Glimpse: On-Demand PoW 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 for 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 such as the Liquid Network (a pegged sidechain of Bitcoin), 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 presenting different technical features (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 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.

The challenge of blockchain interoperability (or *cross-chain communication*) stems from the two functionalities it has to provide: (i) relaying information from a source ledger $\mathcal{L}_S$ to a destination ledger $\mathcal{L}_D$, allowing a user of $\mathcal{L}_D$ to verify that a transaction Tx_S has been included in $\mathcal{L}_S$ without participating in $\mathcal{L}_S$'s consensus protocol (cross-chain verification), and (ii) atomic synchronization of transactions across different chains, e.g., in an atomic swap, a transaction Tx_D on $\mathcal{L}_D$ succeeds if and only if Tx_S was previously posted on $\mathcal{L}_S$ (cross-chain atomicity).

Related Work. Cross-chain verification is typically achieved by running either full nodes or light clients with linear storage overhead in $\mathcal{L}_S$'s length. 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 relaying and processing of a *linear* amount of

information in the chain length, 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. Chain relays verify and store every block header of $\mathcal{L}_S$ within a smart contract on $\mathcal{L}_D$, thereby acting as light clients. The inefficiency of this construction, associated with the lack of incentives for relayers and the high maintenance costs, is arguably one of the reasons why relays are not used in practice.[1] Later on, *super-light clients* with logarithmic complexity were proposed (PoPoW [7], FlyClient [8]), but they either require constant PoW difficulty [7] or an hard fork in Bitcoin [8], and are thus not backward compatible.

More recently, Xie et al. [9] have introduced an Ethereum-compatible bridge with constant size storage, where a zk-SNARK proof guarantees that the blockchain has undergone a state update (either a single block or a batch of them). Each verified state update is stored within a stateful contract, and recent block headers of the source chain are relayed to and stored within the contract until a zero-knowledge proof is made available to finalize and verify the state update. Application-specific contracts can then rely on zkBridge to perform, e.g., SPV verifications. However, zkBridge still requires a linear amount of information relayed from the source to the destination chain. As chain relays, zkBridge still incurs in high maintenance costs and lacks of incentives for relayers, which continuously compute zk proofs and submit them to the bridge contract along with block headers. Moreover, zkBridge can only be deployed and used in destination chains supporting a quasi-Turing complete language, thus excluding important chains holding hundreds of millions of dollars.

Similarly to zkBridge, all current implementations based on the aforementioned client solutions require a quasi-Turing complete scripting language on 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 of 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*, initially proposed by Prestwich [11] and implemented by Summa [12]. Stateless SPV also emerged within the Ethereum research community [13]. Recently, Barbára et al. [14] 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 subchains), 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*: By knowing the to-be-verified transaction upfront, a malicious prover may produce a forged subchain beforehand, leveraging the fact that users on $\mathcal{L}_D$ have no way to ensure that such proof corresponds to the suffix of the correct chain. Since there is no backward time constraint on performing an upfront mining attack, the attacker will eventually succeed in finding enough forged blocks regardless of their mining power and without needing to bribe any miners. This attack is not considered in stateless SPV nor in 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. A summary comparison of the different clients for cross-chain verification can be found in Table 1.

	LC [4–6]	SLC [7, 8]	zkBridge [9]	SSPV [10, 13]	**Glimpse**
Information Relayed:	Linear	Logarithmic	Linear	Constant	Constant
Storage Overhead:	Linear	Logarithmic	Constant	Constant	Constant
Backward Compatibility:	Yes	No	Yes	Yes	Yes
$\mathcal{L}_D$ q-Turing completeness:	Yes	Yes	Yes	Yes	No
Upfront Mining Secure:	Yes	Yes	Yes	No	Yes

Table 1: For light clients (LC), super-light clients (SLC), zkBridge, stateless SPV (SSPV), and Glimpse we show the amount of information relayed, storage overhead, backward compatibility, need for quasi-Turing complete scripting language on $\mathcal{L}_D$, and security w.r.t. upfront mining attacks.

Cross-chain atomicity, i.e., the second core functionality of interoperability, is typically implemented with *lock contracts*, such as Hashed TimeLock Contracts (HTLCs) and adaptor signatures [15]. These secret-based cryptographic techniques 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 within a transaction Tx_S posted on-chain $\mathcal{L}_S$. Lock contracts, however, have fundamental limitations: (i) they require both parties to monitor and actively participate in both chains (e.g., in the context of atomic swaps), (ii) their expressiveness is very limited: they require all transactions to be fully fixed upfront (and pre-signed by the party giving away the coins) since they must depend on the same secret, and (iii) they require one party choosing the secret at the start of the protocol, while the other party learns it later. As a result, lock contracts cannot be used in applications like lending or Proofs-of-Burn, where *the same party* needs to post transactions on both $\mathcal{L}_S$ and $\mathcal{L}_D$ *without the intervention of the other*. We expand on this in Appendix A, where we further discuss the related work.

To summarize, the present research landscape leaves the following research question open: *"Is it possible to design*

[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.

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 encoding a novel on-demand PoW light client in the scripting language of the destination chain.

To achieve constant-size overhead, our light client *assumes knowledge of the current PoW target* and is acting *on-demand, verifying only selected 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 the *contract lifetime*), 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. 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., in 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, Glimpse allows to encode that if any set of transactions satisfying a logical formula expressed as Disjunctive Normal Form[2] (DNF), e.g., $\mathsf{Tx}_S \vee \mathsf{Tx}_S'$, is published on $\mathcal{L}_S$, then Tx_D can be published on $\mathcal{L}_D$. These generalizations allow us to encode a variety of DeFi applications, such as lending, pegs, wrapping/unwrapping of tokens, Proofs-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, thus supporting Bitcoin-based blockchains such as the Liquid Network.[3] The new DeFi protocols for the Liquid Network enabled by Glimpse are de-facto brought into Bitcoin, by pegged conversion of BTC into L-BTC tokens. We further show that only two opcodes are missing to directly support Glimpse on Bitcoin as a destination chain, which adds a further motivation for their inclusion in the ongoing discussion within the community. Notably, Glimpse has full compatibility with Bitcoin as source chain, thus enabling, for the first time, Bitcoin-to-Ethereum cross-chain communication with constant storage, constant amount of relayed information, and no maintenance costs.

[2] A disjunctive normal form formula is a logical formula consisting of a disjunction of conjunctions; it can also be described as an OR of ANDs.

[3] The Liquid Network [16] is a Bitcoin sidechain supported by Blockstream [17] and other major Bitcoin stakeholders, that has anticipated all major upgrades in Bitcoin (SegWit [18], Taproot [19]). The Liquid Network plays a key role in the Bitcoin ecosystem, e.g., El Salvador's Bitcoin bonds are Liquid Network security tokens [20].

$\mathcal{L}_S$ \ $\mathcal{L}_D$	Bitcoin	Bitcoin Cash/SV	Litecoin	Liquid	Ethereum/EVM-chains
Bitcoin	-	✗††	✗†	✓	✓
Bitcoin Cash/SV	✗†	-	✗†	✓	✓
Litecoin	✗†,*	✗††,*	-	✗*	✓
Ethereum PoW	✗†,*	✗††,*	✗†,*	✗*	✓

Table 2: Popular Bitcoin-based and EVM-based Glimpse-compatible source ($\mathcal{L}_S$) and destination ($\mathcal{L}_D$) chains.†: lack of string opcodes. ††: lack of Taproot. *: lack of crypto opcodes.

Table 2 provides a non-exhaustive list of popular chains for which we show the current compatibility when functioning as $\mathcal{L}_S$ or $\mathcal{L}_D$ for Glimpse.

Our further contributions are summarized below:

- 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 properties (Section 5).
- We conduct an economic security analysis to quantify the costs of forgery attacks (affecting any light client) and censorship attacks (harming any timelock-based protocol). Specifically, Glimpse is secure against proof forgeries as long as the value *simultaneously locked on all valid Glimpse contracts* does not exceed a certain threshold (for concrete numbers, \$230M), which is comparable to the total value currently locked on popular bridges. To enhance security against censorship attacks on the destination chain, we further impose an upper bound on the value held by *each single Glimpse contract*, 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 [21, 22] (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 ParentHash is the hash of the previous block, MR is the root of the Merkle tree whose leaves are the transactions in B, 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 with only a *constant amount of data* (with respect to the source chain's length), and assuming the *PoW target is known*.

In particular, Glimpse resembles challenge-response protocols: a *verifier* V challenges a *prover* P to prove the inclusion on $\mathcal{L}_S$ of a *specific* set of transactions Tx_S. Depending on the outcome of the challenge, P and V want to publish on $\mathcal{L}_D$ different pre-selected sets of transactions (Tx_P or Tx_V). To encode this, P and V first agree on the Glimpse specifics and on some consensus parameters of $\mathcal{L}_S$, then they deploy a *Glimpse contract* on $\mathcal{L}_D$. On ledger $\mathcal{L}_S$, an *issuer* I publishes the transaction set Tx_S, and an *untrusted relayer* R provides P with the necessary data to construct a *proof* $\mathcal{P}$, proving 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 Tx_P on $\mathcal{L}_D$. Else, V can post 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 Tx_S on $\mathcal{L}_S$, a prover P that proves the inclusion 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. Depending on the application, parties can play several of these roles at once. E.g., in lending, Proofs-of-Burn, and backed assets (see Section 4.1), P can also play the roles of I and R, being incentivized to get reliable insights on $\mathcal{L}_S$'s state.

We require P and V to each have a key pair $(\mathsf{sk}, \mathsf{pk})$ 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 $\mathcal{L}_D$ (this is application-specific). We assume $\mathcal{L}_D$ to support the same hash function used by the consensus of $\mathcal{L}_S$, and both $\mathcal{L}_S$ and $\mathcal{L}_D$ to allocate the same domain for the hash function, to avoid oversize preimage attacks [23,24]. 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 (recently discussed within the Bitcoin community in the context of Speedy Covenants [25]).

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 [23], 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, the commitment published on $\mathcal{L}_S$ is relayed to $\mathcal{L}_D$, it is verified, 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 posted.

A CCC protocol has to provide atomicity guarantees, which, for Glimpse, we articulate in a weak and a strong variant. *Weak atomicity* ensures that Tx_D appears on $\mathcal{L}_D$ only if Tx_S has been already confirmed on $\mathcal{L}_S$. *Strong atomicity* ensures that Tx_D appears on $\mathcal{L}_D$ if and only if Tx_S has been already 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 the time it takes for a valid transactions to be included on $\mathcal{L}_D$, and let Tx_S and Tx_D be (sets of) transactions for $\mathcal{L}_S$ and $\mathcal{L}_D$, respectively. We refer to n as the number of confirmation blocks that need to be mined on top of a block containing a transaction Tx, for Tx to be considered *stable* [26] on a PoW ledger.

Definition 1 (*Weak Atomicity*)**.** A valid Tx_D is reported stable by honest players on $\mathcal{L}_D$ at time t only if honest players of $\mathcal{L}_S$ have reported Tx_S with at least n confirmations at time $t' \leq t - \Delta_D$: $\mathsf{Tx}_D \in \mathcal{L}_D \implies \mathsf{Tx}_S \in \mathcal{L}_S$.

Definition 2 (*Strong Atomicity*)**.** A valid 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* parties, with *at least one between P and V being honest*. We let γ be the fraction of honest miners the blockchain can tolerate, where the specific value for 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 weak atomicity holds (Section 5).

Then, we extend our model to incorporate rational participants or, in other words, participants exhibiting liveness during the whole duration of the protocol (as defined in Appendix B.2): in this setting, assuming P has direct access to $\mathcal{L}_S$, we show that the Glimpse protocol UC-realizes an ideal functionality $\mathcal{F}_{S-Glimpse}$, and strong atomicity holds (Section 5). Finally, in Section 6 we provide an economic analysis incorporating rational adversaries.

