Generalized Swap Graphs for Blockchain Protocols

Stephan Dübler
Max Planck Institute for Security and Privacy
Bochum, Germany
stephan.duebler@mpi-sp.org

Pedro Moreno-Sanchez
IMDEA Software Institute
Madrid, Spain
pedro.moreno@imdea.org

Clara Schneidewind
Max Planck Institute for Security and Privacy
Bochum, Germany
clara.schneidewind@mpi-sp.org

***Abstract*—Atomic swaps enable two users holding assets in different cryptocurrencies to safely exchange them without relying on a trusted third party. Recent works have extended this idea to *swap graphs*, describing exchanges of assets between multiple users that should be executed atomically. In this work, we observe that the notion of swap graphs can be generalized to describe a broader class of interesting blockchain protocols, which rely on atomic transaction execution. Following this observation, we broaden the class of swap graphs considered in prior work and define a new protocol for securely realizing such graphs from more general primitives. The resulting class of graphs is expressive enough to capture existing multi-party blockchain protocols such as multi-hop payments and crowdfunding.**

***Index Terms*—blockchain, cryptocurrency, atomic swaps**

I. Introduction

Atomic swap protocols allow two users of blockchain-based cryptocurrencies to exchange their assets without the need for a trusted third party while ensuring that (1) if both parties conform to the protocol, then both swaps take place; and (2) if a party misbehaves, the honest party does not lose assets. At its core, an atomic swap is a 2-phase commit protocol where assets are initially committed with respect to a cryptographic secret such that they both are released to the intended receivers if the secret is revealed before a predefined timeout, or refunded to the initial owners otherwise.

Problem generalization. Previous work [3] extends the notion of such bilateral swaps to swaps among multiple parties as described by a strongly connected *swap graph* (see Fig. 1, left). We observe that the core of many blockchain protocols – beyond atomic swaps – consists of the atomic execution of several transactions. An example are multi-hop payments in payment channel networks where a payment is routed along several intermediaries: The protocol security relies on the fact that either all payments along a path are executed or all of them are reverted. Such atomic execution is achieved by a variant of the aforementioned 2-phase commit paradigm: Funds between users are pairwise locked in an initial setup

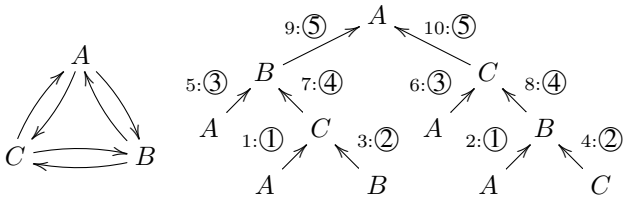

Fig. 1. Swap graph (left) and corresponding swap tree (right).

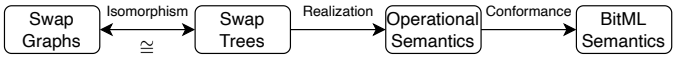

Fig. 2. Overview of the Approach

phase and configured such that they can only be released with the knowledge of a specific cryptographic secret. During the release phase, the parties learn the secrets required for retrieving their funds as soon as the previously locked assets are taken away from them.

Motivated by these common principles, we generalize the concept of swap graphs to be sufficiently expressive for capturing a broader class of protocols such as multi-hop payments in payment channel networks [5], rebalancing [4], and crowdfunding [6]. To this end, we show how to broaden the class of swap graphs beyond strongly connected ones. Next, we give a new protocol for realizing generalized swap graphs based on general primitives supported by most cryptocurrencies.

Protocol generalization. The swap graph protocol in [3] is based on a specific form of *hashed timelock contracts (HTLCs)*. Assets deposited in an HTLC with parameters A, B, Y and t can either (1) be released by user A by publicly revealing a secret x such that $H(x) = Y$ or (2) released to user B after time t. In [3] a more powerful version of this contract is used that operates on a vector $\vec{Y}$ of hash values and a vector $\vec{t}$ of timelocks and ensures that money locked in such contract can only be released by user A if all secrets x_i corresponding to $Y_i \in \vec{Y}$ have been released before time t_i.

