PRIVACY-PRESERVING PROTOCOL FOR ATOMIC SWAP BETWEEN BLOCKCHAINS

by
Kiran Gurung

A thesis

submitted in partial fulfillment

of the requirements for the degree of

Master of Science in Computer Science

Boise State University

May 2020

BOISE STATE UNIVERSITY GRADUATE COLLEGE

DEFENSE COMMITTEE AND FINAL READING APPROVALS

of the thesis submitted by

Kiran Gurung

Thesis Title: Privacy-Preserving Protocol for Atomic Swap Between Blockchains

Date of Final Oral Examination: 13 March 2020

The following individuals read and discussed the thesis submitted by student Kiran Gurung, and they evaluated the presentation and response to questions during the final oral examination. They found that the student passed the final oral examination.

Gaby Dagher, Ph.D.	Chair, Supervisory Committee
Hoda Mehrpouyan, Ph.D.	Member, Supervisory Committee
Casey Kennington, Ph.D.	Member, Supervisory Committee

The final reading approval of the thesis was granted by Gaby Dagher, Ph.D., Chair of the Supervisory Committee. The thesis was approved by the Graduate College.

For “my family, friends and myself”

ACKNOWLEDGMENTS

I want to thank my advisor, Dr. Gaby Dagher, for guiding me on this research endeavour which would not have been possible without his insights and encouragement. His academic excellence and technical expertise were very helpful throughout the research and the writing. His generous attitude made the research journey comfortable and smooth.

I would also like to thank my committee members, Dr. Hoda Mehrpouyanan and Dr. Casey Kennington for their support and advice for improvements in this thesis.

I feel fortunate to be a part of Boise State University with an incredible group of fellow students and faculty. I want to thank them all. Specifically, I am grateful to my fellow graduate student, Joshua Holmes, for his constructive input which was invaluable towards the completion of this thesis.

Finally, I would like to thank my family and friends who were always supportive and encouraging, without them this journey would have been very arduous and lonely.

ABSTRACT

Atomic swap facilitates fair exchange of cryptocurrencies without the need for a trusted authority. It is regarded as one of the prominent technologies for the cryptocurrency ecosystem, helping to realize the idea of a decentralized blockchain introduced by Bitcoin. However, due to the heterogeneity of the cryptocurrency systems, developing efficient and privacy-preserving atomic swap protocols has proven challenging. In this thesis, we propose a generic framework for atomic swap, called PolySwap, that enables fair exchange of assets between two heterogeneous sets of blockchains. Our construction 1) does not require a trusted third party, 2) preserves the anonymity of the swap by preventing transactions from being linked or distinguished, and 3) does not require any scripting capability in blockchain. To achieve our goal, we introduce a novel secret sharing signature (SSSig) scheme to remove the necessity of common interfaces between blockchains in question. These secret sharing signatures allow an arbitrarily large number of signatures to be bound together such that the release of any single transaction on one blockchain opens the remaining transactions for the other party, allowing multi-chain atomic swaps while still being indistinguishable from a standard signature. We provide construction details of secret sharing signatures for ECDSA, Schnorr, and CryptoNote-style Ring signatures. Additionally, we provide an alternative contingency protocol, allowing parties to exchange to and from blockchains that do not support any form of time-locked escape transactions. A successful execution of PolySwap shows that it takes 8.3 seconds to complete an atomic swap between Bitcoin's Testnet3 and Ethereum's Rinkeby (excluding confirmation time).

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AMHL – Anonymous Multi-Hop Lock

ECDSA – Elliptic Curve Digital Signature Algorithm

EdDSA – Edwards-curve Digital Signature Algorithm

HTLC – Hashed Time-Locked Contracts

NIZKP – Non-Interactive Zero Knowledge Proofs

P2P – Peer-to-Peer

P2PKH – Pay to Public Key Hash

PCN – Payment Channel Network

PoW – Proof of Work

SSSig – Secret Sharing Signature Scheme

TTP – Trusted Third Party

UTXO – Unspent Transaction Output

ZKCP – Zero Knowledge Contingent Payment

ZKP – Zero Knowledge Proofs

LIST OF SYMBOLS

$\mathcal{P}_j$	Party j for the protocol
$\mathcal{L}_j$	List of blockchains for Party j
$\mathcal{B}_i^{(j)}$	i^{th} blockchain in List $\mathcal{L}_j$
$T_{\mathcal{B}_i^{(j)}}$	Unsigned transaction in blockchain $\mathcal{B}_i$ in List $\mathcal{L}_j$
$T^C_{\mathcal{B}_i^{(j)}}$	Unsigned claim transaction in blockchain $\mathcal{B}_i$ in List $\mathcal{L}_j$
$T^E_{\mathcal{B}_i^{(j)}}$	Unsigned escape transaction in blockchain $\mathcal{B}_i$ in List $\mathcal{L}_j$
σ	Cryptographic signature
(T, σ)	Tuple representing redeemable transaction
α	Private key for Party 1
β	Private key for Party 2
pk	Shared public key for both parites
ϕ_i	Unlocking secret for party i
Φ_i	Public form of unlocking secret for party i
$\bar{\sigma}$	Set of components of a signature
L_x	List of x type of elements
L_λ	List of order of group of secrets

$[\![c]\!]$	Commitment of plain text c
κ	Key to a commitment
δ	Difficulty parameter for time locks
π	Proof returned by zero knowledge protocols
Π'	Concealed time-locked puzzles
Π	Revealed time-locked puzzles
Ω	Output of polynomial locking algorithm

Chapter 1

INTRODUCTION

Bitcoin [1], introduced in a landmark paper in 2008, is a decentralized digital currency system for secure electronic payments. It relies on a public ledger called blockchain, which is maintained by a peer-to-peer network of participants, following consensus rules based on proof of work, where they expend some computation time to produce certain proofs that can be verified easily. Bitcoin is a pseudonymous system with respect to user privacy. Accounts (or addresses) are hashes of public keys of a public cryptosystem, and the transfer of funds from one account to another is authorized through a signature on the spending (input) transaction. There is nothing in the Bitcoin system that inherently links users to their real-world identities. However, all transactions with their details (such as senders' & receivers' addresses, values) are published to the public ledger. This introduces several problems concerning user privacy in Bitcoin and similar public ledger-based cryptocurrency systems. For example, an adversary can link a cluster of addresses to a user [2], associate it to their personally identifiable information [3], and view their transaction behavior [4]. It also affects the fungibility of Bitcoin. Fungibility is a valuable property required for any currency system, and it refers to the trait that each unit of a token in the system is equivalent in value to another unit and is interchangeable.

In this thesis, we study how to enhance users' privacy during token exchanges, and propose a privacy-preserving protocol for atomic swap between two sets of blockchains.

1.1 Motivation

Various blockchain-based cryptocurrency systems have been proposed and implemented to address different limitations in Bitcoin such as user privacy [5, 6], transaction throughput [7], and distributed applications [8]. With the continuous introduction of new cryptocurrency systems, the demand for mechanisms facilitating interoperability among them has risen; this is evident by the presence of a large number of cryptocurrency exchanges and their daily trading volume [9]. Most of these exchanges are centralized and custodial[1], requiring the users to entrust them with their cryptocurrency assets in order for the users to be able to use their services. However, centralized exchanges are often the target for hackers and exit scams, resulting in users' assets being lost [11, 12]. Atomic swap [13] is the cornerstone of decentralized exchanges, enabling mutually distrusting parties to exchange a cryptocurrency asset for another without requiring the involvement of a trusted third party. Atomic swap is a type of fair exchange [14, 15] where two distrusting parties seek to exchange assets on the condition that either both party receives the other party's asset, or neither party receives anything. It is known that a fair exchange protocol cannot be constructed without a trusted third party (TTP) [16]. Atomic swap, being a fair exchange of cryptocurrencies, is not exempt from this requirement. However, the blockchain itself can be utilized as a TTP, which allows an atomic swap to be realized without an *explicit* TTP.

The concept of atomic swap first surfaced in a BitcoinTalk forum [17], where Tier Nolan proposed a Bitcoin-compatible atomic swap solution without an explicit TTP based on linking transactions together with a secret. The solution uses *Hashed Time-Locked Contracts (HTLC)* refering to a type of transaction which is only spendable in the network

[1] Out of the top 100 cryptocurrency exchanges, only 4 claim to be decentralized [10]

by providing the hash after a fixed duration of time. A hash function is a one way function which maps a pre-image (data of an arbitrary size) to a hash (binary string of fixed length). Using HTLC, transactions can be locked until a pre-image to the hash is released. This allows the two parties to lock two transactions in different blockchains with the same hash, such that when one of the transactions is accepted by the blockchain, the pre-image of the hash is revealed, allowing the other transaction to be redeemed on the other blockchain. However, since the same hash is used to lock both transactions, it is trivial for an observer (global passive adversary) to link them together to an atomic swap which is detrimental for their privacy.