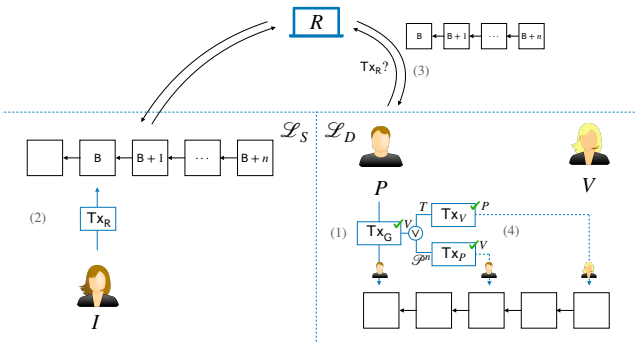

Figure 1: (1) Upon agreeing on the Glimpse specifics, P and V construct Tx_G, Tx_P, and Tx_V. P publishes Tx_G on $\mathcal{L}_D$. (2) I publishes Tx_R on $\mathcal{L}_S$. (3) P queries R to get the data to construct the proof $\mathcal{P}^n$ for Tx_R. (4) P publishes Tx_P with $\mathcal{P}^n$ as part of the witness. Else, if after T the funds in Tx_G are unspent, V publishes Tx_V.

3.2 Protocol Overview

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

P and V prepare transactions Tx_P and Tx_V, both spending the contract's funds, but in different ways; these two transactions are meant to be published by P and V respectively, and are commitments to how the coins must be 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 time T. P signs Tx_V and sends the signature to V, whereas V signs Tx_P and gives the signature to P. They exchange signatures over Tx_G and publish Tx_G on $\mathcal{L}_D$ (if any additional party contributes funds to Tx_G, their signature over Tx_G is also required in order to publish Tx_G). Finally, they hand in 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 $\mathcal{L}_S$ asking for the necessary data to construct the proof $\mathcal{P}^n$ (we describe how $\mathcal{P}^n$ is constructed in Section 3.3). P publishes Tx_P on $\mathcal{L}_D$ by using $\mathcal{P}^n$, V's signature over Tx_P, and their own signature as witnesses. After time T, if the funds in Tx_G are still unspent, V publishes Tx_V on $\mathcal{L}_D$ with P's signature over Tx_V as well as their own as witnesses. The Glimpse instance is now closed and the funds are distributed as agreed in the *Setup* phase.

In Section 3.4, we will explore how this approach can be extended to generalize Tx_S as a set of transactions represented by a disjunctive normal form formula. Figure 1 depicts Glimpse protocol flow.

3.3 Designing the Proof

We show how the Glimpse proof $\mathcal{P}^n$ is constructed by considering stateless SPV as a preliminary proposal. We identify its vulnerabilities to upfront mining attacks and gradually enhance the construction to attain the desired security properties, while also enhancing expressiveness and compatibility.

Stateless SPV. In stateless SPV [13, 14], users convince with a proof $\mathcal{P}^n$ a quasi-Turing complete smart contract hosted on a destination chain $\mathcal{L}_D$ that a transaction Tx has appeared on a PoW source chain $\mathcal{L}_S$. The proof $\mathcal{P}^n$ consists of the header of the block containing Tx, the Merkle inclusion proof for the transaction within such block, and n subsequent confirmation block headers. The smart contract verifies the Merkle proof, checks that each of the $n+1$ headers is a valid child of its parent, and ensures that all headers hold enough PoW, i.e., their hashes are smaller than the pre-defined target.

Various proposals exist for stateless SPV, each with its specific requirements: some demand the contract to check certain fields of the to-be-verified transaction, to guarantee the transaction's intended behavior; others only require a timelock T to limit the time window for the submission of the proof. We show that both requirements are necessary: the first to ensure the transaction is well formed and is indeed the one the parties have agreed upon during the *Setup*, the second to prevent a late submission of the proof (which would break security) and to avoid hostage situations where the funds in the contract are locked indefinitely. We observe that the number n of confirmation blocks in the proof must increase linearly with the timelock T, leading to short lived contracts for practicality.

Upfront Mining Attack. Current stateless SPV designs impose no restriction on Tx, exposing the construction to security risks. In particular, let us consider a malicious P (attacker) wishing to convince the smart contract on $\mathcal{L}_D$ that Tx has been included on $\mathcal{L}_S$. If the attacker knows Tx beforehand, e.g., well before setting up the contract, they could start mining in advance in order to forge a proof. To illustrate this problem, we consider Bitcoin as source chain: Bitcoin is extended by approximately 53k blocks yearly. An attacker with 0.05% of the mining power of the network is expected to find around 6 blocks in less than three months. This means that the attacker can start forging such 6-block proof upfront and proceed by setting up the stateless SPV contract only when the forged proof is ready, e.g., three months later. In this case, the attacker foregoes any potential reward from honestly mining on the main chain, but can claim all the money the contract holds with certainty, regardless of their mining power, and without the need to bribe miners to forge the proof. Even 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 use the same, upfront-mined fake proof in all contracts, thereby cheating multiple users out of their funds at once.

Randomized Tx. Glimpse prevents P from launching an upfront mining attack by asking V to *randomize the transaction* Tx. Specifically, in the *Setup* phase, V samples a uniformly random string $r \xleftarrow{\$} \{0,1\}^\lambda$ (λ is the security parameter) and plugs it into the body of Tx, producing Tx_R. This can be done, e.g., by adding an output of value 0 with spending condition $\texttt{OP\_RETURN}(r)$.[4] The hash of the randomized transaction $\mathcal{H}([\mathsf{Tx}_\mathsf{R}])$ is then hardcoded within Tx_G. Being unable to anticipate r, P cannot start forging $\mathcal{P}^n$ upfront: their computational effort is now restricted to the time window T. Additionally, the random value serves as a *unique identifier, preventing proof replay attacks*.

We highlight that randomizing the transaction does not impose any trust assumption, and the verification remains constant-sized, achieving our design goals. We observe that security against upfront mining attacks restricts Glimpse to verifying transactions that will be included on $\mathcal{L}_S$ "in the future", i.e., only after the contract Tx_G has been set up. Past transactions cannot be randomized anymore and are thus vulnerable to upfront mining. This is a fundamental difference between Glimpse and light clients, which can verify any past transaction instead.

Improving Compatibility. Besides protecting from upfront mining attacks, we show that the Glimpse construction also forgoes the need for stateful smart contracts, opposed to [13, 14] which is designed for quasi-Turing complete contracts. Due to its simplicity, Glimpse can be deployed not only on quasi-Turing complete chains (e.g., Ethereum) but also on *Bitcoin-based chains* such as the Liquid Network. This is achieved by hardcoding the transaction fields and the verification logic in a transaction locking script, which is spendable by using signatures and $\mathcal{P}^n$ as witness. We expand on this in Section 3.5.

3.4 Enhancing Expressiveness

Glimpse cannot yet encode sophisticated applications due to two shortcomings in expressiveness: First, inputs and outputs of Tx_R must be entirely known a priori, which prevents from using it in, e.g., cross-chain lending applications. Second, only single transaction verification is supported, prohibiting verification of, e.g., attestations from multiple oracles. To cater to such use cases, we augment Glimpse to verify (i) parameterized transactions, i.e., transactions which are *not fully known* during the initial *Setup* phase and (ii) *arbitrary combinations of transactions* on $\mathcal{L}_S$ expressed as *disjunctive normal form* formulas.

Parameterized Tx. 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). From Section 2, recall our definition of transaction body $[\mathsf{Tx}] :=$

[4] $\texttt{OP\_RETURN}$ is a Bitcoin script opcode that marks a transaction output as unspendable and can be used to embed up to 80 bytes in a transaction.

$(\mathsf{cntr}_{in}, \mathsf{inputs}, \mathsf{cntr}_{out}, \mathsf{outputs})$, where inputs and outputs are tuples $(\mathsf{txid}, \mathsf{outid})$ and (coins, ϕ), respectively. Now, in Figure 2, we introduce the definition of *description* Desc of a transaction, which allows for parameterized inputs and outputs. Concretely, following Figure 2, txid, outid, and coins can either be static values or parameters x_i acting as *placeholders*. Similarly, to avoid fixing a priori specific parameters for the locking script, we say that ϕ can be a function f which encodes a family of scripts: f takes a fixed number of arguments (or parameters) z_i for a *known spending condition logic* and returns the desired parameterized locking script for the to-be-verified transaction. In other words, the parameterized locking script can be filled with concrete values, e.g., public key, script hash, *after* the *Setup* phase. The spending condition logic that f encodes must be already defined in the *Setup* phase: For instance, if the parties agree on $f(z)$ encoding any P2PKH (Pay-To-Public-Key-Hash), then the output of $f(z)$ is a P2PKH with a placeholder z in the place of the public key hash, and only after the *Setup* phase it accepts any public key hash as replacement for z. Further following Figure 2, inputs and outputs are lists of inputs and outputs as defined above, and cntr_{in} and cntr_{out} are the number of overall inputs and outputs, respectively. While the former can be parameterized, 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 now hardcode within the contract $\mathsf{Tx_G}$ the description Desc, the target $\mathcal{T}_S$, the lifetime T, and the proof size n. By replacing $\mathcal{H}([\mathsf{Tx_R}])$ with Desc, Glimpse can now verify any $\mathsf{Tx_R}$ whose body has the same static data in Desc and any arbitrary realization (specified in a later point in time within the proof) of the parameterized ones. In other words, Desc defines the set of possible transactions Glimpse can accept, yet only one of them can be verified. For example, a Glimpse instance with a parameterized input in Desc, can accept and verify transactions with any input (i.e., any value for txid and outid), but with all other fields matching the ones specified in Desc. For security reasons, the random string sampled by V must always be included in Desc. With $[\mathsf{Tx_R}] \hookleftarrow \mathsf{Desc}$, we denote a transaction $[\mathsf{Tx_R}]$ *compliant with a description* Desc. Given $\mathcal{P}^n$ and the hardcoded Desc, the full transaction body can be reconstructed and hashed by the script of $\mathsf{Tx_G}$.

Verification of DNF Formulas. Glimpse can efficiently verify any DNF formula $\mathcal{F}_S$ over any k literals L_i, which, in our case, are either transactions or descriptions (see Figure 2). To accomplish this, instead of a single $\mathsf{Tx_P}$, P and V create as many sets of transactions ($\mathsf{Tx_T}$, $\mathsf{Tx_F}$, $\mathsf{Tx_P}$) as the number of conjunctive terms in the formula - these sets are kept off-chain. When I publishes on $\mathcal{L}_S$ a combination of transactions specified by $\mathcal{F}_S$, P queries R, constructs the corresponding proofs, 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., falsely claiming a transaction was not published on

$$
\begin{array}{rcl}
\mathsf{txid} & := & \{0,1\}^{256} \mid x_1, \\
\mathsf{outid} & := & \{0,1\}^{32} \mid x_2 \\
\mathsf{coins} & := & \{0,1\}^{64} \mid x_3 \\
\phi & := & f(z_1, \ldots, z_n) \\
\mathsf{inputs} & := & [(\mathsf{txid}, \mathsf{outid})] \mid \mathsf{inputs} \cup [(\mathsf{txid}, \mathsf{outid})] \\
\mathsf{outputs} & := & [(\mathsf{coins}, \phi)] \mid \mathsf{outputs} \cup [(\mathsf{coins}, \phi)] \\
\mathsf{cntr}_{in}, \mathsf{cntr}_{out} & := & \{0,1\}^{1-9} \\
\mathsf{Desc} & := & (\mathsf{cntr}_{in}, \mathsf{inputs}, \mathsf{cntr}_{out}, \mathsf{outputs}) \\
L_i & := & \mathsf{Desc}_i \mid \neg \mathsf{Desc}_i \\
\mathcal{F}_S & := & (L_1 \wedge \ldots \wedge L_k) \vee \ldots \vee (L_1 \wedge \ldots \wedge L_k)
\end{array}
$$

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

Figure 2: Enhanced expressiveness for the verification of parameterized transactions and DNF formulas.

$\mathcal{L}_S$, V can query R, disprove P, and publish $\mathsf{Tx_V}$. We expand on this in Appendix D.

