Invited Paper: Fault-tolerant and Expressive Cross-Chain Swaps

Yingjie Xue  
yingjie_xue@brown.edu  
Brown University  
Providence, RI, United States

Di Jin  
di_jin@brown.edu  
Brown University  
Providence, RI, United States

Maurice Herlihy  
maurice.herlihy@gmail.com  
Brown University  
Providence, RI, United States

ABSTRACT
Cross-chain swaps enable exchange of different assets that reside on different blockchains. Several protocols have been proposed for atomic cross-chain swaps. However, those protocols are not fault-tolerant, in the sense that if any party deviates, no asset transfer can happen. In this paper, we propose two alternative protocols for structuring composable and robust cross-chain swaps. Participants can propose multiple swaps simultaneously and then complete a subset of those swaps according to their needs. Their needs are expressed as predicates which capture acceptable payoff of each participant. Our proposed protocols are thus more expressive due to the introduction of predicates. The proposed protocols are fault-tolerant since, even if some participants deviate, those predicates can still be satisfied, and conforming parties can complete an acceptable set of swaps.

CCS CONCEPTS
• Theory of computation → Distributed algorithms.

KEYWORDS
Cross-chain transactions, Cross-chain swaps, Fault tolerance

ACM Reference Format:
Yingjie Xue, Di Jin, and Maurice Herlihy. 2023. Invited Paper: Fault-tolerant and Expressive Cross-Chain Swaps. In 24th International Conference on Distributed Computing and Networking (ICDCN 2023), January 4–7, 2023, Kharagpur, India. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3571306.3571388

1 INTRODUCTION
Imagine a world where financial transactions span multiple independent ledgers, each managing a different token. Alice’s salary is paid in apricot tokens, managed on the Apricot ledger, but she pays her utility bill using banana tokens, managed on the Banana ledger. The apricot-to-banana exchange rate is volatile, so Alice waits until the first day of each month to buy the banana tokens she needs.

Because Alice does not trust centralized exchanges, she sets up a recurring trust-free atomic swap with some willing counterparty. Suppose Alice and Bob agree to swap some of Alice’s apricot tokens for Bob’s banana tokens using a well-known atomic swap protocol [6, 10] which require both Alice and Bob to place their tokens in escrow. Any such protocol is safe for Alice in the narrow sense that if Bob deviates from the protocol, perhaps by abandoning the swap in the middle, then Alice gets her escrowed tokens back (minus fees). Alice still pays a price because she regains access to her escrowed tokens only after a substantial delay. Alice must then attempt another swap with another counterparty, exposing her to the same risk of inconvenience and delay. Roughly speaking, if Alice eventually succeeds on her $k$th attempt, where each failed swap releases her escrowed tokens only after $t$ hours, then Alice spends about $(k − 1)t$ hours acquiring her banana tokens.

Suppose instead that Alice sets up all $k$ swaps together, and tentatively executes those swaps in parallel. Some swaps (tentatively) succeed and the rest fail. Alice chooses to commit one of the successful swaps (perhaps the one with the best exchange rate), and aborts the rest. As long as one tentative swap succeeds, Alice acquires her banana tokens with a worst-case delay of $t$, not $(k − 1)t$.

Prior cross-chain swap protocols are atomic (all-or-nothing), but not robust: any component failure typically causes the entire exchange to abort, undoing any tentative changes. Alternative paths can be explored only sequentially, not in parallel. This paper proposes novel cross-chain swap protocols that allow parties to explore multiple complex trades in parallel, and to select some satisfactory subset of those trades to be completed.

Devising robust cross-chain swap protocols presents non-obvious challenges. In the apricot-to-banana token swap example, only Alice sets up alternative parallel swaps. What if Bob, too, wants alternatives? (Perhaps he buys banana tokens from Xerxes, or else from Zoe, and then resells them to Alice.) Can Bob and Alice’s parallel swap alternatives compose in a well-defined way? What if their choices interfere?

Prior cross-chain swap protocols [6, 7, 14, 19, 21] typically ask participants to escrow their assets. If all goes well, the escrowed assets are redeemed by their new owners (the swap commits), but if something goes wrong, the escrowed assets are refunded to their original owners (the swap aborts). The “all-or-nothing” nature of these protocols prevents a party from redeeming only a subset of assets. (Some payment channels [1, 12] do support robustness though redundant payments, but these are limited to one-way payments, not support cyclic transfers as in swaps.)

In this paper, we propose two alternative protocols – ProtocolA and ProtocolB, for structuring composable and robust cross-chain swaps. These protocols make different trade-offs.

• We propose a framework where participants can set up multiple tentative swaps and commit only a subset. Participants express their requirements as predicates. The overall exchange is feasible if all parties’ predicates are simultaneously satisfiable.

• We translate the predicates to a system of locks controlled by hashes [10].
• ProtocolA has fast best-case completion time, but requires high collateral: each party must fund a separate escrow for each tentative swap, even though some swaps will abort.
• ProtocolB has slower best-case completion time, but requires lower collateral: the same escrow can be used in multiple alternative swaps.

This paper is organized as follows. Section 2 describes the model, Section 3 elaborates our motivation, Section 4 introduces preliminaries for our proposed protocols. Section 5 describes challenges and some building blocks in proposed protocols. ProtocolA and ProtocolB appear in Section 6 and Section 7 respectively. Section 8 analyzes the protocols’ security and efficiency. Section 9 summarizes related work, and Section 10 considers future directions.

2 MODEL OF COMPUTATION
Although our problem is motivated by cross-chain asset exchanges, nothing in our protocols depends on specific blockchain technologies, or even whether ledgers are implemented as blockchains or some other kind of tamper-proof data store. The fundamental problem addressed here is how to conduct fault-tolerant and safe commerce among mutually-distrusting parties whose assets reside on multiple ledgers.

2.1 Ledgers and Contracts
A blockchain is a ledger (or database) that tracks ownership of assets. A blockchain is tamper-proof, meaning that it can be trusted to process transactions correctly and store data reliably. We assume blockchains support smart contracts (or contracts). A smart contract is a program residing on the blockchain that can own and transfer assets. Contract code and state are public. A contract is deterministic since it is executed multiple times for reliability. A contract can read and write data on the same blockchain where it resides, and it can invoke functions exported by other contracts on the same blockchain, but it cannot directly access data or contracts on other blockchains. A party is a blockchain client, such as a person or an organization. A party can send transactions to be executed on the blockchain. When we say transactions, we mean transactions that happen on a single blockchain, including asset transfers, smart contract initialization, and calling smart contract functions.

2.2 Communication Model
We assume a synchronous communication model where there is a known upper bound $\Delta$ on the time needed for a transaction issued by one party to be included and confirmed in a blockchain and to become visible to others.

2.3 Fault Model
As mentioned, blockchains are assumed to be tamper-proof, and calls to smart contract functions are executed correctly. We rule out attacks on blockchain itself, for example, denial-of-service attacks. Parties can be Byzantine, departing arbitrarily from any agreed-upon protocol. We do not assume Byzantine parties are rational:

---

*The mining process to include a transaction in a block and the confirmation of the block take a non-trivial time, see more in [11]. Local computation is considered instantaneous.*

---

3 MOTIVATION
Suppose Alice owns $Xcoins$ and she wants to buy an NFT from Bob. However, Bob only accepts payment in $Ycoins$. Alice finds an intermediate party, Carol, who accepts her $Xcoins$ and in exchange pays $Ycoins$ to Bob. Since Alice does not want to hold $Ycoins$ if the trade fails, the three of them need to swap their assets atomically: Alice pays Carol $Xcoins$, Carols pays Bob $Ycoins$, and Bob sends the NFT to Alice (Shown in Figure 1a, we call this Example I).

