mathsf{Desc} \in \mathcal{F}_S$ has a newly generated random value in its body. In practice, this can be achieved by P and V running a pre-setup before the protocol, where they both generate a value of length λ uniformly at random $r_P \xleftarrow{\$} \{0,1\}^\lambda$ and $r_V \xleftarrow{\$} \{0,1\}^\lambda$, concatenate them yielding $r_P || r_V$ and add an output $(0, \mathtt{OP_RETURN}(r_P || r_V))$.

Definition 7. A Glimpse protocol Π has *strictly randomized input*, if its environment is restricted in the way defined above.

Parties Exhibiting Liveness. Unfortunately, malicious parties could simply go idle and not post anything on $\mathcal{L}_D$, even though they could, in accordance to what was posted on $\mathcal{L}_S$. This behavior would violate strong atomicity. We therefore introduce the following definition. We also emphasize again, that the outcome that parties can enforce is always non-negative, so they are incentivized to enforce it.

Definition 8. Parties in a Glimpse protocol Π *exhibit liveness*, if they enforce the outcome that corresponds to the transactions on $\mathcal{L}_S$ w.r.t $\mathcal{F}_S$, if they can.

Protocol Description. The protocol proceeds in the same phases *Setup* and *Verify & Commit on* $\mathcal{L}_D$ as the ideal functionality. Because we explicitly omit modelling the issuer of the transaction(s) of $\mathcal{F}_S$ on $\mathcal{L}_S$ (this is external to the protocol, the environment does it and it can proceed in any conceivable way), we do not have the *Commit on* $\mathcal{L}_S$ phase which we show in the simplified pseudocode Figure 4.

1. *Setup:* In this phase, the parties V and P create the necessary transactions to set up a Glimpse contract corresponding to the input data they received, and post the transaction $\mathsf{Tx_G}$ carrying the Glimpse contract. In more detail, if we again consider $\mathcal{F}_S$ where the descriptions are boolean variables and the resulting truth table (as in Appendix B.1), the two parties create a transaction sequence for each of the rows that allows the respective party P or V to enforce their balance with the respective proofs. An example how such a transaction sequence looks like can be seen in Appendix D in Figure D.1, and is formally described later in the protocol.
2. *Verify & Commit on* $\mathcal{L}_D$*:* In this phase, both P and V check $\mathcal{L}_S$ to see if any transactions fulfilling a description in $\mathcal{F}_S$ has appeared there. If yes, they post the transaction sequence which corresponds to the row in the truth table, claiming their respective outcome.

To ease readability of the protocol, we take some key macros out and define them in the following box, before presenting the protocol itself. Note that there is only one protocol, depending on our assumptions, it realizes the functionality with weak or strong atomicity, as we show in Appendix C.

Macros for $\Pi(\mathcal{L}_S, \mathcal{L}_D, \Delta_D, gen\mathcal{P})$

- $\mathsf{genTxsFromF}(\alpha, \mathcal{F}_S, [\mathsf{outcome}_i]_{i\in[1,r]}, T_P, T_V, cp, n, \{\mathtt{inputs}\})$
    1. Create transaction $\mathsf{Tx_G}$, with inputs $\mathsf{Tx_G.inputs} := \{\mathtt{inputs}\}$ and outputs $\mathsf{Tx_G.outputs} := \{\theta_\alpha\} \cup \{\theta_{\varepsilon_i}\}_{i\in[1,var]}$ as list, where var is the number of Desc in $\mathcal{F}_S$, s.t. $\theta_\alpha := (\alpha, \mathsf{MuSig}(pk_P, pk_V) \vee (\mathsf{OneSig}(\mathsf{pk}_V) \wedge T_V))$ and $\theta_{\varepsilon_i} := (\varepsilon_i, (\mathsf{scriptG}(\mathsf{Desc}_i, T_P, cp, n, (P,V)) \wedge \mathsf{OneSig}(\mathsf{pk}_P))) \vee \mathsf{MuSig}(\mathsf{pk}_P, \mathsf{pk}_V))$
    2. Create a truth table for $\mathcal{F}_S$.
    3. For each row in the truth table, do the following.
        (a) Create transactions $\mathsf{Tx_T}$, $\mathsf{Tx_F}$ and $\mathsf{Tx_P}$ (see also Appendix D or Figure D.1) as follows:
        (b) $\mathsf{Tx_T}$ takes as inputs all outputs θ_{ε_i} of $\mathsf{Tx_G}$, where the corresponding input variable Desc_i is set to *true* and $\mathsf{Tx_F}$ takes as inputs all outputs θ_{ε_i} of $\mathsf{Tx_G}$, where the corresponding input variable Desc_i is set to *false*.
        (c) The single output of $\mathsf{Tx_T}$ is $\theta := (\varepsilon, \mathsf{OneSig}(\mathsf{pk}_P)$ and the single output of $\mathsf{Tx_F}$ is $\theta := (\varepsilon, \mathsf{OneSig}(\mathsf{pk}_P) \wedge T_P \vee truevar)$, where $truevar$ is a disjunction of $\mathsf{scriptG}(\mathsf{Desc}_i, T_P, cp, n, (P,V))$ for each input variable Desc_i of the truth table that is set to *true* for this row.
        (d) Finally, $\mathsf{Tx_P}$ takes as inputs θ_α of $\mathsf{Tx_G}$ as well as both outputs of $\mathsf{Tx_T}$ and $\mathsf{Tx_F}$. Its output is $\theta := (\mathsf{outcome}_i.P, \mathsf{OneSig}(\mathsf{pk}_P))$.
    4. We define the result of this as set of tuples $\{(\mathsf{Tx_T}_i, \mathsf{Tx_F}_i, \mathsf{Tx_P}_i)\}_{i\in[1,row]}$, where row is the number of rows in the truth table.
    5. Return $(\mathsf{Tx_G}, \{(\mathsf{Tx_T}_i, \mathsf{Tx_F}_i, \mathsf{Tx_P}_i)\}_{i\in[1,row]})$
- $\mathsf{postTxsFP}(\mathcal{F}_S, T_P, n, \mathsf{Tx_G}, \{(\mathsf{Tx_T}_i, \mathsf{Tx_F}_i, \mathsf{Tx_P}_i, \sigma_V(\mathsf{Tx_F}_i), \sigma_V(\mathsf{Tx_P}_i))\}_{i\in[1,row]})$
    1. If current time is $T_P - \Delta_D$ and output θ_α of $\mathsf{Tx_G}$ is unspent, check if there exist transactions $\{\mathsf{Tx}_i\}$ on $\mathcal{L}_S$, such that for each $\mathsf{Tx}_i \in \{\mathsf{Tx}_i\}$ there exists exactly one $\mathsf{Desc} \in \mathcal{F}_S$, s.t. $[\mathsf{Tx}_i] \hookleftarrow \mathsf{Desc}$.
    2. Looking at the truth table for statement, consider the row k where exactly the description for which Tx_i is on $\mathcal{L}_S$ are marked as *true*.
    3. Extract the corresponding tuple for row k out of the set sent in the parameters of this function, i.e., $(\mathsf{Tx_T}_k, \mathsf{Tx_F}_k, \mathsf{Tx_P}_k, \sigma_V(\mathsf{Tx_F}_k), \sigma_V(\mathsf{Tx_P}_k))$
    4. For each Tx_i and Tx_i where Tx_i is on $\mathcal{L}_S$, construct a proof using the Construct $\mathcal{P}^n_i(\mathsf{Tx}_i, \mathsf{Desc}_i, n)$ function defined in Figure 4, yielding a set of proofs $\{\mathcal{P}^n\}$.
    5. Generate signatures $\sigma_P(\mathsf{Tx_T}_k)$, $\sigma_P(\mathsf{Tx_F}_k)$, $\sigma_P(\mathsf{Tx_P}_k)$.
    6. Send a message "post" for transaction $\mathsf{Tx_T}_k$ with $\sigma_P(\mathsf{Tx_T}_k)$ and $\{\mathcal{P}^n\}$ as witnesses to functionality $\mathcal{L}_D$.
    7. Send a message "post" for transaction $\mathsf{Tx_F}_k$ with $\sigma_V(\mathsf{Tx_F}_k)$ and $\sigma_P(\mathsf{Tx_F}_k)$ as witnesses to functionality $\mathcal{L}_D$.
    8. At time T_P, send a message "post" for transaction $\mathsf{Tx_P}_k$ with $\sigma_V(\mathsf{Tx_P}_k)$ and $\sigma_P(\mathsf{Tx_P}_k)$ to $\mathcal{L}_D$.
- $\mathsf{postTxsFV}(\mathcal{F}_S, T_P, T_V, n, P, V, [\mathsf{outcome}_i]_{i\in[1,r]}, \mathsf{Tx_G})$
    1. If current time is after T_V and output θ_α of $\mathsf{Tx_G}$ is unspent, send a "post" message for a transaction Tx to $\mathcal{L}_D$, where Tx takes as input θ_α of $\mathsf{Tx_G}$ and as output $\theta := (\alpha, \mathsf{OneSig}(\mathsf{pk}_V))$, generate and use signature $\sigma_V(\mathsf{Tx})$ as a witness.
    2. Else, if current time is before T_P and a transaction $\mathsf{Tx_F}$ (as defined in step 3a of $\mathsf{genTxsFromF}$) is on $\mathcal{L}_S$ and a transaction Tx' is on $\mathcal{L}_S$, s.t. it fulfills one of the descriptions of the output of $\mathsf{Tx_F}$, i.e., $[\mathsf{Tx}'] \hookleftarrow \mathsf{Desc}$, do the following.
        (a) Construct a proof using the Construct $\mathcal{P}^n(\mathsf{Tx}', \mathsf{Desc}, n)$ function defined in Figure 4, yielding $\mathcal{P}^n$.
        (b) Send a message "post" for a transaction Tx'' to $\mathcal{L}_D$, where Tx'' takes as input the output of $\mathsf{Tx_F}$ and as output $\theta := (\alpha, \mathsf{OneSig}(\mathsf{pk}_V))$, using $\mathcal{P}^n$ and $\sigma_V(\mathsf{Tx}'')$ as witnesses.

