Glimpse: On-Demand, Cross-Chain Communication for Efficient DeFi Applications on Bitcoin-based Blockchains

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. Solutions based on trusted third parties (TTPs) suffer from security and scalability drawbacks; hence, increasing attention has recently been given to decentralized solutions. Lock contracts (e.g., HTLCs and adaptor signatures) and chain relays emerged as the two most prominent attempts to achieve cross-chain communication without TTPs. Lock contracts enable efficient synchronization of single transactions over different chains but are limited in expressiveness as they only support the development of a restricted class of applications (e.g., atomic swaps). On the other hand, chain relays enable the development of arbitrary cross-chain applications but are extremely expensive to operate in practice because they need to synchronize *every* on-chain transaction, besides assuming a quasi Turing-complete scripting language, which makes them incompatible with Bitcoin-based and scriptless blockchains.**

We introduce Glimpse, a novel *on-demand cross-chain synchronization primitive*, which is both *efficient* in terms of on-chain costs and computational overhead, and *expressive* in terms of applications it supports. The key idea of Glimpse is to synchronize transactions on-demand, i.e., only those relevant to realize the cross-chain application of interest. We present a concrete instantiation which is *compatible* with blockchains featuring a limited scripting language (e.g., Bitcoin-based chains like Liquid), and, yet, can be used as a building block for the design of DeFi applications such as lending, pegs, wrapping/unwrapping of tokens, Proof-of-Burn, and verification of multiple oracle attestations. We formally define and prove Glimpse security in the Universal Composability (UC) framework and conduct an economical security analysis to identify the secure parameter space in the rational setting. Finally, we evaluate the cost of Glimpse for Bitcoin-like chains, showing that 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 basic Bitcoin transaction.

1. Introduction

The plethora of different blockchains that have emerged and gained traction over the past years has yielded a fragmented and diverse landscape. Each blockchain comes with its design and characteristics, attracting users for their privacy properties (e.g., Monero and ZCash), their high throughput (e.g., Algorand) or low fees (e.g., Solana), their unique DeFi ecosystem (e.g., Ethereum) or their robust design (e.g., Bitcoin). Blockchain platforms already hold an impressive amount of investments, users, and developers, with the latter being often reluctant to migrate their assets and contracts to other chains. In this context, the lack of interoperability prevents users from trustlessly leveraging the characteristics and the profit opportunities of different chains, as well as the features of newborn chains. Overall, this fragmentation significantly weakens blockchains' potential and value and erodes the user experience.

Many solutions have been presented to promote cross-chain communication and blockchain interoperability: some rely on a trusted third party (TTP), while others do not impose any trust assumption, such as lock contracts and chain relays. We focus on the latter since trusting a central entity is risky in the context of cryptocurrencies, besides not aligning with their rationale and philosophy. Lock contracts such as Hashed TimeLock Contracts (HTLCs) and adaptor signatures [1] enable the synchronization of individual transactions, possibly on different chains, in a way that these are triggered atomically. The success of lock contracts is due to their relatively simple and elegant design, low on-chain costs, and minimal set of scripting requirements. Despite their broad adoption, their restricted functionality limits their usage in cross-chain applications to the context of atomic swaps swaps [2], [3], [4], [5], as we further discuss in Section 2.3. Conversely, chain relays [6], [7], [8] are an expressive but expensive solution, as they verify and store *every* block header of a source ledger $\mathcal{L}_S$ on a destination ledger $\mathcal{L}_D$, thereby acting as light clients. This hinders their practical deployment due to *extremely high maintenance costs*. Additionally, chain relays require a quasi-Turing complete scripting language on the destination chain, therefore *excluding chains adopting minimal scripting capabilities*, like the Bitcoin-based Liquid, and more. In short, existing solutions either suffer from technical, economic, or compatibility limitations.

In this work, we present Glimpse, a novel *cross-chain synchronization primitive* that allows participants on a *destination ledger* $\mathcal{L}_D$ to obtain *on demand* the desired informa-

tion about the state of a *PoW source ledger* $\mathcal{L}_S$. Intuitively, Glimpse allows a *prover* and a *verifier* to enforce that if a specific set of transactions Tx_S is confirmed on $\mathcal{L}_S$ within a given time, then another set of transactions Tx_D can be published on $\mathcal{L}_D$. Technically, Glimpse is a contract lying 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.

To achieve these properties, we address several challenges. First, to drastically reduce the on-chain costs, we synchronize transactions *on-demand*. More precisely, we forgo the need to verify and store the full list of block headers of $\mathcal{L}_S$ on $\mathcal{L}_D$ as done in relays, verifying instead only that enough work (weighted over a time window) has been done to produce the proof. The protocol design is further enriched by a number of ingredients to make Glimpse secure against a variety of attacks, including proof forgeries, block reorganizations, and upfront mining.

Second, to enhance expressiveness, we fix only partially the expected format of the transaction on $\mathcal{L}_S$ in the contract, leaving parts undefined, which is crucial to support cross-chain applications with a dynamic component (e.g., lending, where it is not known a-priori how the lent money will be used). We generalize our synchronization patterns to those that can be expressed as Disjunctive Normal Forms (DNFs) formulas over transactions, which we leverage to encode a variety of DeFi applications, such as lending, pegs, wrapping/unwrapping of tokens, Proof-of-Burn, and verification of multiple oracle attestations.

Third, the security of Glimpse is based on the fact that it is less profitable to attack Glimpse than to participate in the regular mining process, i.e., by producing a valid block and earning the associated reward. This means that the value locked on $\mathcal{L}_D$ in Glimpse should not exceed a certain threshold to disincentivize malicious parties to cooperate with miners and launch proof forgery or censorship attacks. We formally analyze the security boundaries for the economic value held by Glimpse. Our contributions are:

- We introduce Glimpse, a novel cross-chain synchronization primitive combining efficiency, expressiveness, and compatibility with blockchains offering limited scripting capabilities, e.g., for those based on Bitcoin (Section 3.3);
- We provide an instantiation that is suitable to encode a variety of DeFi applications, including lending, pegs, wrapping/unwrapping of tokens, Proof-of-Burn, and verification of multiple oracle attestations (Section 4);
- We formally analyze Glimpse in the UC framework and characterize the security parameters of Glimpse, including the limits of the economic value Glimpse contracts can hold (Section 5).
- We evaluate the on-chain costs of Glimpse (in Ethereum-like and Bitcoin-like chains), and showing the overall cost is at most 3$, around twice as much as ordinary transactions. We also show how to optimize it with Taproot [9], [10] (Section 6).

2. Background

2.1. 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 $\mathsf{Tx} = (\mathsf{cntr}_{in}; \overrightarrow{\mathsf{input}}; \mathsf{cntr}_{out}; \overrightarrow{\mathsf{output}}; \overrightarrow{\mathsf{witnesses}})$ 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*. We let $[\mathsf{Tx}] := (\mathsf{cntr}_{in}; \overrightarrow{\mathsf{input}}; \mathsf{cntr}_{out}; \overrightarrow{\mathsf{output}})$ be 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.

$\mathsf{cntr}_{in} \in \mathsf{R}_0$ and $\mathsf{cntr}_{out} \in \mathsf{R}_0$ represent the number of elements in the $\overrightarrow{\mathsf{input}}$ and $\overrightarrow{\mathsf{output}}$ lists. Any input within $\overrightarrow{\mathsf{input}}$ is an unspent output from an older transaction, defined by the tuple $:= (\mathsf{txid}; \mathsf{outid})$, with $\mathsf{txid} \in \{0;1\}^{256}$ representing the hash of the old transaction containing the to-be-spent output, and $\mathsf{outid} \in \mathsf{R}_0$ the index of such an output within the output list of the old transaction. These two fields uniquely identify the to-be-spent output. $\overrightarrow{\mathsf{witnesses}} \in \{0;1\}$, also known as *scriptSig* or *unlocking script*, is a list of witnesses *!*, i.e., the data that only the entity entitled to spend the output can provide, thereby authenticating and validating the transaction. Any output in the list $\overrightarrow{\mathsf{output}}$ is a pair $:= (\mathsf{coins};)$ and can be consumed by at most one transaction (i.e., no double-spend). The amount of coins in an output is denoted by $\mathsf{coins} \in \mathsf{R}_0$, whereas the spendability of is restricted by the conditions in , 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.

2.2. Proof-of-Work Consensus

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

In particular, the nodes, called *miners*, repeatedly change the Nonce field of the block header until the hash of the header lies within a *target space* that is smaller (by several

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

2.3. Lock Contract and Chain Relay Limitations

Existing cross-chain communication solutions not relying on a TTP fall into two main categories: lock contracts and chain relays. Lock contracts are an umbrella term for non-custodial locking mechanisms (e.g., Hashed-Timelocked-Contracts, adaptor signatures) that achieve security and atomicity from the hardness of some cryptographic assumptions. Hash locks and adaptor signatures are, for instance, lock contract schemes broadly used to encode blockchain applications such as atomic swaps, payment channels [11], [12], multi-hop payments [13], [14], virtual channels [15], [16], [17], [18], and discreet log contracts [19]. 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.

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 [20]. 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.

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* $\mathcal{L}_D$ without executing a light client.

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

3.1. Assumptions and Models

Cryptographic Assumptions. Hash functions are modeled as random oracles [21]. Digital signature schemes are assumed to be EUF-CMA secure. They comprise three algorithms: $(\mathsf{Gen}; \mathsf{Sign}; \mathsf{Vrfy})$. Gen is a probabilistic polynomial-time (PPT) algorithm on input a security parameter 1^λ returns key pair $(\mathsf{pk}; \mathsf{sk})$. Sign is a PPT algorithm, on input sk and a message m, outputs a digital signature . Vrfy is a deterministic polynomial-time (DPT) algorithm that on input pk, m and returns 1 when the signature over m w.r.t pk is valid and 0 otherwise.

System Model. We assume two blockchains (ledgers) $\mathcal{L}_S$ and $\mathcal{L}_D$ where *consistency* and *liveness* hold [22] and where $\mathcal{L}_S$ uses a PoW consensus as specified in Section 2.2. Glimpse relies on four parties: an *issuer* I that publishes transactions Tx_S on $\mathcal{L}_S$, a prover P that proves the occurrence of Tx_S on $\mathcal{L}_D$, a relayer R (e.g., blockchain explorers, full nodes) that provides parties with the necessary information to construct the proof $\mathcal{P}$, and a verifier V that guarantees *contractual fairness*. These parties can but do not have to be the same, e.g., P and I can be the same party.

We require P and V to have an address (a key pair $(\mathsf{pk},\mathsf{sk})$) on $\mathcal{L}_D$, whereas I to have an address on $\mathcal{L}_S$. The Glimpse contract is deployed on $\mathcal{L}_D$ and holds coins locked in by P and V (and potentially also other users of $\mathcal{L}_D$). We also require $\mathcal{L}_D$ to support the same hash function used by $\mathcal{L}_S$, and both $\mathcal{L}_S$ and $\mathcal{L}_D$ to allocate the same domain for the hash function, to avoid oversize preimage attacks [23], [24]. Furthermore, $\mathcal{L}_D$ needs to support the following functionalities in its programming language: (i) Merkle proof verifications (ii) hash comparisons, and (iii) block header and transaction body reconstructions.[1]

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

Adversarial Model. P and V are *mutually distrustful parties*, with at least one of them being honest. I can

1. We note that (i) and (iii) can be supported by Bitcoin-based chains by simply enabling a concatenation opcode.

be honest or dishonest, meaning she can either publish or not publish Tx_S on $\mathcal{L}_S$ within the established time frame. Similarly, R is untrusted, as she can either provide correct or incorrect information (or no information) to P and V about the state of $\mathcal{L}_S$. We assume a majority of honest miners, where depends on the underlying consensus.

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, $\mathcal{L}_D$ verifies the commitment on $\mathcal{L}_S$ and, upon successful verification, a publicly verifiable commitment, i.e., Tx_D, is posted on $\mathcal{L}_D$. An optional *abort* phase reverts transaction Tx_S on $\mathcal{L}_S$ in case the verification of the commitment failed or the commitment on $\mathcal{L}_D$ is not executed.

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

Definition 1 (*Weak atomicity*)**.** Let Tx_S and Tx_D be (sets of) transactions for $\mathcal{L}_S$ and $\mathcal{L}_D$, respectively. If honest players of $\mathcal{L}_S$ reports Tx_S with at least n confirmations at time t, then a valid Tx_D is provided to honest players of $\mathcal{L}_D$ and reported stable at time $t +$ $_D +$ $_S$: $\mathsf{Tx}_D \in \mathcal{L}_D \Longrightarrow \mathsf{Tx}_S \in \mathcal{L}_S$.

Definition 2 (*Strong atomicity*)**.** Let Tx_S and Tx_D be (sets of) transactions for $\mathcal{L}_S$ and $\mathcal{L}_D$, respectively. Tx_D is reported stable by honest players on $\mathcal{L}_D$ at time $t +$ $_D +$ $_S$ if and only if honest players of $\mathcal{L}_S$ reports Tx_S with at least n confirmations at time t: $\mathsf{Tx}_D \in \mathcal{L}_D \iff \mathsf{Tx}_S \in \mathcal{L}_S$. 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.

3.2. Protocol Overview

In the *Setup* phase, P and V cooperate in the creation of the Glimpse contract Tx_G, which hard-codes the target $\mathcal{T}_S$ of $\mathcal{L}_S$ (*PoW consensus parameter*), as well as the following *Glimpse specifics*: the hashes of the to-be-verified transactions in Tx_S, the contract lifetime T, the number of confirmation blocks in the proofs, and the spending conditions for the funds in the contract.

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

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

We will see that this approach can be generalized so that Tx_S represents a set of transactions, logically expressed as a disjunctive normal form over transactions, that are considered valid by the Glimpse contract. Figure 1 depicts the Glimpse protocol flow.

3.3. Glimpse Design Principles

We describe the protocol design, starting with a straw man proposal and gradually refining it to tackle security challenges.