3.5 Compatibility

As anticipated in Table 2, PoW chains such as Bitcoin, Litecoin, Bitcoin Cash, Bitcoin SV, Ethereum PoW, etc., can be used as the source chain $\mathcal{L}_S$ for Glimpse. As for which chains are supported as destination chain $\mathcal{L}_D$, we need to make some distinctions, because the more restrictive is the scripting language of $\mathcal{L}_D$, the fewer $\mathcal{L}_S$ may be compatible.

In particular, Glimpse requires $\mathcal{L}_D$ to support the hash function used in the PoW consensus of $\mathcal{L}_S$. This strict requirement already rules out some combinations, see the lack of cryptographic primitives in Table 2. For instance, Bitcoin-based chains do not have opcodes for computing Keccak and Scrypt hash functions which are used in Ethereum PoW and Litecoin, respectively. As a result, Ethereum PoW and Litecoin cannot act as source chains for Glimpse contracts deployed on Bitcoin-based chains. On the other hand, EVM-based chains (e.g., Ethereum, Polygon, Binance Smart Chain, Avalanche) as well as chains allowing for a quasi-Turing complete scripting language, can always act as destination chains for Glimpse, no matter what the selected source chain is.

Now, we discuss how the particular design of Glimpse makes it compatible with the Liquid Network, the Bitcoin sidechain. For extended discussion and examples, we refer to Appendix E.

Liquid Network (Full Compatibility). The Liquid Network inherits its design from Bitcoin, while offering an enriched scripting language. Specifically, certain opcodes essential to Glimpse are disabled in Bitcoin but enabled on the Liquid Network. These include: (i) string concatenation (`OP_CAT`), which can be used for Merkle proof verification, block header reconstruction, and transaction body reconstruction, and (ii) `OP_SUBSTR`, which allows splitting strings. This is necessary for comparing hashes (i.e., compare a block header hash to the PoW target): currently, comparisons can only be made between 4-byte strings. Furthermore, the Liquid Network incorporates Taproot [21]: by leveraging Merkelized Abstract

Syntax Trees (MASTs), we can greatly reduce the size and complexity of Glimpse scripts, as shown in Section 7 and exemplified in Appendix E.

3.6 Extend Compatibility: Required Opcodes

In this section, we show which opcodes are missing in order to extend the compatibility of Glimpse to destination chains as Bitcoin, Litecoin, Bitcoin Cash, and Bitcoin SV. Notably, Bitcoin can be $\mathcal{L}_D$ for Glimpse, thereby achieving complete compatibility, with the sole addition of two string opcodes (`OP_CAT`, `OP_SUBSTR`).

Bitcoin and Litecoin (Missing String Opcodes). Bitcoin and Litecoin adopted Taproot, but they deactivated the previously mentioned opcodes back in 2010. If these opcodes for Merkle proof verification (`OP_CAT`) and hash comparison (`OP_SUBSTR` or, alternatively, `OP_LESSTHAN` being capable of comparing 32-byte values) were available, they could effectively support Glimpse using the efficiency of Taproot. It is worth noting that the Bitcoin community has recently engaged in discussions regarding the potential reinstatement of the string concatenation opcode. This consideration comes with the proposal of Speedy Covenants [25]. With our work, we hope to contribute to the ongoing discussion and provide additional motivation for the future enabling of such opcodes.

Bitcoin Cash and Bitcoin SV (Missing Taproot). Bitcoin Cash is the result of a Bitcoin hard fork that took place after Bitcoin moved to SegWit. It has a larger block size and supports more opcodes. Similarly to the Liquid Network, Bitcoin Cash has `OP_CAT` and `OP_SPLIT` (same as `OP_SUBSTR`), but lacks Taproot. When Taproot is not available, one could unroll the MAST, obtaining a large Glimpse script which is, for small n (e.g., $n < 12$), dominated by the opcodes for the Merkle proof verification. In this case, Glimpse could be supported by removing the limit for the maximum number of opcodes allowed in a script (`MAX_OPS_PER_SCRIPT`), or extending it up to 300k. The same applies to Bitcoin SV.

4 Glimpse for Lending and Cross-Chain DeFi

DeFi applications thrive on blockchains supporting quasi-Turing complete smart contracts, but do not exist on Bitcoin-based blockchains. To fill this gap, we show how to use Glimpse for designing a lending protocol for Bitcoin-based chains. We provide pseudocode in Figure 4.

Intuition. We consider a borrower P (also acting as I and R) and a lender V. P has α coins (collateral) on $\mathcal{L}_D$ and wants to take a loan of α' coins on $\mathcal{L}_S$. Having a surplus of coins on $\mathcal{L}_S$, V is willing to grant a loan of α' coins to P. We assume $\alpha > \alpha'$, i.e., the loan is over-collateralized to compensate for price drops of asset α. The lending protocol comprises two steps: (1) an atomic swap, where the loan-granting transaction (Tx_{Loan}) is published on $\mathcal{L}_S$ and the Glimpse transaction (Tx_{G}) holding P's collateral is published on $\mathcal{L}_D$, and (2) a Glimpse protocol, where Tx_{G} returns the α coins to P upon the loan being repaid ($\mathsf{Tx}_{Payback}$) on $\mathcal{L}_S$.

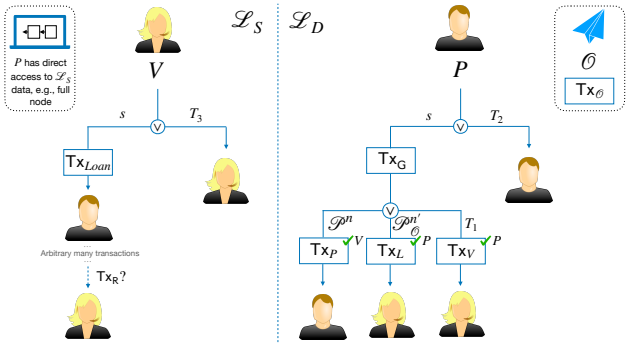

Figure 3: Sketch for the Glimpse-based lending.

As for (1), P and V prepare Tx_{G} holding the α coins of P, and they construct it in such a way that it can be published by revealing a secret s that P knows. P signs Tx_{G} and gives the signature to V. V prepares a transaction Tx_{Loan} transferring α' coins to P and conditioned on the same secret s. V signs it and gives Tx_{Loan} and their signature to P. Via an atomic swap, P publishes Tx_{Loan} on $\mathcal{L}_S$ revealing s to V, and V publishes Tx_{G} on $\mathcal{L}_D$ using the secret s leaked by P.

As for (2), Tx_{G} guarantees that if P repays the loan on $\mathcal{L}_S$ by publishing $\mathsf{Tx}_{Payback}$, P gets back their collateral on $\mathcal{L}_D$. Otherwise, V retains collateral after time T_1.

We discuss the liquidation mechanism at the end of this section, and we expand on (2) below. In Figure 3 we sketch the lending protocol making use of Glimpse.

Setup. P sends $\mathsf{Desc}{:=}(1,[(x)],1,[(\alpha',\mathsf{OneSig}(\mathsf{pk}_V))])$ to V, where Desc is the description of $\mathsf{Tx}_{Payback}$ (note the parameterized input x, which leaves P the freedom to choose the input later on, after having performed arbitrary transactions with the borrowed money). V samples $r \leftarrow \{0,1\}^\lambda$ uniformly at random and includes it within the description, returning $\mathsf{Desc} := (1,[(x)],2,[(\alpha',\mathsf{OneSig}(\mathsf{pk}_V)),(0,\mathtt{OP\_RETURN}(r))])$ to P.

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 $[\mathsf{Tx}_G] := (1,[(\zeta_P)],1,[(\alpha,\mathsf{scriptG}(\mathsf{Desc},T_1,\mathcal{T}_S,n,(P,V)))])$. The locking script generated by $\mathsf{scriptG}$[5] encodes the following: P can get back their α coins by submitting a valid proof $\mathcal{P}^n$ (witness), which proves the inclusion of $[\mathsf{Tx}_{Payback}] \hookleftarrow \mathsf{Desc}$ on $\mathcal{L}_S$; alternatively, V can get the α coins after time T_1. We show the pseudocode for $\mathsf{scriptG}$ in Figure 4.

After setting up $[\mathsf{Tx}_G]$, P constructs $[\mathsf{Tx}_\mathsf{P}] = (1,[\zeta_G],1,[(\alpha,\mathsf{OneSig}(\mathsf{pk}_P))])$ and $[\mathsf{Tx}_\mathsf{V}] := (1,[\zeta_G],1,[(\alpha,\mathsf{OneSig}(\mathsf{pk}_V))])$, where Tx_P (Tx_V) spends the output of Tx_{G} creating a new 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 Glimpse specifics: $(\mathsf{Desc},T_1,\mathcal{T}_S,n,\alpha,\mathsf{scriptG},[\mathsf{Tx}_G],[\mathsf{Tx}_\mathsf{P}],[\mathsf{Tx}_\mathsf{V}],\sigma_P([\mathsf{Tx}_\mathsf{V}]))$.

[5] We show a concrete script example in the extended version of this work [27], Appendix E.

Upon receiving the message from P, V checks the correctness and well-formedness of the Glimpse specifics and checks if $\sigma_P([\mathsf{Tx_V}])$ is a valid signature. Upon successful verification, V signs $\mathsf{Tx_P}$ and sends the signature to P. Upon receiving $\sigma_V([\mathsf{Tx_P}])$ from V, P checks the validity of the signature and, if valid, P signs $[\mathsf{Tx_G}]$ and publishes $\mathsf{Tx_G}$ on $\mathcal{L}_D$ with witness $\omega = \sigma_P([\mathsf{Tx_G}])$.

Commit on $\mathcal{L}_S$. When P wants to pay back the loan on $\mathcal{L}_S$, P posts $\mathsf{Tx}_{Payback}$ such that $[\mathsf{Tx}_{Payback}] \hookleftarrow \mathsf{Desc}$, meaning that $[\mathsf{Tx}_{Payback}]$ is equal to Desc apart from x, which in $[\mathsf{Tx}_{Payback}]$ is replaced by an arbitrary input controlled by P.

Verify & Commit on $\mathcal{L}_D$. P monitors $\mathcal{L}_S$ checking for $\mathsf{Tx}_{Payback}$ inclusion with at least n confirmations. P constructs $\mathcal{P}^n$ by (i) taking the concrete realization x^R of the parameter x, (ii) retrieving the header of the block B including $\mathsf{Tx}_{Payback}$, (iii) constructing the Merkle proof (MP) of $\mathsf{Tx}_{Payback}$ inclusion in B, and (iv) fetching the first n confirmation block headers of B. Formally, $\mathcal{P}^n := (x^R, \mathsf{MP}, \mathsf{header}(\mathsf{B}), \mathsf{confHeaders_n})$, as shown in the pseudocode of Figure 4. P signs $[\mathsf{Tx_P}]$ and gets back their α coins by publishing $\mathsf{Tx_P}$ on $\mathcal{L}_D$ with witness $\omega = (\mathcal{P}^n, \sigma_P([\mathsf{Tx_P}]), \sigma_V([\mathsf{Tx_P}]))$.

After T_1, if the output of $\mathsf{Tx_G}$ is still unspent, V signs $[\mathsf{Tx_V}]$ and publishes $\mathsf{Tx_V}$ with witness $\omega = (\sigma_P([\mathsf{Tx_V}]), \sigma_V([\mathsf{Tx_V}]))$, claiming the α coins in Glimpse. It is crucial for V to publish $\mathsf{Tx_V}$ right after time T_1, otherwise P could maliciously claim the funds by publishing $\mathsf{Tx_P}$ after T_1. On the contrary, T_1 prevents $\mathsf{Tx_V}$ from being valid before T_1.

To ease readability, so far, we have omitted the loan interest rate, which can be easily taken into account in the money distribution of $\mathsf{Tx_P}$; for instance, in Figure 4, one can set $\mathsf{outcomeP} = 0.95$ (thereby leaving the 5% of interest to V).

