Cross-chain Transactions

Narges Shadab Farzin Hooshmand Mohsen Lesani
University of California, Riverside
{nshad001, fhous001, lesani}@ucr.edu

***Abstract*—The value of cryptocurrencies is highly volatile and investors require fast and reliable *exchange* systems. In *cross-chain transactions*, multiple parties exchange assets across multiple blockchains which can be represented as a directed graph $\mathcal{G}$ with vertexes V as parties and edges E as asset transfers. In a simple form, cross-chain transactions are *cross-chain swaps* where each edge e transfers an asset that the head of e already owns. However, in general, a cross-chain transaction includes a *sequence* of exchanges at each blockchain. Further, transactions may have *off-chain steps* and hence may not be *strongly connected*. Given a transaction, *protocol*s are desired that guarantee the following property called *uniformity*. If all parties conform to the protocol, all the assets should be transferred. Further, if any party deviates from the protocol, the conforming parties should not experience any loss. Previous work introduced a uniform protocol for *strongly connected cross-chain swaps* and showed that no uniform protocol exists for transactions that are not strongly connected. We present a uniform protocol for *general cross-chain transactions* with sequenced and off-chain steps when a few *certain parties are conforming*. Further, we prove a new property called *end-to-end* that guarantees that if the source parties pay, the sink parties are paid. We present a *synthesis tool* called XCHAIN that given a high-level description of a cross-transaction can automatically generate smart contracts in Solidity for all the parties.**

I. Introduction

With the promise of an open, verifiable and global financial system, the blockchain technology has attracted attention and investment. Bitcoin's market value surged from less than \$20 billion to more than \$200 billion and venture-capital funding for blockchain were up to \$1 billion in 2017 [1]. Considering the volatility of cryptocurrencies, investors need fast and reliable *exchange* mechanisms and trading platforms [2]–[4]. Although a transfer within a blockchain is atomic, the individual transfers of an exchange *across blockchains* are not atomic. Atomic execution of an exchange is challenging since single transfers are immutable and irreversible.

A *cross-chain transaction* can be modeled as a directed graph where parties are the vertices and the transfers are the edges. Edges are realized by smart contracts on different blockchains. In a simple form, a cross-chain transaction is a *cross-chain swap*. In a cross-chain swap, each edge transfers an asset that the source already owns. Fig. 1a is an example of cross-chain swap (adopted from [5]) in which Alice sends her ethers to Bob, Bob sends his bitcoins to Carol, and Carol sends her Cadillac to Alice. However, in general, a cross-chain transaction includes a *sequence* of exchanges at each blockchain. A party may need to execute steps in sequence instead of executing single independent swaps. For example, a broker may *borrow* an asset and trade it in sequence. For example, Fig. 1c shows a cross-chain transaction with sequencing. Bob wants to trade 3 bitcoins for 3 ethers and Carol wants to trade 3 ethers for 2 bitcoins. Alice mediates this trade and earns 1 bitcoin. Alice receives 3 bitcoins from Bob before she trades 2 of them with Carol. She has to execute two steps in sequence on the bitcoin blockchain: borrowing and then spending. To be able to create contracts for the bitcoins on her outgoing edges, she needs to own them. However, it is Bob that owns them. Thus, to execute this transaction, it should be *transformed* to an *equivalent* transaction shown in Fig. 1d. Two transactions are equivalent if each party's gain is the same in them.

Further, as a special case, transaction graphs can be *strongly connected*. Fig. 1a is an example. However, transactions may have *off-chain steps* and may not be strongly connected. Fig. 1b is an example where Alice eats at Carol's restaurant (off-chain); she sends her ethers to Bob who pays Carol in bitcoins. In addition, the transformed transactions such as Fig. 1d are often not strongly connected.

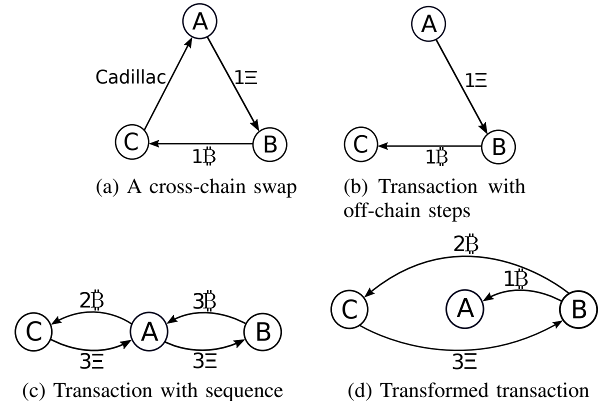


Fig. 1. Cross-chain Transactions

Transaction *protocol*s are desired that given a transaction, guarantee the following liveness and safety properties called *uniformity*. If all parties follow the protocol, the assets on all the edges are transferred. Further, if any of the parties deviates from the protocol, the protocol should be strong enough to prevent loss for conforming parties. Previous work [5] introduced a protocol for *strongly connected cross-chain*

This project was supported by the NSF grant #1657204.

978-1-7281-6680-3/20/\$31.00 2020 IEEE

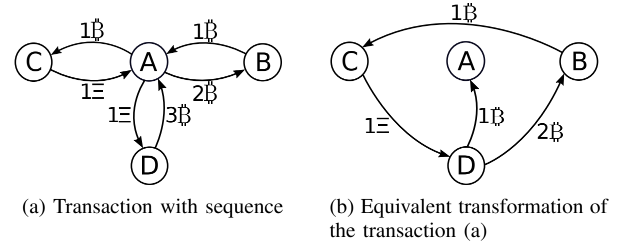
swaps and showed that no uniform protocol exists unless the transactions are strongly connected. In this paper, we assume that a few parties called *representative sources* are conforming. We present a uniform protocol for *general transaction graphs* including graphs that involve a sequence between their steps and graphs with off-chain steps (that may not be strongly connected). We present a *transformation* that given a transaction graph, a vertex and a blockchain, outputs an equivalent transaction graph with no sequence for that vertex on that blockchain. Subsequently, we present a transaction execution *protocol called 3PP* that in addition to uniformity, guarantees a new property called *end-to-end* for transactions with off-chain steps. The end-to-end property states that if the source parties pay, the sink parties are eventually paid. Further, we present a synthesis tool called XCHAIN that given a high-level description of a transaction graph, analyses the transaction and automatically *synthesizes smart contracts* in the Solidity language [6] for all the parties.

In the following sections, we first present basic definitions: the definition of cross-chain transactions, protocols and their properties (II). Then, we present the transformation algorithm (III), the exchange protocol and its properties (IV), and the implementation of the XCHAIN contract synthesis tool (V). Then, we discuss the related works (VI) before conclusion.