The *Bitcoin Lightning Network* [18]—a Payment Channel Network (PCN) [19]—has also used HTLC transactions with a common pre-image to lock all the transactions in a single channel. However, this enables an adversarial node within a payment channel to identify all the other nodes in the channel, thus compromising their privacy. This privacy concern was addressed by Malavolta et al. [20] by using a multi-hop HTLC. The authors later improved on this concept by proposing an anonymous multi-hop lock (AMHL) [21], which do not require HTLC. Nevertheless, their solution is not applicable to atomic swap between heterogeneous blockchains, as it requires their ECDSA and Schnorr based construction to be instantiated over the same cryptographic group. Other solutions for fair exchange of assets in the context of cryptocurrencies have also been proposed [22, 23, 24, 25]. However, these approaches are not generic as they require specific features such as rich scripting capabilities and multisig accounts in the participating blockchain.

There are typically two main privacy concerns with respect to atomic swaps: 1) *Linkability*, where an observer is able to establish a link between atomic swap transactions from different blockchains, due to the use of the same hash to lock claim transactions (i.e. using HTLC to provide atomicity), and 2) *Distinguishability*, where an observer can

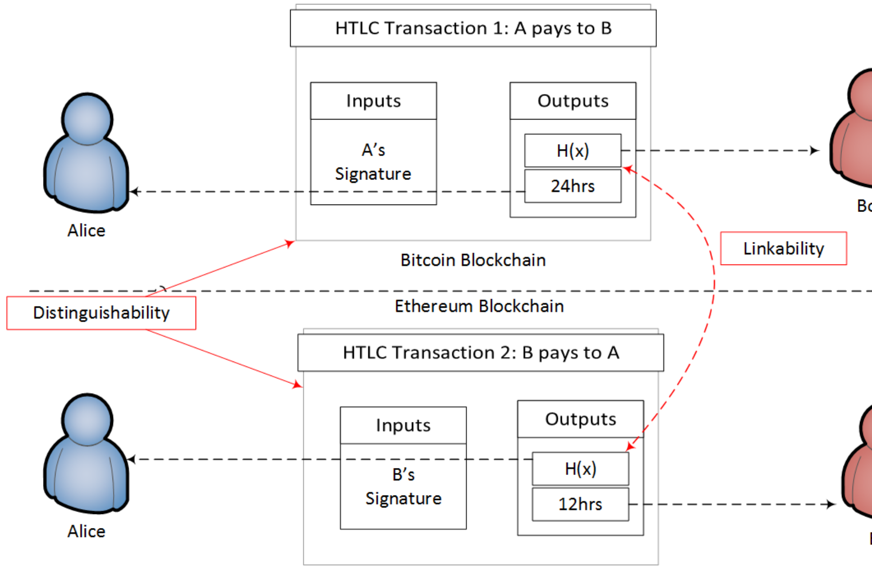


Figure 1.1: Distinguishability and Linkability in HTLC-based atomic swap

distinguish an atomic swap transaction on a blockchain from normal transactions on the same blockchain, which are illustrated in Figure 1.1. We define ***privacy-preserving atomic swap*** as an atomic swap protocol that ensures both unlinkability and indistinguishability (fungibility) of transactions.

1.2 Thesis Statement

The objective of this thesis is to answer the following research question: Can atomic swap between blockchains be achieved at scale without a trusted-third party while preserving users' privacy? More specifically, given two mutually distrusting parties who are interested in exchanging tokens between cryptocurrency blockchains, is it feasible to design an efficient and scalable atomic swap protocol that achieves the privacy-preserving properties, unlinkability and indistinguishability, without the involvement of a trusted-third party?

We answer the research question affirmatively by designing and implementing a privacy-preserving atomic swap protocol, PolySwap, without the presence of any trusted-third party using transactions that are both indistinguishable and unlinkable. We experimentally evaluate PolySwap by executing atomic swap between test networks of Bitcoin and Ethereum blockchains.

1.3 Organization of the Thesis

The rest of the thesis is organized as follows: Chapter 2 introduces required background necessary for the work in this thesis. Chapter 3 reviews different works relating to fair exchange in Bitcoin and other cryptocurrency systems. Chapter 4 provides overview of the protocol along with different building blocks required for the proposed protocol. Chapter 5 describes our solution for privacy-preserving atomic swap. Chapter 6 details the experimental evaluation of proposed protocol. Chapter 7 concludes this thesis and provides directions for future work.

Chapter 2

BACKGROUND

In this chapter, we describe some of the crypytographic preliminaries required for the construction of the proposed solution.

2.1 Shamir's Secret Sharing

Shamir's Secret Sharing scheme [26] is based on the fact that a polynomial of degree $k-1$ requires k distinct points to evaluate. This property is used to implement a (k, n) threshold scheme to share a secret, where at least k unique points out of n are required to determine the secret. Let, $q(x)$ be a polynomial of degree $k-1$. For $x \in [1, n]$, $q(x)$ generates n unique points, which represent the shares of the secret. When at least k of these n points are known, the coefficients of the polynomial $q(x)$ can be calculated to release the secret.

2.2 Time-Locked Puzzles

Time-locked Puzzles [27] are cryptographic puzzles which enable hiding messages for a duration of time. These puzzles guarantee that the receiver cannot see the message until the time duration has elapsed by making the process of solving them inherently sequential, meaning a large number of machines running the solution algorithm cannot solve the puzzle faster than a single machine.

2.3 Commitment Schemes

A commitment scheme enables a party to commit to a chosen secret without revealing it (hiding), while allowing the party to reveal the committed secret later without being able to cheat (binding). Given an elliptic curve group $\mathbb{G}$ of prime order q, the Pedersen commitment [28] of message $m \in \mathbb{Z}_q$ is calculated as $m \cdot G + r \cdot H$, where $r \leftarrow_{\$} \mathbb{Z}_q$ and G & H are base points in the curve. We define a commitment scheme with commitment algorithm $\{[\![c]\!], \kappa\} \leftarrow \mathsf{com}(c, \mathbb{G})$ that produces a Pedersen commitment $[\![c]\!]$ on message c and a key κ to open the commitment in group $\mathbb{G}$. We also define the verification algorithm as $\{0, 1\} \leftarrow \mathsf{V}_{\mathsf{com}}([\![c]\!], c, \kappa)$.

2.4 Zero Knowledge Proofs

Zero Knowledge Proofs (ZKP) [29] are protocols run between a prover and a verifier, where the prover convinces the verifier about the validity of an assertion without revealing anything else besides the fact that the assertion is true. ZKP should satisfy three basic properties: *Completeness*, *Soundness* and *Zero-Knowledge*. A prover can always convince the verifier of the assertion if it is valid (completeness), the verifier rejects the proof with high probability if the assertion is not valid (soundness), and the verifier does not learn anything besides the validity of the assertion (zero-knowledge).

Given a relation $R : \{0, 1\}^* \times \{0, 1\}^* \to \{0, 1\}$ in NP, and a language $\mathcal{L}$ such that $\mathcal{L} = \{x \mid \exists w \text{ s.t. } R(x, w) = 1\}$, we assume there exists a non-interactive zero knowledge functionality with a prover algorithm $\pi \leftarrow \mathsf{P}_{\mathsf{zk}}(x, w)$ and a verifier algorithm $\{0, 1\} \leftarrow \mathsf{V}_{\mathsf{zk}}(\pi)$ where π is the proof additionally containing the public inputs x. We assume that if the verifier has any public inputs included in π beforehand, they will confirm that the public values from π match the values they have–allowing the proofs to be verified on their own. We additionally define a committed Non-Interactive Zero Knowledge Protocol (NIZKP) functionality. The prover algorithm is $\{[\![\pi]\!], \pi, \kappa\} \leftarrow \mathsf{P}_{\text{com-zk}}(x, w)$, where $[\![\pi]\!]$ is the commitment of π and κ is the commitment key. The verifier algorithm is $\{0,1\} \leftarrow \mathsf{V}_{\text{com-zk}}([\![\pi]\!], \pi, \kappa)$. We use superscripts in the algorithms to distinguish different proofs; DH for a Diffie-Hellman proof and DL for a Discrete Log proof.