Protocol $\Pi(\mathcal{L}_S, \mathcal{L}_D, \Delta_D, gen\mathcal{P})$

Parameters:
$\mathcal{L}_S$, $\mathcal{L}_D$... two instances of $\mathcal{G}_{Ledger}$, representing the source and destination blockchain. We let cp be the difficulty of $\mathcal{L}_D$, i.e., this is a parameter of the functionality $\mathcal{G}_{Ledger}$.
$\Delta_D \in \mathbb{N}$ ... the blockchain delay of $\mathcal{L}_D$, i.e., the upper bound on the time it takes from posting a valid transaction Tx to Tx appearing on the the ledger.
$gen\mathcal{P}$ ... a function that takes as input a transaction Tx, a description Desc (as defined in Figure 2) and a consensus parameter cp that is specific to $\mathcal{L}_S$. The function generates a proof that a transaction Tx that matches description Desc, i.e., $[\mathsf{Tx}] \hookleftarrow \mathsf{Desc}$, is on $\mathcal{L}_S$, as a witness that is readable by the scripting of $\mathcal{L}_D$. In our protocol instantiation, we use "Construct $\mathcal{P}_i^{n}$" defined in Figure 4, which uses n as consensus parameter cp.

Variables:

Φ_P ... a set of tuples $(\mathtt{id}, \mathcal{F}_S, T_P, T_V, n, \mathsf{Tx_G}, \{(\mathsf{Tx_T}_i, \mathsf{Tx_F}_i, \mathsf{Tx_P}_i \sigma_V(\mathsf{Tx_F}_i), \sigma_V(\mathsf{Tx_P}_i))\}_{i\in[1,row]})$, where $\mathtt{id} \in \{0,1\}^*$ is an identifier unique to the pair P and V. P and V are in turn both distinct elements of the set of all users $\mathcal{U}$. $\mathcal{F}_S$ is a logical formula as defined in Figure 2. Further, $T_P, T_V, n \in \mathbb{N}$, and $[\mathsf{outcome}_i]$ is a list of outcomes, which in turn are tuples $(\mathsf{outcome}.P, \mathsf{outcome}.V) \in \mathbb{N}^2$. $\mathsf{Tx_G}$ is a transaction and $\{(\mathsf{Tx_T}_i, \mathsf{Tx_F}_i, \mathsf{Tx_P}_i \sigma_V(\mathsf{Tx_F}_i), \sigma_V(\mathsf{Tx_P}_i))\}_{i\in[1,row]}$ is a set of tuples containing transactions and signatures.
Φ_V ... a set of tuples $(\mathtt{id}, \mathcal{F}_S, T_P, T_V, n, \mathsf{Tx_G})$, where $\mathtt{id} \in \{0,1\}^*$ is an identifier unique to the pair P and V. P and V are in turn both distinct elements of the set of all users $\mathcal{U}$. $\mathcal{F}_S$ is a logical formula as defined in Figure 2. Further, $T_P, T_V, n \in \mathbb{N}$. $\mathsf{Tx_G}$ is a transaction.