Simplify Relay Contract. Let us consider, for now, the simple case where the set Tx_S is composed of a single transaction Tx. We recall that a chain relay is a smart contract deployed on a destination chain $\mathcal{L}_D$ that *verifies and stores the full list of block headers* of a source chain $\mathcal{L}_S$: however, this is expensive, and its use is exclusively limited to quasi Turing-complete blockchains.

With this in mind, we try to simplify relays to lightweight Glimpse contract such that, unlike relays, it is no longer able to verify any given transaction. Instead, Glimpse can verify whether or not *one, application-specific transaction* Tx was posted on $\mathcal{L}_S$. This design would allow us to feed the contract with minimal, Tx-specific information, removing the expensive need to continuously verify and store the full list of block headers. However, without a full list of block headers, the contract cannot ad-hoc compute the value of the PoW target $\mathcal{T}_S$ of $\mathcal{L}_S$ anymore.

Instead, the Glimpse contract now needs to be actively made aware of the current $\mathcal{T}_S$. To ensure $\mathcal{T}_S$ is honestly set, P and V have to initially agree on the value of $\mathcal{T}_S$ and *commit to it within the contract, along with the Glimpse specifics*. Concretely, in agreement with V, P publishes on $\mathcal{L}_D$ a contract that hard-codes the *PoW target* $\mathcal{T}_S$ (consensus parameter), the *to-be-verified-transaction hash* $\mathcal{H}([\mathsf{Tx}])$, the contract lifetime T, and the conditions under which the contract's funds can be spent (Glimpse specifics). Concretely, these spending conditions allow P to distribute the funds as

agreed with V in $\mathsf{Tx_P}$, upon submission of a valid proof $\mathcal{P}$ for Tx as part of the witness. Otherwise, after the timeout T, V gets the funds via $\mathsf{Tx_V}$.

The proof $\mathcal{P}$ consists of the Merkle inclusion proof for Tx as well as the block header (without the to-be-computed Merkle root) for the block in $\mathcal{L}_S$ containing Tx. The contract verifies $\mathcal{P}$ by (i) computing the Merkle root using the Merkle inclusion proof and the hard-coded $\mathcal{H}(\mathsf{Tx})$, (ii) reconstructing the full block header and hashing it, and (iii) finally checking if the hash is within the hard-coded $\mathcal{T}_S$. In practice, $\mathcal{P}$ guarantees that *enough work has been done to find a valid block including* Tx. We address difficulty adjustments taking place during the Glimpse lifetime by considering a larger target $\mathcal{T}_S$.

The simplicity of this construction forgoes the need for stateful smart contracts, and it can be deployed, e.g., *in Bitcoin-based chains within a transaction* $\mathsf{Tx_G}$ *that simply encodes this smart contract in the locking script, which is spendable with a witness containing the proof* $\mathcal{P}$.

As is, Glimpse has three weaknesses: (i) $\mathcal{P}$ is not sound as a malicious P could forge a fake one-block proof given a large enough T; it is not robust in case of (ii) block reorganization where a malicious P could submit a valid proof for a transaction Tx that, however, does not end up in the longest chain but to an orphaned branch; and (iii) upfront mining attacks where a malicious P could forge $\mathcal{P}$ upfront by knowing Tx before setting up Glimpse.

Parameterized $\mathcal{P}$ and Randomized Tx. To secure Glimpse against proof forgeries and block reorganizations, we parameterize the proof $\mathcal{P}$ with a *confirmation parameter* n (hard-coded in the $\mathsf{Tx_G}$ locking script), such that $\mathcal{P}^n$ includes n confirmation block headers with hashes within the target $\mathcal{T}_S$ after the block containing Tx. To cheat, P now has to make a significant computational effort to find $n+1$ blocks in time T whose hashes are within $\mathcal{T}_S$. The parties can set n such that P has only a *negligible probability of forging* $\mathcal{P}^n$ *in time* T (see Section 5.1). Instead, P can only construct a valid $\mathcal{P}^n$ by receiving valid information from R about the state of $\mathcal{L}_S$, thus leveraging the work of miners. For increasing n, the risk of block reorganization reduces exponentially.

We prevent P from launching an upfront mining attack by asking V to *randomize the transaction* Tx: for instance, V can sample a uniformly random string $r \leftarrow \{0;1\}^\lambda$ (is the security parameter), and plug it into the Tx body, producing $\mathsf{Tx_R}$. This can be done by adding to an output of value 0 with spending condition $\mathtt{OP\_RETURN}$[2] followed by the random value r. Now the Glimpse transaction $\mathsf{Tx_G}$ must hard code $\mathcal{H}([\mathsf{Tx_R}])$ instead of $\mathcal{H}([\mathsf{Tx}])$. Since P cannot anticipate r, he cannot start forging $\mathcal{P}^n$ upfront, so his computational efforts are restricted to the timeout T.

3.4. Enhancing Expressiveness

As of now, Glimpse cannot yet encode sophisticated applications where, for instance, inputs and outputs of $\mathsf{Tx_R}$ are not entirely known a priori, as for the case of cross-chain lending, nor where one needs to efficiently verify not a single transaction Tx, but a set of transactions, as for multiple oracle attestations. To cater to such use cases, we augment Glimpse as in Figure 2 to verify *transactions which are not fully known during the initial Setup phase* and to capture arbitrary *synchronization patterns* expressed as *Disjunctive Normal Forms* over transactions in $\mathcal{L}_S$.

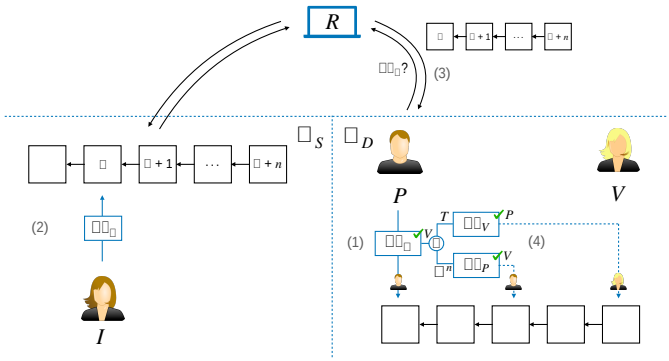

Figure 1: On a high level, Glimpse operates as follows: (1) Upon P and V agreeing on the Glimpse specifics and PoW consensus parameters, they construct $\mathsf{Tx_G}$, $\mathsf{Tx_P}$, and $\mathsf{Tx_V}$. P publishes $\mathsf{Tx_G}$ on $\mathcal{L}_D$. (2) I publishes $\mathsf{Tx_R}$ on $\mathcal{L}_S$.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

Recall our definition transaction bodies $[\mathsf{Tx}] := (\mathsf{cntr}_{in}; \overrightarrow{\mathsf{input}}; \mathsf{cntr}_{out}; \overrightarrow{\mathsf{output}})$, where inputs and outputs are tuples $(\mathsf{txid}; \mathsf{outid})$ and $(\mathsf{coins};\ )$, respectively, from Section 2.1. To verify transactions that are not fully known in the Setup phase, we introduce the *Glimpse description* Desc of a transaction (see Figure 2). In such a description, we allow parameterized inputs and outputs. More concretely, $\mathsf{txid}; \mathsf{outid}$, and coins can either be static data or *variables* x_i. Similarly, to avoid fixing a priori a specific script, we say that in a parameterized input or output can be a function f which encodes a family of scripts. f takes a fixed number of arguments for a well-defined spending condition and returns the desired locking script for the to-be-verified transaction. In other words, this is a parameterized locking script that can be filled with concrete values, e.g., public key, script hash, etc. The script which f encodes must be defined in the *Setup* phase: For instance, if the parties agree on $f(z)$ encoding any P2PKH (Pay-To-Public-Key-Hash), then it would accept any public key hash as parameter z and return the script.

Following Figure 2, $\overrightarrow{\mathsf{input}}$ and $\overrightarrow{\mathsf{output}}$ of descriptions are lists of such parameterized inputs and outputs, and cntr_{in} and cntr_{out} the number of overall inputs and outputs. The latter must be known from the beginning to avoid miners interpreting transactions in an unintended way (see Appendix D). In the *Setup* phase the parties agree on and commit to Desc, $\mathcal{T}_S$, T, and n. By replacing $\mathcal{H}(\mathsf{Tx_R})$ with

2. $\mathtt{OP\_RETURN}$ is a Bitcoin script opcode that marks a transaction output as invalid and can be used to embed up to 80-bytes in a transaction.

$$
\begin{aligned}
\mathsf{txid} &:= f0;1g^{256}\ j\ x_1 \\
\mathsf{outid} &:= f0;1g^{32}\ j\ x_2 \\
\mathsf{coins} &:= f0;1g^{64}\ j\ x_3 \\
&:= f(z_1;\dots;z_n) \\
\overset{!}{\mathsf{input}} &:= [(\mathsf{txid};\mathsf{outid})]\ j\ \mathsf{input}\ [\ [(\mathsf{txid};\mathsf{outid})] \\
\overset{!}{\mathsf{output}} &:= [(\mathsf{coins};\ )]\ j\ \mathsf{output}\ [\ [(\mathsf{coins};\ )] \\
\mathsf{cntr}_{in},\ \mathsf{cntr}_{out} &:= f0;1g^{m} \\
\mathsf{Desc} &:= (\mathsf{cntr}_{in};\overset{!}{\mathsf{input}};\mathsf{cntr}_{out};\overset{!}{\mathsf{output}}) \\
L_i &:= \mathsf{Desc}_i\ j\ :\mathsf{Desc}_i \\
F_S &:= (L_1 \wedge \dots \wedge L_k)\_\dots\_(L_1 \wedge \dots \wedge L_k)
\end{aligned}
$$

$$
8(x_1;\dots;x_3;z_1;\dots;z_n):(F_S\ (\ )\ \mathsf{Tx}_D)
$$

Figure 2: Synchronization Patterns expressed in DNFs

Desc, we can now verify any Tx_R in the set of transactions that share the same description, i.e., any Tx_R whose body has the same static data in Desc and an arbitrary realization for the variable ones. For example any value can be in the place of x_i and any parameter can be given to f, e.g., any public key hash in the example above. We denote this as $[\mathsf{Tx}_R] \leftarrow \mathsf{Desc}$ or a concrete transaction $[\mathsf{Tx}_R]$ *fulfilling a description* Desc. We note that the random string sampled by V must always be included in Desc. The variable realizations are included in the proof $\mathcal{P}^n$. Given $\mathcal{P}^n$ and Desc, the full transaction body can be reconstructed and hashed in the logic of Tx_G.

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

3.5. Compatibility

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

While Glimpse's compatibility with quasi-Turing complete chains as $\mathcal{L}_D$ is obvious, its compatibility with Bitcoin-like chains is not. Glimpse requires the destination chain to have an opcode for the same hash function used for PoW of the source chain: this is a strict requirement that already rules out some combinations in Table 1. E.g., Bitcoin-based chains do not have an opcode for computing Keccack or Scrypt hashes, used in Ethereum PoW and Litecoin, respectively. For this reason, Ethereum PoW and Litecoin can only be source chains for Glimpse contracts deployed on quasi-Turing complete chains. Next, we discuss how the particular design of Glimpse makes it compatible with prominent Bitcoin-based blockchains, such as Liquid, and how it could similarly be supported by Bitcoin, Bitcoin Cash, or Bitcoin Satoshi Vision (SV), e.g., by (re-)enabling string opcodes or Taproot. For detailed discussion and examples, we refer to Appendix D.

Source chain L_S	Destination chain L_D
Bitcoin	Bitcoin^{y}
Bitcoin Cash	Liquid
Bitcoin Satoshi Vision	Bitcoin Cash
Bitcoin Rootstock	Bitcoin SV
Litecoin^{yy}	Litecoin^{y}
Ethereum PoW^{yy}	Any quasi-Turing complete chain
Ethereum Classic^{yy}	

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

Liquid (Fully Compatible). Liquid is a Bitcoin sidechain operating since 2018. It inherits its design from Bitcoin but provides more expressiveness in its scripting language. In particular, the following opcodes crucial to Glimpse are disabled in Bitcoin but enabled on Liquid: (i) string concatenation (`OP_CAT`) for the Merkle Proof verification and block header/transaction reconstruction, and (ii) `OP_SUBSTR` for splitting numbers larger than 4 bytes for comparison (which only allows comparison of 4-byte numbers), i.e., the block header hash and PoW target. Moreover, Liquid adopts Taproot [9], which can significantly reduce the Glimpse script size and complexity with the use of the Merkelized Abstract Syntax Tree (MAST), as discussed in Section 6 and exemplified in Appendix D.

Bitcoin Cash (Missing Taproot). Bitcoin Cash is a blockchain resulting from a Bitcoin hard fork that took place in 2017 after Bitcoin moved to SegWit. It presents a larger block size and a wider variety of supported opcodes. Similarly to Liquid, Bitcoin Cash has `OP_CAT` and `OP_SPLIT` (which is the same as `OP_SUBSTR`). However, Taproot is not enabled, thus we cannot encode the Glimpse script in a single output. While it is possible to unroll the script leaves of the MAST, the script is too large (around 300kB, see Section 6), even though it would be within the transaction size limits. In Bitcoin SV we have the same limitations as in Bitcoin Cash. To support Bitcoin Cash or Bitcoin SV as a destination chain, it would need either Taproot or larger script sizes.