Proof of Group Conversion (GC). As we work across different cryptocurrencies which use different cryptographic groups, it is important to be able to prove that a hidden value in one group is equivalent to a hidden value in another. We define a prover algorithm $\pi \leftarrow \mathsf{P}^{\mathsf{GC}}_{\mathsf{zk}}(\{[\![m]\!]_1, [\![m]\!]_2\}, \{m, \kappa_1, \kappa_2\})$, where $[\![m]\!]_j$ is a hidden form, a commitment or encryption, of message m on key κ_j in group $\mathbb{G}_j$ for $j \in \{1, 2\}$. We also define its accompanying verifier algorithm $\{0,1\} \leftarrow \mathsf{V}^{\mathsf{GC}}_{\mathsf{zk}}(\pi)$. Between discrete log groups, i.e. elliptic curves, this can simply be done through a bit-wise comparison, to show that each bit is the same, and a range proof, to show that the value hidden in the larger group is within the range of the smaller group. We also require a group conversion from Paillier encryption scheme to a smaller elliptic curve group. For this, we will utilize Lindell's proof for L_{PDL} [30].

2.5 Signature Schemes

In the following, we briefly mention some signature schemes used in different cryptocurrencies.

ECDSA Signature. Let $\mathbb{G}$ be an elliptic curve group of prime order q with base point G and $\mathcal{H} : \{0,1\}^* \rightarrow \mathbb{Z}_q$ be a hash function. For a private key, $x \leftarrow_{\$} \mathbb{Z}_q$ and the corresponding public key, $Q = x \cdot G$, to compute an ECDSA signature [31] on message m, sample $k \leftarrow_{\$} \mathbb{Z}_q$ and compute $R = k \cdot G$. Let, $(r_x, r_y) \leftarrow R$, then, the signature is (r, s), where $r \leftarrow r_x$

$\mod q$ and $s \leftarrow k^{-1} \cdot (\mathcal{H}(m) + r \cdot x) \mod q$. We are interested in distributed signing by Lindell [30] and its use to achieve conditional signing by Malavolta *et al.* [21] Popular cryptocurrencies like Bitcoin, Ethereum, Litecoin use ECDSA Signatures.

Schnorr Signature. Let $\mathbb{G}$ be an elliptic curve group of prime order q with base point G and $\mathcal{H} : \{0,1\}^* \rightarrow \mathbb{Z}_q$ be a hash function. For a private key, $x \leftarrow_{\$} \mathbb{Z}_q$ and the corresponding public key, $Q = x \cdot G$, to compute an Schnorr signature on message m, sample $k \leftarrow_{\$} \mathbb{Z}_q$ and compute $R = k \cdot G$. Also, compute $e = \mathcal{H}(R||Q||m)$, then, the signature is (e, s), where $s \leftarrow k - e \cdot x \mod q$. Schnorr Signatures are currently not used in any cryptocurrency system but due to their simplicity and various useful properties in context of cryptocurrencies, different cryptocurrencies including Bitcoin are trying to use them.

Cryptonote-style Signature. Popular privacy focused cryptocurrency Monero uses cryptonote-style signatures. For details of the signature scheme, we refer readers to [5].

Chapter 3

LITERATURE REVIEW

In this chapter, we review the literature in domains which relate to fair exchange in Bitcoin, viz., Fair exchange of digital goods, Secure Multiparty Computation, Payment channels and Interoperability protocols. Table 3.1 summarizes the features of the related approaches, including our proposed protocol.

3.1 Fair Exchange of Digital Goods

Fair exchange of digital goods refers to protocols which enable a customer to pay a merchant for digital goods or services while ensuring that the customer gets what he paid for and the merchant gets paid if the customer receives his goods. Numerous protocols [22, 23, 25, 32, 33, 34] have been proposed for fair exchange of goods over blockchain, most of which are enforced using smart contracts. Zero-Knowledge Contingent Payment (ZKCP) [22] leverages Zero Knowledge Proofs (ZKP) to enable a seller to prove the knowledge of a secret the buyer is interested in. The release of a payment to the buyer is contingent on the seller presenting a key to a smart contract to be used by the buyer to learn the secret. This is possible in blockchains that have rich scripting capabilities, such as Ethereum. Campanelli *et al.* [23] point out a possible violation of the zero-knowledge property in the NIZK proof used in ZKCP if the common reference string (CRS) is maliciously constructed by the buyer, and present a different protocol to address the violation called Zero-Knowledge Contingent Service Payments (ZKCSP) for digital services. It addresses a use case where the proof itself is the good being sold, so ZKCP cannot be used. Banasik *et al.* [33] present ZKCP without scripts over Bitcoin's blockchain, using a standard cut-and-choose technique to construct contracts. Dziembowski *et al.* [25] propose a solution to the same problem in an efficient manner without the use of computationally expensive ZKP. The solution is based on proof of misbehavior—an idea that it is cheaper to prove incorrect behavior than correct behavior—which can be presented to a judge contract in case of disagreement. Goldfeder *et al.* [32] study security and privacy properties offered by different escrow protocols and propose several schemes that are usable over a blockchain. The authors define different metrics that can be used to describe the privacy properties of fair exchange schemes. However, ZKCP and its enhancements are proposed for the exchange of digital goods over blockchain and it is not clear how they can be extended to accomplish atomic swap. In addition, scripting capabilities in the blockchain is a requirement for most of these protocols, while a third party is also needed to enforce fairness.

3.2 Secure Multiparty Computation

Secure multiparty computation over blockchain [35, 24, 36, 37] closely relates to fairness. Bentov and Kumaresan [35] formalized a claim-or-refund functionality among others for secure computation over Bitcoin. Andrychowicz *et al.* [24] describe fair two-party and multi-party computations via bitcoin deposits using Bitcoin-based timed commitments. The commiter pays a deposit to get involved in a computation that is returned only if he opens his commitment within some specific time, introducing a penalty scheme to enforce fairness. Kumaresan *et al.* [36, 37] explore this idea further by improving on its efficiency through the reduction of the total size and number of required transactions.

3.3 Payment Channels

Payment channels over blockchain comprises of off-chain payment protocols guaranteeing eventual transaction finality on the blockchain while providing a varying level of security and privacy. Payment Channel Networks [38, 20, 39, 18], and Payment Channel Hubs [40, 41]—primarily proposed as scaling solutions to blockchain—tackle a similar problem to our protocol concerning guaranteed payments using off-chain transactions. Lightning network [18] enables off-chain payments between distrusting users, where payments are enforced using HTLC transactions. But in order to make cross-chain payments, the participating blockchains must support similar hash-functions. Malavolta *et al*. [20] propose Multi-Hop HTLC protocol to address privacy concerns in such PCNs, i.e., the payment route could be derived if a common hash is used. However, Multi-Hop HTLC's privacy solution does not apply in the case of a single hop [19], which is essentially an atomic swap. To address this issue, AMHL was recently proposed by Malavolta *et al.* [21]. Nevertheless, in order for AMHL to support heterogeneous blockchains, modifications to the cryptocurrency systems are required so that the signature schemes are instantiated over same elliptic curve group. Bolt [40] is an anonymous bidirectional payment channel scheme that provides payment unlinkability and anonymity, but it does not support Bitcoin. TumbleBit [41] is a mixing service that enables unlinkable payments using an untrusted intermediary however, the presence of the intermediary is detrimental to the anonymity of the users in case of collusion.

Table 3.1: Comparative evaluation of techniques for fair exchange (✓=supported property; (✓)=partially supported; ✗=does not support ; * implies the swap is between more than two heterogeneous blockchains; n/a=not applicable)

Protocol	No Intermediaries		# of Chains		Privacy		Required Functionality
	Trusted	Untrusted	Two	Multi*	Unlinkability	Indistinguishability	
Atomic swap [TierNolan] [48]	✓	✓	✓	✗	✗	✗	HTLC
Escrow Protocols [32]	✗	n/a	✓	✓	✗	(✓)	n/a
TumbleBit [41]	✓	✗	(✓)	✗	✓	✗	HTLC
Bolt [40]	✓	✓	✗	✗	✓	✓	Zcash
Lightning [18]	✓	✗	(✓)	✗	✗	✗	HTLC
AMHL [21]	✓	✗	✓	✓	✓	✓	ECDSA/Schnorr Signatures
Xclaim [43]	✓	✓	✓	✗	✓	✗	Scripting capabilities
Proposed Protocol: PolySwap	✓	✓	✓	✓	✓	✓	SSSig reducible signatures

3.4 Interoperability Protocols

Interoperability protocols focus on connectivity and data sharing across blockchains [42, 43, 44, 45, 46, 47]. HyperService [42] describes a platform for interoperability and programmability for heterogeneous blockchains as a third party service, with a focus on programmability of cross-chain decentralized applications for developers. Xclaim [43] proposes interoperability by locking backing cryptocurrency in its native blockchain for equivalent tokens in the issuing cryptocurrency. Nonetheless, fairly expressive scripting capabilities are required in the issuing blockchain. Arwen trading protocol [44] proposes a non-custodial protocol for trading cryptocurrencies over a centralized exchange, but it only supports Bitcoin-derived cryptocurrencies. Gazi *et al.* [46] study sidechains in proof-of-stake blockchains for interoperability. Thomas *et al.* [45] propose an interledger payment protocol; however, the protocol requires both an escrowed transfer mechanism in each blockchain and third parties "connectors" to process payments. Delgado-Segura *et al.* [47] present a data trading protocol in Bitcoin exploiting an ECDSA vulnerability to reveal the private key on release of a signature on the blockchain. However, it requires specific scripts to function which prevents indistinguishability.