Setup

Verifier V

1. Upon $(\mathtt{SETUP}, \mathtt{id}, \mathcal{F}_S, P, [\mathsf{outcome}_i]_{i\in[1,r]}, T_P, T_V, n, \{inputP\}, \{inputV\}) \overset{\tau_V}{\hookleftarrow} \mathcal{Z}$, check that the following holds:
   (a) $[\mathsf{outcome}_i]$ is a list of $r := 2^d$ outcomes, where d is the number of descriptions in $\mathcal{F}_S$ (in other words, the number of rows in the truth table when considering all descriptions in $\mathcal{F}_S$)
   (b) For all rows i of the truth table generated by $\mathcal{F}_S$, where the result is *false*, it must hold that $\mathsf{outcome}_i := (0, \alpha)$.
   (c) For each $\mathsf{outcome}_i$ it must hold that $\mathsf{outcome}_i.P + \mathsf{outcome}_i.V = \alpha$ for some number α
   (d) $T_V > T_P$ are both times in the future
   (e) $n \in \mathbb{N}$
   (f) $\{inputV\}$ is a (potentially empty) set of inputs under control of V and $\{inputP\}$ is a (potentially empty) set of inputs under control of P
   (g) $|\{inputV\} \cup \{inputP\}| > 0$ and the sum of coins stored in $\{inputV\} \cup \{inputP\} \geq \alpha + d \cdot \varepsilon$
   (h) If these checks hold continue, else go idle.
2. $(\mathsf{Tx_G}, \{(\mathsf{Tx_T}_i, \mathsf{Tx_F}_i, \mathsf{Tx_P}_i)\}_{i\in[1,row]}) := \mathsf{genTxsFromF}(\alpha, \mathcal{F}_S, [\mathsf{outcome}_i]_{i\in[1,r]}, T_P, T_V, cp, n, \{\mathtt{inputs}\})$
3. Sign each transaction $\mathsf{Tx_F}_i$ and $\mathsf{Tx_P}_i$ in $\{(\mathsf{Tx_T}_i, \mathsf{Tx_F}_i, \mathsf{Tx_P}_i)\}_{i\in[1,row]}$ and append the signatures to each tuple yielding a set $\{(\mathsf{Tx_T}_i, \mathsf{Tx_F}_i, \mathsf{Tx_P}_i, \sigma_V(\mathsf{Tx_F}_i), \sigma_V(\mathsf{Tx_P}_i))\}_{i\in[1,row]}$
4. Sign $\mathsf{Tx_G}$ yielding $\sigma_V(\mathsf{Tx_G})$
5. Send $(\mathtt{open\text{-}req}, \mathtt{id}, [\mathsf{outcome}_i]_{i\in[1,r]}, T_P, T_V, n, \{inputP\}, \{inputV\}, \mathsf{Tx_G}, \sigma_V(\mathsf{Tx_G}), \{(\mathsf{Tx_T}_i, \mathsf{Tx_F}_i, \mathsf{Tx_P}_i, \sigma_V(\mathsf{Tx_F}_i), \sigma_V(\mathsf{Tx_P}_i))\}_{i\in[1,row]}) \overset{\tau_V}{\hookrightarrow} P$.
6. Add $(\mathtt{id}, \mathcal{F}_S, T_P, T_V, n, \mathsf{Tx_G})$ to Φ_V.

Prover P

7. Upon $(\mathtt{open\text{-}req}, \mathtt{id}, [\mathsf{outcome}_i]_{i\in[1,r]}, T_P, T_V, n, \{inputP\}, \{inputV\}, \mathsf{Tx_G}, \sigma_V(\mathsf{Tx_G}), \{(\mathsf{Tx_T}_i, \mathsf{Tx_F}_i, \mathsf{Tx_P}_i, \sigma_V(\mathsf{Tx_F}_i), \sigma_V(\mathsf{Tx_P}_i))\}_{i\in[1,row]}) \overset{\tau_P}{\hookleftarrow} V$.
8. Perform checks of protocol Setup phase, steps 1a through 1h. If one or more fail, go idle.
9. Verify that $(\mathsf{Tx_G}, \{(\mathsf{Tx_T}_i, \mathsf{Tx_F}_i, \mathsf{Tx_P}_i)\}_{i\in[1,row]})$ is the result of $\mathsf{genTxsFromF}(\alpha, \mathcal{F}_S, [\mathsf{outcome}_i]_{i\in[1,r]}, T_P, T_V, cp, n, \{\mathtt{inputs}\})$. If not, go idle.
10. For each entry of the set $\{(\mathsf{Tx_T}_i, \mathsf{Tx_F}_i, \mathsf{Tx_P}_i, \sigma_V(\mathsf{Tx_F}_i), \sigma_V(\mathsf{Tx_P}_i))\}_{i\in[1,row]}$, check that $\sigma_V(\mathsf{Tx_F}_i)$ and $\sigma_V(\mathsf{Tx_P}_i)$ are valid signatures of V for $\mathsf{Tx_F}_i$ and $\mathsf{Tx_P}_i$, respectively.
11. Verify that $\sigma_V(\mathsf{Tx_G})$ is a valid signature of V for $\mathsf{Tx_G}$.
12. $(\mathtt{id}, \mathcal{F}_S, [\mathsf{outcome}_i]_{i\in[1,r]}, T_P, T_V, n, \{inputP\}, \{inputV\}) \overset{\tau_P}{\hookrightarrow} \mathcal{Z}$.
13. Upon $(\mathtt{id}) \overset{\tau_P}{\hookleftarrow} \mathcal{Z}$, sign $\mathsf{Tx_G}$ yielding $\sigma_P(\mathsf{Tx_G})$.
14. Send a message "post" for transaction $\mathsf{Tx_G}$ with $\sigma_P(\mathsf{Tx_G})$ and $\sigma_V(\mathsf{Tx_G})$ as witnesses to functionality $\mathcal{L}_D$
15. If it appears on the ledger of $\mathcal{L}_D$ at round $\tau_{P1} \leq \tau_P + 1 + \Delta_D$, add $(\mathtt{id}, \mathcal{F}_S, T_P, T_V, n, \mathsf{Tx_G}, \{(\mathsf{Tx_T}_i, \mathsf{Tx_F}_i, \mathsf{Tx_P}_i \sigma_V(\mathsf{Tx_F}_i), \sigma_V(\mathsf{Tx_P}_i))\}_{i\in[1,row]})$ in Φ_P.