II. CROSS-CHAIN TRANSACTIONS

Basic Definitions. A directed *graph* (or digraph) $\mathcal{G}$ is a pair $\langle V, E\rangle$, where V is a finite set of vertexes, and E is a finite set of ordered pairs of distinct vertexes called edges. Consider an edge $e = \langle u, v\rangle$; we say that u is the *head* of e, e is an *outgoing* edge of u, v is the *tail* of e and e is an *incoming* edge of v. The (direct) *predecessors* of a vertex v are the set of vertices u such that $\langle u, v\rangle$ are in E. A vertex is a (local) *source* if it has no predecessors. The (direct) *successors* of a vertex u are the set of vertices v such that $\langle u, v\rangle$ are in E. A vertex is a (local) *sink* if it has no successors. A *path* is a sequence of edges such that the tail of each edge is the head of the next. The *head* of a path is the head of its first edge and the *tail* of a path is the tail of its last edge. A vertex is *reachable* from another if there is a path from the latter to the former. A *cycle* is a path with the same head and tail. A graph is *acyclic* if it has no cycles. A *feedback vertex set* of a graph is a set of vertices whose removal leaves a graph acyclic. A graph is *strongly connected* if all vertices are reachable from each other. The *strongly connected components (SCC)* of a directed graph are its maximal strongly connected subgraphs. A *directed acyclic graphs (DAG)* is a directed graph with no cycles. Every DAG has source and sink nodes. Every vertex of a DAG is reachable from a source and can reach a sink. The *condensation* of a graph is the result of contracting each of its SCCs into a single *super-vertex*. Every condensation graph is a DAG. Consider a condensation graph $\mathcal{C}$ of a graph $\mathcal{G}$. We call a source of $\mathcal{C}$ a *super-source* and a sink of $\mathcal{C}$ a *super-sink*. We call a vertex in a super-source or super-sink of $\mathcal{C}$ a *pseudo-source* or *pseudo-sink* of $\mathcal{G}$ respectively.

Cross-Chain Transactions. A *cross-chain transaction* $\mathcal{T}$ is a directed graph $\langle V, E\rangle$ where each vertex in V is a party and each edge in E is an asset transfer. An edge is a tuple $\langle u, v, b, a\rangle$ where u and v are the sending and receiving parties, b is the blockchain hosting the transfer and a is the transfer amount. Edges may be transferring assets from different blockchains and for different amounts. The transfers are performed by *smart contracts*. Thus, we use vertex and party, and, edge and smart contract interchangeably. In a transaction graph, each party agrees to relinquish assets on its outgoing edges if she receives assets on its incoming edges. Two transactions are *equivalent* if they have the same set of vertices and the net gain of each party is the same in the two transactions. For example, the two transactions in Fig. 1c and Fig. 1d are equivalent.



(a) Transaction with sequence (b) Equivalent transformation of the transaction (a)

Fig. 2. Cross-chain Transactions

Cross-chain Swaps and Cross-chain Transactions. A *cross-chain swap* is a special form of a cross-chain transaction. In a cross-chain swap, each edge transfers an asset that the source already owns. The transactions in Fig. 1a, Fig. 1d and Fig. 1b are all cross-chain swaps. However, in general, a cross-chain transaction involves a *sequence* of exchanges at each blockchain. A party may need to execute steps in sequence instead of executing single independent swaps. For example, a broker may borrow an asset and spend it in sequence. We saw a cross-chain transaction with sequencing in Fig. 1c and its equivalent transaction in Fig. 1d. As another example, Fig. 2a is a transaction where Alice is shorting bitcoin with respect to ethereum. She borrows a bitcoin from Bob and promises to return two. She then sells it for an ether to Carol and then sells the ether for three bitcoins to David! She then returns two bitcoins to Bob and gains a bitcoin herself. Alice has to execute sequences of transfers for both bitcoin and ethereum.

To execute such a transaction, it should be transformed to an equivalent transaction such that each party owns the assets on his outgoing edges. The two transactions in Fig. 2a and Fig. 2b are equivalent since each party's gain is the same. The cross-chain transactions may be strongly connected such as Fig. 1a, or they may have off-chain steps such as Fig. 1b, or they may transform to a not strongly connected graph such as Fig. 1c to Fig. 1d, and Fig. 2a to Fig. 2b.

Transaction Execution. We say that an edge $\langle u, v, b, a\rangle$ is *triggered* if the amount a of asset b is transferred from u to v. After a transaction execution, every vertex v can have four different states: $Deal$: all the incoming and outgoing edges of v are triggered, $NoDeal$: no incoming or outgoing of v is triggered, $UnderWater$: at least one outgoing edge is

triggered, but at least one incoming edge is not triggered, and $FreeRide$: some incoming edges are triggered but no outgoing edge is triggered. Parties only lose in $UnderWater$ states and we will present an algorithm that prevents them. The possible executions of a cross-chain transaction $\mathcal{T}$ are subgraphs of $\mathcal{T}$: an edge is in the subgraph iff it is triggered. An execution is *committed* iff it is a complete subgraph.

Transaction Protocols. Given a cross-chain transaction, a transaction *protocol* defines the steps that parties have to follow. As the correctness property for protocols, previous work [5] defined *uniformity* as the following two conditions. If all parties follow the protocol, the transaction should be committed. If any of the parties deviate from the protocol, the protocol should be strong enough to prevent $UnderWater$ for conforming parties. A party is *rational* if it acts in its own self-interest and deviates from the protocol only if deviation increases its gain. A protocol is desired only if rational parties follow it. A protocol $\mathcal{P}$ is a *Nash equilibrium* if no party improves its gain when it deviates from $\mathcal{P}$. Rational parties can collude with each other to increase their gain at the expense of other parties. A protocol $\mathcal{P}$ is a *Strong Nash equilibrium* if no coalition improves its gain by deviating from $\mathcal{P}$. We note that for cross-chain swaps, the second condition of uniformity implies strong Nash equilibrium. Consider a coalition that deviates from the protocol $\mathcal{P}$. As $\mathcal{P}$ is uniform, no conforming party loses. Therefore, the coalition cannot gain anything.

In general transaction graphs with sources and sinks, the sources want to pay the sinks (with or without an off-chain incentive). For example, in the example in Fig. 1b, Alice has the incentive to pay Carol because she has eaten in her restaurant. Executions of transaction are desired that when the sources pay, the sinks are eventually paid. The two conditions above do not capture this property. In particular, a source or a sink party cannot end up in an $UnderWater$ state as they do not have any incoming and outgoing contracts respectively. We define a new property called *end-to-end* on general graphs that precludes executions where pseudo-sources pay but pseudo-sinks are not. We say that an execution of a transaction is *source-paid* if an outgoing contract of a pseudo-source is triggered. Similarly, we say that it is *sink-paying* if all the incoming contracts of the pseudo-sinks are triggered. A protocol is end-to-end iff for every transaction that it executes, if the transaction is source-paid, it is sink-paying as well.