Chapter 4

PROTOCOL OVERVIEW AND BUILDING BLOCKS

In this chapter, we present definitions and assumptions for PolySwap, provide an overview of PolySwap and present our building blocks required for the main PolySwap protocol.

4.1 Definitions

4.1.1 Adversarial Model

We assume that a *probabilistic polynomial time* adversary $\mathcal{A}$ can corrupt any of the parties during the execution of the protocol. We consider the static corruption model where an adversary controls either party throughout the execution and cannot change parties midway. We further assume that the parties have instantaneous access to each blockchain's *mempool* and can extract signatures from a transaction not yet confirmed by the network. This allows the adversaries to access the unlocking secrets for Secret Sharing Signature (SSSig) prematurely and potentially create opposing transactions.

4.1.2 Security Definitions

In this section, we present general security definitions of a privacy-preserving multi-chain atomic swap (PMAS).

Let, $\mathcal{L}_j$ represent a list of blockchains $\mathcal{B}_i^{(j)}$ for a party $\mathcal{P}_j$ where $j \in \{1, 2\}$ and $i \in \{1, 2, \ldots, \mathcal{L}.\texttt{size}\}$.

Definition 4.1.1. (Valid List pair) A list pair $(\mathcal{L}_1, \mathcal{L}_2)$ is valid for a PMAS if it holds the following:

- Each blockchain $\mathcal{B}_i^{(j)}$ in either $\mathcal{L}_1$ or $\mathcal{L}_2$ for $j \in [1, 2]$ supports escape transactions.$\square$

Definition 4.1.2. *Privacy-preserving Multi-chain Atomic Swap (PMAS).* A privacy-preserving multi-chain atomic swap is a probabilistic polynomial-time interactive protocol run between two parties $\mathcal{P}_1$ and $\mathcal{P}_2$ without any trusted third party with assets in list of blockchains $\mathcal{L}_1$ and $\mathcal{L}_2$ respectively which holds the following properties:

1. **Effectiveness.** A PMAS is effective if, after successful termination of the protocol for parties $(\mathcal{P}_1, \mathcal{P}_2)$ with valid list pairs $(\mathcal{L}_1, \mathcal{L}_2)$, every asset locked in joint accounts in blockchains $\mathcal{B}_i^{(1)}$ in $\mathcal{L}_1$ is under the ownership of $\mathcal{P}_2$ and those in $\mathcal{L}_2$ is under the ownership of $\mathcal{P}_1$.
2. **Termination.** For a party who follows the protocol, PMAS always terminates within a reasonable time with either a $success$ or $abort$ state.
3. **Fairness.** A PMAS is fair if either party does not behave according to the protocol, then an honest party $\mathcal{P}_j$ either retains ownership of all assets in blockchains $\mathcal{B}_i^{(j)}$ in $\mathcal{L}_j$ or gains ownership of all assets in blockchains $\mathcal{B}_i^{(3-j)}$ in $\mathcal{L}_{3-j}$.
4. **Privacy.** An PMAS is privacy-preserving if the following holds:

   - **Indistinguishability** Transactions created during execution of PMAS are indistinguishable from majority of transactions in the blockchain.

   - **Unlinkability.** Transactions created during execution of PMAS is unlinkable to a global passive observer $\mathcal{O}$ if $\mathcal{O}$ cannot guess the link between $T^{(C)}_{\mathcal{B}_i^{(1)}}$ for blockchains $\mathcal{B}_i^{(1)}$ in $\mathcal{L}_1$ and $T^{(C)}_{\mathcal{B}_i^{(2)}}$ for blockchains $\mathcal{B}_i^{(2)}$ in $\mathcal{L}_2$ with probability greater than $\mathsf{negl}(.)$. $\square$

4.1.3 Blockchain Model

Accounts in blockchains are defined with a public-key pair (pk, sk) where the public key pk corresponds to the address of the account while the private key sk functions as the key with which you can spend assets credited to the account by producing a valid signature on a transaction. By assets we mean native tokens issued in the blockchain.

Transactions in a blockchain $\mathcal{B}$ are denoted by $T_{\mathcal{B}}$ and defined by a tuple $(pk_1, pk_2, [t])$ representing a transaction spending from pk_1 to pk_2 on blockchain $\mathcal{B}$, and the transaction may be optionally locked for a time duration t. This time duration may be implemented differently in different blockchains. In Bitcoin, block height is used to emulate time duration. We refrain from denoting the payment value in the transaction tuple as it makes referring to them cumbersome and is of no concern to us assuming that the parties can verify the agreed value upon receiving the transaction in intermediate phases. Unless otherwise specified, a transaction is unsigned–meaning it cannot be redeemed by publishing to the blockchain. A transaction T is redeemable in the blockchain when coupled with its corresponding valid signature, denoted by a tuple (T, σ) where T is the transaction description and σ is a valid signature on T. We do not consider other complex spending conditions as our protocol is based on this form of basic spending condition available in most blockchains. For example, in Bitcoin, this is a simple Pay-to-Public Key Hash (P2PKH) transaction.

Joint accounts are accounts whose asymmetric key pair (corresponding to accounts and keys) is generated by parties using off-chain distributed key generation protocol, and as such, requires distributed signing to generate valid signatures. These accounts are indistinguishable from other accounts on the blockchain. Since they are joint accounts, they can function as escrow to hold assets in the intermediate phases of the protocol. Some blockchains provide this functionality through “multi-signature” addresses. However, we do not use such functionality as doing so may reduce the size of indistinguishability set for transactions. For example, in Bitcoin, a multi-signature address has prefix 3 instead of 1 for a general address.

Escape Transactions. We say a blockchain supports escape transactions if it has native support for creating transactions that spend from accounts not yet present in the system but will be present after expiration of some time duration. Most blockchains that support time-locks also support escape transactions. Monero is an exception to this property.

4.2 PolySwap Overview

PolySwap is a two-party protocol run by parties willing to exchange assets they own in a set of blockchains at once without a trusted third party. The protocol is executed by parties each with a set of blockchains, whose signature scheme is reducible to a secret sharing signature (SSSig), where at least one set supports escape transactions. Figure 4.1 shows the overview of the proposed solution.

Alice and Bob owning assets in Blockchain 1 and Blockchain 2 respectively want to exchange assets. First, both parties jointly create distributed public keys used as joint accounts on each blockchain (AB_1 & AB_2) using instances of SSSig for the signature algorithm used in the blockchain. SSSig enables distributed signing on messages with private outputs of unlocking secrets for each party along with a common partial signature. The assets in joint accounts can only be spent by producing a complete signature which requires unlocking secrets from both parties, thus functioning as an escrow. The SSSig scheme provides indistinguishability of atomic swap transactions from majority of transactions in the same blockchain as these joint accounts are derived from standard public keys

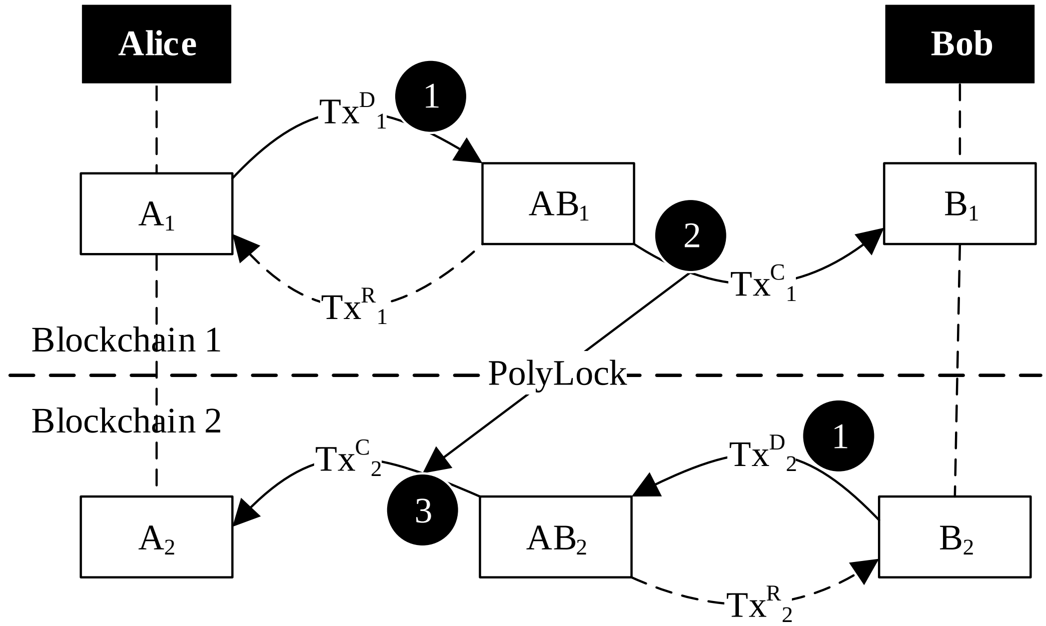


Figure 4.1: Overview of PolySwap

and the complete signature computed jointly is verifiable by standard verification algorithm for a given signature scheme. SSSig is described in details in Section 4.3.1 where we also present constructions for ECDSA, Schnorr and Cryptonote signature which are most widely used in cryptocurrecy systems.