Verify & Commit on $\mathcal{L}_D$: P (in every round τ)

For every $(\mathtt{id}, \mathcal{F}_S, T_P, T_V, n, \mathsf{Tx_G}, \{(\mathsf{Tx_T}_i, \mathsf{Tx_F}_i, \mathsf{Tx_P}_i, \sigma_V(\mathsf{Tx_F}_i), \sigma_V(\mathsf{Tx_P}_i))\}_{i\in[1,row]})$ in Φ_P, execute $\mathsf{postTxsFP}(\mathcal{F}_S, T_P, n, \mathsf{Tx_G}, \{(\mathsf{Tx_T}_i, \mathsf{Tx_F}_i, \mathsf{Tx_P}_i, \sigma_V(\mathsf{Tx_F}_i), \sigma_V(\mathsf{Tx_P}_i))\}_{i\in[1,row]})$.

Verify & Commit on $\mathcal{L}_D$: V (in every round τ)

For every $(\mathtt{id}, \mathcal{F}_S, T_P, T_V, n, \mathsf{Tx_G})$ in Φ_V, execute $\mathsf{postTxsFV}(\theta_P, \theta_V, \mathcal{F}_S, cp, n, T)$.

C Security Proof

In this section, we prove Theorems 1 and 2. We provide the code for the ideal world adversary, the *simulator*, $\mathcal{S}$. The main challenge is for the simulator to provide a simulated transcript that is computationally indistinguishable for the environment $\mathcal{Z}$ from the transcript generated by the real protocol execution. We remark that as with the protocol, there is a single simulator for both $\mathcal{F}_{W-Glimpse}$ and $\mathcal{F}_{S-Glimpse}$, and the difference comes from the assumptions we show below. The following

properties refer to Definitions 4 to 8 in Appendix B.2.

- **Necessity of *genP* being *T-sound* (for Theorems 1 and 2).** Without this property, the environment can simply forge a proof for Tx with non-negligible probability before T expires, without Tx being on $\mathcal{L}_S$. Using this forged proof, they can proceed violate *weak atomicity*, e.g., for example by posting transaction Tx_T even though the corresponding transaction Tx is not on $\mathcal{L}_S$. More formally, this can be shown by a trivial reduction: Assume *weak atomicity* does not hold, we can use the witness in $\mathcal{L}_D$ to extract the proof before T, even though there is no corresponding Tx on $\mathcal{L}_S$. We discuss in Section 6.1 under which conditions the proof generation function "Construct $\mathcal{P}_i^n$" defined in Figure 4 is *T-sound*.
- **Necessity of *strictly randomized input* (for Theorems 1 and 2).** As explained in Appendix B.2 and Section 3.3, without this property the environment has more time than the time from the protocol start until the T. Effectively, with more time than T the environment can potentially forge a proof with non-negligible probability, which leads to similar problems than with *T-soundness* violations.
- **Necessity of parties having *direct access to $\mathcal{L}_S$ and $\mathcal{L}_D$* (for Theorem 2).** Obviously, without direct access to $\mathcal{L}_D$ parties cannot post their transaction to enforce their outcome. However, they also need direct access to $\mathcal{L}_S$, in order to identify if transactions have been posted and to query the necessary information to generate a proof $\mathcal{P}$.
- **Necessity of *genP* being *complete* (for Theorem 2).** Similarly, we require *genP* to be complete, otherwise, there might be a case where even though parties have access to the information on $\mathcal{L}_S$ and a transaction Tx has appeared there, they cannot construct a proof.
- **Necessity of parties *exhibiting liveness* (for Theorem 2).** To achieve strong atomicity, we require parties to exhibit liveness. Indeed, if parties do not post the corresponding transactions on $\mathcal{L}_D$ according to what was posted on $\mathcal{L}_S$, and instead go idle, strong atomicity does not hold. However, as we already argued, every enforceable outcome on $\mathcal{L}_D$ is non-negative for both P and V, so parties who are incentivized to always enforce their correct balance rather than not posting anything.

On a high level, the simulator's job is to keep track of the transactions and witnesses necessary to post the according transactions at the correct moment, which ensures that the execution transcript is the same as in the real world. More formally, the code for the simulator follows.

Simulator for *Setup* phase

a) Case P is honest, V dishonest

1. Upon V sending $(\texttt{open-req}, \texttt{id}, [\mathsf{outcome}_i]_{i\in[1,r]}, T_P, T_V, n, \{inputP\}, \{inputV\}, \mathsf{Tx_G}, \sigma_V(\mathsf{Tx_G}), \{(\mathsf{Tx_T}_i, \mathsf{Tx_F}_i, \mathsf{Tx_P}_i, \sigma_V(\mathsf{Tx_F}_i), \sigma_V(\mathsf{Tx_P}_i))\}_{i\in[1,row]}) \overset{\tau_V}{\hookrightarrow} P$, send $(\texttt{OK}, \texttt{id}, P, [\mathsf{outcome}_i]_{i\in[1,r]}, T_P, T_V, n, \{inputP\}, \{inputV\}) \overset{\tau_V}{\hookrightarrow} \mathcal{F}_{Glimpse}$, If not, go idle.
2. For each entry of the set $\{(\mathsf{Tx_T}_i, \mathsf{Tx_F}_i, \mathsf{Tx_P}_i, \sigma_V(\mathsf{Tx_F}_i), \sigma_V(\mathsf{Tx_P}_i))\}_{i\in[1,row]}$, check that $\sigma_V(\mathsf{Tx_F}_i)$ and $\sigma_V(\mathsf{Tx_P}_i)$ are valid signatures of V for $\mathsf{Tx_F}_i$ and $\mathsf{Tx_P}_i$, respectively.
3. Verify that $\sigma_V(\mathsf{Tx_G})$ is a valid signature of V for $\mathsf{Tx_G}$.
4. Upon $(\texttt{id}) \overset{\tau_P}{\hookleftarrow} P$, sign $\mathsf{Tx_G}$ on P's behalf yielding $\sigma_P(\mathsf{Tx_G})$.
5. Send a message "post" for transaction $\mathsf{Tx_G}$ with $\sigma_P(\mathsf{Tx_G})$ and $\sigma_V(\mathsf{Tx_G})$ as witnesses to functionality $\mathcal{L}_D$
6. If it appears on the ledger of $\mathcal{L}_D$ at round $\tau_{P1} \leq \tau_P + 1 + \Delta_D$, let $\Phi(P) := (\texttt{id}, \mathcal{F}_S, T_P, T_V, n, \mathsf{Tx_G}, \{(\mathsf{Tx_T}_i, \mathsf{Tx_F}_i, \mathsf{Tx_P}_i \sigma_V(\mathsf{Tx_F}_i), \sigma_V(\mathsf{Tx_P}_i))\}_{i\in[1,row]})$.