In contrast to standard HTLCs, the version used in [3] is inherently stateful and as such cannot be implemented in Bitcoin-like currencies, which only support stateless contracts.

We define a new protocol for generalized swap graphs that is built upon a more general primitive and, hence, is not tied to cryptocurrencies with stateful contracts. To this end, we define the notion of conditioned timelock contracts (CTLCs): A CTLC is a generalization of a standard HTLC in that (i) it supports cryptographic conditions given by hard relations and not only hash functions; and (ii) instead of releasing the funds to party A or B, the funds can be released into a follow-up CTLC contract. We argue that CTLCs do not only subsume HTLCs but can even be implemented in cryptocurrencies without any contract support; using ideas presented in [6].

Protocol correctness and security. For proving the new protocol correct and secure, we follow the steps in Fig. 2: We describe the protocol using a tree structure (*swap tree*)

derived from the swap graph. The correspondence between these structures can be shown by a topological isomorphism. For defining the protocol, we associate the edges of a swap tree with CTLCs and define the setup and execution of these CTLCs based on the swap tree. For formally proving the resulting protocol secure, we define an operational semantics for CTLCs. We validate these semantics against a realistic blockchain transaction execution model by showing them to coincide with those of a fragment of BitML [2] – a language for Bitcoin smart contracts, which is proven sound with respect to a computational model of Bitcoin transaction execution.

Constructions and proofs in this paper are work in progress.

II. Swap Graph Protocol

Preliminaries. A swap graph is formally given by a *directed graph* (or just *digraph*), which consists of *vertices* and *arcs*. Here, vertices resemble the (assets of) participating parties and arcs desired transactions[1]. Two nodes are connected if there exists an arc between them and we call a sequence of arcs a *walk*. A digraph is *strongly connected* if, for every two vertices, there exist connecting walks in both directions. If there is a vertex A in the digraph that can be reached starting from every other node we call the graph *in-semiconnected w.r.t.* A. In-semiconnectedness will be our minimum requirement for converting a swap graph into a swap tree. Note that every strongly connected digraph is in-semiconnected w.r.t. every node. The formal definitions are deferred to Appendix A. When referring to swap trees, we will use the notions *edges* and *nodes* instead of arcs and vertices.

We will assume CTLCs to rely on a hard relation R with statement/witness pairs (Y, x), which supports distributive composition operations $\cdot$ and $+$ such for all $(Y_1, x_1), (Y_2, x_2) \in R$ it holds that $(Y_1 * Y_2, x_1 + x_2) \in R$. Here, we will call statements *conditions* and witnesses *secrets*.

Graph-to-tree conversion. Every digraph that is in-semiconnected w.r.t. a leader A can be unfolded into a swap tree by choosing A as a root and connecting all ingoing arcs including their tails. This procedure gets repeated until there are no more ingoing arcs or the current vertex has appeared twice on its walk up to the leader. Given the digraph from Fig. 1 we can choose any vertex as the leader. Choosing A results in the tree on the right. Note that multiple edges in the tree may correspond to a single arc of the swap graph and hence to the same asset (e.g., edges 1 and 6).

Protocol description. Based on this tree and the CTLC primitive, we define a three-phase protocol: First, all participants create secret/condition pairs from R and exchange their statements. Each edge in the tree will correspond to a CTLC with a condition that can be opened with the knowledge of a (composition of) secret(s). The secret for the CTLC of an edge $U \to V$ in the tree is composed of all secrets of the parent edge as well as a fresh secret v_i of party V. The secrets for the example in Fig. 1 are shown in Table I where secrets of party U are denoted u_i. Second, the users set up CTLCs corresponding to the tree edges starting from the leaves up to the root. The timelocks for the CTLCs are derived from the level of the edge in the tree relative to the protocol starting time t_0. Edges in the tree that spend the same asset are locked into a nested CTLC. By the setup order, we ensure that users can never lose funds due to an inconsistent setup: Whenever a user U sets up a CTLC for an outgoing arc at node n either (i) ingoing edges of n are already set up to enable U to acquire all funds they may receive during the swap or (ii) there is another occurrence of U on the path of n to the root and hence U can prevent the execution from reaching n[2].