Next, each party creates time-locked refund transactions from the joint accounts denoted by Tx_1^R and Tx_2^R. This is to prevent the loss of assets in case of malicious behaviour of parties where a party terminates the protocol prematurely (before completion). Prior solutions solved this problem by having time-locked transaction paying back to the owner in case a time window expires within which the atomic swap should have successfully terminated. We follow a similar approach for blockchains that support escape transactions. But for a set of blockchains without support for escape transactions, we solve this problem by releasing a private key share of the party for the joint accounts in such blockchains if any escape transactions for the other set of blockchains is published using the polynomial locking scheme. This scheme is described in details in Section 4.3.3.

Then, each party posts deposit transactions Tx_1^D and Tx_2^D paying to joint accounts. To transfer assets from a joint account to the other party, claim transactions Tx_1^C paying to Bob in Blockchain 1 and Tx_2^C paying to Alice in Blockchain 2 are created. These transactions are signed using SSSig with unlocking secrets for each transaction as output to each party. Since, no party has ability to complete the signature themselves, they cannot post these transactions to the blockchain to acquire assets in joint accounts. In order to enable each party to complete these signatures atomically, we introduce a polynomial locking scheme (PolyLock) for linking and locking unlocking secrets for signatures. The scheme binds together partial signatures for different transactions in the sets of blockchains. Using this polynomial scheme breaks the link between transactions involved in the swap operation as seen in prior HTLC-based atomic swap solutions. This scheme is a key component to enabling multi-chain atomic swap as incorporating multiple partial signatures is trivial by using higher order polynomials. PolyLock is described in Section 4.3.2.

Finally, Alice creates a PolyLock locking her unlocking secrets for claim transactions Tx_1^C and Tx_2^C. The lock is sent to Bob who upon verification sends back his unlocking secret for Alice's claim transaction, Tx_1^C. With this unlocking secret, Alice can complete the signature and post her claim transaction Tx_1^C to the blockchain. After which Bob can extract the full signature from the claim transaction to recover Alice's unlocking secret used to release the PolyLock which gives him the unlocking secret for his claim transaction Tx_2^C which he can post to the respective blockchain. This completes the atomic swap as Alice owns Bob's asset in Blockchain 1 and Bob owns Alice's asset in Blockchain 2, after the confirmation of claim transactions.

4.3 Building Blocks

In this section, we describe the components: *secret sharing signature scheme*, *polynomial locking scheme* and *contingency protocol* which are required as building blocks for PolySwap.

4.3.1 Secret Sharing Signature (SSSig) Scheme

In any cryptocurrency system, cryptographic signatures are required to authorize and verify the ability of a user to spend assets in the system. Recognizing this, we introduce a cryptographic primitive called *Secret Sharing Signatures (SSSig)*. Each of the two parties involved in the creation of a signature work together to partially sign a message, producing a partitioned signature. The partitioned signature consists of the public part (partial signature $\bar{\sigma}$), Party 1's private part (unlocking secret ϕ_1 or a), and Party 2's private part (ϕ_2 or b). The three parts are combined to produce a full signature σ that can be verified using the standard verification algorithm of the signature scheme. Additionally, given a full signature σ, and one of the unlocking secrets ϕ_i, the other unlocking secret ϕ_{3-i} can be computed. These secrets can be chained together to allow the release of one signature to complete another signature or release a key to other valuable information. The following is the formal definition of an SSSig scheme.

Definition 4.3.1. A Secret Sharing Signature (SSSig) scheme:

$$\mathcal{S} = (\mathsf{KeyGen}, \mathsf{PSign}, \mathsf{Complete}, \mathsf{Reveal}, \mathsf{Verify})$$

is run by parties $\mathcal{P}_1$ and $\mathcal{P}_2$ and consists of following algorithms and protocols:

$\{(\alpha, pk), (\beta, pk)\} \leftarrow \langle \mathsf{KeyGen}_{\mathcal{P}_1}(1^n), \mathsf{KeyGen}_{\mathcal{P}_2}(1^n)\rangle$: On input of security parameter 1^n, the key generation protocol returns a shared public key pk and secret keys α & β to $\mathcal{P}_1$ & $\mathcal{P}_1$ respectively.

$\{(\phi_1, \Phi_1, \Phi_2, \bar{\sigma}), (\phi_2, \Phi_2, \Phi_1, \bar{\sigma})\} \leftarrow \langle \mathsf{PSign}_{\mathcal{P}_1}(\alpha, pk, m), \mathsf{PSign}_{\mathcal{P}_2}(\beta, pk, m)\rangle$: On input of respective secret keys α & β, public key pk and message m, the partial signing protocol returns unlocking secrets for signatures on message m as ϕ_1 and ϕ_2 along with their public forms Φ_1 and Φ_2 to $\mathcal{P}_1$ & $\mathcal{P}_1$ respectively and $\bar{\sigma}$ as partial signature to both parties.

$\sigma \leftarrow \mathsf{Complete}_{\mathcal{P}_j}(\phi_1, \phi_2, \bar{\sigma})$: On input of the unlocking secrets ϕ_1 & ϕ_2 and partial signature $\bar{\sigma}$, the complete algorithm produces a full signature σ on message m.

$\phi_{3-j} \leftarrow \mathsf{Reveal}_{\mathcal{P}_j}(\sigma, \phi_j)$: On input of the full signature σ and a unlocking secret ϕ_j, the reveal algorithm produces the other unlocking secret ϕ_{3-j} that completes the signature.

$\{0, 1\} \leftarrow \mathsf{Verify}_{\mathcal{P}_j}(\sigma, pk, m)$: On input of a signature σ on message m for public key pk, the verification algorithm returns 1 for accept and 0 for reject.

Correctness. A SSSig scheme is correct if the verification algorithm Verify always accepts a signature generated by the complete algorithm Complete. $\square$

Our intuition for constructing an SSSig from a standard signature scheme is as follows. We start with a two-party signing scheme which is executed until the last communication such that the completion of the signature is protected by a hard problem for each party. Then, each party takes a commitment of their last communication and proves to the other party that it completes the signature. At this point, we have a perspective SSSig. We now verify that this new protocol meets the requirements of an SSSig. If not, then more signature-specific alteration may be required. For example, ECDSA requires an early value within the signature to be altered.

We present SSSig constructions for the following common schemes: ECDSA, Cryptonote Ring Signature, and Schnorr Signature.

ECDSA-based SSSig Construction

ECDSA-based SSSig construction is based on the ideas of Fast Secure Two-Party ECDSA Signing by Lindell [30] and the modifications by Malavolta *et. al.* [21] to realize AMHLs. Our construction is presented in Protocol 4.1.

Let $\mathbb{G}$ be an elliptic curve group of order q with base point G and $\mathcal{H} : \{0,1\}^* \to \{0,1\}^{|q|}$ be a hash function. Two parties $(\mathcal{P}_1, \mathcal{P}_2)$ generate a shared public key $Q = (x_1 \cdot x_2) \cdot G$ where x_1 is $\mathcal{P}_1$'s share of secret key and x_2 is $\mathcal{P}_2$'s share of secret key. The distributed key is generated by following the key generation protocol presented by Lindell [30] which generates the public key $Q = (x_1 \cdot x_2) \cdot G$ along with a Paillier key-pair (sk_{he}, pk_{he}) owned by $\mathcal{P}_1$.

For the Paillier homomorphic encryption scheme, we denote the encryption function as Enc and decryption function as Dec. $\mathcal{P}_1$ encrypts their secret key as $c_{key} = \mathsf{Enc}(x_1)$ which is used by $\mathcal{P}_2$ to help $\mathcal{P}_1$ produce their half of the signature without revealing either party's key. Recall that multiplication of outputs from Enc denotes homomorphic addition and exponentiating outputs from Enc results a scalar multiplication of the plaintext.