b) Case P is dishonest, V honest

1. Upon V sending $(\texttt{SETUP}, \texttt{id}, \mathcal{F}_S, P, [\mathsf{outcome}_i]_{i\in[1,r]}, T_P, T_V, n, \{inputP\}, \{inputV\}) \overset{\tau_V}{\hookrightarrow} \mathcal{F}_{Glimpse}$, perform checks of protocol Setup phase, steps 1a through 1h. If one or more fail, go idle.
2. $(\mathsf{Tx_G}, \{(\mathsf{Tx_T}_i, \mathsf{Tx_F}_i, \mathsf{Tx_P}_i)\}_{i\in[1,row]}) := \mathsf{genTxsFromF}(\alpha, \mathcal{F}_S, [\mathsf{outcome}_i]_{i\in[1,r]}, T_P, T_V, cp, n, \{\texttt{inputs}\})$
3. Sign each transaction $\mathsf{Tx_F}_i$ and $\mathsf{Tx_P}_i$ in $\{(\mathsf{Tx_T}_i, \mathsf{Tx_F}_i, \mathsf{Tx_P}_i)\}_{i\in[1,row]}$ on behalf of V and append the signatures to each tuple yielding a set $\{(\mathsf{Tx_T}_i, \mathsf{Tx_F}_i, \mathsf{Tx_P}_i, \sigma_V(\mathsf{Tx_F}_i), \sigma_V(\mathsf{Tx_P}_i))\}_{i\in[1,row]}$
4. Sign $\mathsf{Tx_G}$ yielding $\sigma_V(\mathsf{Tx_G})$
5. Send $(\texttt{open-req}, \texttt{id}, [\mathsf{outcome}_i]_{i\in[1,r]}, T_P, T_V, n, \{inputP\}, \{inputV\}, \mathsf{Tx_G}, \sigma_V(\mathsf{Tx_G}), \{(\mathsf{Tx_T}_i, \mathsf{Tx_F}_i, \mathsf{Tx_P}_i, \sigma_V(\mathsf{Tx_F}_i), \sigma_V(\mathsf{Tx_P}_i))\}_{i\in[1,row]}) \overset{\tau_V}{\hookrightarrow} P$.
6. Let $\Phi(V) := (\texttt{id}, \mathcal{F}_S, T_P, T_V, n, \mathsf{Tx_G})$.

c) Case P is honest, V honest

1. Upon V sending $(\texttt{OK}, \texttt{id}, P, [\mathsf{outcome}_i]_{i\in[1,r]}, T_P, T_V, n, \{inputP\}, \{inputV\}) \overset{\tau_V}{\hookrightarrow} \mathcal{F}_{Glimpse}$, perform checks of protocol Setup phase, steps 1a through 1h. If one or more fail, go idle.
2. $(\mathsf{Tx_G}, \{(\mathsf{Tx_T}_i, \mathsf{Tx_F}_i, \mathsf{Tx_P}_i)\}_{i\in[1,row]}) := \mathsf{genTxsFromF}(\alpha, \mathcal{F}_S, [\mathsf{outcome}_i]_{i\in[1,r]}, T_P, T_V, cp, n, \{\texttt{inputs}\})$
3. Sign each transaction $\mathsf{Tx_F}_i$ and $\mathsf{Tx_P}_i$ in $\{(\mathsf{Tx_T}_i, \mathsf{Tx_F}_i, \mathsf{Tx_P}_i)\}_{i\in[1,row]}$ on behalf of V and append the signatures to each tuple yielding a set $\{(\mathsf{Tx_T}_i, \mathsf{Tx_F}_i, \mathsf{Tx_P}_i, \sigma_V(\mathsf{Tx_F}_i), \sigma_V(\mathsf{Tx_P}_i))\}_{i\in[1,row]}$
4. Sign $\mathsf{Tx_G}$ on behalf of V yielding $\sigma_V(\mathsf{Tx_G})$.
5. Let $\Phi(V) := (\texttt{id}, \mathcal{F}_S, T_P, T_V, n, \mathsf{Tx_G})$.
6. Sign $\mathsf{Tx_G}$ on behalf of P yielding $\sigma_P(\mathsf{Tx_G})$.
7. Send a message "post" for transaction $\mathsf{Tx_G}$ with $\sigma_P(\mathsf{Tx_G})$ and $\sigma_V(\mathsf{Tx_G})$ as witnesses to functionality $\mathcal{L}_D$
8. If it appears on the ledger of $\mathcal{L}_D$ at round $\tau_{P1} \leq \tau_P + 1 + \Delta_D$, let $\Phi(P) := (\texttt{id}, \mathcal{F}_S, T_P, T_V, n, \mathsf{Tx_G}, \{(\mathsf{Tx_T}_i, \mathsf{Tx_F}_i, \mathsf{Tx_P}_i \sigma_V(\mathsf{Tx_F}_i), \sigma_V(\mathsf{Tx_P}_i))\}_{i\in[1,row]})$.

Simulator for *Verify & Commit on* $\mathcal{L}_D$: P phase

P is honest

For every (key, value) pair P, $(\texttt{id}, \mathcal{F}_S, T_P, T_V, n, \mathsf{Tx_G}, \{(\mathsf{Tx_T}_i, \mathsf{Tx_F}_i, \mathsf{Tx_P}_i, \sigma_V(\mathsf{Tx_F}_i), \sigma_V(\mathsf{Tx_P}_i))\}_{i\in[1,row]})$ in Φ, execute $\mathsf{postTxsFP}(\mathcal{F}_S, T_P, n, \mathsf{Tx_G}, \{(\mathsf{Tx_T}_i, \mathsf{Tx_F}_i, \mathsf{Tx_P}_i, \sigma_V(\mathsf{Tx_F}_i), \sigma_V(\mathsf{Tx_P}_i))\}_{i\in[1,row]})$ on behalf of P.

Simulator for *Verify & Commit on* $\mathcal{L}_D$: V phase

P is honest

For every (key, value) pair V, $(\texttt{id}, \mathcal{F}_S, T_P, T_V, n, \mathsf{Tx_G})$ in Φ, execute $\mathsf{postTxsFV}(\theta_P, \theta_V, \mathcal{F}_S, cp, n, T)$ on behalf of V.

D DNFs with Glimpse

Glimpse can efficiently encode *Disjunctive Normal Forms (DNFs) over descriptions* (Figure 2) and use them to encode synchronization of transaction combination between a source $\mathcal{L}_S$ and a destination ledger $\mathcal{L}_D$. DNFs express truth tables and logical formulas in terms of *disjunctions of conjunctions of one or more descriptions* (in the context of DNFs, the descriptions are commonly referred to as literals).