Atomic swap protocols [6] are designed to handle this situation. They execute the tentative asset transfers with guaranteed atomicity: either all asset transfers happen, or no transfer happens.

Suppose that Carol is not that reliable and she crashes with 50% probability, but Alice and Bob want the trade to happen in a timely manner (the market might be very volatile). Alice and Bob can mitigate such problem by finding more trading partners (David) to boost their probability of success (Example II in Figure 1b). As long as one of the trading partners is responsive, the trade can succeed.

We call what a participant wants to achieve in a trade their motive, and a set of transactions that satisfies everyone’s motive a feasible swap. In Example I, Alice’s motive is to get the NFT from Bob while paying at most one $Xcoin$ to Carol (she would be perfectly happy if she could get away with not paying), and in Example II, Alice’s motive is to get the NFT from Bob, and paying at most one Xcoin to either Carol or David. Bob has similar motives in Example II. Carol and David have the same motive, either not doing any transaction or trading a Ycoin for an Xcoin. As a result, in Example I there is
This section describes an existing atomic swap protocol, which lays the foundation of our proposed approaches. We start with terminologies in atomic swaps. The notation and terminology needed to define robust cross-chain swaps are also given in this section.

### 4 Directed Graphs

An atomic cross-chain swap is represented by a directed graph. A directed graph or digraph $G = (V, A)$ is a set of vertices $V$ and a set of arcs $A \subseteq V \times V$. Each vertex represents a party, and each arc is labeled with an asset. A path from $u$ to $v$ in $G$ is a sequence of arcs $(u_0, v_1), \ldots, (u_{k-1}, u_k)$ where $u_0 = u$ and $v = v_k$, and each arc $(u_i, u_{i+1}) \in A$. A digraph is strongly connected if there is a path from any vertex to any other vertex.

An arc $(u, v)$ represents a proposed asset transfer from party $u$ to party $v$. All digraphs considered here are strongly connected, implying that all transfers are (perhaps indirect) exchanges. In real life, some exchanges may have off-chain components. For example, if Alice uses a token to pay for a sandwich, we would formally represent the off-chain food transfer as a "virtual" token transferred from the restaurant to Alice.

For brevity, when discussing a swap defined by a digraph, we use the terms vertex and party interchangeably. We use "arc $(a, b)$" as shorthand for "proposed transfer from party $a$ to party $b$", and we say an arc is triggered if that proposed transfer takes place.

### 4.2 Hashlocks, Hashkeys and Hashlock Circuits

Generally speaking, an atomic swap works in the following way: an asset is first escrowed and protected by locks, and then the asset is transferred to the recipient upon unlock. Here we describe the lock mechanism used in the atomic swap protocol described later.

Let Hash(.) be a collision-resistant hash function. A pair of value $s$ and $h$, where $s$ is a secret random value, and $h = Hash(s)$, forms a "lock and key" structure: a lock $h$ can be unlocked when the key $s$ such that $h = Hash(s)$ is shown. A hashlock is such a lock, augmented with more information in the hashkey to keep track of the propagation of the secret $s$ on the graph.

Formally, a hashlock is a hash value $h$. The structure to unlock a hashlock is called a hashkey. Given a digraph $G = (V, A)$, a hashkey on an arc $(u, v)$ is a triple $(s, p, \sigma)$, where $s$ is a randomly-chosen value called a secret, $p$ is a path from $u$ to $v$ in the graph where $u_0 = u$, $u_k$ is the party who chose $s$. $\sigma$ is a path signature, which each party in the path $p$ provides a signature. Recall that sig($m, u$) denotes the signature of a party $u$ signing a message $m$ using his/her secret key $\sigma$. A signature composed recursively by each party $u_j$ in the path signs the signature of $u_{j+1}$ as a message using their secret key. More formally,

$$\sigma = sig(sig(...sig(sig(s, u_k), u_{k-1}), \ldots, u_1), u_0)$$

A hashkey $(s, p, \sigma)$ has its lifespan. The exact definition of the lifespan and more details can be found in the original document [6]. In short, the hashkey $(s, p, \sigma)$’s lifespan is linear to the length of the path: it times out at $|p| \Delta$, where $\Delta$ is the upper bound of message delay defined in Section 2.2.2.

We say a hashkey matches a hashlock if it can unlock the hashlock. On any arc $(u, v)$, its hashlock $h$ can be unlocked by a hashkey $(s, p, \sigma)$ if and only if all of the following conditions hold. (1) $h = Hash(s)$, (2) $p$ is a path from $v$ to the party who chose $s$. (3) $\sigma$ is a valid path signature for $s$ and $p$ constructed as Eq.1. (4) The hashkey does not time out. Roughly speaking, the hashkey structure guarantees that, if a hashlock $h$ on an outgoing arc from $u$ is unlocked, then $h$ on all arcs entering $u$ can be unlocked. A hashlock times out if all possible matching hashkeys time out.

A hashlock circuit $C(u, v)$ for an arc $(u, v)$ is a formula linking hashlocks on that arc via operators $\lor$, $\land$, and $\neg$.

### 4.3 An Atomic Cross-Chain Swap Protocol

Here we describe a protocol [6] that is atomic, but not robust. It does not support alternative swaps, but it does provide a starting point for developing robust protocols.

The swap is represented as a strongly-connected directed graph $G = (V, A)$, where each arc represents a proposed asset transfer. Each transfer along an arc is controlled by a contract. We say an asset is escrowed on an arc when the owner forfeits the asset’s control to the contract on the arc.

The protocol starts with a feedback vertex set (FVS), a set of vertices whose removal leaves the graph without cycles. The vertices in the FVS are called leaders, the rest followers. At the beginning, each leader $l_i$ chooses a random secret $s_i$ and constructs the hashlock $h_i = Hash(s_i)$. After the hashlocks are distributed among the leaders, the protocol can start running on the blockchains. Each contract for an arc is associated with a hashlock circuit $C = \bigwedge_{i \in FVS} h_i$ to protect the escrowed asset. When a hashlock is unlocked, it evaluates to true and when the hashlock circuit evaluates to true, the asset is redeemed by the proposed recipient. We call an arc is triggered when the asset is redeemed.

Overview of the protocol is shown in Figure 2. The protocol consists of two phases: Escrow phase and Redeem phase. Note that transactions sent to blockchains can be included on blockchains and observed by others within $\Lambda$.

In the Escrow phase, each leader escrows their assets on all outgoing arcs (Leader Step a). Then the leaders start waiting for their incoming arcs before they enter redeem phase (Leader Step b). Followers first wait until all the incoming arcs are escrowed (Follower Step a), then they escrow their assets on the outgoing arcs (Follower Step b).