The signing protocol works similar to that presented by Lindell [30] where the secret key is the multiplicative share of x and randomness is multiplicative share of k such that $x = x_1 \cdot x_2$ and $k = k_1 \cdot k_2$ to compute signature $(r, s) = (r, k^{-1} \cdot (\mathcal{H}(m) + r \cdot x))$ where $R = (r_x, r_y) = k \cdot G$ is randomness and $r = r_x$. We modify this signature by having $\mathcal{P}_2$ select a random $k_3 \leftarrow_{\$} \mathbb{Z}_q$ and include it in multiplicative share of k such that $k = k_1 \cdot k_2 \cdot k_3$. Then, the shared signature is computed without k_3 which will serve as the unlocking secret for $\mathcal{P}_2$ and since k_3 was omitted, the shared signature $(r, s'') = (r, k_1^{-1} \cdot k_2^{-1}(x_1 \cdot x_2 \cdot r + \mathcal{H}(m))$ received by $\mathcal{P}_1$ is an incomplete signature on m. For the sake of convenience in later proofs, $(s'')^{-1}$ is used as the unlocking secret for $\mathcal{P}_1$ and k_3 is the unlocking secret for $\mathcal{P}_1$.

$\mathsf{KeyGen}_{\mathcal{P}_1}(1^n)$	$\mathsf{KeyGen}_{\mathcal{P}_2}(1^n)$	$\mathsf{Complete}_{\mathcal{P}_j}(\phi_1, \phi_2, \bar{\sigma})$	$\mathsf{Reveal}_{\mathcal{P}_j}(\sigma, \phi_j, \Phi_{3-j})$
Both parties execute KeyGen by Lindell [30]		Parse $\bar{\sigma}$ as r	Parse σ as (r, s)
$Q = x_1 \cdot x_2 \cdot G, c_{key} = \mathsf{Enc}_{\mathsf{pk_{he}}}(x_1)$	$Q = x_1 \cdot x_2 \cdot G$	$s := (\phi_1 \cdot \phi_2)^{-1}$	$\phi_{3-j} := (s \cdot \phi_j)^{-1}$
$\alpha = \{x_1, sk_{he}\}, pk = \{Q, c_{key}, pk_{he}\}$	$\beta = \{x_2\}, pk = \{Q, c_{key}, pk_{he}\}$	$\sigma := (r, s)$	If $\phi_{3-j} \cdot G \neq \Phi_{3-j}$, $\phi_{3-j} := (-s \cdot \phi_j)^{-1}$
return (α, pk)	return (β, pk)	return σ	return ϕ_{3-j}

$\mathsf{PSign}_{\mathcal{P}_1}(\alpha, pk, m)$		$\mathsf{PSign}_{\mathcal{P}_2}(\beta, pk, m)$
Parse pk as $\{Q, c_{key}, pk_{he}\}$ and α as $\{x_1, sk_{he}\}$		Parse pk as $\{Q, c_{key}, pk_{he}\}$ and β as $\{x_2\}$
$k_1 \leftarrow_\$ \mathbb{Z}_q; R_1 := k_1 \cdot G; e = \mathcal{H}(m)$		$k_3 \leftarrow_\$ \mathbb{Z}_q; e = \mathcal{H}(m)$
$\{[\![\pi_1]\!], \pi_1, \kappa_1\} = \mathsf{P}^{\mathsf{DL}}_{\mathsf{com\text{-}zk}}(\{R_1\}, \{k_1\})$	$\xrightarrow{[\![\pi_1]\!]}$	$k_2 \leftarrow_\$ \mathbb{Z}_q; R_2 := k_2 \cdot k_3 \cdot G$
If $\mathsf{V}^{\mathsf{DL}}_{\mathsf{zk}}(\pi_2) \neq 1$, then abort	$\xleftarrow{R_2, \pi_2}$	$\pi_2 = \mathsf{P}^{\mathsf{DL}}_{\mathsf{zk}}(\{R_2\}, \{k_2 \cdot k_3\})$
	$\xrightarrow{R_1, \pi_1, \kappa_1}$	If $\mathsf{V}^{\mathsf{DL}}_{\mathsf{com\text{-}zk}}([\![\pi_1]\!], \pi_1, \kappa_1) \neq 1$, then abort
$R = k_1 \cdot R_2 = k_1 \cdot k_2 \cdot k_3 \cdot G$		$R = k_2 \cdot k_3 \cdot R_1 = k_1 \cdot k_2 \cdot k_3 \cdot G$
$(r_x, r_y) := R; r = r_x \mod q$		$(r_x, r_y) := R; r = r_x \mod q; \rho \leftarrow_\$ \mathbb{Z}_{q^2}$
		$c_1 = \mathsf{Enc}_{\mathsf{pk_{he}}}(\rho \cdot q + k_2^{-1} \cdot e); v = k_2^{-1} \cdot r \cdot x_2 \mod q$
		$c_2 = (c_{key})^v = \mathsf{Enc}_{\mathsf{pk_{he}}}(k_2^{-1} \cdot r \cdot x_1 \cdot x_2)$
$s' := \mathsf{Dec}_{\mathsf{sk_{he}}}(c')$	$\xleftarrow{c'}$	$c' = c_1 \cdot c_2 = \mathsf{Enc}_{\mathsf{pk_{he}}}(k_2^{-1}(x_1 \cdot x_2 \cdot r + e) + \rho \cdot q)$
$s'' := k_1^{-1} \cdot s' \mod q; \phi_1 = (s'')^{-1}$		$\phi_2 := k_3$
$\Phi_1 = \phi_1 \cdot G; Q_1 = \phi_1 \cdot Q$		$\Phi_2 = k_3 \cdot G; Q_2 = k_3 \cdot Q$
$\{[\![\pi_3]\!], \pi_3, \kappa_3\} = \mathsf{P}^{\mathsf{DH}}_{\mathsf{com\text{-}zk}}(\{\Phi_1, Q_1\}, \{\phi_1\})$	$\xrightarrow{[\![\pi_3]\!]}$	$\pi_4 = \mathsf{P}^{\mathsf{DH}}_{\mathsf{zk}}(\{\Phi_2, Q_2\}, \{k_3\})$
If $\mathsf{V}^{\mathsf{DH}}_{\mathsf{zk}}(\pi_4) \neq 1$, then abort	$\xleftarrow{\pi_4, \Phi_2, Q_2}$	If $\mathsf{V}^{\mathsf{DH}}_{\mathsf{com\text{-}zk}}(\pi_3) \neq 1$, then abort
$(x, y) = e \cdot ((s'')^{-1} \cdot \Phi_2) + r \cdot ((s'')^{-1} \cdot Q_2)$	$\xrightarrow{\pi_3, \kappa_3, \Phi_1, Q_1}$	$(x, y) = e \cdot (k_3 \cdot \Phi_1) + r \cdot (k_3 \cdot Q_1)$
If $x \neq r$, then abort		If $x \neq r$, then abort
$\bar{\sigma} := \{r\}$		$\bar{\sigma} := \{r\}$
return $(\phi_1, \Phi_1, \Phi_2, \bar{\sigma})$		return $(\phi_2, \Phi_2, \Phi_1, \bar{\sigma})$

Protocol 4.1: ECDSA-based SSSig Construction

Given the other party's unlocking secret, each party can trivially compute the full valid signature as $(r, s'' \cdot k_3^{-1}) = (r, k_3^{-1} \cdot k_1^{-1} \cdot k_2^{-1}(x_1 \cdot x_2 \cdot r + \mathcal{H}(m)))$. Additionally, given a valid signature (r, s) and $\mathcal{P}_j$'s unlocking secret ϕ_j, $\mathcal{P}_{3-j}$'s unlocking secret ϕ_{3-j} can be trivially calculated. However, due to the nature of ECDSA signature, both (r, s) and $(r, -s)$ are valid signatures. We perform a simple check with public forms of unlocking secrets, $\phi_{3-j} \cdot G \stackrel{?}{=} \Phi_{3-j}$, to ensure correct values are returned by the Reveal protocol.

Cryptonote-based SSSig Construction

Let $\mathbb{G}$ be the twisted Edwards curve Ed25519. Ed25519 is a group over a finite field $\mathbb{F}_q$, where $q = 2^{255} - 19$, with a base point G of prime order l. The equation for this curve E is defined as $-x^2 + y^2 = 1 - \frac{121665}{121666} \cdot x^2 y^2 \mod q$, $\mathcal{H}_s : \{0, 1\}^* \rightarrow \mathbb{F}_q$ be a cryptographic hash function and $\mathcal{H}_p : E(\mathbb{F}_q) \rightarrow E(\mathbb{F}_q)$ be a deterministic hash function. The construction is presented in Protocol 4.2.