TABLE I
CTLC parameters per tree edge.

No.	*Creation order*	*Involved parties*	*Timelock*	*Secrets*
1	①	$A \to C$	$t_0 + 3\Delta$	a_1, b_2, c_1
2	①	$A \to B$	$t_0 + 3\Delta$	a_2, c_4, b_3
3	②	$B \to C$	$t_0 + 3\Delta$	a_1, b_2, c_2
4	②	$C \to B$	$t_0 + 3\Delta$	a_2, c_4, b_4
5	③	$A \to B$	$t_0 + 2\Delta$	a_1, b_1
6	③	$A \to C$	$t_0 + 2\Delta$	a_2, c_3
7	④	$C \to B$	$t_0 + 2\Delta$	a_1, b_2
8	④	$B \to C$	$t_0 + 2\Delta$	a_2, c_4
9	⑤	$B \to A$	$t_0 + 1\Delta$	a_1
10	⑤	$C \to A$	$t_0 + 1\Delta$	a_2

Once all transactions have been set up, the release phase starts at time t_0. Due to the way that the secrets have been chosen, the CTLC for an edge can be triggered once its parent edge has been triggered. For instance, as soon as party A triggers the CTLC of edge 9, the secret a_1 is revealed to B what allows B to trigger the CTLCs of edges 5 and 7. Furthermore, the increasing timelocks guarantee that whenever the CTLC for an outgoing edge has been pulled from a node there is enough time remaining for it to pull its ingoing ones.

Protocol analysis. We can show that swap trees correctly and securely realize their underlying swap graphs by proving an isomorphism of topological spaces between the two structures. Based on this isomorphism, we can show that (1) if all parties follow the protocol all arcs of the swap graph get executed (correctness) (2) no honest party may be left worse off than in an honest swap execution (security). Our security notion relaxes the one of [3] which requires honest parties to never end up *underwater* – a situation where assets of the party are claimed without the party receiving all assets as indicated by the swap. This is needed as in non-strongly-connected swap graphs (e.g. multi-hop payments) an honest swap graph execution may require participants to come out underwater.

To formally prove that the protocol execution follows the tree structure when being implemented using CTLCs, we give an operational semantics for CTLCs. These CTLC semantics coincides with a fragment of the BitML language [2] for Bitcoin smart contracts, and hence showcases that swap graphs can be provable realized upon Bitcoin-like cryptocurrencies.

[1]For the sake of presentation, we assume here that each party trades exactly one asset and use parties and their assets synonymously.

[2]This only holds for strongly-connected graphs. Achieving a safe setup for other graphs requires additional secrets as we detail in Appendix B.

Applications and Outlook. As outlined in the previous section, we can give a formal proof showing that our novel protocol can be realized in Bitcoin. However, the generality of the underlying CTLC primitive allows for the usage of swap graphs to describe protocols that not only span different cryptocurrencies but may even operate on layer-2 infrastructure like payment channel networks. To substantiate this claim, in future work, we want to draw a formal connection between the proposed CTLC semantics and these systems.

REFERENCES

[1] Jørgen Bang-Jensen and Gregory Gutin. *Classes of directed graphs*, volume 11. Springer, 2018.

[2] Massimo Bartoletti and Roberto Zunino. Bitml: a calculus for bitcoin smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 83–100, 2018.

[3] Maurice Herlihy. Atomic cross-chain swaps. In *Proceedings of the 2018 ACM symposium on principles of distributed computing*, pages 245–254, 2018.

[4] Rami Khalil and Arthur Gervais. Revive: Rebalancing off-blockchain payment networks. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 439–453, 2017.

[5] Giulio Malavolta, Pedro Moreno-Sanchez, Clara Schneidewind, Aniket Kate, and Matteo Maffei. Anonymous multi-hop locks for blockchain scalability and interoperability. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019*, 2019.