Bitcoin and Litecoin (Missing String Opcodes). Bitcoin and Litecoin have both adopted Taproot, but disabled the aforementioned opcodes in 2010. With Taproot at their disposal, they could efficiently support Glimpse if they had opcodes for Merkle root reconstruction and hash comparison available, e.g., if `OP_CAT` and `OP_SUBSTR` are available or if instead of the latter `OP_LESSTHAN` could compare 32-byte values. Interestingly, the Bitcoin community has been recently discussing the string concatenation opcode

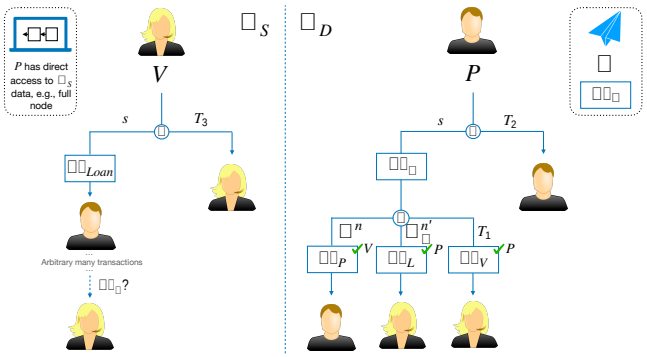

Figure 3: Illustration of Glimpse-based lending for Bitcoin-based blockchains. We have $T_1 < T_2 < T_3$.

and considering enabling it back in the context of Speedy Covenants [25]. We hope this work provides additional motivation for such opcodes to be (re-)enabled in the future.

4. Glimpse for Cross-Chain DeFi

In the last years, Decentralized Finance (DeFi) has gained exceptional traction. In particular, lending and borrowing markets thrive on blockchains supporting quasi Turing-complete smart contracts, as they allow lenders with an excess of funds to supply assets to lending smart contracts, whereas borrowers provide smart contracts with collateral acting as a security deposit for the loan. While trustless cross-chain lending marketplaces are widely used on, e.g., Ethereum, we are still far from having such applications in the Bitcoin-based blockchain ecosystem. To fill this gap, we show how to use Glimpse for designing a lending protocol for Bitcoin-based chains.

4.1. Cross-chain Lending Protocol

Intuition. We consider a borrower P and a lender V. P has coins on $\mathcal{L}_D$ and wants to take a loan of ^{0} coins on $\mathcal{L}_S$. We assume the loan is over-collateralized to compensate for price drops of asset . Having a surplus of ^{0} coins in $\mathcal{L}_S$, V lends the ^{0} coins to P. Intuitively, P locks coins in a Glimpse transaction $\mathsf{Tx_G}$, and V sets up a transaction Tx_{Loan} giving a loan of ^{0} coins to P. Via an atomic swap conditioned to secret s, P publishes Tx_{Loan} on $\mathcal{L}_S$ revealing s to V, and V publishes $\mathsf{Tx_G}$ on $\mathcal{L}_D$ using the same secret s. $\mathsf{Tx_G}$ guarantees that if P gives back to V the ^{0} coins on $\mathcal{L}_S$ within time T, P gets back his coins on $\mathcal{L}_D$. Otherwise, V retains the coin collateral after time T. Figure 3 depicts such a lending construction, which we detail below and later extended to support a liquidation mechanism.

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

We let $_P$ be an unspent output of P holding coins, and $_P$ be an input pointing to $_P$. Then, P constructs $[\mathsf{Tx}_G] := (1; [(\ _P)]; 1; [(\ ; \mathsf{scriptG}(\mathsf{Desc}; T; \mathcal{T}_S; n; (P; V)))])$. The locking script generated by $\mathsf{scriptG}$ can be spent as follows: (i) P can get back the coins by submitting a valid $\mathcal{P}^n$ (witness); else, V can get the coins after time T. T is strictly bigger than one block time of $\mathcal{L}_S$ and $\mathcal{L}_D$. (ii) $\mathcal{P}^n$ proves inclusion for a transaction Tx compliant with Desc, i.e., whose $[\mathsf{Tx}] \leftarrow \mathsf{Desc}$, and (iii) $\mathcal{P}^n$ is verified against the PoW consensus parameter $\mathcal{T}_S$. Figure 5 shows the pseudocode for $\mathsf{scriptG}$ (sG), and Appendix D describes a concrete example.

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

Upon receiving the message from P, if V is interested in opening a Glimpse instance with P, V verifies whether $\mathsf{scriptG}$ returns the intended locking script for $\mathsf{Tx_G}$ and $_P([\mathsf{Tx_V}])$ is a valid signature of P over $[\mathsf{Tx_V}]$. Upon successful verification, V computes the signatures $_V([\mathsf{Tx_P}])$ and $_V([\mathsf{Tx_G}])$, and sends them to P.

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

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

Verify & Commit on $\mathcal{L}_D$. P monitors $\mathcal{L}_S$ checking for Tx being included. If Tx is included within T and has n confirmations, P constructs $\mathcal{P}^n$ by taking the concrete value of x_1 within Tx ($[\mathsf{Tx}]$), computing the Merkle proof (MP) for Tx, retrieving from $\mathcal{L}_S$ the block header without Merkle root for the block B including Tx, and fetching n confirmation block headers (without $\mathsf{ParentHash}$) after B. Concretely, P constructs $\mathcal{P}^n := ([\mathsf{Tx}] \setminus \mathsf{Desc}; \mathsf{MP}; \mathsf{headerWOMR(B)}; \mathsf{confHeaders_n})$. We provide pseudocode for constructing $\mathcal{P}^n$ in Figure 6. Upon computing $\mathcal{P}^n$, P signs $[\mathsf{Tx_P}]$ and publishes $\mathsf{Tx_P}$ on $\mathcal{L}_D$ with witness $! := (\mathcal{P}^n;\ _P([\mathsf{Tx_P}]);\ _V([\mathsf{Tx_P}]))$, thus having back his coins staked in Glimpse.

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

The *Setup*, *Commit on* $\mathcal{L}_S$, and *Verify & Commit on* $\mathcal{L}_D$ phases are the core of Glimpse construction, recurring (with minor application-specific variations) regardless of the

Setup(Desc; T; T_S; n)

1) P sends a transaction description Desc^0 = (cntr$_{in}$; input; cntr$_{out}$; output) to V.
2) Upon receving Desc^0, V samples a random string $r \xleftarrow{\$} \{0,1\}^*$, creates Desc = (cntr$_{in}$; input; cntr$_{out}$ + 1; output $\|$ (0; OP_RETURN(r))), and sends Desc to P.
3) Let $\cdot := \cdot_P$.coins; let $\cdot_P$ point to an unspent output of P.
4) Let [Tx$_G$] := (1; [$\cdot_P$]; 1; [$\cdot_G$:= ($\cdot$; scriptG(Desc; T; T_S; n; (P; V)))]). Let $\cdot_G$ point to $\cdot_G$.
5) Let [Tx$_P$] := (1; [$\cdot_G$]; 2; [($\cdot$ outcomeP; OneSig(pk$_P$)); ($\cdot$ (1 outcomeP); OneSig(pk$_V$))]).
6) Let [Tx$_V$] := (1; [$\cdot_G$]; 2; [($\cdot$ outcomeV; OneSig(pk$_V$)); ($\cdot$ (1 outcomeV); OneSig(pk$_P$))]).
7) P computes σ_P([Tx$_V$]) and sends the specifics (σ_P; Desc; T; T_S; n; $\cdot$; scriptG; [Tx$_G$]; [Tx$_P$]; [Tx$_V$]; σ_P([Tx$_V$])) to V.
8) Upon receiving (σ_P; Desc; T; T_S; n; $\cdot$; scriptG; [Tx$_G$]; [Tx$_P$]; [Tx$_V$]; σ_P([Tx$_V$])), if V is interested in opening a Glimpse instance with P at the given specifics, V signs Tx$_P$ and sends (σ_V([Tx$_P$])) and sends it to P.
9) Upon receiving (σ_V([Tx$_P$])), P verifies if σ_V([Tx$_P$]) is a valid signatures and, upon successful verification, P signs [Tx$_G$] and posts Tx$_G$ with witness $\cdot$:= σ_P([Tx$_G$]).

Commit on L_S (Desc)

1) P posts Tx such that [Tx] - Desc.

Verify & Commit on L_D (Desc; T; n)

1) Upon Tx s.t. [Tx] - Desc being included on L_S with n confirmations before T, P constructs P^n as in Figure 6 and signs Tx$_P$. P posts Tx$_P$ with witness $\cdot$:= (P^n; σ_P([Tx$_P$]); σ_V([Tx$_P$])).
2) If time T has elapsed and $\cdot_G$ is still unspent, V signs Tx$_V$ and posts Tx$_V$ with witness $\cdot$:= (σ_P([Tx$_V$]); σ_V([Tx$_V$])).

Figure 4: Pseudocode for the Glimpse-based lending.

scriptG(Desc; T; cp; n; (P; V))

Let Tx 2 B 2 L_S such that [Tx] - Desc. scriptG generates a locking script with the following logic:

Upon receiving $\cdot$ = (P^n; σ_P([Tx$_P$]); σ_V([Tx$_P$])), where P^n = ([Tx] n Desc; MP; headerWOMR(B); confHeaders$_n$):

1) If [Tx] n Desc matches the expected number of strings and their expected length, reconstruct [Tx]. Else, return $?$.
2) Compute H([Tx]) := txid.
3) Given txid and MP, compute the Merkle root MR for B. Given MR and headerWOMR(B), reconstruct header of B.
4) Compute the hash of B's header and check if it is smaller than cp := T_S. Else, return $?$.
5) If n > 0, for each of the n confirmation blocks, reconstruct the header concatenating headerWOPH(i) with H(B$_{i-1}$).[a] Hash the header and check if it is smaller than T_S. Else, $?$.
6) If (σ_P([Tx$_P$]); σ_V([Tx$_P$])) are valid signatures of P and V, unlock the coins. Else, return $?$.

If t > T, upon receiving $\cdot$ = (σ_P([Tx$_V$]); σ_V([Tx$_V$])), unlock the coins. Else, return $?$.

a. This is equivalent to check whether each block constitutes a valid child of its parent.

Figure 5: scriptG(Desc; T; cp; n; (P; V)) pseudocode.

Construct P^n(Tx; Desc; n)

1) Upon Tx 2 B 2 L_S s.t. [Tx] - Desc, fetch [Tx].
2) Compute the transaction Merkle proof MP for Tx.
3) Fetch the block header of B and remove the Merkle Root: headerWOMR(B) := Header(B) n MR.
4) Let headerWOPH(B) := Header(B) n ParentHash. If n > 0, retrieve confHeaders$_n$:= {headerWOPH(B + 1); : : : ; headerWOPH(B + n)}.
5) Set P^n := ([Tx] n Desc; MP; headerWOMR(B); confHeaders$_n$).

Figure 6: Construct $\mathcal{P}^n$(Tx; Desc; n) pseudocode.

specific use case. We now discuss the liquidation mechanism exclusively for lending.

Liquidation. Should the asset price on $\mathcal{L}_S$ shrink below a predefined liquidity threshold, V must be able to redeem the collateral before T. For this, we assume there exists a trusted oracle $\mathcal{O}$ on $\mathcal{L}_D$ that regularly publishes a transaction Tx$_O$ with the current price of the asset on $\mathcal{L}_S$; for instance, $\mathcal{O}$ can be a Discreet Log Contract-based [19] oracle. If $\mathcal{O}$ is not trusted, we can leverage a set of N different independent oracles, with the promise that if a large enough number of oracles agree on the same price, the liquidation is granted using Glimpse ability to verify DNFs over descriptions. The oracles do not need to coordinate with each other, nor have a common transaction structure. For simplicity, we discuss the case of a single trusted $\mathcal{O}$.

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

To include liquidation, scriptG has to be tweaked to encode the following: (i) if P repays his debt publishing Tx, s.t. [Tx] $\leftarrow$ Desc on $\mathcal{L}_S$ within time T, he can publish Tx$_P$ with witness $\mathcal{P}^n$ and have his collateral back on $\mathcal{L}_D$, (ii) if P defaults the loan, V can publish Tx$_V$ redeeming the collateral after T, and (iii) if before T $\mathcal{O}$ attests the collateral price on $\mathcal{L}_S$ below the predefined liquidity threshold, V can redeem the collateral publishing the liquidation transaction Tx$_L$:= (1; [$_G$]; 1; [($\cdot$; OneSig(pk$_V$))]) with witness $\mathcal{P}^{n^0}_O$ and sell the funds at a discount. The liquidation transaction has to be constructed and signed by P in the *Setup* phase.

Glimpse-based lending enables the first form of trustless peer-to-peer lending on Bitcoin-based chains. Moreover, publishing Tx$_G$ and Tx$_{Loan}$ via atomic swap only requires a single Glimpse instance on $\mathcal{L}_D$. On the other hand, without extensive programmability, funds cannot be pooled, leading to peer-to-peer lending where potential borrowers have to find would-be lenders and agree on the loan's amount and interest rate – reflected in the fund distribution in Tx$_P$. To facilitate matching the demand and supply, we suggest setting up dedicated communication channels or platforms.

4.2. Backed Assets, Proofs-of-Burn, and Oracles

Besides the lending, Glimpse can serve as a lightweight and powerful building block for other interesting applica-

tions, such as *backed assets* (e.g., *sidechain pegs*, *wrapping and unwrapping of tokens*), *Proofs-of-Burn*, and multiple oracle attestations. In this section, we give an intuition about how Glimpse can be leveraged in different use cases.

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

Sidechains are blockchains tightly bound to a pre-existing parent blockchain with the purpose of enabling or extending some features. Users can easily move funds from the parent chain to the sidechain and vice versa through verifiable two-way pegs: assets are locked in an address of the parent chain (sidechain) and are then released on the sidechain (parent chain), ready to be used. Starting from the lending protocol on Section 4.1, we show how to set up pegs with Glimpse. Let us remove the liquidation mechanism from the lending protocol and assume V *can create assets on* $\mathcal{L}_D$: with these two caveats, the same Glimpse-based construction can be used to encode trustless sidechain pegs, where V issues new assets on the sidechain (rather than giving a loan), and P is able to get back the funds on the parent chain by proving he returned the coins to an a priori well-defined peg address on the sidechain.