Liquidation. If the asset price on $\mathcal{L}_S$ drops below a predefined liquidity threshold, V must be able to claim P's collateral before T_1. For this, we assume there exists a trusted oracle O on $\mathcal{L}_D$ that regularly publishes a transaction Tx_O with the real-time price of assets on $\mathcal{L}_S$; for instance, O can be a Discreet Log Contract-based [28] or a voting-based [29] oracle. If O is not trusted, we can consider a set of k independent oracles, with the promise that if a large enough number of oracles agree on the same price, the liquidation is granted by verifying a DNF formula over the oracle transactions' descriptions. Oracles do not need to cooperate, nor have a common transaction structure. For simplicity, we discuss the case of a single trusted O whose Tx_O is described by, e.g., $\mathsf{Desc}_O := (1, [\zeta_i], 2, [\theta_r, (0, \texttt{OP\_RETURN}(\text{real-time-price}))])$.

We note that $\theta_r := (0, \texttt{OP\_RETURN}(r))$ includes the randomness, which now must be taken from $\mathcal{L}_D$ itself, so that Glimpse participants can (only for a short time window!) anticipate it and include it in Desc_O: for example, r can be the hash of the transaction (or block) including the last price update published by the oracle. It is V's responsibility to ensure Desc_O embeds the most recent random string.

To include liquidation, $\mathsf{scriptG}$ has to additionally incorporate the following logic: if, before T_1, O attests the collateral price on $\mathcal{L}_S$ below a predefined liquidity threshold, V can

Setup$(\mathsf{Desc}, T_1, \mathcal{T}_S, n)$:

1. P sends $\mathsf{Desc} := (1, [(x)], 1, [(\alpha', \mathsf{OneSig}(\mathsf{pk}_V))])$ to V.
2. Received Desc, V samples a random $r \xleftarrow{\$} \{0,1\}^\lambda$ and sends $\mathsf{Desc} := (1, [(x)], 2, [(\alpha', \mathsf{OneSig}(\mathsf{pk}_V)), (0, \texttt{OP\_RETURN}(r))])$ to P.
3. Let $\alpha := \theta_P.\mathsf{coins}$ and let ζ_P point to an unspent output of P.
4. Let $[\mathsf{Tx}_G] := (1, [\zeta_P], 1, [\theta_G := (\alpha, \mathsf{scriptG}(\mathsf{Desc}, T_1, \mathcal{T}_S, n, (P, V)))])$.
5. Let ζ_G point to θ_G. Let $[\mathsf{Tx_P}] := (1, [\zeta_G], 2, [(\alpha \cdot \mathsf{outcomeP}, \mathsf{OneSig}(\mathsf{pk}_P)), (\alpha \cdot (1 - \mathsf{outcomeP}), \mathsf{OneSig}(\mathsf{pk}_V))])$ and $[\mathsf{Tx_V}] := (1, [\zeta_G], 1, [(\alpha, \mathsf{OneSig}(\mathsf{pk}_V))])$.
6. P computes $\sigma_P([\mathsf{Tx_V}])$ and sends $(\mathsf{Desc}, T_1, \mathcal{T}_S, n, \alpha, \mathsf{scriptG}, [\mathsf{Tx}_G], [\mathsf{Tx_P}], [\mathsf{Tx_V}], \sigma_P([\mathsf{Tx_V}]))$ to V.
7. Upon receiving $(\mathsf{Desc}, T_1, \mathcal{T}_S, n, \alpha, \mathsf{scriptG}, [\mathsf{Tx}_G], [\mathsf{Tx_P}], [\mathsf{Tx_V}], \sigma_P([\mathsf{Tx_V}]))$, V checks the Glimpse parameters. If they are correct and well-formed, V signs $[\mathsf{Tx_P}]$ and sends $\sigma_V([\mathsf{Tx_P}])$ to P.
8. P verifies if $\sigma_V([\mathsf{Tx_P}])$ is a valid signature. 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 $\mathsf{Tx}_{Payback}$ s.t. $[\mathsf{Tx}_{Payback}] \hookleftarrow \mathsf{Desc}$.

Verify & Commit on $\mathcal{L}_D$ (Desc, T_1, n):

1. If $\mathsf{Tx}_{Payback}$ has n confirmations, P constructs $\mathcal{P}^n$ (as shown below) 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. After T_1, if θ_G is still unspent, V signs $[\mathsf{Tx_V}]$ and posts $\mathsf{Tx_V}$ with witness $\omega := (\sigma_P([\mathsf{Tx_V}]), \sigma_V([\mathsf{Tx_V}]))$.

Construct $\mathcal{P}^n(\mathsf{Desc}, n)$ for $\mathsf{Tx}_{Payback}$:

1. Isolate x_i^R, i.e., the realization of every parameter x_i in Desc.
2. Fetch the header of the block B including $\mathsf{Tx}_{Payback}$ and compute the Merkle proof MP for $\mathsf{Tx}_{Payback}$.
3. Retrieve the first n confirmation headers after B.
4. Return $\mathcal{P}^n := (x_i^R, \mathsf{MP}, \mathsf{header}(\mathsf{B}), \mathsf{confHeaders}_n)$.

$\mathsf{scriptG}(\mathsf{Desc}, T_1, \mathcal{T}_S, n, (P, V))$ outputs a locking script that:

- Upon receiving $\omega = (\mathcal{P}^n, \sigma_P([\mathsf{Tx_P}]), \sigma_V([\mathsf{Tx_P}]))$, does the following:
  1. If x^R is a valid realization of x, i.e., it matches the expected format and length, reconstruct $[\mathsf{Tx}_{Payback}]$. Else, return $\perp$.
  2. Compute $\mathcal{H}([\mathsf{Tx}_{Payback}])$.
  3. Given $\mathcal{H}([\mathsf{Tx}_{Payback}])$ and $\mathsf{header}(\mathsf{B})$, verify the Merkle proof MP. If successfully verifies, reconstruct the header of B. Else, return $\perp$.
  4. If the hashes of B and of the n confirmation blocks are smaller than $\mathcal{T}_S$, go to the next step. Else, return $\perp$.
  5. If $(\sigma_P([\mathsf{Tx_P}]), \sigma_V([\mathsf{Tx_P}]))$ are valid signatures of P and V over $[\mathsf{Tx_P}]$, unlock the coins. Else, return $\perp$.
- Upon receiving $\omega = (\sigma_P([\mathsf{Tx_V}]), \sigma_V([\mathsf{Tx_V}]))$, if the current time $t > T_1$, unlock the coins. Else, return $\perp$.

Figure 4: Glimpse pseudocode for cross-chain lending.

claim the collateral by publishing a liquidation transaction $\mathsf{Tx_L} := (1, [\zeta_G], 1, [(\alpha, \mathsf{OneSig}(\mathsf{pk}_V))])$ s.t. $[\mathsf{Tx_L}] \hookleftarrow \mathsf{Desc}_O$, with witness $\mathcal{P}_O^{n'}$, being $\mathcal{P}_O^{n'}$ the proof for Tx_O, i.e., the most recent oracle attestation. The liquidation transaction is constructed in the *Setup* phase, during which P signs it and gives their signature to V.

Our construction enables the first form of trustless peer-to-peer lending on chains having limited scripting capabilities. We note that in Bitcoin-based chains funds cannot be pooled, resulting in borrowers having to seek for lenders. To facilitate matching the demand and supply, we suggest setting up dedicated communication channels or platforms.

4.1 Other Applications

In this section, we outline how Glimpse can be used for different applications.

Backed Assets. We refer to *backed assets* as assets issued on a ledger $\mathcal{L}_D$ that are backed by a cryptocurrency or another asset on a ledger $\mathcal{L}_S$. This category includes assets that are issued on sidechains and backed on parent chains, such as Liquid tokens L-BTC backed by BTC, but also encompasses wrapped tokens, for example, WBTC in Ethereum backed by BTC in Bitcoin.

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. Let us remove the liquidation mechanism from the lending protocol in Section 4 and assume *V can create assets on $\mathcal{L}_D$*: with these two caveats, the same construction can be used to encode trustless pegs, where V issues new assets on the sidechain and locks them in the Glimpse contract (rather than giving a loan), and P can get them by proving to have sent some assets to the peg address on the parent chain. Similarly, P can get back the funds on the parent chain by proving to have returned the coins to the peg address on the sidechain.

Equivalently, Glimpse can be also used to wrap and unwrap tokens. Wrapped tokens are digital assets that represent other underlying assets, typically from a different blockchain. They are issued (wrapping) on $\mathcal{L}_D$ when the corresponding original tokens have been locked on $\mathcal{L}_S$, and they are then destroyed or locked (unwrapping) to release the original ones.

Proofs-of-Burn. Proofs-of-Burn prove that a certain amount of cryptocurrency or other valuable assets has been burnt, i.e., has been sent to an unspendable address or a special smart contract where they become permanently irretrievable. Proofs-of-Burn are used, e.g., as a bootstrapping mechanism to get assets on a new chain. In Proofs-of-Burn Glimpse is used as for backed assets, with the only difference being that funds are moved unidirectionally.

Proofs of Oracle Attestations. Let us assume there exist on $\mathcal{L}_S$ k different oracles posting information about real-world events (e.g., real-time prices for currencies). On $\mathcal{L}_D$, a user wants to verify k oracle attestations for a specific event. In this case, Glimpse can be used to verify the formula $\mathcal{F}_S = (\mathsf{Desc}_1 \wedge \cdots \wedge \mathsf{Desc}_k)$, with Desc_i being the description of the transaction published by the i-th oracle O_1. We note that oracles do not have to cooperate, nor to operate on the same chain.

Off-chain Glimpse and its Applications. State channels [30–32] and generalized channels [15] enable Payment Channel Networks (PCNs) that offer off-chain the same functionality as the underlying chain. We can thus host Glimpse on PCNs, thereby remarkably improving its performance and scalability, and enabling a new range of applications on layer-2. In particular, instead of posting the contract on $\mathcal{L}_D$, Glimpse can be encoded in a standard channel update, be kept off-chain, and posted on-chain only to resolve disputes. In this way, the contract can be iteratively updated off-chain within subsequent channel updates, allowing for changes of the Glimpse specifics on the run. For instance, one could efficiently and securely extend the contract lifetime by updating the randomness in the contract as well as inside the to-be-verified transaction,[6] while accordingly adjusting the number of confirmation blocks for the proof. E.g., if initially $T = 2$ hour and $n = 10$, one can update the randomness and extend T by 1 hour minutes only asking for $n = 5$ confirmation blocks: security is preserved and the proof verification cost is decreased. Overall, hosting Glimpse contracts on PCNs dramatically improves its practicality (parameters changed on the run to accommodate users' needs) and its cost (no on-chain transactions in the optimistic case).

Application-wise, payment channel hubs may employ Glimpse contracts to set up betting hubs, where users connected to the hub bet on a certain Tx_S being published on-chain within an absolute time T. If Tx_S is published, the users and the hub reflect the correct outcome in a channel update within time $t < T$. If any party (user or hub) misbehave, the counterparty can post the contract on-chain (thereby closing the channel) and enforce the correct outcome. Glimpse also provides an out-of-the-box solution for enabling off-chain applications synchronized to an on-chain event. Examples include cross-chain payments based on [30,33] or cross-chain virtual channels based on [34].

5 Security in the UC Framework

We model Glimpse in the synchronous Global Universal Composability (GUC) framework [35], closely following prior work [15, 31, 32]. First, we state that according to the basic assumptions in Section 3.1, Glimpse achieves weak

[6] We note that this can be done only if the particular application for which Glimpse is used allows for updates of the randomness in the not-yet-posted transaction to be verified.

atomicity. Next, we show that by additionally assuming liveness of the parties and direct access to $\mathcal{L}_S$ for P (and V, in case of DNF verification), Glimpse achieves strong atomicity.