[6] Sri AravindaKrishnan Thyagarajan, Giulio Malavolta, and Pedro Moreno-Sanchez. Universal atomic swaps: Secure exchange of coins across all blockchains. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1299–1316. IEEE, 2022.

APPENDIX

A. Graphtheory

Definition A.1. A *directed graph* (or just *digraph*) $\mathfrak{D}$ consists of a non-empty finite set $\mathfrak{N}$ of *vertices* or *nodes* and a finite set $\mathfrak{A}$ of ordered pairs of distinct vertices called *arcs*. We call $\mathfrak{N}$ the *vertex set* or *set of nodes* and $\mathfrak{A}$ the *set of arcs*, $\mathfrak{D}$ is then defined as $\mathfrak{D} := (\mathfrak{N}, \mathfrak{A})$. For an arc $(A, B) \in \mathfrak{A}$ we call A its *tail*, B its *head*. Two nodes $A, B \in \mathfrak{N}$ are *connected* if there exists an arc $(A, B) \in \mathfrak{A}$ or $(B, A) \in \mathfrak{A}$, which will be denoted with $A \to B$ or $B \to A$ respectively. Also, requiring the two vertices forming an arc to be distinct removes loops from one node back to itself from the set of arcs [1, p.3].

Definition A.2. A *walk* in $\mathfrak{D} = (\mathfrak{N}, \mathfrak{A})$ is an alternating sequence $W := A_1 a_1 A_2 a_2 A_3 ... A_{k-1} a_{k-1} A_k$ of $A_i \in \mathfrak{N}$ and $a_j \in \mathfrak{A}$ such that A_i and A_{i+1} are end-vertices of a_i for every $i \in \{1, 2, ..., k-1\}$, $k \geq 2$. Furthermore, if A_i and A_{i+1} are the tail and head of a_i respectively, for every $i \in \{1, 2, ..., k-1\}$, then W is called a *directed walk* or *diwalk*. We say that W is a *diwalk from* A_1 *to* A_k or an (A_1, A_k)-*diwalk* [1, p.7].

Definition A.3. $\mathfrak{D} = (\mathfrak{N}, \mathfrak{A})$ is *strongly connected* if

$$\forall A, B \in \mathfrak{N} : \exists\, (A, B)\text{-diwalk} \wedge \exists\, (B, A)\text{-diwalk [1, p.7].}$$

Definition A.4. Let $A \in \mathfrak{N}$ be an arbitrarily chosen vertex from $\mathfrak{D}$, then we define the *reachable set* of A as

$$N_{\mathfrak{D}}(A) := \{C \in \mathfrak{N} \setminus \{A\} \mid \exists\, (C, A)\text{-diwalk}\} \text{ [1, p.16].}$$

Definition A.5. Let $\mathfrak{D} = (\mathfrak{N}, \mathfrak{A})$ be a digraph and $A \in \mathfrak{N}$. Then we call $\mathfrak{D}$ *in-semiconnected w.r.t.* A if and only if

$$N_{\mathfrak{D}}(A) \cup \{A\} = \mathfrak{N}.$$

B. Setup for in-semiconnected digraphs

When the swap graph is not strongly connected but only in-semiconnected w.r.t. a leader the situation can come up where a party A only appears in a leaf and not another time further up in the tree. This is problematic as it is possible that only A sets up its outgoing transaction but the other participants are not. In this case, A would be discontent with the outcome. For instance in a Multi-Hop Payment like the one in Fig. 3 where A wants to pay C through an intermediary B the party A only wants to set up its CTLC if it can guarantee that the funds get forwarded to C. Here digraph and tree coincide with C being the leader. We can prevent undesired outcomes by adding a

$$A \xrightarrow{a,b,c} B \xrightarrow{a,c} C$$

Fig. 3. Multi-Hop Payment

secret a of A to all CTLCs, which can be revealed by A between the setup and execution phase. This mechanism is only part of the protocol if the graph is not strongly connected, as it makes for another party which could potentially delay the execution.