Similar to pegs, Glimpse can be used to wrap and unwrap tokens. Wrapped tokens allow one to represent on a chain $\mathcal{L}_D$ an asset that does not have any native representation on a different chain $\mathcal{L}_S$. Wrapped tokens can be issued on $\mathcal{L}_D$ when a corresponding amount of native tokens are locked on $\mathcal{L}_S$, and they can subsequently be released (unwrapped) on the native chain when the user locks up the wrapped ones on $\mathcal{L}_D$.

Proofs-of-Burn. Proof-of-Burn is a bootstrapping mechanism allows users to prove on a destination ledger $\mathcal{L}_D$ they burnt some coins on the source ledger $\mathcal{L}_S$, meaning they sent coins to a verifiable unspendable address. By verifying such proof, the user can obtain the correspondent amount of native assets on $\mathcal{L}_D$.

The construction for a Proof-of-Burn is very similar to the one for backed assets, with the only difference being that funds are moved unidirectionally.

Proofs of Oracle Attestations. A noteworthy application enabled by Glimpse for Bitcoin-based chains is the following: let us assume on $\mathcal{L}_S$ there exist k oracles posting information about real-world events, such as real-time prices for currencies, and on $\mathcal{L}_D$ one wants to efficiently verify k oracle attestations for a specific event. In case of two oracles, $\mathcal{O}1$ and $\mathcal{O}2$, $\mathsf{Tx_G}$ verifies $\mathcal{F}_S = (\mathsf{Desc}_1 \wedge \mathsf{Desc}_2)$, with Desc_1 being the description for $\mathcal{O}1$ and Desc_2 the one for $\mathcal{O}2$. We note that $\mathcal{O}1$ and $\mathcal{O}2$ do not have to cooperate nor operate on the same chain; their chains need to run PoW consensus.

5. Security Analysis

In this Section, we formally analyze the security of Glimpse in the Global Universal Composability (GUC) framework [26]. First, we state that according to basic assumptions, Glimpse achieves weak atomicity. Next, we show that by assuming the liveness of parties and access to $\mathcal{L}_S$, Glimpse achieves strong atomicity. Finally, we investigate under which conditions the basic assumptions underlying the security of Glimpse hold against rational parties. To that end, we calculate the adversary's costs for breaking the assumptions we make for the GUC model by carrying out proof forgery and censorship attacks.

Security in the UC Framework. We model Glimpse in the synchronous Global Universal Composability (GUC) framework [26], closely following prior work [17], [1], [27]. We use a global clock $\mathcal{G}_{Clock}$ [26] and authenticated channels with guaranteed delivery $\mathcal{G}_{GDC}$ [17] to model time and communication. We assume static corruption. We use the functionality $\mathcal{G}_{Ledger}$ defined in [28] to model a ledger $\mathcal{L}$. We use a concrete instantiation as specified in [28], where the parameters are chosen such that the ledger achieves both *liveness* and *consistency*, which is defined in [22]. We define two very similar ideal functionalities $\mathcal{F}_{W\ \ Glimpse}$ and $\mathcal{F}_{S\ \ Glimpse}$ (see Appendix A.1), formalizing our desired properties of *weak atomicity* and *strong atomicity* in the general case, respectively. More concretely, the ideal functionality is parameterized over two ledgers $\mathcal{L}_A$ or $\mathcal{L}_B$. The functionality observes these ledgers and, after two parties have registered to it, will ensure that the respective transactions are posted on $\mathcal{L}_A$ or $\mathcal{L}_B$ such that *weak atomicity* or *strong atomicity* holds.

We then formally model our Glimpse protocol in the UC framework (see Appendix A.2), and prove that realizes $\mathcal{F}_{W\ \ Glimpse}$ or $\mathcal{F}_{S\ \ Glimpse}$, depending on the underlying assumptions. In a nutshell, this is done by designing an ideal world adversary (or simulator) $\mathcal{S}$ and showing that no PPT *environment* can computationally distinguish between interacting with the real world protocol 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 is "as secure", i.e., has the same properties as $\mathcal{F}_{W\ \ Glimpse}$ or $\mathcal{F}_{S\ \ Glimpse}$. This is formalized in Appendix A. In Appendix B 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 A.2.

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

Theorem 2. *Given functionalities $\mathcal{G}_{Clock}$, $\mathcal{G}_{GDC}$, the protocol is instantiated with two ledger instantiations $\mathcal{L}_A$ and $\mathcal{L}_B$ of $\mathcal{G}_{Ledger}$, a delay $_B \in \mathsf{N}$ and a proof generation function $\mathsf{gen}\mathcal{P}$, where $\mathsf{gen}\mathcal{P}$ is complete and T-sound, and where has strictly randomized input,* ***and where all parties have direct access to $\mathcal{L}_A$ and $\mathcal{L}_B$, and where parties***

exhibit liveness, *then the protocol UC-realizes the ideal functionality* $\mathcal{F}_{S\ Glimpse}$.

We now study the cost of two types of attacks that violate the assumptions from the theorems above by bribing miners or conducting feather fork attacks. We first analyze an attack on *T-soundness*, where miners try to forge a proof, and then an attack on the *liveness* of a ledger via transaction censorship. The aim is to find under which parameters these assumptions hold and Glimpse achieves the desired properties.

5.1. Proof Forgery Attack

In the UC security model, we assume the generated proofs are *T-sound*, which means that proof forgery attacks are successful only with negligible probability. In this economic analysis, we examine under which parameters this assumption holds for Glimpse if players are rational, i.e., they act to maximize their profit.

Let us consider the case where an adversarial P bribes the miners of $\mathcal{L}_S$ to forge a proof for his Glimpse contract, and promise to reward them with the coins held by Glimpse on $\mathcal{L}_D$. We take one step further and we augment the adversary's power such that each prover P_i having *active Glimpse instances on top of the same* $\mathcal{L}_S$ is malicious and cooperates to launch a joint attack. In this case, corrupted miners of $\mathcal{L}_S$ can optimize their computational overhead by forging a single proof for all the contracts: upon receiving from P_i the transactions to include in the forgery, they forge a single block B^f including them all, and then they mine n confirmation blocks on top of B^f in time T, where n and T are the average proof size and the lifetime of the Glimpse instances.

Since we assume the worst case where all provers collude, we denote $\mathcal{G}$ as the set of *all* the *active* Glimpse contracts on top of $\mathcal{L}_D$ sharing the same source chain $\mathcal{L}_S$. To analyze when this attack constitutes a threat in practice, we need to have knowledge of the *cumulative economic value held by the contracts in* $\mathcal{G}$. In practice, we can say that honest verifiers should mark a Glimpse contract Tx_G as such, e.g., by adding an output $(0; \mathsf{OP\_RETURN}(\text{“This is Glimpse contract: } T; n; \text{ ”}))$. This way, honest parties can monitor the cumulative value of *all active Glimpse contracts* with honest verifiers on top of $\mathcal{L}_D$ sharing the same source chain $\mathcal{L}_S$, and avoid opening a new Glimpse if that value exceeds the security levels we discuss below.

To make matters worse, there may be Glimpse contracts for one source chain $\mathcal{L}_S$ on m different destination chains. In this case, the adversary is basically controlling every prover on every destination chain. In order to compute the cumulative value $\mathcal{G}$, an honest verifier would have to access each destination chain $\mathcal{L}_{Di}$. As this might be infeasible, we propose possible countermeasures: honest verifiers may use a public bulletin board where they announce their Glimpse contract, or use some heuristic to estimate $\mathcal{G}$, e.g., computing the cumulative value for their chain and multiplying it with the number of compatible destination chains. We leave a more rigorous analysis of how honest verifiers can compute $\mathcal{G}$ in the face of an adversary this powerful as future work. We assume honest verifiers can compute $\mathcal{G}$.

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

The adversary controlling all provers will bribe the miners, and *the bribe can be up to the cumulative value locked in* $\mathcal{G}$. To study the economic cost of a proof forgery attack, we need to compare the miners' expected gain when mining honestly to their expected gain when executing the attack. If the latter is higher than the former, then rational miners will launch the attack. Let R be the number of coins given as block reward on $\mathcal{L}_S$, V_S is the USD value of 1 coin of $\mathcal{L}_S$, $\mathsf{E}^{\mathsf{B}}_{L_S}[T]$ the number of expected blocks on $\mathcal{L}_S$ in T, and $_r$ the attacker's *relative* mining power on $\mathcal{L}_S$. Then, the expected gain for honest miners is given by:

$$\mathsf{E}[\mathsf{H}] = \mathsf{R} \cdot V_S \cdot \mathsf{E}^{\mathsf{B}}_{L_S}[T] \cdot \ _r; \tag{1}$$

We note that $0 \leq \ _r \leq 1$ and $_r <$, being the fraction of honest miners. Conversely, the miners' expected gain when executing the attack depends on how many blocks they need to forge, how much mining power they hold, and the fluctuation (we pessimistically consider a price drop) of the bribe value during the attack. Let $_c$ be the bribe value, $_i$ the percentage price drop of the bribe in the native asset of $\mathcal{L}_{D_i}$ (the i-th destination chain) over the duration of the attack, and V_{D_i} the USD value of 1 coin of $\mathcal{L}_{D_i}$. Being the attacker's hashing power (hashes per second), the number of hashes computed in T is $\mathsf{N} :=$ $\cdot T$. Considering N repeated, independent, and equally distributed hashes, and being $P_{v,T}$ the probability to find a valid hash given a target $\mathcal{T}$, the binomial probability that given $\mathcal{T}$ the attacker finds at least n valid blocks in time T is:

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

The miners' expected gain for the forgery attack is thus given by:

$$\mathsf{E}[\mathsf{F}] = \ _c \cdot \sum_{k=1}^{m} ((1 - \ _k) \cdot V_{D_k}) \cdot P^T_{n,T}; \tag{3}$$

For Glimpse to be economically secure, it has to be more profitable to honestly mine blocks rather than launch a proof forgery attack: $\mathsf{E}[\mathsf{F}] < \mathsf{E}[\mathsf{H}]$. This yields the upper bound for the cumulative number of coins $_c$ held in all active Glimpse instances on the same $\mathcal{L}_S$:

$$_c < \frac{\mathsf{R} \cdot V_S \cdot \mathsf{E}^{\mathsf{B}}_{L_S}[T] \cdot \ _r}{\sum_{k=1}^{m} ((1 - \ _k) \cdot V_{D_k}) \cdot P^T_{n,T}}; \tag{4}$$

We recall that any honest user willing to open a new Glimpse instance of value must first retrieve and compute the cumulative number of coins $^{0}_{c}$ for the Glimpse in $\mathcal{G}$ (on $\mathcal{L}_S$), and check that $+ \, ^{0}_{c} < \, _{c}$. The $_{c}$ varies considerably with varying parameters: in particular, the block reward and the USD value of the coins play a dominant role. As the block reward and the gap between currencies' values increases, the coins that can be securely locked in Glimpse increase.

We note that if miners with a large share of $_{r}$ redirect their mining power to forging a proof, the attack could be detected, as a drop in the network computational power can be observed. For example purposes, we make this simplification: let us assume all the active Glimpse instances are between Bitcoin ($\mathcal{L}_S$) and Liquid ($\mathcal{L}_D$) with proofs having, on average, 5 confirmation blocks and a lifetime $T \sim 1$ hour. We consider the largest Bitcoin target $\mathcal{T}$ over the year 2022 with 19 leading zeros, and $_{r} = 23\%$ of attacker hashing fraction, comparable to the largest Bitcoin mining pool in 2022. We also consider BTC and Liquid prices averaged on November 2022. In Figure 7 we show $_{c}$ as a function of the price drop . With minimal price drop over the attack time window, all the active Glimpse instances can hold ~ 100k bitcoin, i.e., 162 million USD in November 2022.

5.2. Censorship Attack

In general, we have so far assumed the ledgers underlying Glimpse fulfill consistency and liveness. However, a malicious verifier V could attempt violating liveness for $\mathcal{L}_S$ or $\mathcal{L}_D$ by launching a censorship attack. Indeed, V might try to censor the transaction(s) on $\mathcal{L}_S$ to unfairly get more coins on $\mathcal{L}_D$. Alternatively, V could reach the same goal by censoring P's transactions on $\mathcal{L}_D$. We analyze the censorship attack without distinguishing between $\mathcal{L}_S$ or $\mathcal{L}_D$, as V will *always attack the weakest chain*, i.e., the cheapest one to attack. A rational V will attack Glimpse only if censoring the transaction(s) leads to a larger expected profit than the one V would get behaving honestly. V can adopt two strategies to launch the attack: She can either bribe the miners not to include the transaction(s) or, if she is a miner herself, she can announce a feather fork attack. We discuss these two scenarios below.

Bribery. Closely following [29], we define the *bribing game* as a Markov game running in $T + 1$ sequential stages, a stage being the period between two mined blocks. In each stage, every miner can choose between censoring the transaction pointed out by V and therefore following the attack or including the transaction in her block, hence refusing to play the game. We define the bribing game as *safe* if, after eliminating the strictly dominated strategies, the only action for each miner in stage one is to play *refuse*.

The discussion moves from the following assumptions: (i) miners are rational, i.e., they always try to maximize their profit and, if they can choose, they always follow dominant strategies; (ii) miners do not create forks; (iii) each miner's relative hash power $_{r}$ is publicly known and remains constant during the attack; (iv) the attacker and the victim of the bribing attack have no hash power of their own; (v) all miners can see time-locked transactions that will be valid in the future; (vi) the Glimpse lifetime T is a time lock expressed in number of blocks, and each block generation is equivalent to a tick of the clock; finally, (vii) block rewards and fees generated outside the Glimpse protocol are constant and do not impact on the attack.

We refer to a miner whose relative hashing power is $_{r} < \frac{f}{\alpha}$ as a *weak miner*, and we let $_{w}$ be the sum of the relative hashing powers (or the sum of probabilities to be selected as next block proposer) of all weak miners in the system. We refer to f as the fee of the to-be-censored transaction, to as *the bribe, i.e., the economic value of the single Glimpse contract*. We reasonably consider $> f$. As proved in [29], the following theorem holds:

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

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

Therefore, to secure Glimpse from censorship attacks, we require $< \frac{f}{\mu_r}$ and T as in Theorem 3. For instance, to censor a transaction in Bitcoin considering the average fee per transaction being 1.4 USD (over July-October 2022) and 0.01% being the minimum fraction of hashing power for a miner, the value locked in a *single* Glimpse should not have been higher than 1400 USD.