We use a global clock $\mathcal{G}_{Clock}$ [35] and authenticated channels with guaranteed delivery $\mathcal{G}_{GDC}$ [32] to model time and communication. We assume a static corruption model, where the adversary decides which set of parties to corrupt before the execution of the protocol. We use the instantiation of the functionality $\mathcal{G}_{Ledger}$ defined in [36] to model a ledger $\mathcal{L}$, where the parameters are chosen such that the ledger achieves both *liveness* and *consistency* as defined in [26]. We define two 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 of DNF verification, respectively. More concretely, the ideal functionality is parameterized over two ledgers $\mathcal{L}_S$ or $\mathcal{L}_D$. After P and V parties have registered to it, the functionality ensures that the respective transactions are posted on $\mathcal{L}_S$ or $\mathcal{L}_D$, 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 probabilistic polynomial time *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 the functionalities $\mathcal{G}_{Clock}$ and $\mathcal{G}_{GDC}$, the protocol Π is instantiated with two ledger instantiations $\mathcal{G}_{Ledger}$ for $\mathcal{L}_S$ and $\mathcal{L}_D$, and has strictly randomized inputs. Let $\Delta_D \in \mathbb{N}$ be the wait time of $\mathcal{L}_D$ and $gen\mathcal{P}$ a proof generation function that is T-sound. Then, the protocol Π UC-realizes the ideal functionality $\mathcal{F}_{W-Glimpse}$.*

Theorem 2. *Given the functionalities $\mathcal{G}_{Clock}$ and $\mathcal{G}_{GDC}$, the protocol Π is instantiated with two ledger instantiations $\mathcal{G}_{Ledger}$ for $\mathcal{L}_S$ and $\mathcal{L}_D$, and has strictly randomized inputs Let $\Delta_D \in \mathbb{N}$ be the wait time of $\mathcal{L}_D$ and $gen\mathcal{P}$ a proof generation function that is complete and T-sound. **All parties have direct access to $\mathcal{L}_S$ and $\mathcal{L}_D$, and they 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. As for light clients, the security of Glimpse relies on the assumption that the underlying chains operate under a well-designed incentive mechanism that ensures an honest majority of miners, thereby guaranteeing consistency and liveness.

However, introducing cross-chain or cross-layer applications brings forth new external profit opportunities for miners. If the total value locked in the application exceeds the amount of reward offered by the internal incentive mechanism, miners may stop adhering to the protocol rules with the purpose of maximizing their profit. This is true for *all cross-chain and cross-layer applications and bridges* [37]: atomic swaps [38], chain relays [6], payment channels [39], bridges [9, 40], etc. In particular, Glimpse and chain relays equally suffer from forgery attacks, where miners use their computational power to forge a fake subchain of blocks (suffix) rather than mining honestly.

In this section, we study the cost for an adversary bribing miners to mount a proof forgery attack, thereby compromising the security of Glimpse. Furthermore, we study the cost for an adversary bribing miners to mount a censorship attack towards, e.g., $\mathsf{Tx_P}$ or $\mathsf{Tx_V}$, thereby compromising the security of Glimpse and violating the liveness of the underlying blockchain. For these attacks, we define the *secure parameter space* specific to Glimpse.

6.1 Proof Forgery Attack

In a *proof forgery attack* a malicious prover (attacker) bribes the miners of the source chain to convince them forging a fake proof, i.e., an invalid extension of the longest chain. With such a fake proof, the attacker can fool the Glimpse contract and steal the funds in it. This attack, however, is not limited to Glimpse, but threatens light clients and chain relays as well. Light clients operate under the assumption that the majority of the mining power is in the hands of honest miners, resulting in a good chain quality [26]. If this assumption is broken, a light client can also be fooled to accept a fake n-block suffix. This attack becomes profitable if the total value locked in Glimpse (or in any cross-chain applications relying on a light client or chain relay)[7] is larger than the profit miners make when mining honestly.

Attack Strategy. Let us consider a powerful attacker consisting of all provers having an *active* Glimpse contract on the *same* $\mathcal{L}_S$ but (potentially) *different* $\mathcal{L}_D$. The attacker bribes the miners of $\mathcal{L}_S$ to forge a proof for all these Glimpse contracts at once. The bribe consists of the coins held by all the active Glimpse contracts over the considered $\mathcal{L}_D$. The miners following the attack can optimize their computational effort by forging *a single proof for all the contracts*: they can create a single fake block B^f including all the transactions the attacker wants them to include (the transactions the Glimpse contracts are conditioned on), and then mine n confirmation blocks on

[7] Without creating any fork on the main chain, corrupted miners can create a fake light client (or chain relay) suffix that will not be part of the blockchain as, e.g., it contains invalid transactions which are rejected by full nodes.

top of B^f in time T, with n and T being averaged over the active Glimpse instances.

Total Value Locked in Active Glimpse Contracts. To understand when this attack constitutes a real threat, we first need to know the economic resources the attacker has at their disposal for bribing the miners. This is given by the *total value locked* in *all* the *simultaneously active* Glimpse contracts operating over the same $\mathcal{L}_S$. We denote the total value with α_T. Note that α_T must also take into account similar active cross-chain protocols relying on the same proof design [41], i.e., checking that enough PoW has been done within a fixed time frame.

Computing α_T requires monitoring all the destination chains which support Glimpse, and look for the *active* Glimpse contracts (we recall that, for practicality, Glimpse contracts are meant to be active only for a short time). As this could be unpractical, we propose possible alternative solutions: honest parties may use a public bulletin board where they announce their Glimpse contracts (we assume at least one between P and V is honest, see Section 3.1) or, alternatively, use some heuristics to estimate α_T, e.g., computing the total value for the most used Glimpse destination chain and multiplying it by the number of compatible chains. For the analysis that follows, we assume honest parties are able to compute α_T. We leave a more rigorous analysis of how one can compute α_T in the face of such a powerful adversary as future work.

Model. We require the number n of confirmation blocks for Glimpse to be at least equal to the minimum number of blocks for which the probability of a temporary fork (or ordinary block reorganization) is negligible. With this, we prevent a malicious P fooling the contract by submitting a proof taken from an orphaned branch of $\mathcal{L}_S$. Finally, we require the probability of n being larger than the number of honest blocks mined for $\mathcal{L}_S$ in T to be negligible. With this, we exclude P from being unable to construct a proof because n is larger than the number of blocks honestly appended to $\mathcal{L}_S$ over the time window T.

To study the proof forgery attack, we need to consider the expected gain for honest miners ($\mathbb{E}[\mathsf{H}]$) and the expected gain for corrupted miners mounting the forgery attack ($\mathbb{E}[\mathsf{F}]$). The first represents the money miners would get from following the protocol rules (block rewards and transaction fees). The second is the profit from the attack, i.e., the value of the bribe that miners would get from the attacker if they successfully forge the proof. The miners' total profit is therefore given by $\mathbb{E}[\mathsf{F}] - \mathbb{E}[\mathsf{H}]$. If the total profit is positive, then the attacker (and cooperating miners) is incentivized to mount the attack.

Expected Gain for Honest Miners. Let R be the block reward (in USD) on $\mathcal{L}_S$, $\mathbb{E}[\mathsf{B}]$ the number of expected blocks on $\mathcal{L}_S$ in T, and $\mu_r \leq 1-\gamma$ [8] the attacker's *relative* mining power on $\mathcal{L}_S$ (which gives the attacker's probability of finding a valid block). The expected gain for honest miners is:

$$\mathbb{E}[\mathsf{H}] = \mathsf{R} \cdot \mathbb{E}[\mathsf{B}] \cdot \mu_r. \tag{1}$$

Expected Gain for Corrupted Miners. The expected gain for corrupted miners depends on n, on their mining power, and on the fluctuation (in USD) of the bribe value during the time window T of the attack. Let μ be the corrupted miners' hashing power (in hashes per second); then, the number of hashes computed in T is given by $\mathsf{N} := \mu \cdot T$. We consider N repeated, independent, and equally distributed hash evaluations, and we let $P_{\mathcal{T}}$ be the probability of finding a hash smaller than a target $\mathcal{T}$. In time T, miners will be able to forge a proof consisting of n confirmation blocks with (binomial) probability given by:

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

Let α_T be the bribe the attacker offers to miners, and δ be the total expected percentage price drop (over all different $\mathcal{L}_D$) of the bribe after time T. The expected gain for corrupted miners is given by:

$$\mathbb{E}[\mathsf{F}] = \alpha_T \cdot P_{n,\mathcal{T}} \cdot (1-\delta). \tag{3}$$

We observe that if corrupted miners hold a significant share of the computational power of the network, this attack becomes detectable, undermining users' trust in the chain.

Secure Parameter Space. Glimpse is economically secure when it is more profitable for miners to honestly mine blocks rather than mounting a proof forgery attack. In other words, Glimpse is secure when the total profit for the attacker (and cooperating miners) is negative, i.e., when $\mathbb{E}[\mathsf{F}] < \mathbb{E}[\mathsf{H}]$. This inequality yields the *upper bound* for the total value α_T locked in *simultaneously active* Glimpse contracts over the same $\mathcal{L}_S$:

$$\alpha_T < \frac{\mathsf{R} \cdot \mathbb{E}[\mathsf{B}] \cdot \mu_r}{P_{n,\mathcal{T}} \cdot (1-\delta)}. \tag{4}$$

Before opening a new Glimpse contract of value α over $\mathcal{L}_S$, an honest party must first compute the total value α_T locked in all the active Glimpse over $\mathcal{L}_S$, and make sure that the new total value locked, i.e., $\alpha + \alpha_T$, fulfills the inequality in Equation (4). We stress that α_T is the *upper bound at each point in time* and that, for practicality, Glimpse contracts are meant to have a short lifetime. Put differently, Glimpse is a dynamic and fast protocol that can move large amounts of money capped by α_T at each point in time. The value α_T may considerably fluctuate depending on the underlying chains, e.g., the block reward of the source chain, and on the market conditions, e.g., the values (in USD) of the assets involved.

Example. Let us assume all active Glimpse contracts have Bitcoin as $\mathcal{L}_S$ and the Liquid Network as $\mathcal{L}_D$. These contracts have, on average, $n = 5$ and $T = 1$ hour. We consider a Bitcoin

[8] $\gamma \in [0,1]$ is the fraction of honest miners the blockchain can tolerate.

target having 19 leading zeros (the largest in 2022), an attacker controlling 23% of the total hashing power (largest mining pool in 2022), and the average prices for BTC and L-BTC at November 2022. Without price drop, $\alpha_T \sim 230$ million USD as of January 2023: this upper bound compares to the one for other bridges, e.g., Gravity [40, 42]. However, while in Glimpse funds are locked for a short time and α_T is the maximum at each point in time, other bridges (e.g., Gravity) have worst performances, as funds are locked for long periods.

6.2 Censorship Attack

Glimpse makes use of a timelock and, as any other protocol relying on timelocks [38, 39, 43], it suffers from *censorship attacks*, i.e., attacks on the liveness of the underlying chains. We investigate how an attack on the liveness of the destination chain harms Glimpse, and we estimate the cost of mounting such an attack.

In a censorship attack, a malicious verifier (attacker) bribes block proposers (validators in Proof-of-Stake, or miners, in PoW) to not include a specific transaction (in our case, e.g., $\mathsf{Tx_P}$) on chain. Rational proposers, however, will only censor $\mathcal{L}_D$ if they gain something from doing so. Therefore, the economic security of Glimpse depends on the conditions making this attack profitable for V (and the proposers).

Closely following [38], we define the *bribing game* as a Markov game running in $T+1$ sequential stages, where a stage is the period between two blocks. In each stage, the block proposer chooses between censoring the transaction pointed out by V (playing the game), or including the transaction in the block (refusing to play the game). The bribing game is *safe* if, after eliminating the strictly dominated strategies, the only action left in stage one for each block proposer is to *refuse* the bribery and include the transaction.

The analysis stems from these assumptions: (i) block proposers are rational, i.e., they always try to maximize their profit and, if they can choose, they always follow dominant strategies; (ii) block proposers do not create forks; (iii) the probability μ_r of each player to be selected as block proposer is publicly known and is constant during the attack; (iv) the attacker and the victim are not block proposers; (v) all block proposers can see timelocked transactions that will be valid in the future; (vi) the Glimpse lifetime T is a timelock expressed in number of blocks; finally, (vii) block rewards and fees generated outside the Glimpse protocol are constant and do not affect the attack.

A *weak block proposer* as a player whose probability to be selected as the next block proposer is $\mu_r < \frac{f}{\alpha}$. We let μ_w be the sum of the probabilities of all weak block proposers in the system, f be the fee of the to-be-censored transaction, and α the value of the bribe, i.e., the economic value hold by the (*single*!) Glimpse contract under attack. Let $\alpha > f$. As proved in [38], the following theorem holds:

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

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

Secure Parameter Space. Glimpse is secure from censorship attacks when $\alpha < \frac{f}{\mu_r}$ and T fulfills Equation (5).