Thus, we make uniformity stronger by adding the third conjunct. A protocol $\mathcal{P}$ is *uniform* iff (1) If all parties follow $\mathcal{P}$, then the transaction is committed. (2) If any set of parties deviate from $\mathcal{P}$, no conforming party finishes with an $UnderWater$ state. (3) $\mathcal{P}$ is end-to-end.

Previous work [5] showed that if the transactions are not *strongly connected*, no uniform (and strong Nash equilibrium) protocol exists. Intituively, when the transaction is not strongly connected, there should be a set of vertices that pay another set but not vice versa. The first set of vertices can form a coalition that only trades internally and does not pay the other set. Thus, the first set can increase its gain by deviating from the protocol. We note that if the source vertices are conforming, no coalition

T_1 $\textsc{TransformSeq}(\mathcal{T}, v, b)$
T_2 let $\langle V, E\rangle := \mathcal{T}$ in
T_3 $E' := E$
T_4 $ivs := \{\langle u, a\rangle \mid \langle u, v, b, a\rangle \in E\}$
T_5 $ovs := \{\langle u', a'\rangle \mid \langle v, u', b, a\rangle \in E\}$
T_6 $E' := E' \setminus (\{\langle u, v, b, a\rangle \mid \exists a.\ \langle u, a\rangle \in ivs\} \cup \{\langle v, u', b, a\rangle \mid \exists a'.\ \langle u', a'\rangle \in ovs\})$
T_7 foreach $(\langle u', a'\rangle \in ovs)$
T_8 if $(ivs \neq \emptyset)$
T_9 $ovs = ovs \setminus \{\langle u', a'\rangle\}$
T_{10} foreach $(\langle u, a\rangle \in ivs)$
T_{11} $ivs = ivs \setminus \{\langle u, a\rangle\}$
T_{12} $E' := E' \cup \{\langle u, u', b, min(a, a')\rangle\}$
T_{13} if $(a > a')$
T_{14} $ivs := ivs \cup \{\langle u, a - a'\rangle\}$
T_{15} if $(a' > a)$
T_{16} $a' := a' - a$
T_{17} if $(ivs = \emptyset)$
T_{18} $ovs = ovs \cup \{\langle u', a'\rangle\}$
T_{19} else continue T_7
T_{20} if $(ivs \neq \emptyset)$
T_{21} $E' := E' \cup \{\langle u, v, b, a\rangle \mid \langle u, a\rangle \in ivs\}$
T_{22} if $(ovs \neq \emptyset)$
T_{22} $E' := E' \cup \{\langle v, u', b, a'\rangle \mid \langle u', a'\rangle \in ovs\}$
T_{23} return $\langle V, E'\rangle$

Fig. 3. Sequence Transformation

can increase its gain by deviation. Any coalition of vertices is reachable from a source (or includes a source). The coalition cannot include a conforming source. Further, the coalition will lose some assets on the path from the conforming source to the coalition. In this paper, we show that if a few parties called representative sources are conforming, uniform protocols exist for general transaction graphs including not strongly connected graphs.

III. Sequence Transformation

A transaction has a *sequence* between an incoming edge and an outgoing edge of a vertex if both edges transfer assets on the same blockchain. The vertex often does not own the asset that is on the outgoing edge and only acquires it when the incoming edge is triggered. For example, in Fig. 1c, Alice does not own the bitcoins that she should pass to Carol before she acquires them from Bob. Similarly, Alice does not own the outgoing ethers. Importantly, a transaction that involves a sequence cannot be directly executed. A contract should be issued for each edge of the graph by the head vertex of the edge. If the head vertex does not own the asset, it cannot issue the contract. For example, in Fig. 1c, Alice cannot issue contracts to pay Bob and Carol. Therefore, the first step for the market clearing service is to *transform* the transaction graph to an *equivalent* transaction graph such that each vertex owns the assets on its outgoing edges. We remember that two transaction graphs are equivalent if the net gain of each vertex is the same in the two transactions. The transaction in Fig. 1c can be transformed to the equivalent transaction in Fig. 1d. The gain of each party stays the same. Alice acquires 1 bitcoin, Bob spends 3 bitcoins and acquires 3 ethers and Carol spends 3 ethers and acquires 2 bitcoins. Intuitively, the transformation bypasses the middle party so that paying parties own the assets. Now, we explain the transformation more precisely.

The function TRANSFORMSEQ in Fig. 3 transforms the given transaction $\mathcal{T}$ to an equivalent transaction with no sequence at the given vertex v and blockchain b. The function should be called for every vertex and blockchain. The input transaction $\mathcal{T}$ is $\langle V, E\rangle$ (lines T_1-T_2) and the output is an equivalent transaction with the same set of vertices V and transformed edges E' with no sequence at vertex v and blockchain b. The set E' is first set to be equal to E (lines T_3) and is gradually transformed. The algorithm looks for a trade sequence and transforms it. The set ivs is the set of pairs of vertices u that pay v, and their amounts a (line T_4) and similarly, ovs is the set of pairs of vertices u' that are paid by v and their amounts a' (line T_5). The incoming and outgoing edges to v on b are removed from E (line T_6). The algorithm iterates over all the outgoing edges to vertices u' with amounts a' (lines T_7-T_9). To have an equivalent transaction, for each such edge, it iterates over incoming edges from vertices u with amounts a (lines T_{10}-T_{11}). A new direct edge is added from u to u' with the minimum of the amounts a and a' (line T_{12}). If the incoming amount a is more than the outgoing amount a', an incoming edge with the extra input amount is added (lines T_{13}-T_{14}). If the outgoing amount a' is more than the incoming amount a, the remainder output amount should still be compensated using the other incoming edges (lines T_{15}-T_{16}) or if there is no incoming edges left, an outgoing edge with the remained amount should be added back (lines T_{17}-T_{18}). Otherwise, this outgoing edge is totally compensated and the iteration continues for the next (line T_{19}). After iteration, if any paying vertices remain in ivs (line T_{20}), edges form those vertices with their corresponding amounts are added to the current vertex v (line T_{21}). There is a similar case for the outgoing edges (lines T_{22}-T_{23}).