Feather Fork. We now consider the case where V has some computational mining power . V can launch a feather fork attack by publicly announcing she will not include a specific transaction in her blocks, therefore forking the chain for a certain number of blocks. Concretely, suppose honest miners include such transaction a block B of the main chain: in this case, V forks the chain from block $B - 1$ and keeps forking until b blocks are appended to the main chain after the block holding the blacklisted transaction.

We define successful a *feather fork attack of order* b if an attacker manages to append $b + 1$ consecutive blocks to her own forked chain before other miners append at least b consecutive blocks to the main chain. Since rational miners do not want to lose their reward, they will avoid mining blocks containing the blacklisted transactions, as they are likely to be excluded from the main chain (while other blocks do not). We consider the case where V announces a feather fork attack of order b and promises a bribe (the bribe is again the value locked in Glimpse) to any miner that joins her in the fork endeavor. A miner with relative hash power $_{r}$ will follow the fork attack if $> \frac{f}{\mu_r^b}$. Put differently, Glimpse is safe against feather fork attacks with bribe payment if

$$\leq \frac{f}{\frac{b}{r}}. \tag{6}$$

To launch a successful attack, the attacker has to pay a lower bribe in the bribe attack rather than in the feather fork combined with bribing. In Figure 8 we show the USD

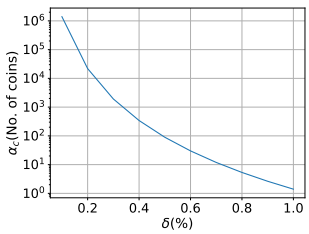

Figure 7: $_c$ as function of for $n = 5$, $T \sim 1$ hour, $_r$ = 23%, and $\mathcal{T}$ with 19 leading zeros.

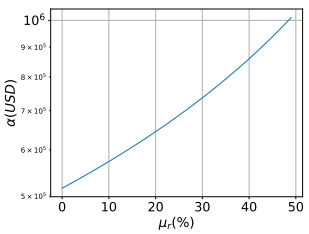

Figure 8: For $f = 1{:}4$, we show the USD value Glimpse that can hold for different $_r$.

value a single Glimpse can hold for different $_r$ assuming $f = 1{:}4$.

Secure Parameter Space for Glimpse. To have solid security guarantees, the economic value locked in Glimpse has to fulfill twofold conditions: First, to prevent proof forgery attacks, the *cumulative* value of all the active Glimpse instances having the same source chain (but potentially different destination chains) is bound by Equation (4). Second, to protect Glimpse against censorship attacks carried out via bribery or feather fork attack, the economic value of each *single* Glimpse instance is bound by Theorem 3 if the attacker has no computational power or by Equation (6) if the attacker has computational power.

6. Evaluation

We discuss the Glimpse costs in Ethereum- and Bitcoin-like chains and we analyze the computational and communication overhead.

Overhead in Ethereum-like Chains. We now consider the Ethereum main chain, but the same discussion also applies to any Ethereum-based chain, e.g., Ethereum Classic and Ethereum PoW. In Ethereum, the cost of a transaction is measured in *gas*, which together with a gas-price specified by the issuer of the transaction results in the fees expressed in the native currency. Every computation consumes an amount of gas proportional to its complexity, and every data that is stored on-chain consumes an amount of gas proportional to its length. The computational costs of Glimpse come from the proof $\mathcal{P}^n$ verification, which, for a given target $\mathcal{T}$, consists of a Merkle proof verification for a specific transaction (or description), the transaction body and block header reconstruction, and final hash comparison, as shown in Figure 5. A Merkle tree with k leaves has a Merkle proof of size $O(\log_2 k)$. This leads to Merkle proof verification cost scaling logarithmically in the number of transactions in a block. Each of the n confirmation blocks in $\mathcal{P}^n$ yields an overhead of 36k gas.

Besides these computational costs, the Glimpse contract has to initially store the source chain target $\mathcal{T}$ as well as either the to-be-published transaction hash or a transaction description. In Ethereum the data are stored in 32-bytes slots and for each slot 20k gas are consumed: this leads to 40k gas storage cost for the target and the transaction hash, or to 188k in the case of target and ∼300-bytes description. We have implemented an open-source cost evaluation which can be found in an anonymized Github repository [30].

Glimpse has lower on-chain costs compared to the overall costs of some optimized and naive relay solutions, such as Ethrelay [8] and zkRelay [7]. For Ethrelay, each block header submission results in an average cost of 280k gas, whereas the inclusion of a transaction is verified via SPV combined with an advanced search algorithm that checks for main chain membership. For relatively recent blocks, this leads to a gas consumption of 110k gas. Using zkRelay, the submission of a batch of blocks of arbitrary size costs 522k gas, including the validation of the zero-knowledge proof and the storage costs. The proof validation alone results in 351k gas. To verify that a transaction has been included in a block on the source blockchain, users have to provide the relay contract with a Merkle proof for verifying the block inclusion in the batch and a Merkle proof for the SPV.

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

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

To cope with the limited scripting capabilities of Bitcoin-like chains, when Taproot is supported, the parties can use Merkelized Abstract Syntax Trees (MAST) [31]. A MAST can compress many scripts into a single root of a Merkle tree and we use it to efficiently encode our Glimpse contract. The MAST first needs to be constructed and then exchanged off-chain. The MAST size depends on (i) the number of variable inputs and outputs in Desc, as their well-formedness needs to be checked with dedicated opcodes, (ii) the number of confirmation blocks in $\mathcal{P}^n$, (iii) the number of transactions in the block determining the number of levels in the Merkle tree, and (iv) the size of the description, being the static data to be hard-coded in the script. For example, in a single to-be-verified transaction Tx of ∼350 bytes, $n = 6$ confirmation blocks, one variable input or output, the upper bound for the MAST is 10 MB. For DNF formulas, parties need to compute and exchange the MAST for each literal. For a detailed discussion see Appendix D.

We provide a theoretical estimation for a Glimpse transaction size in Liquid, where we have Taproot and all the necessary string opcodes. Assuming two P2PKH inputs and one P2TR output for Tx_G, we have a transaction size of approximately 350 bytes. For Tx_P, assuming one P2TR input (Glimpse witness + selected script of the MAST) and two P2PKH outputs, the size is again roughly 350

bytes. Instead, considering one P2TR input and two P2PKH outputs, the size of $\mathsf{Tx_V}$ is about 200 bytes. Concretely, in November 2022, users' fees for $\mathsf{Tx_G}$ and $\mathsf{Tx_P}$ would amount to 1.5$ each, whereas for $\mathsf{Tx_V}$ to 0.84$. The total cost would be at most 3$, in line with the costs for standard transactions. In the *Setup* phase, parties need to exchange $\mathsf{Tx_G}$, $\mathsf{Tx_P}$ and $\mathsf{Tx_V}$, as well as the description for the to-be-verified transaction. For verifying a DNF formula with m literals, the parties need to exchange $4 \cdot 2^m$ transactions, and V has to send to P $2 \cdot 2^m$ signatures.

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

The computations overhead consist of the creation and verification of signatures, the creation and verification of the MAST, and the construction of the proofs, all of which can be performed using commodity hardware.

7. Related Work

The idea of chain relays first appeared with BTC Relay [6], 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 and not compensated by user's fees.

Westerkamp et al. [7] introduced zkRelay which batches multiple headers. Their validity is verified off-chain and proven on-chain via zkSNARKs. zkRelay has constant verification costs and releases the target ledger from processing and storing every single block header of the source blockchain. Although the on-chain costs are lower than for BTC Relay, a maintenance overhead for the off-chain computation and for on-chain storage remain. Furthermore, the users' costs for transaction inclusion verification are doubled, as both the block inclusion in the batch and the transaction inclusion in the block have to be verified.

Efficient relay contracts for Ethereum on Ethereum Classic (and vice versa) are even more challenging to design, as Ethereum has a complex and ASIC-resistant consensus in place. To this end, Frauenthaler et al. [8] propose Ethrelay, a relay that employs an optimistic approach: Block headers are optimistically accepted as valid and only validated on-demand. The computational costs per header are cut out, but the storage costs persist.

Zamyatin et al. [20] 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.

Protocol	*Commit on* L_S			*Verify & Commit on* L_D		Expressiveness
	Ass.	Comp.	Consensus	Ass.	Comp.	
Universal Atomic Swap [2]	Sync	①	Any	Sync	①	Secret-based logic (Adapt. Sigs)
HTLC Atomic Swap [5], [3], [4]	Sync	①	Any	Sync	①	Secret-based logic (HTLC)
Glimpse (this work)	Sync	①	PoW	Sync	②	DNF formulas over transactions (or descriptions)
Bidirectional chain relays [7], [8], [6], [32]	Sync	③	PoW, PoS	Sync	③	Arbitrary logic
XCLAIM [20], XCC [33]	TTP	①	PoW, PoS	Sync	③	Arbitrary logic

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

Another conceptually and technically different solution for cross-chain communication is atomic swaps, originally introduced by Herlihy [5]. Atomic swaps allow multiple parties to exchange assets across multiple blockchains in a distributed and coordinated manner. Herlihy designed a swap protocol that uses HTLCs, i.e., contracts storing a pair $(h; t)$ and ensuring that if the contract receives the secret s such that $h = \mathcal{H}(s)$ before time t has elapsed, then the ownership of the asset locked in the contract are transferred to the counter party. This secret-based solution is cheap and elegant, but, contrarily to Glimpse, it can only be used in an asymmetric setting, where the party posting a transaction on $\mathcal{L}_S$ cannot be the same one posting a transaction on $\mathcal{L}_D$, besides being limited in expressiveness.

Thyagarajan et al. [2] enhanced the compatibility of atomic swaps to all existing chains without hash locks or timelocks,enabling atomic swaps that are simultaneously non-custodial (no TTP), universal (compatible with all (current and future) blockchains), and multi-asset (supporting the exchange of multiple coins in a single atomic swap), but no other applications.

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

8. Conclusion

We presented Glimpse, a new *on-demand* trustless cross-chain communication primitive for PoW source chains. Glimpse is *efficient* in terms of on-chain costs and computational overhead, *expressive* for the information complex-

ity it can transfer across chains, *compatible* with Bitcoin-based chains. It is proven secure in the UC framework and through an economical security analysis, where we quantify the costs of breaking basic security assumptions. We demonstrated how the expressiveness of Glimpse enables the design of sophisticated cross-chain applications, such as lending, backed-assets, and multiple oracle attestations. The crucial assumption on the underlying scripting language is the capability to verify Merkle proofs: Glimpse is already compatible with several cryptocurrencies and we believe this work has the potential to shape the evolution of others too, by adding arguments in favor of, e.g., the `OP_CAT` opcode, whose inclusion in Bitcoin is currently under discussion.

This work opens up a number of interesting research directions. For instance, we intend to explore how to support different consensus mechanisms, such as Proof-of-Stake or Proof-of-Space. Furthermore, we plan to investigate how to lift the Glimpse construction off-chain, in order to make it compatible with the Lightning Network and open up the design of off-chain applications operating over different, and possibly cross-chain, payment channel networks.

Acknowledgments

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

References

[1] 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.

[2] 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.

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

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

[5] M. Herlihy, "Atomic cross-chain swaps," *CoRR*, vol. abs/1801.09515, 2018. [Online]. Available: http://arxiv:org/abs/1801:09515

[6] "Btc relay," http://btcrelay:org/, 2016.

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

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

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

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

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

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

[13] 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.

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

[15] L. Aumayr, P. M. Sanchez, A. Kate, and M. Maffei, "Donner: UTXO-based virtual channels across multiple hops," 2021, https://eprint:iacr:org/2021/855.

[16] 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.

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

[18] 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.

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

[20] 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.

[21] J. Katz and Y. Lindell, *Introduction to Modern Cryptography, Second Edition*, 2nd ed. Chapman & Hall/CRC, 2014.

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

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

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

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

[29] T. Nadahalli, M. Khabbazian, and R. Wattenhofer, "Timelocked bribing." Berlin, Heidelberg: Springer-Verlag, 2021. [Online]. Available: https://doi:org/10:1007/978-3-662-64322-8_3

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

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

[32] M. Westerkamp and M. Diez, "Verilay: A verifiable proof of stake chain relay," in *IEEE International Conference on Blockchain and Cryptocurrency, ICBC 2022, Shanghai, China, May 2-5, 2022*, 2022.

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

[34] 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.

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

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

[37] "Bitcoin core," https://github.com/bitcoin/bitcoin/blob/master/src/script/script.h, 2022.

Appendix A. Modeling Glimpse in the UC-Framework

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

We assume synchronous communication, i.e., the protocol execution proceeds in synchronized rounds. We follow [34], [17], formalizing these rounds with a global ideal functionality $\mathcal{G}_{Clock}$ which can be seen as a global clock. At a high level, this functionality proceeds to the next round only after all honest parties indicate that they are ready to do so. Every party knows what the current round is.

We model message exchange between parties via authenticated communication channels with guaranteed delivery after one round. This notion is formalized with the functionality $\mathcal{G}_{GDC}$ (e.g., [17]), and basically, this means that if a party P sends a message to Q in round t, Q will receive this message exactly at the beginning of round $t+1$. The adversary $\mathcal{A}$ has the power to observe the content of messages between parties and can reorder any messages sent within the same round, but $\mathcal{A}$ cannot delay, modify or censor messages or insert new messages on an honest party's behalf. We assume that any computation made by entities and communication that does not involve two parties, but rather a special entity such as $\mathcal{A}$ or $\mathcal{Z}$, takes zero rounds.