Example. For instance, considering a $2 transaction fee in Ethereum and the lowest probability to be selected as block proposer being $1.8 \cdot 10^{-6}$ [44], each Glimpse contract in Ethereum can hold at most 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

On-chain Costs for EVM Chains. We now consider Ethereum, but a similar discussion also applies to any EVM-based chain. In Ethereum, the cost of a transaction is measured in *gas*: every computation consumes an amount of gas proportional to its complexity, and the data stored on-chain consume an amount of gas proportional to its byte length. The computational cost of Glimpse stems from the proof verification, which consists of a Merkle proof verification, a transaction body and a block header reconstruction, and hash comparisons (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, as shown in Table 3. Each of the n confirmation blocks in $\mathcal{P}^n$ yields an overhead of 36k gas for the hash comparison.

Besides these computational costs, the Glimpse contract has to store the PoW target of the source chain as well as the hash of the to-be-verified transaction(s) - or the transaction description(s). 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 approximately 188k gas in the case of target and a ∼300-bytes description. We have implemented an open-source cost evaluation which can be found in a Github repository [45].

Glimpse has lower on-chain costs compared to optimized relay solutions, such as Ethrelay [6] and zkRelay [5] (presented in Appendix A), and zkBridge. With 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. With zkRelay, the submission of a batch of blocks costs 522k gas, including the verification of the zero-knowledge proof and the storage costs. The proof verification 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 two Merkle proofs: one for verifying the block inclusion in

the batch, and one for verifying the transaction inclusion in the block. With zkBridge, each zk proof verification alone already results in 230k gas.

While the costs of relays and zkBridge for verifying transaction inclusion in a block are analogous to the ones of Glimpse, the crucial difference lies in the maintenance costs. Chain relays continuously incur in high maintenance costs for *relaying, verifying, and storing the full list of block headers of* $\mathcal{L}_S$; similarly, zkBridge requires relayers to continuously relay block headers, compute zk proofs, and submit the headers and the corresponding proofs on-chain. Glimpse, on the other hand, does not have any maintenance cost because of its on-demand nature. With Glimpse, the verification of a single fully known transaction via a proof comprising of 5 confirmation blocks has an upper bound of 330k gas: we stress that this is a *one-time* fee, compared to the continuous 280k gas for each block header submission of Ethrelay and 522k gas per batch submission of zkRelay.

On-chain Costs for Bitcoin-based Chains. In Bitcoin-based chains, transaction fees are usually proportional to the size in bytes of the transaction. In Bitcoin, for instance, this results in a few satoshi per byte as of November 2022.

To cope with the limited scripting capabilities, Glimpse parties can use Taproot, which using Merkelized Abstract Syntax Trees (MAST) [46] allows to commit to multiple scripts within a single one, i.e., a Pay-To-Taproot (P2TR) script which contains the root of the tree. The size of a MAST for Glimpse depends on (i) the number of undefined inputs and outputs in Desc, because their well-formedness needs to be checked with dedicated opcodes, (ii) the number of confirmation blocks in $\mathcal{P}^n$, because their hashes have to be compared with the PoW target, (iii) the number of transactions within the block, because it affects the number of levels in the tree, and (iv) the size of Desc, being the description hard-coded in the script. For example, for a single to-be-verified transaction Tx of $\sim$350 bytes, 6 confirmation blocks, and one parameterized input or output, the upper bound for the MAST is 10MB. For DNF formulas, parties need to compute and exchange the MAST for each literal. For a detailed discussion see Appendix E.

We theoretically estimate the size of transactions Tx_G, Tx_P, and Tx_V on the Liquid Network, where Taproot and all necessary string opcodes are available. Assuming Tx_G has two P2PKH inputs and one P2TR output, the transaction size is approximately 350 bytes. Assuming Tx_P has one P2TR input and two P2PKH outputs, the size is again roughly 350 bytes. Instead, assuming Tx_V has one P2TR input and two P2PKH outputs, its size is of 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\$, similar to the costs of standard Bitcoin transactions.

Computational Overhead. The computational overhead is minimal: parties need to create and verify signatures, and construct proofs. If the destination chain is the Liquid Network, parties also need to construct and verify the MAST. All these operations can be performed using commodity hardware.

No. Tx in B	$\mathcal{P}^{n=0}$	$\mathcal{P}^{n=5}$
$2^6{+}1 \leq \mathsf{Tx} \leq 2^7$	92k gas	273k gas
$2^7{+}1 \leq \mathsf{Tx} \leq 2^8$	95k gas	276k gas
$2^8{+}1 \leq \mathsf{Tx} \leq 2^9$	99k gas	280k gas
$2^9{+}1 \leq \mathsf{Tx} \leq 2^{10}$	103k gas	283k gas
$2^{10}{+}1 \leq \mathsf{Tx} \leq 2^{11}$	106k gas	287k gas
$2^{11}{+}1 \leq \mathsf{Tx} \leq 2^{12}$	111k gas	291k gas

Table 3: On-chain costs for EVM chains for proof verification with $n = 0$ and $n = 5$, for different number of Tx in B.

Communication Overhead. We now discuss the communication overhead for Glimpse. In the *Setup* phase, parties need to exchange Tx_G, Tx_P and Tx_V, as well as the descriptions of the to-be-verified transactions. Verifying a DNF formula with m literals requires the parties to exchange $4 \cdot 2^m$ transactions and the respective signatures. For Bitcoin-based chains supporting Taproot, parties also need to exchange the MAST, which roughly amounts to 10MB.

8 Conclusion

We present Glimpse, an *on-demand light client for cross-chain communication* that only requires constant-size storage. Glimpse enables many applications such as lending, Proofs-of-Burn, proofs of oracle attestations, and off-chain applications, while remarkably retaining low on-chain costs and *compatibility* with chains having limited scripting capabilities. The security and atomicity properties of Glimpse are proven within the UC framework. By conducting an economic analysis which considers rational players, we provide the secure parameter space for Glimpse.

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, 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," 2022. [Online]. Available: https://rb.gy/hvo01

[2] "Hackers have stolen \$1.4 billion this year using crypto

bridges. Here's why it's happening," 2022, https://shorturl.at/yGJT3.

[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.

[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, "ETHRelay: 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.

[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.

[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.

[10] M. Bartoletti and R. Zunino, "BitML: A calculus for bitcoin smart contracts," in *ACM SIGSAC Conference on Computer and Communications Security, CCS*, 2018.

[11] "How to validate Bitcoin payments in Ethereum (for only 700k gas!)," 2018, https://medium.com/summa-technology/cross-chain-auction-technical-f16710bfe69f.

[12] "Summa," https://github.com/summa-tx/bitcoin-spv.

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

[14] 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.

[15] 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.

[16] "The Liquid Network," https://blockstream.com/liquid/.

[17] "Blockstream," https://blockstream.com.

[18] "What the heck is SegWit," 2020, https://medium.com/bitbees/what-the-heck-is-segwit-3f58b7352b1c.

[19] "Bitcoin's Taproot upcoming upgrade and how it matters to the network," 2021, https://tokenize.exchange/blog/article/bitcoin-taproot-upcoming-upgrade.

[20] "Salvador Bitcoin Bonds," 2021, https://rb.gy/fcku5.

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

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

[23] 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.

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

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

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

[27] G. Scaffino, L. Aumayr, Z. Avarikioti, and M. Maffei, "Glimpse: On-demand PoW light client with constant-size storage for DeFi," Cryptology ePrint Archive, Paper 2022/1721, 2022, https://eprint.iacr.org/2022/1721.

[28] T. Dryja, "Discreet Log Contracts," https://adiabat.github.io/dlc.pdf.

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

[30] 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*.

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

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

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

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

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

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

[37] "Vitalik Buterin on cross-chain applications," https://rb.gy/hvo01, 2022.

[38] T. Nadahalli, M. Khabbazian, and R. Wattenhofer, "Timelocked bribing," in *Financial Cryptography and Data Security*, N. Borisov and C. Diaz, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2021.

[39] 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.

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

[41] "Summa proofs are not composable," 2019. [Online]. Available: https://medium.com/@dionyziz/summa-proofs-are-not-composable-57b87825f428

[42] "Value locked in Ethereum L1 bridges," 2023, https://www.theblock.co/data/scaling-solutions/scaling-overview/value-locked-of-ethereum-l1-bridges.

[43] 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.

[44] "Beacon scan," https://beaconscan.com/statistics.

[45] "Glimpse Github," https://github.com/Glimpse-CrossChainPrimitive/Glimpse.

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

[47] 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.

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

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

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

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

[52] 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.

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

[54] "Submarine swap in Lightning Network," https://wiki.ion.radar.tech/tech/research/submarine-swap, 2021.

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

[56] M. Westerkamp and M. Diez, "Verilay: A verifiable Proof of Stake chain relay," in *IEEE International Conference on Blockchain and Cryptocurrency, ICBC*, 2022.

[57] T. Bugnet and A. Zamyatin, "XCC: Theft-resilient and collateral-optimized cryptocurrency-backed assets," Cryptology ePrint Archive, 2022.

[58] "Bitcoin Wiki: Payment channels," 2018, https://en.bitcoin.it/wiki/Payment_channels.

[59] 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.

[60] 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.

[61] 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.

[62] 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.

[63] "The Bitcoin Script language," https://betterprogramming.pub/the-bitcoin-script-language-e4379908448f, 2021.

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

[65] "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.

Frauenthaler et al. [6] propose Ethrelay, a relay adopting an optimistic approach: Block headers are optimistically accepted 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. [47] 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.

	Commit on $\mathcal{L}_S$			*Ver.&Comm. on* $\mathcal{L}_D$		Expressiveness
	Ass.	SR	Consensus	Ass.	SR	
Universal Atomic Swap [52]	Sync	①	Any	Sync	①	Secret-based logic
HTLC-based Swap [49, 54, 55]	Sync	①	Any	Sync	①	Secret-based logic
Glimpse	Sync	①	PoW	Sync	②	DNF formulas
Chain relays [4–6, 56]	Sync	③	PoW,PoS	Sync	③	Arbitrary logic
XCLAIM [47], XCC [57]	TTP	①	PoW,PoS	Sync	③	Arbitrary logic
Gravity Bridge [40]	TTP	③	PoS,BFT	TTP	③	Arbitrary logic

Table 4: State-of-the-art CCC protocols w.r.t.: (i) the assumption they make (Trusted Third Party (TTP) or Synchrony), (ii) their scripting requirements (SR), (iii) the consensus they operate on, and the expressiveness they achieve.

Gravity [40] 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. Gravity assumes an honest super majority of validators.

Another conceptually and technically different solution for cross-chain communication is atomic swaps, which likely originated from a forum user TierNolan [48] and was later analyzed by, e.g., Herlihy [49] or Xu et al. [50]. Atomic swaps allow multiple parties to exchange assets across multiple blockchains in a distributed and coordinated manner. There exist numerous proposals for atomic swaps, e.g., Gugger [51] proposed a construction compatible with Monero. Thyagarajan et al. [52] 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. [53] proposed atomic swaps that require no timelock (or even timelock puzzle) on one of the chains.

Table 4 compares Glimpse to other state-of-the-art cross-chain solutions. 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[9],

[9] HTLCs are contracts storing a pair (h,t) and ensuring that if the contract

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 [30, 58], multi-hop payments [33, 59], virtual channels [32, 34, 60, 61], and discreet log contracts [28]. 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 [47]. 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 $\mathcal{U}$ 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 model) 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 [32, 62], 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.

receives the secret s such that $h = \mathcal{H}(s)$ before time t, then the ownership of the asset locked in the contract is transferred to the counter party.