For example, the transaction shown in Fig. 1c that has sequences for both bitcoin and ethereum on the vertex A can be transformed to the equivalent transaction in Fig. 1d. There are 3 incoming bitcoins from the predecessor B and 2 outgoing bitcoins to the successor C. The vertex A is bypassed to transfer 2 bitcoins directly from B to C. The remaining 1 bitcoin is transferred from B to A herself. Further, the vertex A is bypassed to transfer 3 ethers directed from C to B. Similarly, the transaction in Fig. 2a can be transformed to the equivalent transaction in Fig. 2b. The vertex A is bypassed to transfer 1 bitcoin directly from B to C, 1 ether directly from C to D, and 2 ethers from D to B. The remaining 1 bitcoin is transferred from D to A. In the transaction of Fig. 2b, although an apparent sequence exists on Bob, every party owns the outgoing assets. thus, the transaction is executable.

IV. TRANSACTION PROTOCOL

In this section, we incrementally present the *three-phase protocol (3PP)* to execute cross-chain transactions and prove its properties. We assume that sequences are removed from the input transactions by the transformation presented in III.

Distrust Tie. In a transaction, each party is willing to transfer assets on her outgoing edges if she receives assets on her incoming edges. A uniform protocol should prevent any conforming party from getting $UnderWater$ if other parties deviate from it. In particular, no party should find herself paying her outgoing edges but not receiving her incoming edges. A contract that is processed by a blockchain is irreversible. Further, contracts executed by two blockchains are independent and not atomic. A party cannot risk issuing unrestrained contracts to transfer her assets. In return, the other parties may not issue contracts to transfer assets to her. As the two parties symmetrically do not trust each other, there is a *tie* in the order of issuing contracts.

Hashed Timelock Contracts. To break the tie in the order of issuing contracts, *hashed timelock contracts* are used. A hashed timelock contract is locked by a *secret* s and expires after a certain time period t. It stores the two values *hash* h and *timeout* t where h is the result of applying a known hash function H to the secret s. The contract takes the secret as an input and calculates its hash. If the resulting hash matches h, it transfers the asset. Thus, by the collision-resistance property of the hash function, the contract can be triggered only by the correct secret; we say that the contract is *hashlocked*. Further, if the contract does not receive the matching secret before the time t, it *refund*s the asset; we say that the contract is *timelocked*. The tie between two parties of a transaction can be broken using hashed timelock contracts as follows: Alice generates a secret, calculates its hash and creates a contract with that hash to transfer her asset to Bob. Although she has issued the contract, she can protect her asset by holding the secret. Bob has received an incoming contract with a hash. He creates a contract with the same hash to transfer his asset to Alice. He knows that his contract can be triggered only with the matching secret and then he can learn and use the secret to trigger Alice's contract. To give Bob enough time, the timeout for Alice's contract should be twice as much as Bob's contract. In the case of inaction from the counterparty, each party is refunded by the contract after the timeout is elapsed.

Leaders and Followers. What is the protocol to safely issue contracts for multi-party transactions? The two-party exchange that we saw above is generalized to strongly connected graphs. To break the tie in the order of cycles in the graph, a *feedback vertex set* called *leaders* generate secrets. Every contract should be locked with all these secrets. A party that generates and holds a secret protects assets on her outgoing contracts even if her incoming contracts are not issued yet. Thus, leaders can safely initiate the creation of contracts by issuing their outgoing contracts. Parties other than the leaders are called *follower*s. A follower issues outgoing contracts from him once he observes all the incoming contracts to him. Previous work [5] presented a two-phased protocol and proved that if the graph is strongly connected, contract creation eventually propagates to every edge in the graph.

We assume that the transaction is assembled by a market-clearing service. Each party offers the service the trades he is willing to do. In addition, each party creates a secret s and calculates its hash h by applying a known hash function H. The party sends its hash value h together with his offers to the service. The service matches offers from several parties to

THREE-PHASE PROTOCOL (3PP)

P_1 ▷ Phase 1: Contract Creation
P_2 if (the party is a *leader*)
P_3 Issue all the outgoing contracts.
P_4 Wait for all the incoming contracts and validate them;
P_5 on invalid incoming contracts, stop.
P_6 if (the party is a *representative source*)
P_7 Wait for messages from all *representative sink*s.
P_8 else ▷ the party is a *follower*
P_9 Wait for all the incoming contracts and validate them;
P_{10} on invalid contracts, stop.
P_{11} Issue all the outgoing contracts.
P_{12} if (the party is a *representative sink*)
P_{13} Send messages to all *representative source*s.
P_{14} ▷ Phase 2: Release and Propagation of Secrets
P_{15} if (the party is in the *feedback vertex set*)
P_{16} Release the secret on the incoming contracts.
P_{17} Apply every secret that appears on the
P_{18} outgoing contracts on the incoming contracts.
P_{19} ▷ Phase 3: Relay and Propagation of Secrets
P_{20} if (the party is a *representative source*)
P_{21} Upon receipt of a secret, pass it to *pseudo-sinks*.
P_{22} else
P_{23} Apply secrets that are received from sources and
P_{24} from outgoing contracts to incoming contracts.

Fig. 4. Three-phase Transaction Protocol (3PP). The feedback vertex set, and the representative sources and sinks are given by the market-clearing service. The leader set is a feedback vertex set and representative sources.

assemble a transaction. Then, it analyses the transaction graph to find the leaders. It sends to the parties their roles (i.e. leader, follower, etc.), the parties that they will directly trade with and the hash values for the leaders. Then, the parties can execute the transaction with no coordination from the clearing service. Further, the parties do not need to trust the clearing service as they can check the validity of the contracts that they receive.

As we saw earlier, *general transaction graphs* may not be strongly connected and may have sources or sinks. Thus, if the contract creation starts from the feedback vertex set, it does not necessarily propagate to all vertices and some contracts may not be created. In particular, source vertices are not in the feedback vertex set and do not have any incoming edges. Therefore, contract creation does not propagate to sources. Further, we note that parties can protect their outgoing assets in two ways. The first is to be a leader, generate a secret, lock the contract with its hash and keep the secret. The second is to wait for all incoming contracts and then publish the outgoing contracts with the same hashes collected from the incoming contracts. Sources do not have any incoming edges. Thus, the only way to protect their outgoing assets is to include them in the leader set. More generally, a graph may not have source vertices but its condensation has super-source vertices. The vertices in a super-vertex are strongly connected and are reachable from each other. We call a vertex in a super-source, a pseudo-source. The market clearing service can choose an arbitrary pseudo-source in each super-source as a *representative source*. The leader set should include the representative sources in addition to the feedback vertex set.

Fig. 4 presents the three phase protocol (3PP). In the first phase, the contracts are created. In the second phase, leaders in the feedback vertex set release their secrets and the secrets are propagated. In the third phase the representative sources send the collected secrets to the representative sinks and the secrets are fully propagated.