The total amount of assets by a party escrowed is called the party’s collateral. During the Escrow phase if any expected escrow does not arrive for an extended period, the party should just abort the protocol.

\[\text{More precisely, it times out at } (\text{MaxPathLength}(G) + |p| \Delta) \text{ after the protocol starts. Here we focus on brevity other than precision.} \]
In the Redeem phase, hashkeys are sent to contracts that manage their assets to unlock hashlocks. For a leader $l_i$, if it has observed that all its incoming arcs are escrowed, it constructs a hashkey $(s_i, l_i, \sigma_i, s_i(l_i))$ and sends it to corresponding contracts representing all its incoming arcs (Leader Step c). The hashlock $h_i$ associated with the contracts can be unlocked upon receiving $(s_i, l_i, \sigma_i, s_i(l_i))$. Then both leaders and followers can propagate the secrets (Leader Step d and Follower Step e) using the hashkey structure. For a party $u$, when a hashlock on its outgoing arcs is unlocked by a hashkey $(s, p, \sigma)$, it can construct and send the hashkey $(s, u||p, \sigma(u))$ to all contracts managing its incoming arcs to unlock the same hashlock.

The hashkey mechanism guarantees that if any party observes a matching hashkey sent to its outgoing arc, there is one more $\Delta$ allowing it to send a new matching hashkey to all of its incoming arcs. If all secrets are propagated correctly, all hashlocks are unlocked, and all the assets will be redeemed.

There is an additional time-out structure that ensures that assets cannot be escrowed forever, which is described in the original document [6]. In short, if a hashlock cannot be unlocked anymore, the asset will be refunded.

In summary, the atomic swap protocol satisfies liveness: if all parties conform, all asset transfers happen. It also satisfies safety: a conforming party’s assets are all refunded if $u$ does not receive all its incoming assets. We call atomic swaps as all-or-nothing swaps because of those properties.

### Figure 2: The Atomic Swap Protocol

#### Leader $l_i$
- a) Escrow assets on all its outgoing arcs.
- b) Wait until all incoming assets are escrowed.
- c) If $b$ finishes, send the hashkey $(s, l_i, \sigma_i, s_i(l_i))$ to all its incoming arcs.
- d) If a hashkey $(s_i, l_i, \sigma_i, s_i(l_i))$ is sent to its outgoing arcs, send a new hashkey $(s, l_i|l_i||p, \sigma_i(l_i))$ to all of its incoming arcs.

#### Follower $f$
- a) Wait until all incoming assets are escrowed.
- b) If $a$ finishes, escrow assets on all its outgoing arcs.
- c) If a hashkey $(s, p, \sigma)$ is sent to its outgoing arcs, send a new hashkey $(s, f||p, \sigma(f||p))$ to all of its incoming arcs.

#### Redeem Phase
- a) Escrow assets on all its outgoing arcs.
- b) Wait until all incoming assets are escrowed.
- c) If $a$ finishes, escrow assets on all its outgoing arcs.
- d) If a hashkey $(s, p, \sigma)$ is sent to its outgoing arcs, send a new hashkey $(s, f||p, \sigma(f||p))$ to all of its incoming arcs.

### 4.4 Predicates

The directed graph formalism suffices to describe all-or-nothing swaps, where success means all proposed transfers take place. We now define a way to describe swaps where parties also accept outcomes where only a subset of the proposed swaps take place.

We first define a boolean variable $v$ on each arc: $v$ is true if the transfer it represents takes place, and false otherwise. An acceptable outcome for a party $x$ can be represented with a boolean predicate of its incoming and outgoing arcs. Suppose party $x$ has incoming arcs $i_1$, $i_2$, outgoing arcs $o_1$, $o_2$, and $x$ wants to exchange $o_1$ with $i_1$, or exchange $o_2$ with $i_2$. Naively one would construct the predicate as $(o_1 \wedge i_1) \lor (o_2 \wedge i_2)$. Looking at it closely, this predicate definition has two problems. Firstly, it does not capture safety. When $o_1 \wedge i_1$ is true, this predicate will be true even if $o_2$ is true, but this would mean that $x$ is paying both $o_1$ and $o_2$, i.e., $x$ overpays. Secondly, it does not allow $x$ to accept greedy outcome. If $i_1$ is true, and all other arcs are false, it is perfectly acceptable to $x$, but our predicate evaluates false to this situation.

To capture their expected exchanges (liveness), safety requirements, and allow greediness, each party has liveness and safety requirements, characterizing acceptable outcomes in the (partial) success and failure cases respectively. First, we consider safety requirements. For each possible outgoing asset, there is an income predicate associated with it. That captures the payoff that, if this party pays this asset, what should they get at least to be safe. Note that, a participant may want some exchanges to happen atomically, say does not pay an asset unless getting a bundle of other assets even though get a subset of them already gives him/her a better payoff. This is also characterized as a safety requirement. The safety requirement for party $x$ is given by a predicate $S_x$.

In Example I, we use the arc $"(a, c)"$ as shorthand for the "asset labeling arc" $(a, c)$ is transferred from party $a$ to party $c". Alice’s safety predicate is

$$S_a := (a, c) \implies (b, a).$$

meaning that if Alice transfers her assets to Carol $"(a, c)"$ then Bob transfers his assets to her $"(b, a)"$. Importantly, Alice can be greedy. Her predicate is satisfied if she gets something for nothing: that is, if Bob pays her but she somehow does not pay Carol. This predicate is also satisfied if no payments are made, an outcome that Alice may not prefer, but considers acceptable because it leaves her no worse off. The no payment scenario incentivizes Alice to try alternatives.

Formally, for every asset $y$ that a party $x$ may pay, there is an income predicate $I^T_x$. In addition, there is a predicate $O_y$ over those outgoing payments to make sure $x$ does not overpay. For example, the predicate may require that at least two of three outgoing assets are paid.

In Example II, Alice’s predicate is

$$S_a := ((a, c) \implies (b, a)) \wedge ((a, d) \implies (b, a)) \wedge \neg ((a, c) \wedge (a, d)).$$

The first two clauses say that if Alice pays either Carol or David, then she gets Bob’s NFT in exchange, and the third clause says that Alice does not want to pay both.

In general, if party $x$ has potential exchanges with parties $u_1, \ldots, u_k$, then $x$’s safety predicate has the form:

$$S_x := \bigwedge_{i=1}^k ((x, u_i) \implies I^T_{x,u_i}) \wedge O_x,$$

where each implication clause states that if $x$ pays $u_i$ it gets the agreed-upon amount in return, and the final clause limits how many of the outgoing payments’ clauses $(x, u_i)$ can be true. For example $O_x$ might say “no more than one of the $k$ transfers can occur”, or “no more than $m_1$”, or “at least $m_2$”, and so on.

Next, we consider liveness requirements. The liveness requirement for party $x$ is given by a predicate $L_x$. First, the liveness predicate contains the safety predicate since this is a property that
should always hold. In addition to safety, liveness characterizes that something good should happen. That means, one of the income predicates $x$ specified previously in safety requirements should be true. That is,

$$L_x := S_x \land \bigwedge_{i=1}^{k} f_i(x,a_i)$$

The predicate for each party $x$ is denoted as $P_x$, which consists of two predicates $L_x$ and $S_x$. Unless otherwise specified, $P_x$ means $S_x$, since $S_x$ should always be true, while, reasonably, $L_x$ is true only if a party completes an asset transfer. Thus, $L_x$ is implied implicitly in $P_x$, since if a party transfers an asset to someone, one of its income predicates in $S_x$ must be true.

4.5 Example

Suppose Alice would like to exchange 1 Xcoin for 1 Ycoin. She sets up alternative trades with Bob and Carol, but she is willing to trade with only one of them. Bob expects to exchange 1 Ycoin for 1 Xcoin, with either Alice or Carol, but not both. Carol is willing to trade with Alice, or Bob, or both.