In the key generation step, the two parties $(\mathcal{P}_1, \mathcal{P}_2)$ execute a distributed key generation protocol in the malicious model to generate a distributed spend key $B := (b_1 + b_2) \cdot G$ with b_1 as $\mathcal{P}_1$'s secret and b_2 as $\mathcal{P}_2$'s secret. Also, a view key $A = a \cdot G$ is created by a party with the discreet log a known to both parties so that each party can independently check to recognize their transactions. However, in order to spend their output, knowledge of either the full spend key or the full signature is required.

Since the signature scheme is a ring signature, other ring members are simulated as we are only concerned with our part of the signature which we assume is indexed at s in the ring for each ring members $i \in [0, n]$, which is released by our signing scheme. This can be achieved by having parties divide the one-time private key $x = x_1 + x_2$ where $x_1 = b_1$ for $\mathcal{P}_1$ and $x_2 = b_2 + \mathcal{H}_s(aR)$ for $\mathcal{P}_2$.

$\underline{\mathsf{KeyGen}_{\mathcal{P}_1}(1^n)}$
Spend key, $B := (b_1 + b_2) \cdot G$
$a \leftarrow\$ \mathbb{Z}_q$
View key, $A = a \cdot G$ $\xrightarrow{a}$
$\alpha = \{a, b_1\}; pk = \{A, B\}$
return (α, pk)

$\underline{\mathsf{KeyGen}_{\mathcal{P}_2}(1^n)}$
Spend key, $B := (b_1 + b_2) \cdot G$
$A = a \cdot G$
$\beta = \{a, b_2\}; pk = \{A, B\}$
return (β, pk)

$\underline{\mathsf{Complete}_{\mathcal{P}_j}(\phi_1, \phi_2, \bar{\sigma})}$
Parse $\bar{\sigma}$ as $\{I, c_i, r_i \mid i \in [0, n], i \neq s\}$
$r_s = \phi_1 + \phi_2 = r_{s,1} + r_{s,2}$
$\sigma = \{I, c_i, r_i \mid i \in [0, n]\}$
return σ

$\underline{\mathsf{Reveal}_{\mathcal{P}_j}(\sigma, \phi_j)}$
Parse σ as $\{I, c_i, r_i \mid i \in [0, n]\}$
Extract r_s from σ
$\phi_{3-j} = r_s - \phi_j$
return ϕ_{3-j}

$\underline{\mathsf{PSign}_{\mathcal{P}_1}(\alpha, pk, m)}$
Parse pk as $\{A, B\}$ and α as $\{a, b_1\}$

$\underline{\mathsf{PSign}_{\mathcal{P}_2}(\beta, pk, m)}$
Parse pk as $\{A, B\}$ and β as $\{a, b_2\}$

Execute Cryptonote ring protocol and simulate other ring members.
Creates all standard ring signature components except the true signature indexed at s.
For each other ring member i, q_i, w_i are generated and are held by $\mathcal{P}_1$ and $\mathcal{P}_2$

$\mathcal{P}_1$:
$x_1 = b_1; I_1 = x_1 \mathcal{H}_p(P_s); q_{s,1} \leftarrow\$ \mathbb{Z}_q$
$L_{s,1} = q_{s,1}G; R_{s,1} = q_{s,1} \cdot \mathcal{H}_p(P_s)$
$P_{s,1} = x_1 \cdot G; P_{s,2} = P_s - P_{s,1}$
$\{[\![\pi_1]\!], \pi_1, \kappa_1\} = \mathsf{P}^{\mathsf{DH}}_{\mathsf{com\text{-}zk}}(\{P_{s,1}, I_1\}, \{x_1\})$
$\{[\![\pi_2]\!], \pi_2, \kappa_2\} = \mathsf{P}^{\mathsf{DH}}_{\mathsf{com\text{-}zk}}(\{L_{s,1}, R_{s,1}\}, \{q_{s,1}\})$

$\mathcal{P}_2$:
$x_2 = b_2 + \mathcal{H}_s(aR); I_2 = x_2 \mathcal{H}_p(P_s); q_{s,2} \leftarrow\$ \mathbb{Z}_q$
$L_{s,2} = q_{s,2}G; R_{s,2} = q_{s,2} \cdot \mathcal{H}_p(P_s)$
$P_{s,2} = x_2 \cdot G; P_{s,1} = P_s - P_{s,2}$
$\pi_3 = \mathsf{P}^{\mathsf{DH}}_{\mathsf{zk}}(\{P_{s,2}, I_2\}, \{x_2\})$
$\pi_4 = \mathsf{P}^{\mathsf{DH}}_{\mathsf{zk}}(\{L_{s,2}, R_{s,2}\}, \{q_{s,2}\})$

$\xrightarrow{[\![\pi_1]\!], [\![\pi_2]\!]}$
$\xleftarrow{P_{s,2}, I_2, L_{s,2}, R_{s,2}, \pi_3, \pi_4}$

$\mathcal{P}_1$:
If $\mathsf{V}^{\mathsf{DH}}_{\mathsf{zk}}(\pi_3) \vee \mathsf{V}^{\mathsf{DH}}_{\mathsf{zk}}(\pi_4) \neq 1$, then abort

$\xrightarrow{P_{s,1}, I_1, L_{s,1}, R_{s,1}, \pi_1, \pi_2, \kappa_1, \kappa_2}$

$\mathcal{P}_2$:
If $\mathsf{V}^{\mathsf{DH}}_{\mathsf{com\text{-}zk}}([\![\pi_1]\!], \pi_1, \kappa_1) \vee \mathsf{V}^{\mathsf{DH}}_{\mathsf{com\text{-}zk}}[\![\pi_2]\!], \pi_2, \kappa_2) \neq 1$, abort

$\mathcal{P}_1$:
$I = I_1 + I_2, ; L_s = L_{s,1} + L_{s,2}$
$R_s = R_{s,1} + R_{s,2}$
$c = \mathcal{H}_s(m, L_1, L_2, ...L_n, R_1, R_2, ...R_n)$
$c_s = c - \sum_{i=1}^{n} w_i; r_{s,1} = q_{s,1} - c_s \cdot x_1$
$L'_{s,1} = r_{s,1} \cdot G, R'_{s,1} = r_{s,1} \cdot \mathcal{H}_p(P_s)$
$\{[\![\pi_5]\!], \pi_5, \kappa_5\} = \mathsf{P}^{\mathsf{DH}}_{\mathsf{com\text{-}zk}}(\{L'_{s,1}, R'_{s,1}\}, \{r_{s,1}\})$

$\mathcal{P}_2$:
$I = I_1 + I_2; L_s = L_{s,1} + L_{s,2}$
$R_s = R_{s,1} + R_{s,2}$
$c = \mathcal{H}_s(m, L_1, L_2, ...L_n, R_1, R_2, ...R_n)$
$c_s = c - \sum_{i=1}^{n} w_i; r_{s,2} = q_{s,2} - c_s \cdot x_2$
$L'_{s,2} = r_{s,2} \cdot G, R'_{s,2} = r_{s,2} \cdot \mathcal{H}_p(P_s)$
$\pi_6 = \mathsf{P}^{\mathsf{DH}}_{\mathsf{zk}}(\{L'_{s,2}, R'_{s,2}\}, \{r_{s,2}\})$

$\xrightarrow{[\![\pi_5]\!]}$
$\xleftarrow{L'_{s,2}, R'_{s,2}, \pi_6}$

$\mathcal{P}_1$:
If $\mathsf{V}^{\mathsf{DH}}_{\mathsf{zk}}(\pi_6) \neq 1$, then abort

$\xrightarrow{L'_{s,1}, R'_{s,1}, \pi_5, \kappa_5,}$

$\mathcal{P}_2$:
If $\mathsf{V}^{\mathsf{DH}}_{\mathsf{com\text{-}zk}}([\![\pi_5]\!], \pi_5, \kappa_5) \neq 1$, then abort

$\mathcal{P}_1$:
Verify $L_s = L'_{s,1} + L'_{s,2} + c_s P_s$
Verify $R_s = R'_{s,1} + R'_{s,2} + c_s I$
$\phi_1 := r_{s,1}; \Phi_1 = L'_{s,1}; \Phi_2 = L'_{s,2}$
$\bar{\sigma} := \{I, c_i, r_i \mid i \in [0, n], i \neq s\}$
return $(\phi_1, \Phi_1, \Phi_2, \bar{\sigma})$

$\mathcal{P}_2$:
Verify $L_s = L'_{s,1} + L'_{s,2} + c_s P_s$
Verify $R_s = R'_{s,1} + R'_{s,2} + c_s I$
$\phi_2 := r_{s,2}; \Phi_1 = L'_{s,1}; \Phi_2 = L'_{s,2}$
$\bar{\sigma} := \{I, c_i, r_i \mid i \in [0, n], i \neq s\}$
return $(\phi_2, \Phi_2, \Phi_1, \bar{\sigma})$