Modeling the Ledger. For modeling the ledger we refer to the functionality $\mathcal{G}_{Ledger}$ as defined in [28]. Concretely, we use a concrete instantiation also as specified in [28], where the parameters are chosen such that the ledger achieves both *liveness* and *consistency* (or just consistency), which is defined in [22]. 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 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 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 $\mathsf{EXEC}^{\mathcal{F}_{prelim}}_{,\mathcal{A},\mathcal{Z}}(\ ; z)$ the output of $\mathcal{Z}$ interacting with and $\mathcal{A}$ on input a security parameter and an auxiliary input z. Further, we denote $_{\mathcal{F}}$ as the ideal protocol of the ideal functionality $\mathcal{F}_{Glimpse}$ with access to the same preliminary functionalities $\mathcal{F}_{prelim}$. $_{\mathcal{F}}$ is a trivial protocol where parties merely forward any input to $\mathcal{F}_{Glimpse}$. The output of an environment $\mathcal{Z}$ on input and an auxiliary input z interacting with $_{\mathcal{F}}$ and an ideal world adversary $\mathcal{S}$ (also called *simulator)* is denoted as $\mathsf{EXEC}^{\mathcal{F}_{prelim}}_{\phi_{\mathcal{F}},\mathcal{S},\mathcal{Z}}(\ ; z)$.

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

Definition 3. A real world protocol 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

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

where $\approx^c$ denotes computational indistinguishability.

A.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 $_D$, 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., $[\mathsf{Tx}] \leftarrowtail \mathsf{Desc}$,

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 C).

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 $(m) \xrightarrow{t} X$ as sending message (m) to party X in round t and denote $(m) \xleftarrow{t} X$ 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.

$\mathcal{F}_{Glimpse}(\mathcal{L}_S, \mathcal{L}_D, \ \ _D)$
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
$_D$ 2 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:

... a set of tuples $(\mathtt{id}; \mathcal{F}_S; T_P; T_V; n; P; V; [\mathsf{outcome}_i]; \mathsf{Tx}_G)$, where $\mathtt{id}$ 2 $f0; 1g$ 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 U. $\mathcal{F}_S$ is a logical formula as defined in Figure 2. Further, $T_P; T_V; n$ 2 N, and $[\mathsf{outcome}_i]$ is a list of outcomes, which in turn are tuples $(\mathsf{outcome}{:}P; \mathsf{outcome}{:}V)$ 2 N^2.

Setup

1) Upon $(\mathtt{SETUP}; \mathtt{id}; \mathcal{F}_S; P; [\mathsf{outcome}_i]_{i2[1;r]}; T_P; T_V; n; finputPg; finputVg)$ - 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 $\mathsf{outcome}_i := (0; \ )$.
   d) For each $\mathsf{outcome}_i$ it must hold that $\mathsf{outcome}_i{:}P + \mathsf{outcome}_i{:}V =$ for some number
   e) $T_V > T_P$ are both times in the future
   f) n 2 N
   g) $finputVg$ is a (potentially empty) set of inputs under control of V and $finputPg$ is a (potentially empty) set of inputs under control of P
   h) $jfinputVg \ [\ finputPgj > 0$ and the sum of coins stored in $finputVg \ [\ finputPg$ + d
   i) If these checks hold continue, else go idle.
2) Send $(\mathtt{id}; \mathcal{F}_S; [\mathsf{outcome}_i]_{i2[1;r]}; T_P; T_V; n; finputPg; finputVg)$, $\overset{+1}{!}$ P, receive $(\mathtt{id})$ $\overset{+1}{-}$ P.
3) At round $_1$ + 1 + $_D$, if a transaction Tx_G appears on $\mathcal{L}_D$ which takes $finputPg$ and $finputVg$ as input and has at least one output holding coins, add $(\mathtt{id}; \mathcal{F}_S; T_P; T_V; n; P; V; [\mathsf{outcome}_i]_{i2[1;r]}; \mathsf{Tx}_G)$ in .

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

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

For every $(\mathtt{id}; \mathcal{F}_S; T_P; T_V; n; P; V; [\mathsf{outcome}_i]_{i2[1;r]}; \mathsf{Tx}_G)$ in , if current round is smaller than T_P, do the following.

1) If there is a transaction Tx on $\mathcal{L}_D$, such that Tx spends output of Tx_G and has two outputs $_P := (x; \mathsf{OneSig}(\mathsf{pk}_P))$ and $_V := (y; \mathsf{OneSig}(\mathsf{pk}_P))$, s.t. x $\mathsf{outcome}_i{:}P$ and y $\mathsf{outcome}_i{:}V$ corresponds to the k-th element in the list $[\mathsf{outcome}_i]_{i2[1;r]}$. Check the k-th row in the truth table corresponding to $\mathcal{F}_S$.
2) For each description Desc 2 $\mathcal{F}_S$ which is set to *true* in the k-th row in the truth table, a transaction Tx_i with n subsequent blocks, s.t. $[\mathsf{Tx}_i]$ - Desc, must be on $\mathcal{L}_S$. Additionally, for

each description Desc 2 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. $[\mathsf{Tx}_j]$ - Desc on L_S. If this does not hold, output $(\mathsf{id}, \mathtt{error})$.

(b) Strong atomicity (Functionality F_S $_{Glimpse}$)

Verify & Commit on L_D from (a) (in every round)

Verify & Commit on L_D: P (in every round)

For every $(\mathsf{id}, F_S, T_P, T_V, n, P, V, [\mathsf{outcome}_i]_{i \in [1;r]}, \mathsf{Tx}_G)$ in where P is honest and of Tx_G is unspent, do the following.

1) If current round is T_P $_D$. Let $\{\mathsf{Tx}_i\}$ be the set of all transactions on 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 Desc $\in F_S$ where $[\mathsf{Tx}_i]$ - Desc. Evaluate the statement F_S by setting to *true* all descriptions for $[\mathsf{Tx}_i]$ whose corresponding transaction Tx_i is on 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 L_D after at most 2 $_D$ rounds, such that Tx spends output of Tx_G and has two outputs $_P$:= $(x, \mathsf{OneSig}(\mathsf{pk}_P))$ and $_V$:= $(y, \mathsf{OneSig}(\mathsf{pk}_P))$, s.t. x $\mathsf{outcome}_i{:}P$ and y $\mathsf{outcome}_i{:}V$ of the k-th element in the list $[\mathsf{outcome}_i]_{i \in [1;r]}$. If no such transaction appears within said time, output $(\mathsf{id}, \mathtt{error})$.

Verify & Commit on L_D: V (in every round)

For every $(\mathsf{id}, F_S, T_P, T_V, n, P, V, [\mathsf{outcome}_i]_{i \in [1;r]}, \mathsf{Tx}_G)$ in where V is honest and of Tx_G is unspent, do the following.

1) If current round is T_P $_D$. Let $\{\mathsf{Tx}_i\}$ be the set of all transactions on 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 Desc $\in F_S$ where $[\mathsf{Tx}_i]$ - Desc. Evaluate the statement F_S by setting to *true* all descriptions for $[\mathsf{Tx}_i]$ whose corresponding transaction Tx_i is on 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 of Tx_G and as output := (; $\mathsf{OneSig}(\mathsf{pk}_V)$), must appear on L_D within $_D$ rounds. If no such transaction appears within said time, output $(\mathsf{id}, \mathtt{error})$.

A.2. Glimpse Protocol

In this section we present the formal UC protocol of Glimpse. 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 4.1, 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 $\mathcal{L}_S$, $\mathcal{L}_D$, $_D$ (which is explicitly stated for readability, even though it is implicitly given by $\mathcal{L}_D$), as well as over a function parameter $\mathit{gen}\mathcal{P}$, which should generate a proof $\mathcal{P}$ for $\mathcal{L}_D$ that a transaction has appeared on $\mathcal{L}_S$. The two ledger parameters $\mathcal{L}_S$ and $\mathcal{L}_D$ have to have the same properties as the ledger parameters of the functionality, which 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 $\mathit{gen}\mathcal{P}$. The proof generation function is specific to $\mathcal{L}_S$ and $\mathcal{L}_D$ and is parameterized over a transaction Tx, a description Desc (as defined in Figure 2) and a consensus parameter *cp* that is specific to $\mathcal{L}_S$. The function generates a proof proving that a transaction Tx that matches description Desc, i.e., $[\mathsf{Tx}] \leftarrow \mathsf{Desc}$, 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 6, which uses n as consensus parameter *cp*. We require this function $\mathit{gen}\mathcal{P}$ to have the following properties: *complete* and *T-sound*.

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

Definition 5. A function $\mathit{gen}\mathcal{P}$ 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 4.1, 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 6 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 , we put a restriction on the environment to only send an $\mathcal{F}_S$, where every $\mathsf{Desc} \in \mathcal{F}_S$ has a newly generated random value in its body. In practice, this can be achieved by P and V running a pre-setup before the protocol, where they both generate a value of length 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 $(0, \mathsf{OP\_RETURN}(r_P || r_V))$.

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

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

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

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

1) *Setup:* In this phase, the parties V and P create the necessary transactions to set up a Glimpse contract corresponding to the input data they received, and post the transaction $\mathsf{Tx_G}$ carrying the Glimpse contract. In more detail, if we again consider $\mathcal{F}_S$ where the descriptions are boolean variables and the resulting truth table (as in Appendix A.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 C in Figure C.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 B.

Macros for $(\mathcal{L}_S, \mathcal{L}_D, \ , \mathit{genP})$

genTxsFromF($; \mathcal{F}_S, [\mathsf{outcome}_i]_{i \in [1;r]}, T_P, T_V, cp, n,$ finputsg)

1) Create transaction $\mathsf{Tx_G}$, with inputs $\mathsf{Tx_G}$.input := finputsg and outputs $\mathsf{Tx_G}$.output := $[$ $_i g_{i \in [1;var]}$ as list, where var is the number of Desc in $\mathcal{F}_S$, s.t. := (; MuSig(pk_P, pk_V) _ (OneSig(pk_V) ^ T_V)) and $_i$:= ($_i$; (scriptG($\mathsf{Desc}_i, T_P, cp, n, (P,V)$) ^ OneSig($\mathsf{pk}_P$))) _ 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 C or Figure C.1) as follows:
   b) $\mathsf{Tx_T}$ takes as inputs all outputs $_i$ of $\mathsf{Tx_G}$, where the corresponding input variable Desc_i is set to *true* and $\mathsf{Tx_F}$ takes as inputs all outputs $_i$ of $\mathsf{Tx_G}$, where the corresponding input variable Desc_i is set to *false*.
   c) The single output of $\mathsf{Tx_T}$ is := (; OneSig(pk_P)) and the single output of $\mathsf{Tx_F}$ is := (; OneSig(pk_P) ^ T_P _ $truevar$), where $truevar$ is a disjunction of 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 of $\mathsf{Tx_G}$ as well as both outputs of $\mathsf{Tx_T}$ and $\mathsf{Tx_F}$. Its output is := (outcome$_i$:P; OneSig(pk_P)).
4) We define the result of this as set of tuples $(\mathsf{Tx_T}_i, \mathsf{Tx_F}_i, \mathsf{Tx_P}_i)g_{i \in [1;row]}$, where row is the number of rows in the truth table.
5) Return $(\mathsf{Tx_G}, (\mathsf{Tx_T}_i, \mathsf{Tx_F}_i, \mathsf{Tx_P}_i)g_{i \in [1;row]})$

postTxsFP($\mathcal{F}_S, T_P, n, \mathsf{Tx_G}, (\mathsf{Tx_T}_i, \mathsf{Tx_F}_i, \mathsf{Tx_P}_i,$ $_V(\mathsf{Tx_F}_i)$; $_V(\mathsf{Tx_P}_i))g_{i \in [1;row]}$)

1) If current time is T_P $_D$ and output 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. [$\mathsf{Tx}_i$] - 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., $(\mathsf{Tx_T}_k, \mathsf{Tx_F}_k, \mathsf{Tx_P}_k,$ $_V(\mathsf{Tx_F}_k)$; $_V(\mathsf{Tx_P}_k))$
4) For each Tx_i and Tx_i where Tx_i is on $\mathcal{L}_S$, construct a proof using the Construct $P_i^n(\mathsf{Tx}_i, \mathsf{Desc}_i, n)$ function defined in Figure 6, yielding a set of proofs $\{P^n\}$.
5) Generate signatures $_P(\mathsf{Tx_T}_k)$, $_P(\mathsf{Tx_F}_k)$, $_P(\mathsf{Tx_P}_k)$.
6) Send a message "post" for transaction $\mathsf{Tx_T}_k$ with $_P(\mathsf{Tx_T}_k)$ and $\{P^n\}$ as witnesses to functionality $\mathcal{L}_D$.
7) Send a message "post" for transaction $\mathsf{Tx_F}_k$ with $_V(\mathsf{Tx_F}_k)$ and $_P(\mathsf{Tx_F}_k)$ as witnesses to functionality $\mathcal{L}_D$.
8) At time T_P, send a message "post" for transaction $\mathsf{Tx_P}_k$ with $_V(\mathsf{Tx_P}_k)$ and $_P(\mathsf{Tx_P}_k)$ to $\mathcal{L}_D$.

postTxsFV($\mathcal{F}_S, T_P, T_V, n, P, V,$ [outcome$_i$]$_{i \in [1;r]}$, $\mathsf{Tx_G}$)

1) If current time is after T_V and output of $\mathsf{Tx_G}$ is unspent, send a "post" message for a transaction Tx to $\mathcal{L}_D$, where Tx takes as input of $\mathsf{Tx_G}$ and as output := (; OneSig(pk_V)), generate and use signature $_V(\mathsf{Tx})$ as a witness.
2) Else, if current time is before T_P and a transaction $\mathsf{Tx_F}$ (as defined in step 3a of genTxsFromF) is on $\mathcal{L}_S$ and a transaction Tx^0 is on $\mathcal{L}_S$, s.t. it fulfills one of the descriptions of the output of $\mathsf{Tx_F}$, i.e., [Tx^0] - Desc, do the following.
   a) Construct a proof using the Construct $P^n(\mathsf{Tx}^0, \mathsf{Desc}, n)$ function defined in Figure 6,

yielding P^n.

b) Send a message "post" for a transaction Tx^{00} to L_D, where Tx^{00} takes as input the output of Tx_F and as output := (; OneSig(pk_V)), using P^n and $_V(\mathsf{Tx}^{00})$ as witnesses.