5 ROADMAP AND BUILDING BLOCKS

This section describes a roadmap to devise a robust cross-chain swap protocol, the challenges, and the building blocks from which we construct our approaches.

5.1 Roadmap

A robust cross-chain swap can be described by a digraph $G = (V, A)$ accompanied with a set of predicates from all participants. Each arc in $G$ is a proposed asset transfer. An arc is set to true if the asset transfer happens. A predicate is satisfied if it evaluates to true. Is there a trade that satisfies every party’s predicate? Whether such a trade exists is the satisfiability problem. If not, then the proposed trade is infeasible.

We should first find solutions that satisfy all parties’ predicates. A solution $s$ is an assignment that satisfies all parties’ predicates,

$$\{ (a_1 \mapsto true, a_2 \mapsto false, b_1 \mapsto true, b_2 \mapsto false, c_1 \mapsto false, c_2 \mapsto false) \}$$

After finding solutions, we can map them to feasible swaps. In each solution $s$, we look at the set of arcs that $s$ assigns to true, which forms a digraph denoted as $G_s$. The set of transfers in $G_s$, if executed atomically, will satisfy all parties. A digraph $G_s$ is called a feasible swap, or an alternative.

If there is more than one such feasible swap, it is tempting to execute all the alternatives in parallel, because some alternatives might fail. The following describes challenges that arise if we try to execute the alternatives in parallel.

5.2 Challenges

1. Alternatives may conflict: in one solution, Alice pays Bob and not Carol, but in another, she pays Carol and not Bob. Completing both swaps would cause Alice to overpay.
2. Alternatives may charge twice for the same transfers: if there are two solutions where Alice pays Bob, then Alice has to escrow the same amount of assets twice. Alice’s collateral would exceed the value of the assets she trades away.
3. Trades are not independent when one alternative’s digraph is a subgraph of the other. For example, Figure 4 shows two solutions. In $s_1$, Alice and Carol trade only with one another, setting only $a_2$, $c_1$ to true. Suppose Alice is the only leader. In $s_2$, Alice trades with Carol, and Carol with Bob, setting $a_2$, $c_1$, $b_2$ to true. Suppose Alice and Bob are leaders. To complete both alternatives, Alice would have to create and release secrets for both $s_1$, $s_2$. She might not have an incentive to participate in $s_2$, since that alternative provides her no additional robustness.

5.3 Building Blocks in Proposed Protocols

To tackle the challenges, we first describe some concepts and building blocks for our proposed protocols.

5.3.1 Mapping Arc Assignments to Swap Digraphs. A robust cross-chain swap is described by a predicated directed graph defined as $(P, G)$, where $G$ is the digraph of all proposed transfers, and $P$ is the trade where no assets are transferred is acceptable to all parties, but we exclude that solution as trivial.
the set of all parties’ requirements. \( \phi(P) \) denotes the conjunction of predicates \( p \in P \):

\[
\phi(P) = \bigwedge_{p \in P} p.
\]

Given a digraph \( G = (V, A) \), an assignment is a map \( \alpha : A \to \{\text{true}, \text{false}\} \) that assigns a Boolean to each arc. For any assignment \( \alpha \), \( \phi(P)(\alpha) \) denotes the value of \( \phi(P) \) under assignment \( \alpha \). A solution \( s \) is an assignment where \( \phi(P)(s) = \text{true} \). A swap digraph \( G_s \) contains the set of arcs that \( s \) assigns to true, denoted as \( G_s = \{(a, b) \in \mathcal{A} | (a, b) \to \text{true} \in s \} \).

Some \( G_s \) is a subgraph of a larger graph \( G_{s'} \). We define this relation as inclusion \( \subseteq \).

**Definition 1. Inclusion \( \subseteq \).** Consider solutions \( s, s' \) where \( \phi(P)(s) = \phi(P)(s') = \text{true} \). \( G_s = (V_s, A_s) \) and \( G_{s'} = (V_{s'}, A_{s'}) \). If \( V_s \subseteq V_{s'} \) and \( A_s \subseteq A_{s'} \), then we say \( s \subseteq s' \).

5.3.2 **Redundancy Providers.** Some parties may prepare redundant trades for fault tolerance even though they do not intend to complete all of those trades.

**Definition 2.** Let \( (P, G) \) be a predicated directed graph, and \( x \) a vertex in \( G \). If there are solutions \( s_1, s_2 \), where \( x \in V_{s_1} \) and \( x \in V_{s_2} \), such that \( \phi(P)(s_1) = \phi(P)(s_2) = \text{true} \), but setting all the arcs in \( G_{s_1} \) and \( G_{s_2} \) to true makes \( P_x = \text{false} \), then \( s_1 \) and \( s_2 \) are called conflicting solutions, and \( x \) is called a redundancy provider.

5.3.3 **Swap Schemes.** Next we describe a swap scheme derived from the atomic swap protocol [6] with minor changes, which lays the foundation for the robust protocols defined later.

A swap scheme is a tuple \((G, \mathcal{H})\), where \( G \) is a digraph and \( \mathcal{H} \) the set of hashlocks on each arc. The circuit \( C = \wedge_{h \in \mathcal{H}} h \) denotes that all hashlocks in \( \mathcal{H} \) need be unlocked to trigger any arc. Each arc has the same hashlock set \( \mathcal{H} \). In a swap scheme, a swap \((G, \mathcal{H})\) is executed in two sequential phases: the Escrow Phase (denoted as \( \text{Swap.Escrow}(G, \mathcal{H}) \)), and the Redeem Phase (denoted as \( \text{Swap.Redeem}(G, \mathcal{H}) \)).

Compared to the original swap protocol [6] described in Section 4, the only difference in our swap scheme is that, the set \( \mathcal{H} \) in the original scheme is the set of hashlocks generated by leaders, while in ours, \( \mathcal{H} \) also contains a set of hashlocks generated by redundancy providers. As a result, in the Redeem Phase, redundancy providers also need to send their hashkeys like leaders do.

The interface we defined in the swap scheme works as follows.

- The same as the original swap protocol.
- \( \text{Swap.Redeem}(G, \mathcal{H}) \)
  - If \( x \) is a leader or a redundancy provider, send hashkeys as a leader in the original swap protocol.
  - Once any party \( x \) receives a hashkey on their outgoing arc, it sends a new hashkey to their incoming arcs as in the original swap protocol.

We intend to run different swap schemes in parallel. Here we define a maximal set of compatible schemes that can be completed.

**Definition 3.** A maximal set of schemes. A set of schemes is **maximal**, if all schemes in the set are not conflicting, and there is no new swap schemes can be added to the set without having a conflict with some of existing schemes.

5.4 **Two Protocols and a Trade-off**

We propose two novel protocols for composable and robust cross-chain swaps. These protocols make different trade-offs: one optimizes for time at the cost of higher collateral, and the other makes the opposite choice.

**ProtocolA** prioritizes time over collateral. In the best case, when all parties conform to the protocol and avoid delays, the trade can be completed in a short time. In the worst case, when all trades fail, the parties’ assets are refunded quickly. The catch is that parties must provide higher collateral: parties must create separate escrows for each alternative, temporarily tying up more assets than are eventually traded.