Contract Creation. In the first phase (lines P_1-P_{13}) the leaders issue their outgoing contracts (lines P_2-P_3) and wait for their incoming contracts (line P_4). A follower issues outgoing contracts from himself once he observes all the incoming contracts to him (lines P_8-P_{11}). Each party *validate*s its incoming contracts. The contracts should be paying the right amount and protected by the hash values of the leaders. Parties proceed only if the validation is successful (lines P_5 and P_{10}). Every contract is created with a timeout to return the escrowed assets to the original owner if the transaction does not proceed as expected. We will consider how the timeout values are calculated after we see the structure of the protocol.

Representative sources kick-start the propagation in the super-sources. Further, the order tie in cycles is broken by the feedback vertex set and the representative sources. Therefore, if all the parties comply with the protocol, contract creation eventually propagates to the whole graph and all the contracts are issued. However, the protocol looks out for deviation. Before releasing the secrets in the next phases, the leaders wait to receive their expected incoming contracts (line P_4). They stop if the incoming contracts are invalid (line P_5). To make sure that the contract creation has propagated throughout the graph, representative sources wait and move to the next phase only after sinks notify that they have received their incoming contracts. More precisely, general graphs have super-sinks. The market clearing service can choose an arbitrary pseudo-sink in each super-sink as a *representative sink*. If the representative sink can observe its incoming contracts, the enclosing super-sink should have received its incoming contracts. Further, as the vertices in a super-vertex are strongly connected and are reachable from each other, the other pseudo-sink vertices in the super-sink will eventually receive their incoming contracts. Therefore, before moving to the next phase, the representative sources wait to receive messages from representative sinks (lines P_6-P_7). The representative sinks send a message to the representative sources when they receive their incoming contracts (lines P_{12}-P_{13}). The representative sources authenticate their received messages to ascertain the identity of the senders, using orthogonal cryptographic mechanisms.

Secret Release and Propagation. In the second phase, the secrets of the feedback vertex set are *released* and *propagated* (lines P_{14}-P_{18}). In the first phase, the feedback vertex set has observed that their expected incoming contracts are created. Thus, they *release* their secrets on their incoming contracts (lines P_{15}-P_{16}). Parties monitor their outgoing contracts. If a party observes that a secret is applied to one of her outgoing contracts, she applies it to her incoming contracts (lines P_{17}-P_{18}). Parties have the incentive to apply secrets to their incoming contracts to receive the assets. Therefore, the secret of each leader in the feedback vertex set propagates from her to parties *reachable* in the reverse order of contracts. To commit the transaction, all of its contracts should be triggered. Therefore,

all secrets should reach all parties. In a strongly connected graph, every vertex is reachable from every other vertex and the secrets trivially propagate to all vertices. However, in general graphs, not all the vertices are reachable (backwards) from the feedback vertex set. Intuitively, the secrets propagate back to sources (more precisely pseudo-sources) of the graph. Since secrets do not propagate to every vertex, a follow-up phase should complete the propagation.

Secret Relay and Propagation. The second phase of the 3PP protocol propagated secrets back to pseudo-sources. In the third phase (lines P_{19}-P_{24}), the secrets are *relayed* from the pseudo-sources to the pseudo-sinks and *propagated* back from them. More precisely, if the party is a representative source, she relays every secret that she has to the pseudo-sinks (lines P_{20}-P_{21}). These secrets include the secrets of the feedback vertex set and the secret of the representative source herself. Other parties propagate secrets that are received from sources and their outgoing contracts to their incoming contracts (lines 22-24). We note that although the second and third phases are conceptually two phases, the sources can propagate secrets to sinks as soon as they receive them.

We now calculate the time complexity of 3PP. Contract creation propagates from sources to sinks. The distance between the sources and sinks is at most the diameter $diam(\mathcal{T})$ of the transaction graph $\mathcal{T}$. Thus, it takes $O(diam(\mathcal{T}))$ time to create the contracts. The secrets are propagated from the feedback vertex set back to the sources and sent from the sources to the sinks and then propagated from the sinks back to sources. Thus, it takes $O(diam(\mathcal{T}))$ time to propagate the secrets. Therefore, the time complexity is $O(diam(\mathcal{T}))$.

Timeouts. Previous work [5] proposed timeouts for strongly connected graphs. We extend it for general graphs where a leader may be indirectly reachable.

The timeouts on the incoming contracts of each party should leave enough time Δ for the party to put the secrets that she observes on her outgoing contracts to her incoming contracts. A first idea is to simply set the timeout of the incoming contracts to the maximum of the timeouts of the outgoing contracts plus the time Δ. However, this approach is not applicable in cycles. Instead, different timeouts are set for different paths to a leader. In a transaction graph $\mathcal{T}$, each contract is hashlocked with hashes from each leader. Consider a leader v_l with the secret s and the hash h. The leader v_l may be directly reachable from a node v. In addition, a leader may be indirectly reachable through a pseudo-sink v_s and a representative source w_s where there is a path from v to v_s and then there is a path from w_s to v_l. The secrets propagate through indirect paths when representative sources relay secrets to pseudo-sinks. Each contract $\langle u, v\rangle$ is hashlocked for h that can be opened by multiple hashkeys. There is a hashkey σ and an associated timeout t for each path p from v to v_l. The hashkey σ is the result of each party in the path from v_l back to u signing s. The time out t is $(diam(\mathcal{T}) + |p| + 1) \times \Delta$ time units after the start of the protocol, where $diam(\mathcal{T})$ is the diameter of $\mathcal{T}$ and $|p|$ is the length of the path and accounts for the propagation time of the secret back from the leader. A hashlock on a contract has timed out when all of the hashkeys have timed out.

Uniformity of 3PP. We prove that 3PP is uniform if the representative sources comply with it.

Lemma 1 (Uniformity): If the representative source vertices are conforming, 3PP is uniform.

Uniformity is the conjunction of three conditions: (1) If all parties follow 3PP, then the transaction is committed. (2) If any set of parties deviate from 3PP, no conforming party finishes with an $UnderWater$ state. (3) $\mathcal{P}$ is end-to-end. We proved each separately in the appendix [7] section VIII.

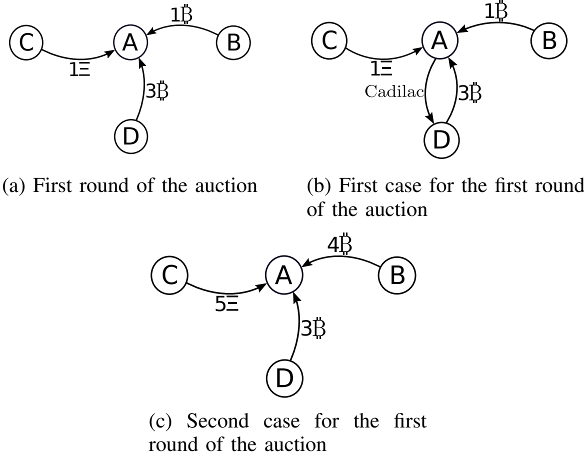
As we stated in section II, when the transaction is not strongly connected, there should be a set of vertices that pay another set but not vice versa. The first set of vertices can form a coalition that only trades internally and does not pay the other set. Every coalition is reachable from a representative source. Thus, the assumption about the conformity of the representative sources opens up possibilities to circumvent the impossibility result.