Let us consider two oracles O_1 and O_2 operating on $\mathcal{L}_S$ (they could also be deployed on different ledgers) and regularly posting information about a real-world event. On $\mathcal{L}_D$, prover P and verifier V, e.g., bet on a specific outcome of the event by locking $\frac{\alpha}{2}$ coins each in the Glimpse transaction $\mathsf{Tx_G}$ and by conditioning the spendability of the coins to a specific outcome being attested by at least one-out-of-two oracles. If either O_1 or O_2 attests the desired outcome for the event, P can get the α coins by providing a valid proof $\mathcal{P}^n$; otherwise, if the coins are still unspent after T, V can redeem them. We recall that the selected outcome for the event is specified within the descriptions and hard coded within $\mathsf{Tx_G}$. We let Desc_1 and Desc_2 be descriptions for O_1 and O_2, respectively, and we let the to-be-verified DNF formula be $\mathcal{F}_S = (\mathsf{Desc}_1 \wedge \neg\mathsf{Desc}_2) \vee (\neg\mathsf{Desc}_1 \wedge \mathsf{Desc}_2) \vee (\mathsf{Desc}_1 \wedge \mathsf{Desc}_2)$. The Glimpse protocol proceeds as follows.

Setup. Let θ_P and θ_V be unspent outputs on $\mathcal{L}_D$ holding $\frac{\alpha}{2}$ coins each and controlled by P and V, respectively. We let $\alpha := \theta_P.\mathsf{coins} + \theta_V.\mathsf{coins}$ be the value locked in Glimpse and we denote with ζ_P and ζ_V the inputs spent by θ_P and θ_V, respectively. The parties construct $[\mathsf{Tx}_G] := (2, [\zeta_P, \zeta_V], 3, [\theta_\alpha, \theta_{\varepsilon_1}, \theta_{\varepsilon_2}])$, such that $\theta_\alpha := (\alpha, (\mathsf{MuSig}(\mathsf{pk}_P, \mathsf{pk}_V)) \vee (\mathsf{OneSig}(\mathsf{pk}_V) \wedge T_3))$, $\theta_{\varepsilon_1} := (\varepsilon, (\mathsf{scriptG}(\mathsf{Desc}_1, T_1, \mathcal{T}_S, n_1, P)) \vee \mathsf{MuSig}(\mathsf{pk}_P, \mathsf{pk}_V))$, and $\theta_{\varepsilon_2} := (\varepsilon, (\mathsf{scriptG}(\mathsf{Desc}_2, T_1, \mathcal{T}_S, n_2, P)) \vee \mathsf{MuSig}(\mathsf{pk}_P, \mathsf{pk}_V))$.

When encoding DNFs, $\mathsf{Tx_G}$ has as many outputs holding a negligible amount as the number of literals in the DNF (e.g., in this case we have two: $\theta_{\varepsilon_1}, \theta_{\varepsilon_2}$), plus an additional one holding the Glimpse value. As opposed to the case with a single transaction, we now require P to also sample a random string and include it within each descriptions. Then, on $\mathcal{L}_D$, *for each term in $\mathcal{F}_S$'s disjunction*, the parties need to create the set of transactions $(\mathsf{Tx_T}, \mathsf{Tx_F}, \mathsf{Tx_P})_i$, where:

- $\mathsf{Tx_T}$ allows P to prove the inclusion of the oracle's transaction which attests the desired outcome for the event. $\mathsf{Tx_T}$ takes as inputs the outputs θ_{ε_i} whose scripts verify the descriptions which are not negated in the term, and has a single output $\theta := (\varepsilon, \mathsf{OneSig}(\mathsf{pk}_P))$.
- $\mathsf{Tx_F}$ protects V from P falsely claiming some transaction was not published. $\mathsf{Tx_F}$ takes as inputs the outputs θ_{ε_i} whose scripts verify the descriptions which are negated in the term. $\mathsf{Tx_F}$ has as many outputs as the number of inputs, each one of the form $\theta_i := (\varepsilon, (\mathsf{scriptG}(\mathsf{Desc}_i, T_2, \mathcal{T}_S, n_i, (P, V))$. $\mathsf{Tx_F}$ allows V to react to P's false claim by submitting a proof within T_2, thereby proving i-th transaction inclusion and spending θ_i. For some terms of the disjunction, $\mathsf{Tx_F}$ is not needed, e.g., $(\mathsf{Desc}_1 \wedge \mathsf{Desc}_2)$.
- $\mathsf{Tx_P}$ takes as inputs the output θ of $\mathsf{Tx_T}$, all the outputs θ_i of $\mathsf{Tx_F}$, and the output θ_α of $\mathsf{Tx_G}$. If all the outputs of $\mathsf{Tx_F}$ are still unspent, $\mathsf{Tx_P}$ allows P to get the Glimpse coins.

Finally, parties create one single transaction $\mathsf{Tx_V}$ that allows V to spend the output θ_α. Figure D.1 shows an example of transaction set $(\mathsf{Tx_G}, (\mathsf{Tx_T}, \mathsf{Tx_F}, \mathsf{Tx_P})_i, \mathsf{Tx_V})$ for the term $(\mathsf{Desc}_1 \wedge \neg\mathsf{Desc}_2)$ of $\mathcal{F}_S$.

Then, P signs $[\mathsf{Tx_V}]$ and $([\mathsf{Tx_F}])_{\forall i}$, and sends to V $(\zeta_P, \zeta_V, \mathsf{Desc}_1, \mathsf{Desc}_2, T_1, T_2, \mathcal{T}_S, n_1, n_2, \alpha, \mathsf{scriptG}, [\mathsf{Tx}_G], ([\mathsf{Tx_T}], [\mathsf{Tx_F}], [\mathsf{Tx_P}])_{\forall i}, [\mathsf{Tx_V}], \sigma_P([\mathsf{Tx_V}]), \sigma_P([\mathsf{Tx_F}])_{\forall i})$. Upon receiving the message, if V is interested in opening a Glimpse instance with P at the received parameters, after checking correctness of the parameters and well-formedness of transactions, V signs $\mathsf{Tx_G}$, $(\mathsf{Tx_F})_{\forall i}$ and $(\mathsf{Tx_P})_{\forall i}$, and sends the signatures to P.

Upon receiving $(\sigma_V([\mathsf{Tx_G}]), \sigma_V([\mathsf{Tx_F}])_{\forall i}, \sigma_V([\mathsf{Tx_P}])_{\forall i})$ from V, P checks if the signatures are valid. If so, P publishes $\mathsf{Tx_G}$ on $\mathcal{L}_D$, while both parties locally store the tuples $(\mathsf{Tx_T}, \mathsf{Tx_F}, \mathsf{Tx_P})_{\forall i}$ and the signatures.

Commit on $\mathcal{L}_S$. The oracles publish transactions attesting the outcome of the selected real-world event.