**ProtocolB** prioritizes collateral over time. A party can use a single escrow for multiple alternatives. The catch is that the trade takes longer to settle: transfers require a hard timeout even when all parties conform to the protocol and avoid unnecessary delay. **ProtocolB** also provides more fault-tolerance than **ProtocolA**. Detailed comparison can be found in our full version paper [18].

6 PROTOCOLA: HIGHER COLLATERAL

6.1 **Overview**

This section describes **ProtocolA**, a protocol that favors time over collateral. Given a predicated digraph \((P, G)\), we first find a set of solutions \( s_1, \ldots, s_k \) by identifying sets of arcs such that executing trades on those sets satisfies \( \phi(P) \). Such solutions could be generated by an all-SAT solvers [16, 17, 20], though it may not be necessary to identify all solutions. The parties run these solutions in parallel. The follow summarizes **ProtocolA**.

- To control conflicting trades, each redundancy provider uses distinct hashlocks in distinct solutions. In the Redeem Phase, that party can choose which hashlocks to unlock.
- A party can add additional hashlocks after an asset is escrowed, allowing overlapping arcs in different solutions, indexed by their hashlocks.
- If \( G_{s_1} \sqsubseteq G_{s_2} \), then the hashlocks on \( G_{s_1} \) are reused in \( G_{s_2} \).

6.2 **Detailed Construction**

First, the parties set up a mutually-agreeable trade, and express their requirements in the form of predicates, yielding a predicated digraph \((P, G)\). We assume the predicates are reasonable and if not, a party can be rejected to join in the trade. A preliminary **Market**
Clearing Phase decides what swaps are feasible, and what hashlocks to use in each swap.

6.2.1 Market Clearing Phase.

Find solutions. Given \((P, G)\), the first step is to find a set of solutions acceptable to all parties, perhaps by applying an all-SAT solver to \(\phi(P)\), yielding assignments \(a\) for which \(\phi(P)(a)\) evaluates to true. If we do not need all solutions, we can stop after finding enough assignments. Suppose we have found \(k\) solutions \(S = \{s_i|\phi(P)(s_i) = true, i \in [1, k]\}\). We rule out solutions that are not strongly connected, since if the graph is not strongly connected, some rational parties have incentive to deviate [6].

Sort solutions. Each digraph \(G_i\) corresponds to a swap, and we construct schemes to execute the swap as atomic swap does. Since we plan to reuse hashlocks if some solutions \(s, s'\) satisfies \(s \subseteq s'\), solutions are sorted by inclusion. This can be done trivially by comparing each pair of solutions \((s_i, s_j)\). We use a directed graph \(T\) to depict their relation (Figure 5), where an arc from \(s_i\) to \(s_j\) means \(s_i \subseteq s_j\). \(T\) is a directed acyclic graph (DAG) since \(\subseteq\) is not reflexive. In Figure 5, each node is a solution and each arc is a \(\subseteq\) relation. For example, \(s_1 \subseteq s_4 \subseteq s_5\). If one solution, say \(s_8\), is reachable from another solution, say \(s_1\), then \(s_1 \subseteq s_8\). (There is no need for a direct arc \((s_1, s_8)\) since inclusion can be inferred). Note that if there is an arc \((s, s')\) in \(T\), then no \(s''\) exist such that \(s \subseteq s'' \subseteq s'\). If one solution is not reachable from another, then those solutions are incomparable.

We call solutions that are not reachable by any other solutions root solutions. The solutions that they can reach directly are called their children. Solutions that do not reach other solutions are called leaves. A path to a leaf node \(v\) is denoted as \(q = [v_0, \ldots, v]\), where \(v_0\) is a root node and \(v\) is reachable from \(v_0\). In the graph \(T\), all paths from all roots to reachable leaf nodes in the tree are denoted as \(Q(T(S))\), where \(T(S)\) is a DAG of solutions in the set \(S\).

Assign hashlocks. After sorting solutions in \(S\), we are ready to assign hashlocks to swap digraph \(G_i = (V_i, A_i)\) where \(s \in S\).

We first assign hashlocks to the root solutions. For any root solution \(s\), if the corresponding \(G_i\) is cyclic, like in atomic swap, then we choose a feedback vertex set (FVS). The vertices in FVS are called leaders \(L\). Although finding a minimum feedback vertex is NP-complete, there exists an efficient 2-approximation [2]. Recall that two exchanges are conflicting if there exists a party \(x\) such that \(P_x = false\) if both exchanges are completed. We identify the set of redundancy providers \(R_P\) by checking whether a party \(x\) is involved in two conflicting solutions such that \(P_x = false\) if two conflicting exchanges are both completed. The set of hashlock generators is \(HG = R_P \cup L\). Each party \(x\) in \(HG\) generates a hash \(h_x^s = Hash(\theta_x^s)\) meaning the hashlock is used for solution \(s\) generated by a party \(x\), and the secret is \(\theta_x^s\). For all arcs in solution \(s\), the set of hashlocks \(H_s\) is \(H_s = \{h_x^s \forall x \in HG\}\) and the corresponding circuit is \(C_s = \lor h \forall h \in H_s\).

After assigning hashlocks for root solutions, we move to their children. Their children will reuse the hashlocks from them. Note that a root can have multiple children, and a child can have multiple parents. For this reason, a solution’s hashlock may be used by multiple children, and a solution may reuse hashlocks from multiple parents. From each root node in the graph \(T\), we search all paths from the root to its children in the tree until the leaf. A path is denoted as \(q = [s_0, s_1, s_2, \ldots, s_k]\), where each \(s_i\) denotes a node and \(s_0\) is a root node. For ease of exposition, we used \((G_i, H_{s,q})\) to mean that all arcs in \(G_i\) are assigned hashlock set \(H_{s,q}\) for solution \(s\) in the path \(q\). For root solutions, \(H_{s,q}\) is the same for all \(q\) since \(q = s\).

For non-root node \(s\), \(H_{s,q}\) is different for different \(q\).

Starting with a root solution \(s_0\), we assign hashlocks for solutions reachable from \(s_0\). Here we show how to assign hashlocks for solutions in a path \(q = \{s_0, s_1, s_2, \ldots, s_k\}\) starting from \(s_0\). For \(i \in [1, k]\), assume \(s_{i-1}\) is associated with hashlock set \(H_{s_{i-1},q}\). Then, for \(s_i\), the hashlocks are set using the following steps.

- Compute \(G_{s_i|s_{i-1}} = A_{s_i} \setminus A_{s_{i-1}}\) and find hashlock generators for \(G_{s_i|s_{i-1}}\). We also use \(s_i \setminus s_{i-1}\) to denote \(G_{s_i|s_{i-1}}\) when there is no confusion.
- If \(G_{s_i|s_{i-1}}\) is acyclic, then no leader is introduced.
- If \(G_{s_i|s_{i-1}}\) is cyclic, then new leaders in \(G_{s_i|s_{i-1}}\) are chosen. Denote by \(L_{s_i|s_{i-1}}\) the set of leaders in \(G_{s_i|s_{i-1}}\).
- If redundancy providers are introduced in \(G_{s_i|s_{i-1}}\), then redundancy providers are added. Let \(R_P^{s_i|s_{i-1}}\) denote the redundancy providers in \(G_{s_i|s_{i-1}}\).
- New hashlock generators \(H_{G_{s_i|s_{i-1}}} = L_{s_i|s_{i-1}} \cup R_P^{s_i|s_{i-1}}\)
- Each party in \(H_{G_{s_i|s_{i-1}}}\) generates a new hashlock. The set of hashlocks generated is denoted as \(H_{s_i|s_{i-1}}\).
- The hashlocks for \(s_i\) on the path \(q \in Q(T(S))\) is \(H_{s_i,q} = H_{s_{i-1},q} \cup H_{s_i|s_{i-1}}\).