Examples. We now consider the execution of the protocol on a few examples. Consider the transaction in Fig. 1b. Alice is trying to pay Carol through an exchange with Bob. The transaction graph is acyclic. Thus, the leader set is only Alice as a representative source. Alice generates a secret and calculates the hash of it. Then, Alice creates a contract to Bob. Bob observes this incoming contract and creates a contract to Carol. Carol, the sink, observes the incoming contract and notifies Alice, the representative source, that she has received the incoming contract. In response, Alice passes her secret to Carol. Carol applies the secret to the incoming contract from Bob and receives a bitcoin. Thus, Bob learns the secret and applies it to the incoming contract from Alice to receive an ether. Thus, the transaction is successfully committed.

The second example shown in Fig. 2b is similar. There is a cycle between Bob, Carol and Dave and they constitute the super-source of the graph. Any one of them can be the representative source and the leader. Let's assume that Carol is that party. Carol generates a secret, calculates its hash and creates a contract to David. David observes the incoming contract and in response, creates contracts to Alice and Bob. In response, Bob creates a contract to Carol. Alice, the sink, notifies Carol, the representative source, that she has received her incoming contract. In response, Carol releases her secret to Alice. Alice applies the received secret to the contract coming from David. Then, David learns and applies the secret to the contract coming from Carol. Similarly, Carol applies it to the contract coming from Bob and Bob applies it to the contract coming from David. Thus, the transaction is successfully committed.

The presented approach can be extended to the auction problem illustrated in Fig. 5. Alice wants to sell her Cadilac in the auction, and Carol, Bob and David want to buy it. The auction is executed in rounds. The first round is shown in Fig. 5a. Carol, Bob and David have to generate their own secrets and calculate its hash. Then, each of them creates a contract with the hash of his or her own secret and sends

it to Alice. Alice compares the suggested bids and selects the highest offer. Let's assume that the highest bid of this round is 3 bitcoins. Then, Alice notifies all parties that the highest bid is 3 bitcoins. If no one suggests a higher price, then David wins the auction as shown in Fig. 5b. In this case, Alice creates a contract with the hash of the David's secret to pass her Cadilac to him. Then, David applies his secret to the incoming contract to get the Cadilac and Alice learns the secret and applies it to the contract coming from David to get 3 bitcoins. The second case is that someone may want to suggest a higher price. Fig. 5c is an example that Carol and Bob want to increase the deposit of their contracts to 5 ethers and 4 bitcoins respectively. Thus, the next round starts with new offers. This process may repeat multiple times.



(a) First round of the auction

(b) First case for the first round of the auction

(c) Second case for the first round of the auction

Fig. 5. The auction graphs

```
contract EdgeContract {
  address senderP, receiverP;
  bytes20[] hashedSecret;
  uint delta, diameter;
  bool[] unlocked, leaders;
  uint initTime;
  uint256 amount;
  function initiate() { /* initialize variables */ }
  function redeem_path_i (bytes32 _secret, Sig _sig) {
   /* The tail of receiverP of the path_i is the leader.
      the length of the path_i is len. */
    if (msg.sender == receiverP
        & H(_secret) == hashedSecret[leader]
        & verify(_sig, _secret)
        & block.timestamp <
        initTime + (diameter + len + 1) * delta)
      unlocked[leader] = true;
  }
  function claim() {
    if (msg.sender == receiverP
       & foreach leader: unlocked[leader] == true)
      Pay amount to receiverP;
  }
  function refund() {
    if (msg.sender == senderP & exists leader:
       /* with maximum length maxlen for paths to it */
         unlocked[leader] == false
         & block.timestamp >=
           initTime + (diameter + maxlen + 1) * delta)
      Pay amount to senderP;
} }
```

Fig. 6. Pseudo code of Smart Contract for an edge in Solidity.

```
function redeem_i (bytes32 _secret) {
  /* leader generates the secret and maxLen
     is maximum path length from source to leader. */
  if (msg.sender == receiverP
     & H(_secret) == hashedSecret[leader]
     & block.timestamp <
     initTime + (diameter + maxLen + 1) * delta)
    unlocked[leader] = true;
}
```

Fig. 7. Special redeem function

TABLE I
EVALUATION OF XCHAIN

#	#Nodes	#Leaders	Synthesis Time (ms)	Average Gas	Average Price (Ether)
1	2	1	294	1400331	0.02801
2	3	1	324	1440434	0.02881
3	4	1	330	1722722	0.03445
4	5	1	310	1520925	0.03042
5	6	2	318	1673709	0.03347
6	7	2	343	1716544	0.03433
7	8	4	351	2070887	0.04142

V. IMPLEMENTATION

In this section, we present our synthesis tool called XCHAIN and its experimental results. XCHAIN is a Java application that takes the transaction graph as input. It also inputs the address of each party in each blockchain. XCHAIN analyzes and transforms the input graph and synthesizes contracts for each edge of the transformed graph in the Solidity language [6]. XCHAIN can be used by a market clearing service.

Given a transaction graph, XCHAIN performs the following tasks in sequence: (1) It first finds the SCCs of the transaction graph. (2) It then finds a feedback vertex set for each of the SCCs using a 2-approximation algorithm [8]. (3) It calculates the condensation graph, finds its sources and sinks and chooses representative sources and sinks. (4) It determines the leader set as the union of the feedback vertex set of each SCC and representative sources. (5) It finds all the possible paths for each of the parties to each of the leaders. As we saw in IV, for the hashlock of each party for each leader, different timeouts are set for different paths. (6) It generates hashed timelock contracts in the Solidity language.

Fig 6 shows the outline of a contract that is generated for each edge in Solidity (with pseudo-code for readability). The contract has an `initiate` function that initializes the state of the contract when the sender deploys it. It has multiple `redeem` functions one for each path `path_i` from the receiver party to each leader. A `redeem` function gets the secret `_secret` of a leader and a hashkey `_sig` as input. It checks that the caller is the receiver party, the hash of `_secret` is the known hash value for `leader`, the hashkey `_sig` can be decoded back to the `_secret`, and considering the length `len` of the path, the hashkey has not timed out. If all the checks pass, the hashlock of the `leader` is unlocked. The contract also has a `claim` function for the receiver party to receive the asset. If the caller is the receiver party, and the hashlock for all the leaders are unlocked, then the receiver can claim the escrowed

amount. The contract also includes a `refund` function for the sender to reclaim the escrowed amount if there is an unlocked and timed out hashlock. The hashlock of a leader times out when even the hashkey of its longest path times out. As we show in Fig. 7, we generate specific `redeem` functions for pseudo-sink parties. Pseudo-sinks receive secrets directly from the representative sources. They have to check the hashkeys. Thus, the redeem functions for pseudo-sink parties only take the secrets as input and check if hashing them result in the expected hash values.