Verify & Commit on $\mathcal{L}_D$. P and V monitor $\mathcal{L}_S$ (or make use of a trusted relayer) checking for the inclusion of transactions matching descriptions Desc_1 and Desc_2. P constructs the proofs for the oracles' transactions attesting the event he bet on, and get the Glimpse coins by posting $\mathsf{Tx_T}_i$, $\mathsf{Tx_F}_i$, and $\mathsf{Tx_P}_i$ corresponding to the term occurred. E.g., if both oracles published the event outcome P bet on, then P will post the set of transaction corresponding to the term $(\mathsf{Desc}_1 \wedge \mathsf{Desc}_2)$ in the formula.

V checks whether the set of transactions published by P corresponds to the correct term of $\mathcal{F}_S$ realized by the oracles. If V detects any misbehavior from P, can react within T_2 and spend one of $\mathsf{Tx_F}$'s outputs, thereby invalidating $\mathsf{Tx_P}_i$. V can then publish $\mathsf{Tx_V}$ and get the Glimpse coins after T_3. We note that $T_1 < T_2 < T_3$.

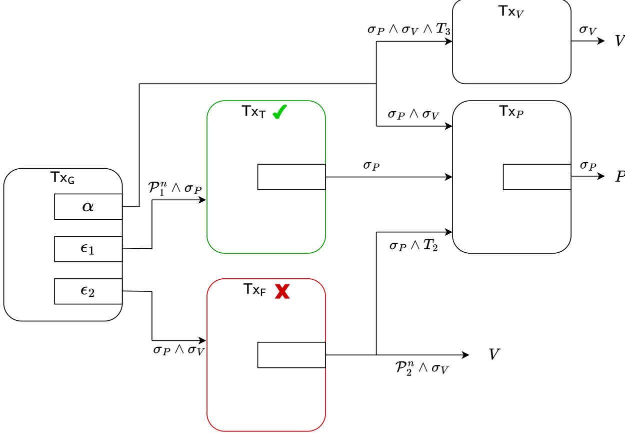

Figure D.1: Set $(\mathsf{Tx}_\mathsf{G}, (\mathsf{Tx}_\mathsf{T}, \mathsf{Tx}_\mathsf{F}, \mathsf{Tx}_\mathsf{P})_i, \mathsf{Tx}_\mathsf{V})$ of transactions to be constructed for verifying a DNF with two literals. The arrows refer to the term $i = (\mathsf{Desc}_1 \wedge \neg \mathsf{Desc}_2)$ of $\mathcal{F}_S$.

Remarks. We stress that parties can set a specific fund distribution for multiple different outcomes for P. DNF verification with Glimpse requires V to construct and submit a proof $\mathcal{P}^n$ in case P is cheats: This means that V needs to interact with R to obtain $\mathcal{L}_S$'s data (or to run a full node).

In general, although increasing the off-chain communication overhead, this construction results in up to three on-chain transactions in the optimistic case, regardless of the complexity of the DNF to verify.

E Glimpse Script example for Bitcoin-like chains

We present examples for the Glimpse locking and unlocking scripts used in Glimpse for Bitcoin-based source and destination chains. In particular, we construct the script for Liquid, where we have the Taproot optimization and the necessary opcodes for concatenating strings as well as comparing hashes. Finally, we show how to cope with the lack of Taproot by discussing Glimpse for Bitcoin Cash.

For Liquid, we have the following setting: Taproot is enabled (granting access to the MAST functionality), the opcodes `OP_CAT` as well as `OP_SUBSTR` are active. We point the reader to [56, 57] for a high-level description of how locking and unlocking scripts work and for the Bitcoin-like chains transaction format.

Example. To ease readability, we consider the simple case where P publishes Tx_P with witness $\mathcal{P}^{n=0}$ on Liquid as a result of Tx being published on Bitcoin. The Tx_G in Liquid hard codes the description $\mathsf{Desc} = (1, [(x_1)], 1, [(\theta)])$ having a variable input. Assume Tx being part of block B, accommodates 4 transactions in total, as in Figure E.1.

Scripts. The Bitcoin scripting language is stack-based and only comprises two types of values: *opcodes*, i.e., the instructions, and *data*, e.g., public keys, signatures, hashes. It processes instructions sequentially, meaning the locking and

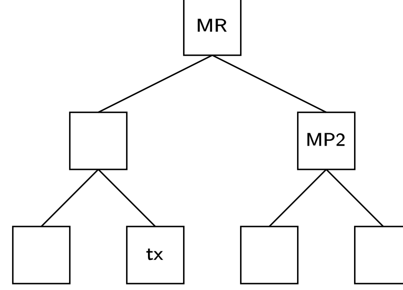

Figure E.1: Merkle tree of four transactions. The two elements MP1 and MP2 we need for reconstructing the Merkle root are marked, along with Tx.

unlocking scripts execute one after the other. If the whole computation ultimately yields true, the validation is successful. We recall from Section 2 that an unspent output locks some funds employing a locking script and, to spend such funds, one needs to provide some witness as unlocking script. We consider Tx_P solely spending Tx_G's output θ_G. We recall Bitcoin-like chains process instructions sequentially: For Tx_P to spend θ_G, the locking script of θ_G is executed after the witness for Tx_P. If the computation ultimately yields true, the validation is successful.

We denote data by using angle brackets, i.e., `<data>`, and to ease readability we implicitly assume that data to be pushed on the stack uses the `OP_PUSHDATA` opcode and followed by the data byte-length. We now provide the witness and locking script for the simple case described above, along with a high-level description. As an amusing exercise, we let the reader verify the whole computation correctness step by step.

Unlocking Script (Witness). In line 1 of Figure E.2, we have

```
1  <σP> <σV>
2  <HeaderSuffix> <HeaderPrefix>
3  <MP2> <MP1>
4  <txid> <outid>
```

Figure E.2: Example of witness containing the realization for the x_1 input of Desc (to be read from bottom to top and from left to right). The witness for Glimpse is the proof itself.

P and V signatures over Tx_P necessary to verify the 2-2 multi-signature spending condition. In line 2, we have the block header suffix and prefix (`HeaderSuffix`, `HeaderPrefix`) which, along with the to-be-computed Merkle root, give the block header. In line 3, we have the Merkle proof elements that, along with the hash of Tx, allow to reconstruct the Merkle root for the transactions in B. Finally, line 4 shows the realization of x_1 (`txid` and `outid`).