6.2.2 Execute the protocol on chain. After assigning hashlocks for solutions, each solution \(s\) can be described as a set of swap schemes \(Swap(G_i, H_{s,q}), q \in Q(T(S))\). Those swap schemes can be executed in parallel on the chain. Denote the solutions output by all-SAT solvers as \(S = \{s_1, \ldots, s_k\}\).

For each party \(x\), we first find all solutions involving it, i.e. \(x \in V_s\). Then for each such solution, it executes a separate swap scheme \(Swap(G_i, H_{s,q})\) for all \(q \in Q(T(S))\), in three phases called escrow, select and redeem.

Escrow Phase. Each party in \(Swap(G_i, H_{s,q})\) runs \(Swap.Escrow(G_i, H_{s,q})\) in parallel. If an asset is already escrowed, it is not escrowed again. Instead, the asset’s circuit is updated with an OR gate: \(C_{s_i,q} \lor C_{current}\). Suppose \(C_{current}\) means the current hashlock circuit on an arc. \(C_{s_i,q}\) is the hashlock circuit for \(Swap(G_i, H_{s,q})\), which is the conjunction of all hashes in \(H_{s_i,q}\). Symmetrically, parties do not require incoming assets to be escrowed twice, only that the hashlocks on those incoming assets are updated to \(C_{s_i,q} \lor C_{current}\). A party can update the hash circuit on its outgoing arc using OR gate since it adds more possibilities to be redeemed and does not affect the ability to redeem using the current hash circuits.

Select Phase. After the escrow phase is finished, the parties select which swap scheme should proceed, since some swap schemes are conflicting so that not all of them can be completed. First, redundancy providers run an agreement procedure to decide which set of swap schemes they would like to complete. The agreement procedure is not the main focus of the protocol, here we give an algorithm to reach an agreement without considering its efficiency.

A redundancy provider is randomly chosen as a proposer. That party proposes a maximal set of compatible schemes \(S_{prop}\) (which can
be generated by a greedy algorithm) from all swap schemes where no party deviates in the escrow phase. The rest of the redundancy providers are voters, who vote whether to complete those schemes. If the proposer is conforming, $S_\ell$ should be acceptable for all of them, and conforming voters should all vote yes. If some of voters do not vote yes, then this scheme is removed from $S_\ell$. Another round lets the proposer add schemes to $S_\ell$ to make sure $S_\ell$ is maximal. The role of proposer can be replaced by a program shared among all parties, which observes all escrows on the chain and deterministically outputs a maximal set of schemes $S_\ell$. Redundancy providers then vote on each scheme in this set $S_\ell$. The search for $S_\ell$ ends when all redundancy providers vote yes on each proposed swap scheme, or there are no new swap scheme to be added to $S_\ell$. For fast settlement, this protocol can run on the side. For example, once the escrow phase in a swap scheme $s_1$ is completed, the protocol can start to decide whether to complete the swap.

Redeem Phase. After $S_\ell$ is chosen by the redundancy providers, the parties proceed as follows. We use a tuple $(s, q)$ to denote the swap scheme $\text{Swap}(G_s, H_{s,q})$. A leader or redundancy provider $x$ needs extra consideration before they proceed to the redeem phase for a scheme $(s, q) \in S_\ell$ because of the reuse of hashlocks. They proceed in $\text{Swap}(G_s, H_{s,q})$ only if the hashlock $h$ that $x$ generated for $(s, q)$ satisfies: for all $\text{Swap}(G, H)$ where $h \in H, x$ receives all incoming escrows in Swap$(G, H)$. Otherwise, it does not proceed. This requirement guarantees that, if a leader/redundancy provider releases a hashkey for a hashlock $h$, and any outgoing arc is triggered in any scheme $\text{Swap}(G_s, H)$ who uses this $h$, then they can get assets from incoming arcs in this scheme, leaving them no worse off. See more analysis in Section 8. If $x$ is not a leader or redundancy provider, no extra consideration is required before they proceed. For each selected swap scheme $\text{Swap}(G_s, H_{s,q})$, those who proceed run $\text{Swap.Redeeem}(G_s, H_{s,q})$.

An asset escrowed can be redeemed by a counterparty if its hashlock circuit evaluates to true, where we assign true to hashlocks that have been unlocked, and false to the rest. Recall that the circuit is composed of the disjunction of hashlock circuits of each independent swap scheme. That means, if the circuit of any independent swap scheme evaluates to true, a swap happens.

7 PROTOCOLB: LOWER COLLATERAL

Suppose Alice wants to exchange one apricot token for one banana token. Using ProtocolA, Alice sets up tentative trades with Bob, Carol, and David. She must escrow three apricot tokens, one for each possible trade. In the end, she will commit one trade, spending one token, and cancel the rest, reclaiming the other two tokens. Nevertheless, she must have three tokens at hand to provide collateral for the alternative trades.

In this section, we describe ProtocolB, a protocol that allows Alice to provide collateral for all three alternatives with the same token. The catch is that ProtocolB requires a hard timeout to complete the trade. In both the best and worst cases, ProtocolB takes time $4\Delta$ 4, while ProtocolA requires less time since participants can complete an alternative immediately after its escrow phase is completed. Note that even with a hard timeout, ProtocolB requires less time than attempting the alternative trades sequentially.

7.1 Overview

Here is a high-level sketch of ProtocolB. We focus on the difference between ProtocolA and ProtocolB in the description.

Predicates reflect the reuse of assets. To start, parties express their exchange requirements just as ProtocolA. The difference between ProtocolA and ProtocolB is that, in ProtocolA, each arc represents a unique asset, while in ProtocolB, some arcs can represent the same asset, e.g. one token is reused on multiple arcs. To cater for that change, each participant provides an additional predicate: for different arcs that represent the same assets, at most one of them is assigned true. Given the participants’ new predicates, we find assignments to satisfy all the predicates.

Solutions are sorted by preferences. Suppose there are $k$ solutions. We assign hashlocks as in ProtocolA (the definition of redundancy providers change a bit, explained later). For an asset that has multiple different recipients, solutions are sorted according to participants’ preferences to indicate who has priority to get the asset. For example, if Alice’s escrowed asset $a_1$ is transferred to Bob in swap1 but transferred to Carol in swap2, then Alice, Bob, and Carol rank swap1 and swap2 by preference.

Circuits use negations to indicate preferences. Suppose swap1 is preferred than swap2, and the circuit for those swaps are $C_1$ and $C_2$, respectively. Each circuit also indicates the recipient when they evaluate to true. To implement this priority, the circuit on the escrowed asset $a_1$ would be $(\neg C_1 \land C_2) \lor C_1$, indicating that if $C_1$ evaluates to true, swap1 will be completed and swap2 will only be completed if $C_1$ evaluates to false.