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., [32]), 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 [36]. Concretely, we use a concrete instantiation also as specified in [36], where the parameters are chosen such that the ledger achieves both *liveness* and *consistency* (or just consistency), which is defined in [26]. 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 $\mathtt{EXEC}^{\mathcal{F}_{prelim}}_{\Pi,\mathcal{A},\mathcal{Z}}(\lambda, z)$ 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 $\mathtt{EXEC}^{\mathcal{F}_{prelim}}_{\phi_{\mathcal{F}},\mathcal{S},\mathcal{Z}}(\lambda, z)$.

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

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

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 $(\texttt{SETUP}, \texttt{id}, \mathcal{F}_S, P, [\mathsf{outcome}_i]_{i\in[1,r]}, T_P, T_V, n, \{inputP\}, \{inputV\}) \xleftarrow{\tau} V$, 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 $(\texttt{id}, \mathcal{F}_S, [\mathsf{outcome}_i]_{i\in[1,r]}, T_P, T_V, n, \{inputP\}, \{inputV\}) \xhookrightarrow{\tau+1} P$, receive $(\texttt{id}) \xhookleftarrow{\tau+1} P$.
3. At round $\tau_1 \leq \tau + 1 + \Delta_D$, if a transaction 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 $(\texttt{id}, \mathcal{F}_S, T_P, T_V, n, P, V, [\mathsf{outcome}_i]_{i\in[1,r]}, \mathsf{Tx}_\mathsf{G})$ in Φ.

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

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

For every $(\texttt{id}, \mathcal{F}_S, T_P, T_V, n, P, V, [\mathsf{outcome}_i]_{i\in[1,r]}, \mathsf{Tx}_\mathsf{G})$ 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 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. $[\mathsf{Tx}_i] \hookrightarrow \mathsf{Desc}$, 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. $[\mathsf{Tx}_j] \hookrightarrow \mathsf{Desc}$ on $\mathcal{L}_S$. If this does not hold, output $(\texttt{id}, \texttt{error})$.

(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 $(\texttt{id}, \mathcal{F}_S, T_P, T_V, n, P, V, [\mathsf{outcome}_i]_{i\in[1,r]}, \mathsf{Tx}_\mathsf{G})$ in Φ where P 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 $[\mathsf{Tx}_i] \hookrightarrow \mathsf{Desc}$. 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 $(\texttt{id}, \texttt{error})$.

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

For every $(\texttt{id}, \mathcal{F}_S, T_P, T_V, n, P, V, [\mathsf{outcome}_i]_{i\in[1,r]}, \mathsf{Tx}_\mathsf{G})$ 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 $[\mathsf{Tx}_i] \hookrightarrow \mathsf{Desc}$. 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 $(\texttt{id}, \texttt{error})$.

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 $genP$, 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 $genP$**.** 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 separately sampling a value of length λ uniformly at random $r_P \leftarrow^{\$} \{0,1\}^{\lambda}$ and $r_V \leftarrow^{\$} \{0,1\}^{\lambda}$, concatenate them yielding $r_P || r_V$ and add an output $(0, \texttt{OP\_RETURN}(r_P || r_V))$.

Definition 7. A Glimpse protocol Π has *strictly randomized inputs*, 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 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 $\mathcal{P}_i^n(\mathsf{Tx}_i, \mathsf{Desc}_i, n)$ 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. $(\mathsf{Tx}_\mathsf{G}, \{(\mathsf{Tx}_{\mathsf{T}i}, \mathsf{Tx}_{\mathsf{F}i}, \mathsf{Tx}_{\mathsf{P}i})\}_{i\in[1,row]}) := \mathsf{genTxsFromF}(\alpha, \mathcal{F}_S, [\mathsf{outcome}_i]_{i\in[1,r]}, T_P, T_V, cp, n, \{\mathtt{inputs}\})$
3. Sign each transaction $\mathsf{Tx}_{\mathsf{F}i}$ and $\mathsf{Tx}_{\mathsf{P}i}$ in $\{(\mathsf{Tx}_{\mathsf{T}i}, \mathsf{Tx}_{\mathsf{F}i}, \mathsf{Tx}_{\mathsf{P}i})\}_{i\in[1,row]}$ and append the signatures to each tuple yielding a set $\{(\mathsf{Tx}_{\mathsf{T}i}, \mathsf{Tx}_{\mathsf{F}i}, \mathsf{Tx}_{\mathsf{P}i}, \sigma_V(\mathsf{Tx}_{\mathsf{F}i}), \sigma_V(\mathsf{Tx}_{\mathsf{P}i}))\}_{i\in[1,row]}$
4. Sign Tx_G yielding $\sigma_V(\mathsf{Tx}_\mathsf{G})$
5. Send $(\mathtt{open\text{-}req}, \mathtt{id}, [\mathsf{outcome}_i]_{i\in[1,r]}, T_P, T_V, n, \{inputP\}, \{inputV\}, \mathsf{Tx}_\mathsf{G}, \sigma_V(\mathsf{Tx}_\mathsf{G}), \{(\mathsf{Tx}_{\mathsf{T}i}, \mathsf{Tx}_{\mathsf{F}i}, \mathsf{Tx}_{\mathsf{P}i}, \sigma_V(\mathsf{Tx}_{\mathsf{F}i}), \sigma_V(\mathsf{Tx}_{\mathsf{P}i}))\}_{i\in[1,row]}) \overset{\tau_V}{\hookrightarrow} P$.
6. Add $(\mathtt{id}, \mathcal{F}_S, T_P, T_V, n, \mathsf{Tx}_\mathsf{G})$ to Φ_V.

Prover P

7. Upon $(\mathtt{open\text{-}req}, \mathtt{id}, [\mathsf{outcome}_i]_{i\in[1,r]}, T_P, T_V, n, \{inputP\}, \{inputV\}, \mathsf{Tx}_\mathsf{G}, \sigma_V(\mathsf{Tx}_\mathsf{G}), \{(\mathsf{Tx}_{\mathsf{T}i}, \mathsf{Tx}_{\mathsf{F}i}, \mathsf{Tx}_{\mathsf{P}i}, \sigma_V(\mathsf{Tx}_{\mathsf{F}i}), \sigma_V(\mathsf{Tx}_{\mathsf{P}i}))\}_{i\in[1,row]}) \overset{\tau_P}{\hookleftarrow} V$.
8. Perform checks of protocol Setup phase, steps 1a through 1h. If one or more fail, go idle.
9. Verify that $(\mathsf{Tx}_\mathsf{G}, \{(\mathsf{Tx}_{\mathsf{T}i}, \mathsf{Tx}_{\mathsf{F}i}, \mathsf{Tx}_{\mathsf{P}i})\}_{i\in[1,row]})$ 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 $\{(\mathsf{Tx}_{\mathsf{T}i}, \mathsf{Tx}_{\mathsf{F}i}, \mathsf{Tx}_{\mathsf{P}i}, \sigma_V(\mathsf{Tx}_{\mathsf{F}i}), \sigma_V(\mathsf{Tx}_{\mathsf{P}i}))\}_{i\in[1,row]}$, check that $\sigma_V(\mathsf{Tx}_{\mathsf{F}i})$ and $\sigma_V(\mathsf{Tx}_{\mathsf{P}i})$ are valid signatures of V for $\mathsf{Tx}_{\mathsf{F}i}$ and $\mathsf{Tx}_{\mathsf{P}i}$, respectively.
11. Verify that $\sigma_V(\mathsf{Tx}_\mathsf{G})$ is a valid signature of V for 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 Tx_G yielding $\sigma_P(\mathsf{Tx}_\mathsf{G})$.
14. Send a message "post" for transaction Tx_G with $\sigma_P(\mathsf{Tx}_\mathsf{G})$ and $\sigma_V(\mathsf{Tx}_\mathsf{G})$ 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 $(\mathtt{id}, \mathcal{F}_S, T_P, T_V, n, \mathsf{Tx}_\mathsf{G}, \{(\mathsf{Tx}_{\mathsf{T}i}, \mathsf{Tx}_{\mathsf{F}i}, \mathsf{Tx}_{\mathsf{P}i}, \sigma_V(\mathsf{Tx}_{\mathsf{F}i}), \sigma_V(\mathsf{Tx}_{\mathsf{P}i}))\}_{i\in[1,row]})$ in Φ_P.

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

For every $(\mathtt{id}, \mathcal{F}_S, T_P, T_V, n, \mathsf{Tx}_\mathsf{G}, \{(\mathsf{Tx}_{\mathsf{T}i}, \mathsf{Tx}_{\mathsf{F}i}, \mathsf{Tx}_{\mathsf{P}i}, \sigma_V(\mathsf{Tx}_{\mathsf{F}i}), \sigma_V(\mathsf{Tx}_{\mathsf{P}i}))\}_{i\in[1,row]})$ in Φ_P, execute $\mathsf{postTxsFP}(\mathcal{F}_S, T_P, n, \mathsf{Tx}_\mathsf{G}, \{(\mathsf{Tx}_{\mathsf{T}i}, \mathsf{Tx}_{\mathsf{F}i}, \mathsf{Tx}_{\mathsf{P}i}, \sigma_V(\mathsf{Tx}_{\mathsf{F}i}), \sigma_V(\mathsf{Tx}_{\mathsf{P}i}))\}_{i\in[1,row]})$.

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

For every $(\mathtt{id}, \mathcal{F}_S, T_P, T_V, n, \mathsf{Tx}_\mathsf{G})$ 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 $(\texttt{open-req}, \texttt{id}, [\mathsf{outcome}_i]_{i\in[1,r]}, T_P, T_V, n, \{inputP\}, \{inputV\}, \mathsf{Tx_G}, \sigma_V(\mathsf{Tx_G}), \{(\mathsf{Tx}_{\mathsf{T}i}, \mathsf{Tx}_{\mathsf{F}i}, \mathsf{Tx}_{\mathsf{P}i}, \sigma_V(\mathsf{Tx}_{\mathsf{F}i}), \sigma_V(\mathsf{Tx}_{\mathsf{P}i}))\}_{i\in[1,row]}) \overset{\tau_V}{\hookrightarrow}$ P, perform checks of protocol Setup phase, steps 1a through 1h and perform the following checks.
2. Verify that $(\mathsf{Tx_G}, \{(\mathsf{Tx}_{\mathsf{T}i}, \mathsf{Tx}_{\mathsf{F}i}, \mathsf{Tx}_{\mathsf{P}i})\}_{i\in[1,row]})$ is the result of $\mathsf{genTxsFromF}(\alpha, \mathcal{F}_S, [\mathsf{outcome}_i]_{i\in[1,r]}, T_P, T_V, cp, n, \{\texttt{inputs}\})$.
3. For each entry of the set $\{(\mathsf{Tx}_{\mathsf{T}i}, \mathsf{Tx}_{\mathsf{F}i}, \mathsf{Tx}_{\mathsf{P}i}, \sigma_V(\mathsf{Tx}_{\mathsf{F}i}), \sigma_V(\mathsf{Tx}_{\mathsf{P}i}))\}_{i\in[1,row]}$, check that $\sigma_V(\mathsf{Tx}_{\mathsf{F}i})$ and $\sigma_V(\mathsf{Tx}_{\mathsf{P}i})$ are valid signatures of V for $\mathsf{Tx}_{\mathsf{F}i}$ and $\mathsf{Tx}_{\mathsf{P}i}$, respectively.
4. Verify that $\sigma_V(\mathsf{Tx_G})$ is a valid signature of V for $\mathsf{Tx_G}$.
5. If these checks succeed, send $(\texttt{SETUP}, \texttt{id}, \mathcal{F}_S, P, [\mathsf{outcome}_i]_{i\in[1,r]}, T_P, T_V, n, \{inputP\}, \{inputV\}) \overset{\tau_V}{\hookrightarrow} \mathcal{F}_{Glimpse}$ on V's behalf. Otherwise go idle.
6. Upon P sending $(\texttt{id}) \overset{\tau_P:=\tau_V+1}{\longrightarrow} \mathcal{F}_{Glimpse}$, sign $\mathsf{Tx_G}$ on P's behalf yielding $\sigma_P(\mathsf{Tx_G})$. Else go idle.
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 $\Phi(P) := (\texttt{id}, \mathcal{F}_S, T_P, T_V, n, \mathsf{Tx_G}, \{(\mathsf{Tx}_{\mathsf{T}i}, \mathsf{Tx}_{\mathsf{F}i}, \mathsf{Tx}_{\mathsf{P}i}, \sigma_V(\mathsf{Tx}_{\mathsf{F}i}), \sigma_V(\mathsf{Tx}_{\mathsf{P}i}))\}_{i\in[1,row]})$.