Protocol $(L_S; L_D; \ _D; genP)$

Parameters:
L_S, L_D ... two instances of G_{Ledger}, representing the source and destination blockchain. We let *cp* be the difficulty of L_D, i.e., this is a parameter of the functionality G_{Ledger}.
$_D$ 2 N ... the blockchain delay of 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.
genP ... 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 L_S. The function generates a proof that a transaction Tx that matches description Desc, i.e., [Tx] - Desc, is on L_S, as a witness that is readable by the scripting of L_D. In our protocol instantiation, we use "Construct P_i^n" defined in Figure 6, which uses n as consensus parameter *cp*.

Variables:

$_P$ ... a set of tuples $(\mathtt{id}; F_S; T_P; T_V; n; \mathsf{Tx}_G; f(\mathsf{Tx}_{Ti}; \mathsf{Tx}_{Fi}; \mathsf{Tx}_{Pi\ V}(\mathsf{Tx}_{Fi}); \ _V(\mathsf{Tx}_{Pi}))g_{i2[1;row]})$, where $\mathtt{id}$ 2 $f0; 1g$ 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 U. F_S is a logical formula as defined in Figure 2. Further, $T_P; T_V; n$ 2 N, and $[\mathsf{outcome}_i]$ is a list of outcomes, which in turn are tuples $(\mathsf{outcome}{:}P; \mathsf{outcome}{:}V)$ 2 N^2. Tx_G is a transaction and $f(\mathsf{Tx}_{Ti}; \mathsf{Tx}_{Fi}; \mathsf{Tx}_{Pi\ V}(\mathsf{Tx}_{Fi}); \ _V(\mathsf{Tx}_{Pi}))g_{i2[1;row]}$ is a set of tuples containing transactions and signatures.
$_V$ ... a set of tuples $(\mathtt{id}; F_S; T_P; T_V; n; \mathsf{Tx}_G)$, where $\mathtt{id}$ 2 $f0; 1g$ 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 U. F_S is a logical formula as defined in Figure 2. Further, $T_P; T_V; n$ 2 N. Tx_G is a transaction.

Setup

Verifier V

1) Upon $(\mathtt{SETUP}; \mathtt{id}; F_S; P; [\mathsf{outcome}_i]_{i2[1;r]}; T_P; T_V; n; inputPg; inputVg) \ ^V$ - 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 F_S (in other words, the number of rows in the truth table when considering all descriptions in F_S)
   b) For all rows i of the truth table generated by F_S, where the result is *false*, it must hold that $\mathsf{outcome}_i := (0; \ )$.
   c) For each $\mathsf{outcome}_i$ it must hold that $\mathsf{outcome}_i{:}P$ + $\mathsf{outcome}_i{:}V$ = for some number
   d) $T_V > T_P$ are both times in the future
   e) n 2 N
   f) $inputVg$ is a (potentially empty) set of inputs under control of V and $inputPg$ is a (potentially empty) set of inputs under control of P
   g) $jinputVg$ [$inputPgj > 0$ and the sum of coins stored in $inputVg$ [$inputPg$ + d
   h) If these checks hold continue, else go idle.
2) $(\mathsf{Tx}_G; f(\mathsf{Tx}_{Ti}; \mathsf{Tx}_{Fi}; \mathsf{Tx}_{Pi})g_{i2[1;row]})$:= genTxsFromF(; F_S; $[\mathsf{outcome}_i]_{i2[1;r]}$; T_P; T_V; *cp*; n; finputsg)
3) Sign each transaction Tx_{Fi} and Tx_{Pi} in $f(\mathsf{Tx}_{Ti}; \mathsf{Tx}_{Fi}; \mathsf{Tx}_{Pi})g_{i2[1;row]}$ and append the signatures to each tuple yielding a set $f(\mathsf{Tx}_{Ti}; \mathsf{Tx}_{Fi}; \mathsf{Tx}_{Pi}; \ _V(\mathsf{Tx}_{Fi}); \ _V(\mathsf{Tx}_{Pi}))g_{i2[1;row]}$
4) Sign Tx_G yielding $_V(\mathsf{Tx}_G)$
5) Send $(\mathtt{open\text{-}req}; \mathtt{id}; [\mathsf{outcome}_i]_{i2[1;r]}; T_P; T_V; n; inputPg; inputVg; \mathsf{Tx}_G; \ _V(\mathsf{Tx}_G); f(\mathsf{Tx}_{Ti}; \mathsf{Tx}_{Fi}; \mathsf{Tx}_{Pi}; \ _V(\mathsf{Tx}_{Fi}); \ _V(\mathsf{Tx}_{Pi}))g_{i2[1;row]}) \ ^{V}, _P$.
6) Add $(\mathtt{id}; F_S; T_P; T_V; n; \mathsf{Tx}_G)$ to $_V$.

Prover P

7) Upon $(\mathtt{open\text{-}req}; \mathtt{id}; [\mathsf{outcome}_i]_{i2[1;r]}; T_P; T_V; n; inputPg; inputVg; \mathsf{Tx}_G; \ _V(\mathsf{Tx}_G); f(\mathsf{Tx}_{Ti}; \mathsf{Tx}_{Fi}; \mathsf{Tx}_{Pi}; \ _V(\mathsf{Tx}_{Fi}); \ _V(\mathsf{Tx}_{Pi}))g_{i2[1;row]}) \ ^P$ - V.
8) Perform checks of protocol Setup phase, steps 1a through 1h. If one or more fail, go idle.
9) Verify that $(\mathsf{Tx}_G; f(\mathsf{Tx}_{Ti}; \mathsf{Tx}_{Fi}; \mathsf{Tx}_{Pi})g_{i2[1;row]})$ is the result of genTxsFromF(; F_S; $[\mathsf{outcome}_i]_{i2[1;r]}$; T_P; T_V; *cp*; n; finputsg). If not, go idle.
10) For each entry of the set $f(\mathsf{Tx}_{Ti}; \mathsf{Tx}_{Fi}; \mathsf{Tx}_{Pi}; \ _V(\mathsf{Tx}_{Fi}); \ _V(\mathsf{Tx}_{Pi}))g_{i2[1;row]}$, check that $_V(\mathsf{Tx}_{Fi})$ and $_V(\mathsf{Tx}_{Pi})$ are valid signatures of V for Tx_{Fi} and Tx_{Pi}, respectively.
11) Verify that $_V(\mathsf{Tx}_G)$ is a valid signature of V for Tx_G.
12) $(\mathtt{id}; F_S; [\mathsf{outcome}_i]_{i2[1;r]}; T_P; T_V; n; inputPg; inputVg) \ ^P$, $!$ Z.
13) Upon $(\mathtt{id}) \ ^P$ - Z, sign Tx_G yielding $_P(\mathsf{Tx}_G)$.
14) Send a message "post" for transaction Tx_G with $_P(\mathsf{Tx}_G)$ and $_V(\mathsf{Tx}_G)$ as witnesses to functionality L_D
15) If it appears on the ledger of L_D at round $_P$ 1 + $_P$ + $_D$, add $(\mathtt{id}; F_S; T_P; T_V; n; \mathsf{Tx}_G; f(\mathsf{Tx}_{Ti}; \mathsf{Tx}_{Fi}; \mathsf{Tx}_{Pi\ V}(\mathsf{Tx}_{Fi}); \ _V(\mathsf{Tx}_{Pi}))g_{i2[1;row]})$ in $_P$.

Verify & Commit on L_D: P (in every round)

For every $(\mathtt{id}; F_S; T_P; T_V; n; \mathsf{Tx}_G; f(\mathsf{Tx}_{Ti}; \mathsf{Tx}_{Fi}; \mathsf{Tx}_{Pi}; \ _V(\mathsf{Tx}_{Fi}); \ _V(\mathsf{Tx}_{Pi}))g_{i2[1;row]})$ in $_P$, execute postTxsFP(F_S; T_P; n; Tx_G; $f(\mathsf{Tx}_{Ti}; \mathsf{Tx}_{Fi}; \mathsf{Tx}_{Pi}; \ _V(\mathsf{Tx}_{Fi}); \ _V(\mathsf{Tx}_{Pi}))g_{i2[1;row]}$).

Verify & Commit on L_D: V (in every round)

For every $(\mathtt{id}; F_S; T_P; T_V; n; \mathsf{Tx}_G)$ in $_V$, execute postTxsFV($_P$; $_V$; F_S; *cp*; n; T).

Appendix B. 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 A.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 5.1 under

which conditions the proof generation function "Construct $\mathcal{P}_i^n$" defined in Figure 6 is T-*sound*.

Necessity of *strictly randomized input* (for Theorems 1 and 2). As explained in Appendix A.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 $gen\mathcal{P}$ being *complete* (for Theorem 2). Similarly, we require $gen\mathcal{P}$ 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 $(\texttt{open-req}, \mathsf{id}, [\mathsf{outcome}_i]_{i2[1;r]}, T_P, T_V, n, \mathit{finputPg}, \mathit{finputVg}, \mathsf{Tx}_G, \sigma_V(\mathsf{Tx}_G), f(\mathsf{Tx}_{Ti}, \mathsf{Tx}_{Fi}, \mathsf{Tx}_{Pi}, \sigma_V(\mathsf{Tx}_{Fi}), \sigma_V(\mathsf{Tx}_{Pi}))g_{i2[1;row]})$, $\overset{V}{!}$ P, send $(\mathsf{OK}, \mathsf{id}, P, [\mathsf{outcome}_i]_{i2[1;r]}, T_P, T_V, n, \mathit{finputPg}, \mathit{finputVg})$, $\overset{V}{!}$ $F_{Glimpse}$, If not, go idle.
2) For each entry of the set $f(\mathsf{Tx}_{Ti}, \mathsf{Tx}_{Fi}, \mathsf{Tx}_{Pi}, \sigma_V(\mathsf{Tx}_{Fi}), \sigma_V(\mathsf{Tx}_{Pi}))g_{i2[1;row]}$, check that $\sigma_V(\mathsf{Tx}_{Fi})$ and $\sigma_V(\mathsf{Tx}_{Pi})$ are valid signatures of V for Tx_{Fi} and Tx_{Pi}, respectively.
3) Verify that $\sigma_V(\mathsf{Tx}_G)$ is a valid signature of V for Tx_G.
4) Upon (id) $\overset{P}{\text{-}}$ P, sign Tx_G on P's behalf yielding $\sigma_P(\mathsf{Tx}_G)$.
5) Send a message "post" for transaction Tx_G with $\sigma_P(\mathsf{Tx}_G)$ and $\sigma_V(\mathsf{Tx}_G)$ as witnesses to functionality $\mathcal{L}_D$
6) If it appears on the ledger of $\mathcal{L}_D$ at round $\tau_P + 1 + \Delta_D$, let $(P) := (\mathsf{id}, F_S, T_P, T_V, n, \mathsf{Tx}_G, f(\mathsf{Tx}_{Ti}, \mathsf{Tx}_{Fi}, \mathsf{Tx}_{Pi}, \sigma_V(\mathsf{Tx}_{Fi}), \sigma_V(\mathsf{Tx}_{Pi}))g_{i2[1;row]})$.

b) Case P is dishonest, V honest

1) Upon V sending $(\mathsf{SETUP}, \mathsf{id}, F_S, P, [\mathsf{outcome}_i]_{i2[1;r]}, T_P, T_V, n, \mathit{finputPg}, \mathit{finputVg})$, $\overset{V}{!}$ $F_{Glimpse}$, perform checks of protocol Setup phase, steps 1a through 1h. If one or more fail, go idle.
2) $(\mathsf{Tx}_G, f(\mathsf{Tx}_{Ti}, \mathsf{Tx}_{Fi}, \mathsf{Tx}_{Pi})g_{i2[1;row]}) := \mathsf{genTxsFromF}(\ , F_S, [\mathsf{outcome}_i]_{i2[1;r]}, T_P, T_V, cp, n, \mathit{finputsg})$
3) Sign each transaction Tx_{Fi} and Tx_{Pi} in $f(\mathsf{Tx}_{Ti}, \mathsf{Tx}_{Fi}, \mathsf{Tx}_{Pi})g_{i2[1;row]}$ on behalf of V and append the signatures to each tuple yielding a set $f(\mathsf{Tx}_{Ti}, \mathsf{Tx}_{Fi}, \mathsf{Tx}_{Pi}, \sigma_V(\mathsf{Tx}_{Fi}), \sigma_V(\mathsf{Tx}_{Pi}))g_{i2[1;row]}$
4) Sign Tx_G yielding $\sigma_V(\mathsf{Tx}_G)$
5) Send $(\mathsf{open\text{-}req}, \mathsf{id}, [\mathsf{outcome}_i]_{i2[1;r]}, T_P, T_V, n, \mathit{finputPg}, \mathit{finputVg}, \mathsf{Tx}_G, \sigma_V(\mathsf{Tx}_G), f(\mathsf{Tx}_{Ti}, \mathsf{Tx}_{Fi}, \mathsf{Tx}_{Pi}, \sigma_V(\mathsf{Tx}_{Fi}), \sigma_V(\mathsf{Tx}_{Pi}))g_{i2[1;row]})$ $\overset{V}{!}$ P
6) Let $(V) := (\mathsf{id}, F_S, T_P, T_V, n, \mathsf{Tx}_G)$.

c) Case P is honest, V honest

1) Upon V sending $(\mathsf{OK}, \mathsf{id}, P, [\mathsf{outcome}_i]_{i2[1;r]}, T_P, T_V, n, \mathit{finputPg}, \mathit{finputVg})$, $\overset{V}{!}$ $F_{Glimpse}$, perform checks of protocol Setup phase, steps 1a through 1h. If one or more fail, go idle.
2) $(\mathsf{Tx}_G, f(\mathsf{Tx}_{Ti}, \mathsf{Tx}_{Fi}, \mathsf{Tx}_{Pi})g_{i2[1;row]}) := \mathsf{genTxsFromF}(\ , F_S, [\mathsf{outcome}_i]_{i2[1;r]}, T_P, T_V, cp, n, \mathit{finputsg})$
3) Sign each transaction Tx_{Fi} and Tx_{Pi} in $f(\mathsf{Tx}_{Ti}, \mathsf{Tx}_{Fi}, \mathsf{Tx}_{Pi})g_{i2[1;row]}$ on behalf of V and append the signatures to each tuple yielding a set $f(\mathsf{Tx}_{Ti}, \mathsf{Tx}_{Fi}, \mathsf{Tx}_{Pi}, \sigma_V(\mathsf{Tx}_{Fi}), \sigma_V(\mathsf{Tx}_{Pi}))g_{i2[1;row]}$
4) Sign Tx_G on behalf of V yielding $\sigma_V(\mathsf{Tx}_G)$.
5) Let $(V) := (\mathsf{id}, F_S, T_P, T_V, n, \mathsf{Tx}_G)$.
6) Sign Tx_G on behalf of P yielding $\sigma_P(\mathsf{Tx}_G)$.
7) Send a message "post" for transaction 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 $\tau_P + 1 + \Delta_D$, let $(P) := (\mathsf{id}, F_S, T_P, T_V, n, \mathsf{Tx}_G, f(\mathsf{Tx}_{Ti}, \mathsf{Tx}_{Fi}, \mathsf{Tx}_{Pi}, \sigma_V(\mathsf{Tx}_{Fi}), \sigma_V(\mathsf{Tx}_{Pi}))g_{i2[1;row]})$.