Locking Script. In line 1 and 2, the script ensures `<txid>` and `<outid>` have the expected byte-length, i.e., `<outid>` of 4-bytes and `<txid>` of 32-bytes. This step is necessary as *Glimpse needs to verify the input and output strings of the witness are not malicious*: concretely, *the strings have to be interpreted by the nodes as intended at the beginning, not changing the validation process, e.g., by injecting malicious*

```
1   OP_SIZE <4> OP_EQUALVERIFY
2   <1> OP_PICK OP_SIZE <32> OP_EQUALVERIFY
    OP_DROP
3   OP_CAT <txSuffix> OP_CAT  <txPrefix>
    OP_SWAP OP_CAT OP_HASH256
4   OP_CAT OP_HASH256 OP_SWAP OP_CAT OP_HASH256
5   OP_CAT OP_SWAP OP_CAT OP_HASH256
6   <target> OP_SUBSTR <4> OP_ROT OP_ROT
    OP_SUBSTR <4> OP_ROT OP_ROT OP_LESSTHAN
    OP_VERIFY OP_SWAP
7   OP_SUBSTR <4> OP_ROT OP_ROT OP_SUBSTR <4>
    OP_ROT OP_ROT OP_LESSTHAN OP_VERIFY OP_SWAP
8   OP_SUBSTR <4> OP_ROT OP_ROT OP_SUBSTR <4>
    OP_ROT OP_ROT OP_LESSTHAN OP_VERIFY OP_SWAP
9   OP_SUBSTR <4> OP_ROT OP_ROT OP_SUBSTR <4>
    OP_ROT OP_ROT OP_LESSTHAN OP_VERIFY OP_SWAP
10  OP_SUBSTR <4> OP_ROT OP_ROT OP_SUBSTR <4>
    OP_ROT OP_ROT OP_LESSTHAN OP_VERIFY OP_SWAP
11  OP_SUBSTR <4> OP_ROT OP_ROT OP_SUBSTR <4>
    OP_ROT OP_ROT OP_LESSTHAN OP_VERIFY OP_SWAP
12  OP_SUBSTR <4> OP_ROT OP_ROT OP_SUBSTR <4>
    OP_ROT OP_ROT OP_LESSTHAN OP_VERIFY OP_SWAP
13  OP_LESSTHAN OP_VERIFY
14  <pk_V> OP_CHECKSIG <pk_P> OP_CHECKSIGADD <2>
    OP_NUMEQUAL
```

Figure E.3: Example of locking script contained in one branch of the MAST.

instructions or data. For this, Glimpse first verifies the number of strings and their length: this is possible because, even if they are a priori undefined, they are of known number and size. In line 3, the transaction body is reconstructed by concatenating `<txid>` and `<outid>` with the Desc (`<txSuffix>`, `<txPrefix>`) - we stress the description must be hard-coded in the locking script so that no malicious party can tamper with it or change it during the lifetime of Glimpse. The transaction body is finally hashed. In line 4, the transaction Merkle root MR of block B is computed using txid, `<MP2>`, and `<MP1>`. In line 5, B's header is reconstructed by concatenating `<HeaderPrefix>`, the Merkle root, and `<HeaderSuffix>`, and it is finally hashed. From line 6 to 13 we check the header hash is smaller than the target (`<target>`): since there is no opcode for hash comparison, we essentially split (`OP_SUBSTR`) the two hashes in 4-bytes shares and compare them all. Finally, in line 14, we check validity of the signatures of P and V, satisfying the 2-2 multi-signature condition.

E.1 Taproot: Merkelized Abstract Syntax Tree (MAST)

Real-world use cases are not as simple as the example we just presented, as the number of transactions per block varies and is unpredictable a priori. The position of Tx within the block is also unpredictable. Since there are no loops in Bitcoin script, we need to explicitly provide a script for each possible size of the the merkle tree in the block header and each position of the to-be-verified transaction in the merkle tree. As we see below, we can use MAST to efficiently encode this size blow-up in a constant size output, but estimating the number of opcodes in total is more difficult.

MAST. Luckily, in some chains as Bitcoin, Litecoin, and Liquid, Taproot comes to the rescue by enabling the *Merkelized Abstract Syntax Tree* (MAST) functionality, also known as *script path spending* or *TapTree*. On a high level, a MAST is a Merkle tree whose leaves are scripts allowing a user to commit not to a single spending script but to a Merkle tree of scripts or, concretely, to a Merkle root. The user chooses which script to execute at spending time, when the inclusion of the chosen script within the committed tree has to be proven revealing the public Taproot internal key, the Merkle proof to the Taproot leaf, and the to-be-executed script in the leaf. For Glimpse the parties can thus construct a MAST whose leaves are the scripts for all the possible realizations of number of transactions and positions of the to-be-verified transaction in of the block.

Number of Transactions in a Block: Say, the number of transactions in a Bitcoin block is at most 4000 (the average is closer to 2000), so we can assume to have an upper limit of $2^{12} = 4096$ transactions in a block. By design, Bitcoin Merkle trees have an even number of elements on each level, as every last element in an odd position gets duplicated. This affects the number of elements in the Merkle proof, such that if one has k leaves, with $2^n \leq k \leq 2^{n+1}$, the number of elements will be the same as for a tree of 2^{n+1} leaves. It follows that from 2^0 to 2^{12} transactions in a block, we have to encode the Merkle root reconstruction for only 13 different trees.

Position of the Transaction in the Block: Assuming $\sum_{i=0}^{12} 2^i = 8191$ different leaves in the tree, we obtain 8191 scripts in the MAST; however, we consider 8192 different scripts, as we also include spending condition for the verifier. Taproot limits set to 2^{128} the maximum number of scripts allowed within the MAST [14]. Furthermore, the largest script to reconstruct the Merkle root is when the transaction Merkle tree has 2^{12} leaves, resulting in 36 opcodes. Considering that the number of opcodes for all the other checks and validations is not larger than 100 opcodes, we are well within the Taproot limits, where 201 is the maximum number of opcodes allowed per script [58].

Without MAST. If the MAST feature is unavailable on the destination blockchain, as is the case for Bitcoin Cash, Glimpse can still be encoded, although with a more complex script. Indeed, one could unroll the MAST tree and encode the branches with nested `if-else` conditions. Of course, this leads to a large script whose number of opcodes is given by $\sum_{l=0}^{\log_2(M)} 2^l \cdot (3l+3)+1$, where M is the maximum number of transactions in a block. Concretely, $\sum_{l=0}^{\log_2(M)} 2^l$ gives the total number of scripts necessary to consider the different possible positions of the transaction within the tree, while

$\sum_{l=0}^{\log_2(M)}(3l+3)+1$ is the maximum number of opcodes per script (upper bound). For instance, being 550 transactions/hour the throughput of Bitcoin Cash, we reasonably assume $M = 1000$: this results in an upper bound of 136k opcodes, each opcode size being of 1 bytes. While this is by far within the transaction size limits, Bitcoin-like chains limit the maximum number of opcodes within a transaction (`MAX_OPS_PER_SCRIPT` is 201 Bitcoin Cash and 500 in Bitcoin SV). Missing Taproot, one can use these chains as Glimpse destination chains only if this constraint is removed.