7.2 Detailed Construction

7.2.1 Market Clearing Phase. First, participants express their exchange requirements as before. Taking predicates $P_x$ from a party $x$, there will be an addition restriction $r_x$ due to the fact that multiple arcs represent the same asset. $r_x$ is defined as: for arcs that represent the same asset, at most one of those arcs can be true. This can be expressed in the same way as predicates we define in Section 4.4. Then, the new predicate for $x$ is

$$p_{new}^x = P_x \land r_x.$$  

For convenience, $p_{new}^x$ is called $P_x$ from now on. The set of new predicates are called $P$.

We find assignments that satisfy $\phi(P)$. The solutions are denoted by $S = \{s_1, \ldots, s_i, \ldots, s_k\}$. We sort the solutions by inclusion, and organize them into a DAG, and find hashlocks for each solution as ProtocolA, except the redundancy provider is defined differently.

**Definition 4.** Suppose $s_1$ and $s_2$ are two solutions for a predicated graph $(P, G)$. A party $x$ is a redundancy provider in ProtocolB if, it is a redundancy provider defined in Def. 2 and completing both $s_1$ and $s_2$ does not conflict with $r_x$.

The reason why the definition is updated is that, if $s_1$ and $s_2$ shares the same asset with different recipients (i.e. completing both $s_1$ and $s_2$ with $r_x$), it is impossible to complete both. In
other words, \( x \) does not provide redundant collateral in \( s_1 \) and \( s_2 \). It is not a redundancy provider in this case.

In addition to sorting solutions into a DAG by inclusion, we also sort them by participants’ preferences. Assume there is a protocol which allows the participants to agree on a ranking: a total order on the solutions. Let \( S' := \{s^*_1, s^*_2, \ldots, s^*_k\} \) be the set of sorted solutions, where \( s^*_i \) precedes \( s^*_j \) if \( i < j \). In other words, \( s^*_j \) is preferred over \( s^*_i \). To distinguish, we call the circuit \( C_{s^*_i}' \) in the original swap scheme as old circuit, the one in ProtocolB as new circuit. For each swap scheme \((s^*_j, q)\), the new hashlock circuit is

\[
C^\text{new}_{s^*_j q} := \bigwedge_{i < j, q'q \in Q(T(S))} s^*_j \text{ conflicts with } s^*_i \neg C_{s^*_i q'} \land C_{s^*_j q'}
\]

The new circuit implements the following logic: a swap \( s^*_j \) is completed if and only if the hashlocks in \( s^*_i \) are unlocked, and there is no preceding conflicting swap \( s^*_i (i < j) \) such that hashlocks of \( s^*_j \) that can be unlocked.

### 7.2.2 Running the protocol on chain

This phase is similar to the previous protocol, but it only include two phases: Escrow Phase and Redeem Phase.

**Escrow Phase.** Each participant runs \( \text{Swap}(G_i, \mathcal{H}_{i, q}). \text{Escrow} \). Note that the circuit corresponding to \( \mathcal{H}_{i, q} \) is \( C_{s^*_i q} \) now as defined above. If an asset \( a \) is already escrowed, \( C(a) := C_{\text{current}(a)} \lor C_{s^*_i q} \). Suppose \( C_{\text{current}(a)} \) means the current hashlock circuit on an arc \( a \). If an asset participates in multiple swaps with different recipients, the \( \mathcal{H}_{i, q} \) also specifies the recipient in this swap.

**Redeem Phase.** We say a hashlock set \( \mathcal{H}_{i} \) is unlocked when all hashlocks in \( \mathcal{H}_{i} \) are unlocked, and a hashlock set \( \mathcal{H}_{i} \) times out if any hashlock in \( \mathcal{H}_{i} \) times out. We cannot let parties simply run \( \text{Swap}(G_i, \mathcal{H}_{i, q}). \text{Redeem} \) since it is not safe. For example, if a party’s one outgoing arc has \( \mathcal{H}_{i} \) unlocked, and all \( \mathcal{H}_{j} \) where \( j < i \) times out, then this outgoing arc will be triggered as in \( \text{Swap}(G_i, \mathcal{H}_{i, q}) \). However, if another outgoing arc has \( \mathcal{H}_{i} \) unlocked where \( j < i \), then this party’s incoming arcs will have \( \mathcal{H}_{i} \) unlocked, then all incoming arcs can be triggered in swap scheme \( \text{Swap}(G_i, \mathcal{H}_{i, q}) \). Completing payments in two different schemes may produce a worse payoff. To overcome this problem, we use to a broadcast scheme to synchronize the state of hashlocks. Assume there is a broadcast scheme where there is an upper bound \( t_0 \) to synchronize all hashlocks such that, if a hashlock \( h \) is unlocked on any arc, then all arcs’ hashlocks \( h \) can be unlocked. The redeem phase takes \( \text{MaxPathLength}(G) \Delta + t_0 \), where \( \text{MaxPathLength}(G) \) is the length of the longest path in the graph. We provide a broadcast scheme based on a modification of hashkeys.

The key behind our design is, once a hashkey appears on any arc of a party, this party can relay it to all its related arcs, both outgoing arcs and incoming arcs. We first transform the directed graph \( G \) into an undirected graph \( G' \). If there is more than one arc between two vertices, we just add one to the undirected graph. Then, a hashkey corresponds to a simple path in the undirected graph. It times out after \( \text{MaxPathLength}(G) + |p| \Delta \), where \( G \) is the original directed graph containing all participants, and \( p \) is a path in the transformed undirected graph \( G' \). The Redeem phase ends after \( \text{MaxPathLength}(G) + \text{MaxPathLength}(G') \Delta \).

An asset transfer happens on an arc if one clause of its hashlock circuit is true. Each clause corresponds to hashlock circuits in one swap scheme, which is composed of the negations of conflicting preceding solutions’ (old) circuits and the current solution’s (old) hashlock circuit. The clause is true only if both of the following two conditions are met.

- All conflicting preceding solutions’ old circuits are false until timeout. That means the contract needed to wait until it timed out to decide whether conflicting preceding solutions’ old circuits are unlocked.
- Current solution’s old hashlock circuit need to evaluate to true. That means all hashes in the old hashlock circuit need to be unlocked before they time out.

All contracts agree on the order of conflicting solutions. A solution is triggered only if the conflicting preceding solutions are not triggered. Each arc will at most complete one asset transfer to one recipient.

### 7.3 Ranking Solutions

Solutions that can conflict with each other must be ranked to decide which one is preferable. This ranking is established by some kind of negotiation. Participants who are not proposers can accept if the proposed order is acceptable and leave the deal otherwise. We can design a more sophisticated protocol to make a smarter decision. Details are left to interested readers.

## 8 ANALYSIS AND PROOF

In this section, we analyze our proposed protocols. We provide definitions of required properties below, and detailed proof for those properties can be found in our full version paper [18].

### 8.1 Security Properties

Let \( S \) be the set of solutions output by an all-SAT solver. Here are some desired properties for both protocols.

**Definition 5. Universal Liveness:** if all parties are conforming, then a maximal set of compatible exchanges out of \( S \) can be completed.

**Definition 6. Local Liveness:** in a swap scheme \((G, \mathcal{H})\), if all involved parties are conforming, and this scheme is selected (in ProtocolA) or no scheme with higher priority is completed (in ProtocolB), then the asset transfers in \( G \) can be completed.

**Definition 7. Safety:** A conforming party \( x \) will never end up worse off, that is, \( x \) never pays an asset unless its liveness predicate is true, and never overpays (as defined by the safety predicate in Section 4.4).