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

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

P is honest

For every (key, value) pair P, $(\texttt{id}, \mathcal{F}_S, T_P, T_V, n, \mathsf{Tx_G}, \{(\mathsf{Tx}_{\mathsf{T}i}, \mathsf{Tx}_{\mathsf{F}i}, \mathsf{Tx}_{\mathsf{P}i}, \sigma_V(\mathsf{Tx}_{\mathsf{F}i}), \sigma_V(\mathsf{Tx}_{\mathsf{P}i}))\}_{i\in[1,row]})$ in Φ, execute $\mathsf{postTxsFP}(\mathcal{F}_S, T_P, n, \mathsf{Tx_G}, \{(\mathsf{Tx}_{\mathsf{T}i}, \mathsf{Tx}_{\mathsf{F}i}, \mathsf{Tx}_{\mathsf{P}i}, \sigma_V(\mathsf{Tx}_{\mathsf{F}i}), \sigma_V(\mathsf{Tx}_{\mathsf{P}i}))\}_{i\in[1,row]})$ 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 Verifying DNF Formulas with Glimpse

As introduced in Section 3.4, Glimpse can efficiently verify *DNF formulas over descriptions* (Figure 2), allowing to synchronize *any logical combination of transactions on* $\mathcal{L}_S$ with corresponding transactions on $\mathcal{L}_D$. DNF formulas express truth tables in terms of disjunctions (OR) of conjunctions (AND) of one or more descriptions.

We explain how Glimpse achieves this degree of expressiveness by showing a concrete example. Let us consider two oracles, i.e., O_1 and O_2, operating on $\mathcal{L}_S$ (they can also operate on different chains) and regularly posting information about a real-world event. On $\mathcal{L}_D$, prover P and verifier V lock $\frac{\alpha}{2}$ coins each in a Glimpse contract ($\mathsf{Tx_G}$) and, e.g., they bet on a specific outcome for the event: if at least one oracle attests the desired outcome for the event (1-out-of-2 threshold), P can claim the α coins by proving to $\mathsf{Tx_G}$ that at least one oracle has published a transaction attesting the established outcome for the event; if the coins are still unspent after time T, V can claim the coins.

We recall that the to-be-verified outcome for the event is specified in the description Desc of each transaction and is hardcoded within $\mathsf{Tx_G}$. We let Desc_i be the description for the transaction published by the oracle $\mathcal{O}_i$, and we let

$$\mathcal{F}_S = (\mathsf{Desc}_1 \wedge \neg \mathsf{Desc}_2) \vee (\neg \mathsf{Desc}_1 \wedge \mathsf{Desc}_2) \vee (\mathsf{Desc}_1 \wedge \mathsf{Desc}_2)$$

be the DNF formula Glimpse has to verify. Glimpse proceeds as follows:

Setup. θ_P and θ_V are unspent outputs on $\mathcal{L}_D$ holding $\frac{\alpha}{2}$ coins each and controlled by P and V, respectively. We denote with $\alpha := \theta_P.\mathsf{coins} + \theta_V.\mathsf{coins}$ the value locked in Glimpse and with ζ_P and ζ_V inputs spending 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 $\theta_\alpha := (\alpha, (\mathsf{MuSig}(\mathsf{pk}_P, \mathsf{pk}_V)) \vee (\mathsf{MuSig}(\mathsf{pk}_P, \mathsf{pk}_V) \wedge T_3))$, $\theta_{\varepsilon_1} := (\varepsilon_1, (\mathsf{scriptG}(\mathsf{Desc}_1, T_1, \mathcal{T}_S, n_1, P)))$, and $\theta_{\varepsilon_2} := (\varepsilon_2, (\mathsf{scriptG}(\mathsf{Desc}_2, T_1, \mathcal{T}_S, n_2, P)))$. We denote with ε_i the smallest possible amount of cash. Its value does not matter, since it is just used to enable the construction.

This means that when Glimpse verifies DNF a formula, $\mathsf{Tx_G}$ has to have as many outputs (θ_{ε_i}) as the number of transactions in the formula (in this case, two: $\theta_{\varepsilon_1}, \theta_{\varepsilon_2}$), plus an additional one holding the Glimpse value (θ_α). We now require *both P and V* to sample a random string and embed it in the transaction descriptions. Beside $\mathsf{Tx_G}$, the parties create a set of transactions $(\mathsf{Tx_T}, \mathsf{Tx_F}, \mathsf{Tx_P})_i$ for each disjunctive term in the formula. In our example, $\mathcal{F}_S$ has three terms: $(\mathsf{Desc}_1 \wedge \neg \mathsf{Desc}_2)$, $(\neg \mathsf{Desc}_1 \wedge \mathsf{Desc}_2)$, and $(\mathsf{Desc}_1 \wedge \mathsf{Desc}_2)$, therefore we will have three sets of transactions $(\mathsf{Tx_T}, \mathsf{Tx_F}, \mathsf{Tx_P})$. 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 $\mathcal{F}_S$. In general, $(\mathsf{Tx_T}, \mathsf{Tx_F}, \mathsf{Tx_P})_i$ is constructed as follows:

- $\mathsf{Tx_T}$ allows P to prove the inclusion of the oracles' transactions attesting the desired outcome for the event. $\mathsf{Tx_T}$ spends the outputs θ_{ε_i} for the Desc_i in the term (but not for the $\neg\mathsf{Desc}_i$), and it has a single output $\theta_T := (\varepsilon, \mathsf{OneSig}(\mathsf{pk}_P))$.
- $\mathsf{Tx_F}$ allows V to submit a proof for a transaction being posted on $\mathcal{L}_S$, as a reaction to a malicious P falsely claiming the transaction was not posted. $\mathsf{Tx_F}$ spends the outputs θ_{ε_i} for the $\neg\mathsf{Desc}_i$ in the term (but not for the Desc_i), and it has as many outputs as the number of its inputs, each one of the form $\theta_{F,i} := (\varepsilon_i, (\mathsf{scriptG}(\mathsf{Desc}_i, T_2, \mathcal{T}_S, n_i, (V, P))$. For some terms, $\mathsf{Tx_F}$ is not needed at all, e.g., $(\mathsf{Desc}_1 \wedge \mathsf{Desc}_2)$.
- $\mathsf{Tx_P}$ allows P to claim the α coins if all the outputs of $\mathsf{Tx_F}$ are still unspent. It spends θ_T, all the $\theta_{F,i}$, and θ_α.

Finally, the parties create transaction $\mathsf{Tx_V}$ allowing V to spend the output θ_α after time T_3. We have $T_1 < T_2 < T_3$.

At this point, P signs $[\mathsf{Tx_V}]$ and sends to V the Glimpse specifics $(\mathsf{Desc}_1, \mathsf{Desc}_2, T_1, T_2, \mathcal{T}_S, n_1, n_2, \alpha, \mathsf{scriptG}, [\mathsf{Tx}_G], ([\mathsf{Tx_T}], [\mathsf{Tx_F}], [\mathsf{Tx_P}])_{i,\forall i}, [\mathsf{Tx_V}], \sigma_P([\mathsf{Tx_V}]))$. V checks if the Glimpse specifics are well-formed, and verifies the validity of P's signature. If everything is correct, V signs $[\mathsf{Tx_G}]$ and $[\mathsf{Tx_P}]_{i,\forall i}$, and sends the signatures to P. P checks if V's signatures are valid and, if so, posts $\mathsf{Tx_G}$ on $\mathcal{L}_D$.

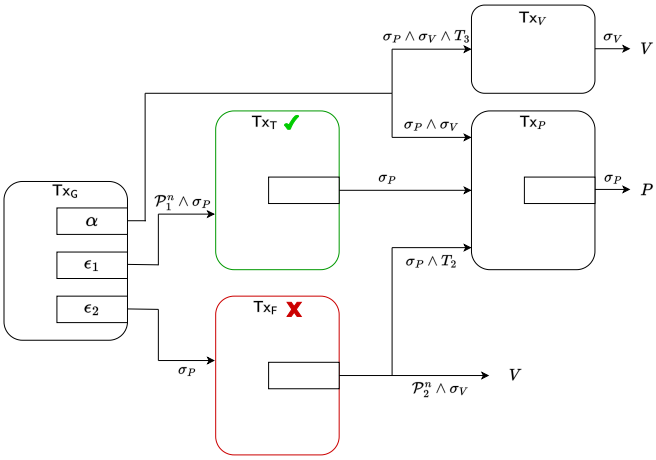

Figure D.1: Set $(\mathsf{Tx_G}, (\mathsf{Tx_T}, \mathsf{Tx_F}, \mathsf{Tx_P})_i, \mathsf{Tx_V})$ of transactions for efficiently verifying the term $\mathsf{Desc}_1 \wedge \neg\mathsf{Desc}_2$ of the DNF formula in the example.

Commit on $\mathcal{L}_S$. The oracles publish transactions attesting the outcome of the real-world event, e.g., $\mathcal{O}_1$ publishes Tx_1 s.t. $[\mathsf{Tx}_1] \hookleftarrow \mathsf{Desc}_1$ and $\mathcal{O}_2$ publishes Tx_2 s.t. $[\mathsf{Tx}_2] \hookleftarrow \mathsf{Desc} \neq \mathsf{Desc}_2$. This is, e.g., the case depicted in Figure D.1.

Verify & Commit on $\mathcal{L}_D$. P and V monitor $\mathcal{L}_S$ (or query a relayer R) checking for the inclusion of transactions matching descriptions Desc_1 and Desc_2. P constructs the corresponding proofs, and claims the coins by posting the corresponding set $(\mathsf{Tx_T}, \mathsf{Tx_F}, \mathsf{Tx_P})_i$.

V checks whether the set of transactions published by P corresponds to the correct term of $\mathcal{F}_S$ realized by the oracles. If P does not spend θ_α, V can publish $\mathsf{Tx_V}$ and get the Glimpse coins after T_3. If P misbehaves, V can react within T_2 and spend one of $\mathsf{Tx_F}$'s outputs, thereby invalidating $\mathsf{Tx_P}_i$ and claiming the money via $\mathsf{Tx_V}$.

Remarks. We observe that when verifying DNF formulas with Glimpse, V needs to be able to query the relayer R (or run a full node), as he needs to construct and submit proofs in case P cheats. Although increasing the off-chain communication overhead, this construction results in up to three on-chain transactions in the optimistic case, no matter the complexity of the formula to verify.

E Glimpse Script Example for Liquid

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 en-

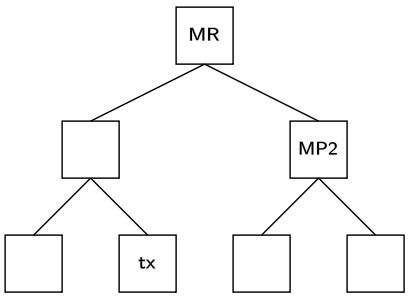

Figure E.1: Merkle tree of four transactions. The two elements MP1 and MP2 we need for reconstructing the Merkle root are marked, along with Tx.

abled (granting access to the MAST functionality), the opcodes OP_CAT as well as OP_SUBSTR are active. We point the reader to [63, 64] 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 $\mathsf{Tx_P}$ with witness $\mathcal{P}^{n=0}$ on Liquid as a result of Tx being published on Bitcoin. The $\mathsf{Tx_G}$ 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 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 $\mathsf{Tx_P}$ solely spending $\mathsf{Tx_G}$'s output θ_G. We recall Bitcoin-like chains process instructions sequentially: For $\mathsf{Tx_P}$ to spend θ_G, the locking script of θ_G is executed after the witness for $\mathsf{Tx_P}$. 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 $\mathsf{Tx_P}$ 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

```
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 <pkV> OP_CHECKSIG <pkP> OP_CHECKSIGADD <2>
   OP_NUMEQUAL
```

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

> 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 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 [21]. 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 [65].

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.