We have evaluated XCHAIN using transaction use-cases with varying number of vertices and leaders. The use-cases are available in the appendix [7] . We evaluate the time that it takes to synthesize the contracts and also the price to execute the generated contracts. We ran XCHAIN on an Intel Core i7 processor with 3.5 GHz speed and 16 GB Memory with MacOS version 10.14.3 with kernel version Darwin 18.2.0 and the JDK version 10.0.1. We used Remix's JavaScript Ethereum VM environments [9] to deploy the contracts. The cost needed for transactions to be committed is measured by gas. The consumed gas depends on upcodes of the contract's bytecode [10]. We recorded the gas consumption which is calculated and reported by Remix. This includes a call to `initiate`, multiple calls to `redeem`, and a call to `claim`. We calculated the average gas consumption over the generated contracts and reported the average. The value of one unit of a Gas is set to 0.00000002 ethers to calculate the average price. Table I represents the results. XCHAIN can generate contracts in less than a second for the use-cases. Further, although the execution of the contracts involves hashing, decoding and checking, the average price for the execution of a contract is a few hundredths of ether. In addition, we observe that the gas consumption depends on the number of hashlocks in the contract that is determined by the number of leaders.

VI. Related Works

Protocols. The fair exchange problem arises when two parties want to exchange their assets. The outcome of a fair exchange must be either that the two parties end up trading their assets, or that they both keep their assets. This problem has been studied even before the blockchain technology [11]–[15]. The optimistic fair exchange protocol [11] relies on invisible trusted parties: parties that work as a background service and intervene only in case of a misbehaviour. Similarly, the secure group barter protocol [12] studies multi-party barter with semi-trusted agents. To the best of our knowledge, it was back in 2013 when the notion of cross-chain swaps first emerged in an online forum [16]. Atomic cross-chain swap is since a known problem to the blockchain community [16]–[19]. The two wiki pages [17] and [16] proposed protocols for bilateral swaps: two-party transactions. Later, platforms such as Komodo BarterDEX [20] and deCRED [21] emerged. However, these projects offer bilateral swaps and do not support the notion of borrowing and sequence [2]–[4]. Previous works [5], [22] presented a protocol for strongly connected cross-chain swaps and showed that no uniform protocol exists unless the transactions are strongly connected. This paper considers general transaction graphs including graphs with a sequence and graphs with off-chain steps that are not strongly connected.

Verification and Synthesis. Programming smart contracts is a subtle task; a simple flaw in a critical contract can lead to catastrophes. For example, DAO hack [23] led to the loss of $55M worth of digital money. Therefore, formal verification of blockchain protocols as well as smart contracts has been an area of interest. A verification effort [24] uses a combination of temporal logic and strategy logic to precisely specify pre-conditions and post-conditions for a bilateral swap protocol and applies the MCK model checker to verify it. Post-condition for such an algorithm states that a fair trade must happen between two parties. Automatic generation of smart contracts has also been studied in the literature [25]–[27]. In [27], a function in a contract on one chain can call functions in contracts on other chains. In [26], smart contracts are automatically generated from semantic rules and constraint specifications. In [25], smart contracts are automatically generated from institutional specifications. Pluralize [28] addresses the problem of trusting external sources of information such as IoT sensors for contracts. It introduces a new formal framework equipped with a formal smart contract language to specify accountability for external sources and to verify contracts based on their accountability.

Scilla [29], [30] is an intermediate-level programming language that presents stateful contracts as communicating state transition systems. This language can be translated to the Gallina language of the Coq proof assistance to verify temporal properties of contracts formally. Further, the translated contracts can be compiled to bytecode to execute on blockchain nodes. A similar work [31] translates contracts from Solidity to an embedding in F^* and uses it to verify and compile the contracts. EthIR [32] is a bytecode analysis tool for ethereum contracts. It decompiles the bytecode into a high-level representation called rule-based form that keeps the control-flow and data-flow analysis results.

Matching and Optimization. The matching problem is to maximize the number [33] or to minimize the cost [34] of pair-wise matchings for the given objects. In the remote area of kidney exchange markets, clearing algorithms for barter exchange are used [35]–[37]. Optimization techniques have been applied to maximize the profit of off-chain transactions [38]. Further, routing protocols have been optimized for off-chain transactions [39]. These algorithms can be applied to maximize the matchings or fees. Given a multi-party transaction, this paper presents a protocol to safely execute it.

VII. Conclusion

This paper presented the 3PP protocol for general cross-chain transactions with both sequenced and off-chain steps. It showed the uniformity of the protocol with the additional end-to-end property that guarantees that if the source parties pay, the sink parties are paid. It presented the XCHAIN synthesis tool that analyzes high-level descriptions of transactions and quickly generates contracts in Solidity.

REFERENCES

[1] "Blockchain beyond the hype: What is the strategic business value?" https://www.mckinsey.com/business-functions/digital-mckinsey/our-insights/blockchain-beyond-the-hype-what-is-the-strategic-business-value.

[2] G. Zyskind, C. Kisagun, and C. Fromknecht, "Enigma catalyst : A machine-based investing platform and infrastructure for crypto-assets," 2017.

[3] "Secret auction smart contracts with enigma: A walkthrough," https://blog.enigma.co/secret-auction-smart-contracts-with-enigma-a-walkthrough-ec27f89f9f7c, [Online; accessed 27-January-2019].

[4] "Towards a decentralized data marketplace part 2," https://blog.enigma.co/towards-a-decentralized-data-marketplace-part-2-1362c8e11094, [Online; accessed 27-January-2019].

[5] M. Herlihy, "Atomic cross-chain swaps," in *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, ser. PODC '18. New York, NY, USA: ACM, 2018, pp. 245–254. [Online]. Available: http://doi.acm.org/10.1145/3212734.3212736

[6] "Solidity documentation," https://solidity.readthedocs.io/en/latest/index.html, [Online; accessed 23-January-2019].

[7] Shadab, Hooshmand, and Lesani, "Appendix," 2020.

[8] A. Becker and D. Geiger, "Optimization of pearl's method of conditioning and greedy-like approximation algorithms for the vertex feedback set problem," *Artif. Intell.*, vol. 83, no. 1, pp. 167–188, May 1996. [Online]. Available: http://dx.doi.org/10.1016/0004-3702(95)00004-6

[9] "Remix framework," https://remix.ethereum.org/, 2019, [Online; Feb-2019].