**Definition 8. Fault-tolerance:** A party can complete an exchange according to its liveness predicate as long as there exists one swap scheme \( s \) where all parties are conforming, and \( s \) is chosen by negotiation (in ProtocolA) or no scheme with higher priority is completed (in ProtocolB). If this party is involved in \( m \) swap schemes (out of the swap schemes output by an all-SAT solver), we say they can tolerate \( (m - 1) \) failed schemes.

**Definition 9. Protocol is a strong Nash equilibrium strategy if no coalition improves its payoff when its members cooperatively deviate from the protocol.**
We compare the two proposed protocols with prior protocols. There is an extensive body of research on blockchain interoperability. Mercan et al. [9] propose protocols to improve transaction success rate in payment networks by improving payment channel networks’ performance with better routing strategies and ways to address imbalances.

8.2 Protocol Comparisons

We compare the two proposed protocols with prior protocols. Details can be found in our full version paper [18].

9 RELATED WORK

There is an extensive body of research on blockchain interoperability [3]. Some research addresses general-purpose cross-chain communication, focusing on the problem of reliably communicating the source chain’s internal state to a target chain. Other research addresses atomic asset transfers, for example, Alice trades her bitcoin for Bob’s ether.

This paper’s protocols focus on cross-chain asset transfers. Note that we do not mint or burn any assets. An asset transfer occurs between a sender and a receiver. There are many prior proposals for asset exchanges. Some are centralized [5], and some use connectors to route packetized payments across different blockchains [15]. These protocols are surveyed elsewhere [3, 13].

Many prior works utilize hashed timelock contracts (HTLCs). HTLCs allow one party to safely exchange assets with another party without the need for a trusted third party. The only trust anchors are the blockchains themselves, and any blockchain that supports smart contracts supports HTLCs. HTLCs are the basis for the Lightning network [11], for two-party atomic cross-chain swaps [4, 10, 19], and for multi-party atomic cross-chain swaps [6] on strongly-connected digraphs. In [14], the authors integrate off-chain steps to deal with swaps whose digraphs may not be strongly connected.

Cross-chain transactions that tolerate deviating participants are studied by Bagaria et al. [1], which proposes a technique called Boomerang to be used on top of multi-path routing schemes to construct redundant payment paths. A payment is split into \( n \) paths, each of which carries the same amount of payment. The payee can trigger \( t \) paths out of \( n \), where \( t \) yields the intended amount of payment. If the payee tries to cheat by collecting payments from more than \( t \) paths, all payments will be voided. The approach can tolerate participants on \( (n - t) \) paths being faulty. A limitation of this approach is that it has to split the payments to \( n \) equal shares and it does not support heterogeneous payments along different paths. Spear [12] improves Boomerang by allowing different payment amounts on different paths. However, Spear only works for single payments from one payer to one payee, e.g., from Alice to Bob, and the multi paths are disjoint. It is not straightforward to generalize these protocols to a cyclic graph, multiple paths overlap, and multiple parties have set up tentative redundant payments. Mercan et al. [9] propose protocols to improve transaction success rate in payment networks by improving payment channel networks’ performance with better routing strategies and ways to address imbalances.

10 REMARKS

We refer interested readers to our full version paper [18], which provides conclusions and interesting discussions, such as on how to choose the number of alternatives, how to achieve more fault-tolerance with a variant of hashkeys.

REFERENCES

[1] Vivek Bagaria, Joachim Neu, and David Tse. 2020. Boomerang: Redundancy improves latency and throughput in payment-channel networks. In International Conference on Financial Cryptography and Data Security. Springer. 304–324.

[2] Ann Becker and Dan Geiger. 1996. Optimization of Pearl’s method of conditioning and greedy-like approximation algorithms for the vertex feedback set problem. Artificial Intelligence 83, 1 (1996), 167 – 188. https://doi.org/10.1016/0004-3702(95)00004-6.

[3] Rafael Belchior, André Vasconcellos, Sérgio Guerreiro, and Miguel Correia. 2021. A survey on blockchain interoperability: Past, present, and future trends. ACM Computing Surveys (CSUR) 54, 8 (2021), 1–41.

[4] Trey Griffith. July, 2018. Sparkswap: Trade across blockchains without custody risk. https://medium.com/sparkswap/sparkswap-trade-across-blockchains-without-custody-risk-a6bfe0801586.

[5] Ethan Heilman, Sebastien Lipmann, and Sharon Goldberg. 2020. The arwen trading protocols. In International Conference on Financial Cryptography and Data Security. Springer. 156–173.

[6] Maurice Herlihy. 2018. Atomic cross-chain swaps. In Proceedings of the 2018 ACM symposium on principles of distributed computing (PODC ’18). ACM, New York, NY, USA, 245–254. https://doi.org/10.1145/3212734.3212736. Number of pages: 10 Place: Egham, United Kingdom tex.acm.id: 3212736.

[7] Maurice Herlihy, Barbara Liiskov, and Liuba Shrira. 2021. Cross-chain deals and adversarial commerce. The VLDB journal (2021), 1–19.

[8] Don Johnson, Alfred Menezes, and Scott Vanstone. 2001. The elliptic curve digital signature algorithm (ECDSSA). International journal of information security 1, 1 (2001), 36–63.

[9] Suat Mercan, Enes Erdin, and Kemal Akkaya. 2021. Improving transaction success rate in cryptocurrency payment channel networks. Computer Communications 166 (2021), 196–207.

[10] Tier Nolan. May, 2013. Alt chains and atomic transfers. https://bitcointalk.org/index.php?topic=193281.0. Bitcoin Forum.

[11] Joseph Poon and Thaddeus Dryja. 2016. The bitcoin lightning network: Scalable off-chain instant payments.

[12] Sonbol Rahimpour and Majid Khabbazian. 2021. Spear: fast multi-path payment with redundancy. In Proceedings of the 3rd ACM Conference on Advances in Financial Technologies. 183–191.

[13] Peter Robinson. 2021. Survey of crosschain communications protocols. Computer Networks 200 (2021), 108488.

[14] Narges Shadab, Farzin Houshmamed, and Molsen Lesani. 2020. Cross-chain Transactions. In 2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC) IEEE, 1–9.

[15] Stefan Thomas and Evan Schwartz. 2015. A protocol for interledger payments. https://blockchainlab.com/pdf/interledger.pdf.

[16] Takahisa Toda. 2015. All Solutions SAT Repository. http://www.sd.is.uic.ac.jp/~toda/code/allsat.html.

[17] Takahisa Toda and Takehide Soh. 2016. Implementing efficient all solutions SAT solvers. Journal of Experimental Algorithmics (JEA) 21 (2016), 1–44.

[18] Yingjie Xue, Di Jin, and Maurice Herlihy. 2022. Invited Paper: Fault-tolerant and economic-efficient redemption with redundancy. In Proceedings of the 3rd ACM Conference on Advances in Financial Technologies. 183–191.

[19] Yixin Yu, Pramod Subramanyan, Nestan Tsiskaridze, and Sharad Malik. 2014. All-SAT using minimal blocking clauses. In 2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems. IEEE, 86–91.

[20] Jean-Yves Zie, Jean-Christophe Deneuvile, Jérémy Briffaut, and Benjamin Nguyen. 2019. Extending atomic cross-chain swaps. In Data Privacy Management, Cryptocurrencies and Blockchain Technology. Springer, 219–229.