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

P is honest

For every (key, value) pair P, $(\mathsf{id}, F_S, T_P, T_V, n, \mathsf{Tx}_G, f(\mathsf{Tx}_{Ti}, \mathsf{Tx}_{Fi}, \mathsf{Tx}_{Pi}, \sigma_V(\mathsf{Tx}_{Fi}), \sigma_V(\mathsf{Tx}_{Pi}))g_{i2[1;row]})$ in , execute $\mathsf{postTxsFP}(F_S, T_P, n, \mathsf{Tx}_G, f(\mathsf{Tx}_{Ti}, \mathsf{Tx}_{Fi}, \mathsf{Tx}_{Pi}, \sigma_V(\mathsf{Tx}_{Fi}), \sigma_V(\mathsf{Tx}_{Pi}))g_{i2[1;row]})$ on behalf of P.

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

P is honest

For every (key, value) pair V, $(\mathsf{id}, F_S, T_P, T_V, n, \mathsf{Tx}_G)$ in , execute $\mathsf{postTxsFV}(\sigma_P, \sigma_V, F_S, cp, n, T)$ on behalf of V.

Appendix C. DNF Formulas with Glimpse

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

Let us consider two oracles $\mathcal{O}_1$ and $\mathcal{O}_2$ operating on $\mathcal{L}_S$ (they could also be on different ledgers), and regularly posting information about a real-world event. On $\mathcal{L}_D$, prover P and verifier V, e.g., bet on a specific outcome of the event

by locking $\frac{\alpha}{2}$ coins each in the Glimpse transaction $\mathsf{Tx_G}$ and condition the spendability of the coins to a specific outcome being attested by at least one of the two oracles. If either $\mathcal{O}_1$ or $\mathcal{O}_2$ attests the desired outcome for the event, P can get the coins by providing a valid proof $\mathcal{P}^n$; otherwise, if the coins are still unspent V can claim the coins after T. We recall that the selected outcome for the event is defined within the descriptions, and hard coded within $\mathsf{Tx_G}$. We let Desc_1 and Desc_2 be descriptions for $\mathcal{O}_1$ and $\mathcal{O}_2$, respectively, and we let the to-be-verified DNF formula be $\mathcal{F}_S = (\mathsf{Desc}_1 \wedge \neg\mathsf{Desc}_2) \vee (\neg\mathsf{Desc}_1 \wedge \mathsf{Desc}_2) \vee (\mathsf{Desc}_1 \wedge \mathsf{Desc}_2)$. The Glimpse protocol proceeds as follows.

Setup. Let $_P$ and $_V$ be unspent outputs on $\mathcal{L}_D$ controlled by P and V respectively, holding $\frac{\alpha}{2}$ coins each. These are the funds P and V want to lock in Glimpse. We let := $_P$:coins + $_V$:coins, and we denote with $_P$ and $_V$ the inputs that $_P$ and $_V$ point to.

The parties construct $[\mathsf{Tx}_G]$:= $(2; [\ _P;\ _V]; 3; [\ _\alpha;\ _{\epsilon_1};\ _{\epsilon_2}])$, such that the output $_\alpha$:= $(\ ; (\mathsf{MuSig}(\mathsf{pk}_P; \mathsf{pk}_V)) \vee (\mathsf{OneSig}(\mathsf{pk}_V) \wedge T_3))$ holds the Glimpse value, and the outputs $_{\epsilon_1}$:= $(\ ; (\mathsf{scriptG}(\mathsf{Desc}_1; T_1; \mathcal{T}_S; n_1; P)) \vee \mathsf{MuSig}(\mathsf{pk}_P; \mathsf{pk}_V))$ and $_{\epsilon_2}$:= $(\ ; (\mathsf{scriptG}(\mathsf{Desc}_2; T_1; \mathcal{T}_S; n_2; P)) \vee \mathsf{MuSig}(\mathsf{pk}_P; \mathsf{pk}_V))$ hold the Glimpse scripts.

We note that now $\mathsf{Tx_G}$ has as many additional outputs as the number of literals in the DNF ($_{\epsilon_1}$; $_{\epsilon_2}$), each one holding a negligible amount (e.g., 1 satoshi). Conversely from the single transaction case, we now require P to also plug in randomness in the descriptions. Then, on $\mathcal{L}_D$, *for each disjunctive term in* $\mathcal{F}_S$, parties need to:

- Create a transaction $\mathsf{Tx_T}$ that allows P to prove inclusion for the transactions published by the oracles. To do so, $\mathsf{Tx_T}$ takes as inputs the $_{\epsilon_i}$ for the *non-negated literals*, and has a single output := $(\ ; \mathsf{OneSig}(\mathsf{pk}_P))$.
- Create a transaction $\mathsf{Tx_F}$ that protects V from P falsely claiming some transaction was not published. $\mathsf{Tx_F}$ takes as inputs the $_{\epsilon_i}$ outputs of the *negated literals*, and has as many outputs as the number of inputs. Outputs are of the form $_i$:= $(\ ; (\mathsf{scriptG}(\mathsf{Desc}_i; T_2; \mathcal{T}_S; n_i; (P; V))$. $\mathsf{Tx_F}$ enables V to react to P's false claim by submitting a proof within T_2 thereby proving i-th transaction inclusion and spending $_i$. In some cases $\mathsf{Tx_F}$ is not needed, as for $(\mathsf{Desc}_1 \wedge \mathsf{Desc}_2)$.
- Create a transaction $\mathsf{Tx_P}$ that takes as inputs the output of $\mathsf{Tx_T}$, the $_i$ outputs of $\mathsf{Tx_F}$, and the output $_\alpha$. If V has not spent any $\mathsf{Tx_F}$ output publishing $\mathsf{Tx_F}$, $\mathsf{Tx_P}$ allows P to spend the funds in $\mathsf{Tx_G}$.

Then, parties create a *unique* transaction $\mathsf{Tx_V}$ that spends the output $_\alpha$. At this point, P signs $\mathsf{Tx_V}$ and $[\mathsf{Tx_F}]_{i=1,2}$ and sends message $(\ _P;\ _V; \mathsf{Desc}_1; \mathsf{Desc}_2; T_1; T_2; \mathcal{T}_S; n_1; n_2;\ ; \mathsf{scriptG}; [\mathsf{Tx}_G]; ([\mathsf{Tx_T}]; [\mathsf{Tx_F}]; [\mathsf{Tx_P}])_{i=1,2}; [\mathsf{Tx_V}];\ _P([\mathsf{Tx_V}]);\ _P([\mathsf{Tx_F}])_{i=1,2})$ to V. Upon receiving the message, if V is interested in opening a Glimpse instance with P at the given parameters, after checking correctness of the parameters and well-formedness of transactions, V signs $\mathsf{Tx_G}$, $\mathsf{Tx_F}$ and $\mathsf{Tx_P}_{i=1,2}$, and sends the signatures to P.

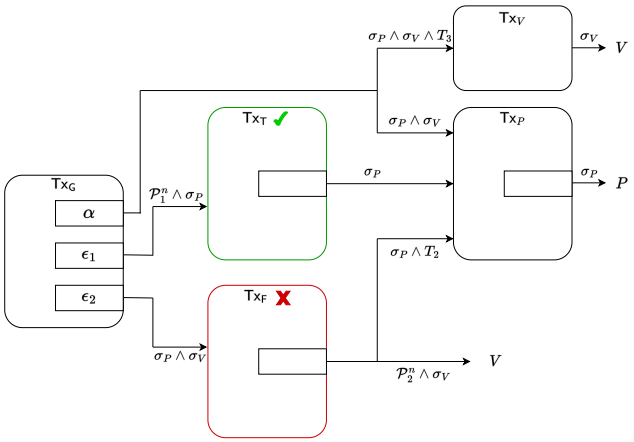

Figure C.1: Set $(\mathsf{Tx_G}; (\mathsf{Tx_T}; \mathsf{Tx_F}; \mathsf{Tx_P})_i; \mathsf{Tx_V})$ of transactions to be constructed for verifying a DNF with two literals. The arrows refer to the term $i = (\mathsf{Desc}_1 \wedge \neg\mathsf{Desc}_2)$ of $\mathcal{F}_S$.

Upon receiving $(\ _V([\mathsf{Tx_G}]);\ _V([\mathsf{Tx_F}]);\ _V([\mathsf{Tx_P}])_{i=1,2})$ from V, P checks signatures validity. If valid, P publishes $\mathsf{Tx_G}$ on $\mathcal{L}_D$, while both parties locally store the tuples $(\mathsf{Tx_T}; \mathsf{Tx_F}; \mathsf{Tx_P})_i$ and the signatures.

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

Verify & Commit on $\mathcal{L}_D$. *P and V* monitor $\mathcal{L}_S$ (or make use of a trusted relayer) checking for transactions matching descriptions Desc_1 and Desc_2 being published. If at least one of such transactions are published, P constructs the proofs and publishes the transactions $\mathsf{Tx_T}_i$ and $\mathsf{Tx_F}_i$ corresponding to the event realization. V checks whether the set of transactions published by P corresponds to the correct term of $\mathcal{F}_S$ realized by the oracles. If V detects any misbehavior from P, V can react within T_2 by constructing a proof and spending one of $\mathsf{Tx_F}$'s outputs. In this way, V spends an input of $\mathsf{Tx_P}_i$, thereby invalidating $\mathsf{Tx_P}_i$. V can consequently publish $\mathsf{Tx_V}$, redeeming the coins after T_3. If V does not spend any of $\mathsf{Tx_F}$'s outputs, P will get the funds after T_2 via $\mathsf{Tx_P}_i$. If P does not publish any set of transactions, V can similarly redeem the coins after T_3. We note that $T_1 < T_2 < T_3$. Figure C.1 shows an example of transaction set $(\mathsf{Tx_G}; (\mathsf{Tx_T}; \mathsf{Tx_F}; \mathsf{Tx_P})_i); \mathsf{Tx_V}$ for $i = (\mathsf{Desc}_1 \wedge \neg\mathsf{Desc}_2)$ of $\mathcal{F}_S$.

Remarks. We stress that parties can set a specific fund distribution for multiple different outcomes for P. Contrarily to what we have for a simple 1-transaction verification, this optimized Glimpse for DNF verification requires V to construct and submit a proof $\mathcal{P}^n$ in the pessimistic case, where P is cheating. Therefore, for Glimpse, V needs to interact with R to obtain $\mathcal{L}_S$'s data (or to run a full node).

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

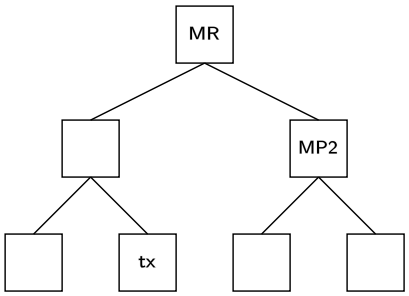

Figure D.1: Merkle tree of four transactions. The two elements MP1 and MP2 we need for reconstructing the Merkle root are marked, along with Tx.

Appendix D.
Glimpse Script example for Bitcoin-like chains

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

For Liquid, we have the following setting: Taproot is enabled (granting access to the MAST functionality), the opcodes OP_CAT as well as OP_SUBSTR are active. We point the reader to [35], [36] 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 $\mathsf{Desc} = (1; [(x_1)]; 1; [(\ )])$ having a variable input. Assume Tx being part of block B, accommodates 4 transactions in total, as in Figure D.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 2.1 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 $_G$. We recall Bitcoin-like chains process instructions sequentially: For $\mathsf{Tx_P}$ to spend $_G$, the locking script of $_G$ 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 D.2, we have 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).

```
1  < P> < V>
2  <HeaderSuffix> <HeaderPrefix>
3  <MP2> <MP1>
4  <txid> <outid>
```

Figure D.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.

Locking Script. In line 1 and 2, the script ensures <

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

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

txid> and <outid> have the expected byte-length, i.e., <outid> of 4-bytes and <txid> of 32-bytes. This step is necessary as *Glimpse needs to verify the input and output strings of the witness are not malicious*: concretely, *the strings have to be interpreted by the nodes as intended at*

the beginning, not changing the validation process, e.g., by injecting malicious 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.

D.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 [9]. 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 [37].

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.