[10] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, pp. 1–32, 2014.

[11] S. Micali, "Simple and fast optimistic protocols for fair electronic exchange," in *Proceedings of the Twenty-second Annual Symposium on Principles of Distributed Computing*, ser. PODC '03. New York, NY, USA: ACM, 2003, pp. 12–19. [Online]. Available: http://doi.acm.org/10.1145/872035.872038

[12] M. Franklin and G. Tsudik, "Secure group barter: Multi-party fair exchange with semi-trusted neutral parties," in *Financial Cryptography*, R. Hirchfeld, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 90–102.

[13] M. Ben-Or, O. Goldreich, S. Micali, and R. L. Rivest, "A fair protocol for signing contracts," in *Automata, Languages and Programming*, W. Brauer, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1985, pp. 43–52.

[14] N. Asokan, M. Schunter, and M. Waidner, "Optimistic protocols for fair exchange," in *Proceedings of the 4th ACM Conference on Computer and Communications Security*, ser. CCS '97. New York, NY, USA: ACM, 1997, pp. 7–17. [Online]. Available: http://doi.acm.org/10.1145/266420.266426

[15] N. Asokan, V. Shoup, and M. Waidner, "Optimistic fair exchange of digital signatures," *IEEE Journal on Selected Areas in Communications*, vol. 18, pp. 593–610, 1997.

[16] Tier Nolan, "Alt chains and atomic transfers," https://bitcointalk.org/index.php?topic=193281.msg2224949#msg2224949, 2013, [Online; accessed 23-January-2019].

[17] bitcoinwiki, "Atomic swap," https://en.bitcoin.it/wiki/Atomic_swap, [Online; accessed 23-January-2019].

[18] ——, "Hashed timelock contracts," https://en.bitcoin.it/wiki/Hashed_Timelock_Contracts, [Online; accessed 23-January-2019].

[19] S. Bowe and D. Hopwood, "Hashed time-locked contract transactions," https://github.com/bitcoin/bitcoin/pull/7601, [Online; accessed 23-January-2019].

[20] "Barterdex: Decentralised exchange and cryptocurrency market," https://github.com/KomodoPlatform/BarterDEX, [Online; accessed 27-January-2019].

[21] deCRED, "On-chain atomic swaps for decred and other cryptocurrencies," https://github.com/decred/atomicswap, [Online; accessed 27-January-2019].

[22] M. Herlihy, B. Liskov, and L. Shrira, "Cross-chain deals and adversarial commerce," *arXiv preprint arXiv:1905.09743*, 2019.

[23] "A hacking of more than $50 million dashes hopes in the world of virtual currency," https://www.nytimes.com/2016/06/18/business/dealbook/hacker-may-have-removed-more-than-50-million-from-experimental-.cybercurrency-project.html, 2016, [Online; accessed 23-January-2019].

[24] R. van der Meyden, "On the specification and verification of atomic swap smart contracts," *arXiv preprint arXiv:1811.06099*, 2018.

[25] C. K. Frantz and M. Nowostawski, "From institutions to code: Towards automated generation of smart contracts," in *2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS* W)*. IEEE, 2016, pp. 210–215.

[26] O. Choudhury, N. Rudolph, I. Sylla, N. Fairoza, and A. Das, "Auto-generation of smart contracts from domain-specific ontologies and semantic rules," in *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. IEEE, 2018, pp. 963–970.

[27] P. Robinson, D. Hyland-Wood, R. Saltini, S. Johnson, and J. Brainard, "Atomic crosschain transactions for ethereum private sidechains," *arXiv preprint arXiv:1904.12079*, 2019.

[28] Z. Dargaye, A. D. Pozzo, and S. Tucci Piergiovanni, "Pluralize: a trustworthy framework for high-level smart contract-draft," *CoRR*, vol. abs/1812.05444, 2018. [Online]. Available: http://arxiv.org/abs/1812.05444

[29] I. Sergey, A. Kumar, and A. Hobor, "Scilla: a smart contract intermediate-level language," *arXiv preprint arXiv:1801.00687*, 2018.

[30] ——, "Temporal properties of smart contracts," in *ISoLA*, 2018.

[31] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Zanella-Béguelin, "Formal verification of smart contracts: Short paper," in *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, ser. PLAS '16. New York, NY, USA: ACM, 2016, pp. 91–96. [Online]. Available: http://doi.acm.org/10.1145/2993600.2993611

[32] E. Albert, P. Gordillo, B. Livshits, A. Rubio, and I. Sergey, "Ethir: A framework for high-level analysis of ethereum bytecode," *arXiv preprint arXiv:1805.07208*, 2018.

[33] R. M. Kaplan, "An improved algorithm for multi-way trading for exchange and barter," *Electron. Commer. Rec. Appl.*, vol. 10, no. 1, pp. 67–74, January 2011. [Online]. Available: http://dx.doi.org/10.1016/j.elerap.2010.08.001

[34] H. N. Gabow and R. E. Tarjan, "Faster scaling algorithms for general graph matching problems," *J. ACM*, vol. 38, no. 4, pp. 815–853, October 1991. [Online]. Available: http://doi.acm.org/10.1145/115234.115366

[35] Z. Jia, P. Tang, R. Wang, and H. Zhang, "Efficient near-optimal algorithms for barter exchange," in *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*, ser. AAMAS '17. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2017, pp. 362–370. [Online]. Available: http://dl.acm.org/citation.cfm?id=3091125.3091181

[36] J. P. Dickerson, D. F. Manlove, B. Plaut, T. Sandholm, and J. Trimble, "Position-indexed formulations for kidney exchange," in *Proceedings of the 2016 ACM Conference on Economics and Computation*, ser. EC '16. New York, NY, USA: ACM, 2016, pp. 25–42. [Online]. Available: http://doi.acm.org/10.1145/2940716.2940759

[37] D. J. Abraham, A. Blum, and T. Sandholm, "Clearing algorithms for barter exchange markets: Enabling nationwide kidney exchanges," in *Proceedings of the 8th ACM Conference on Electronic Commerce*, ser. EC '07. New York, NY, USA: ACM, 2007, pp. 295–304. [Online]. Available: http://doi.acm.org/10.1145/1250910.1250954

[38] O. Ersoy, S. Ross, and Z. Erkin, "How to profit from payments channels," *arXiv preprint arXiv:1911.08803*, 2019.

[39] P. Hoenisch and I. Weber, "Aodv–based routing for payment channel networks," in *International Conference on Blockchain*. Springer, 2018, pp. 107–124.