Protocol 4.2: Cryptonote-based SSSig Construction

Recall that the one-time private key for producing ring signatures in Cryptonote protocol is calculated as $x = b + \mathcal{H}_s(aR)$ where R is the randomness encoded into the transaction. The corresponding public key is $P_s = x \cdot G$ and is calculated as $P_s = \mathcal{H}_s(aR)G + B$. Similarly, parties jointly calculate intermediate values $L_s = (q_{s,1}+q_{s,2})\cdot G$ and $R_s = (q_{s,1}+q_{s,2})\cdot \mathcal{H}_p(P_s)$ where $q_{s,1}$ & $q_{s,2}$ is randomly chosen by $\mathcal{P}_1$ & $\mathcal{P}_2$ respectively. Next, both parties calculate the non-interactive challenge as $c = \mathcal{H}_s(m, L_0, .., L_s, .., L_n, R_0, .., R_s, .., R_n)$, where all values except m, L_s, R_s are simulated values calculated with $\{q_i, w_i, P_i \mid i \in [0, n], i \neq s\}$. Next, each party calculates part of the the response $c_s = c - \sum_{i=1}^{n} w_i$. With this value, $\mathcal{P}_1$ calculates the other part of the response $r_{s,1} = q_{s,1} - c_s \cdot x_1$ and $\mathcal{P}_2$ calculates $r_{s,2} = q_{s,1} - c_s \cdot x_2$ as their unlocking secrets. The verification of each intermediate value is done using public forms of secrets and appropriate zero knowledge proofs. The value $\bar{\sigma} := \{I, c_i, r_i \mid i \in [0, n], i \neq s\}$ is output as partial signature to both parties. The signature can then be completed trivially by calculating $r_s = r_{s,1} + r_{s,2}$ once a party has the other party's unlocking secret (to include it in the partial signature).

Schnorr-based SSSig Construction

Schnorr-based construction for SSSig is comparatively simpler due to the linear structure of the signature. The construction is presented in Protocol 4.3.

Let $\mathbb{G}$ be an elliptic curve group of order q with base point G and $\mathcal{H} : \{0,1\}^* \to \{0,1\}^{|q|}$ be a hash function. Two parties $\mathcal{P}_1$ and $\mathcal{P}_2$ generate a shared public key $Q = (x_1 + x_2) \cdot G$ with $\mathcal{P}_1$'s private key share x_1 and $\mathcal{P}_2$'s private key share x_2. To generate partial signatures on a message, each party $\mathcal{P}_1$ and $\mathcal{P}_2$ selects $k_1 \leftarrow_\$ \mathbb{Z}_q$ & $k_2 \leftarrow_\$ \mathbb{Z}_q$ respectively to compute $R_1 = k_1 \cdot G$ & $R_2 = k_2 \cdot G$. Next, the two parties compute common randomness $R = R_1 + R_2$ which is used by each party to locally compute $e = H(Q||R||m)$.

$\mathsf{KeyGen}_{\mathcal{P}_1}(1^n)$		$\mathsf{KeyGen}_{\mathcal{P}_2}(1^n)$	$\mathsf{Complete}_{\mathcal{P}_j}(\phi_1, \phi_2, \bar{\sigma})$	$\mathsf{Reveal}_{\mathcal{P}_j}(\sigma, \phi_j)$
$x_1 \leftarrow_\$ \mathbb{Z}_q; pk_1 = x_1 \cdot G$		$x_2 \leftarrow_\$ \mathbb{Z}_q; pk_2 = x_2 \cdot G$	Parse $\bar{\sigma}$ as $\{R\}$	Parse σ as (R, s)
$\{[\![\pi_1]\!], \pi_1, \kappa_1\} = \mathsf{P}^{\mathsf{DL}}_{\mathsf{com\text{-}zk}}(\{pk_1\}, \{x_1\})$	$\xrightarrow{[\![\pi_1]\!]}$	$\pi_2 = \mathsf{P}^{\mathsf{DL}}_{\mathsf{zk}}(\{pk_2\}, \{x_2\})$	$s = \phi_1 + \phi_2$	$\phi_{3-j} = s - \phi_j$
If $\mathsf{V}^{\mathsf{DL}}_{\mathsf{zk}}(\pi_2) \neq 1$, then abort	$\xleftarrow{pk_2, \pi_2}$		Verify (R, s) for message m	
	$\xrightarrow{pk_1, \pi_1, \kappa_1}$	If $\mathsf{V}^{\mathsf{DL}}_{\mathsf{com\text{-}zk}}([\![\pi_1]\!], \pi_1, \kappa_1) \neq 1$, then abort	$\sigma := (R, s)$	
$Q := pk_1 + pk_2 = (x_1 + x_2) \cdot G$		$Q := pk_1 + pk_2 = (x_1 + x_2)$	return σ	return ϕ_{3-j}
$\alpha := \{x_1\}; pk := \{Q, pk_1, pk_2\}$		$\beta := \{x_2\}; pk := \{Q, pk_2, pk_1\}$		
return (α, pk)		return (β, pk)		

$\mathsf{PSign}_{\mathcal{P}_1}(\alpha, pk, m)$		$\mathsf{PSign}_{\mathcal{P}_2}(\beta, pk, m)$
Parse pk as $\{Q\}$ and α as $\{x_1\}$		Parse pk as $\{Q\}$ and β as $\{x_2\}$
$k_1 \leftarrow_\$ \mathbb{Z}_q; R_1 := k_1 \cdot G$		$k_2 \leftarrow_\$ \mathbb{Z}_q; R_2 := k_2 \cdot G$
$\{[\![\pi_1]\!], \pi_1, \kappa_1\} := \mathsf{P}^{\mathsf{DL}}_{\mathsf{com\text{-}zk}}(\{R_1\}, \{k_1\})$	$\xrightarrow{[\![\pi_1]\!]}$	$\pi_2 = \mathsf{P}^{\mathsf{DL}}_{\mathsf{zk}}(\{R_2\}, \{k_2\})$
If $\mathsf{V}^{\mathsf{DL}}_{\mathsf{zk}}(\pi_2) \neq 1$, then abort	$\xleftarrow{R_2, \pi_2}$	
	$\xrightarrow{R_1, \pi_1, \kappa_1}$	If $\mathsf{V}^{\mathsf{DL}}_{\mathsf{com\text{-}zk}}([\![\pi_1]\!], \pi_1, \kappa_1) \neq 1$, then abort
$R := R_1 + R_2; e = H(Q\|\|R\|\|m)$		$R := R_1 + R_2; e = H(Q\|\|R\|\|m)$
$\phi_1 := s_1 = k_1 - x_1 \cdot e; \Phi_1 := \phi_1 \cdot G$		$\phi_2 := s_2 = k_2 - x_2 \cdot e; \Phi_2 := \phi_2 \cdot G$
$\{[\![\pi_3]\!], \pi_3, \kappa_3\} = \mathsf{P}^{\mathsf{DL}}_{\mathsf{com\text{-}zk}}(\{\Phi_1\}, \{\phi_1\})$	$\xrightarrow{[\![\pi_3]\!]}$	$\pi_4 = \mathsf{P}^{\mathsf{DL}}_{\mathsf{zk}}(\{\Phi_2\}, \{\phi_2\})$
If $\mathsf{V}^{\mathsf{DL}}_{\mathsf{zk}}(\pi_4) \neq 1$,then abort	$\xleftarrow{\Phi_2, \pi_4}$	
	$\xrightarrow{\Phi_1, \pi_3, \kappa_3}$	If $\mathsf{V}^{\mathsf{DL}}_{\mathsf{com\text{-}zk}}([\![\pi_3]\!], \pi_3, \kappa_3) \neq 1$, then abort
If $\Phi_1 + \Phi_2 \neq R - e \cdot Q$, then abort		If $\Phi_1 + \Phi_2 \stackrel{?}{=} R - e \cdot Q$, then abort
$\bar{\sigma} := \{R\}; \phi_1 := s_1$		$\bar{\sigma} := \{R\}; \phi_2 := s_2$
return $(\phi_1, \Phi_1, \Phi_2, \bar{\sigma})$		return $(\phi_2, \Phi_2, \Phi_1, \bar{\sigma})$

Protocol 4.3: Schnorr-based SSSig Construction

Next, $\mathcal{P}_1$ calculates their unlocking secret as $s_1 = k_1 - x_1 \cdot e$ and $\mathcal{P}_2$ calculates their unlocking secret as $s_2 = k_2 - x_1 \cdot e$. To check the validity of these unlocking secrets, $\mathcal{P}_1$ and $\mathcal{P}_2$ exchange the public form of these secrets as $S_1 = s_1 \cdot G$ and $S_2 = s_2 \cdot G$ along with a proof of knowledge of discrete log and verify whether $S_1 + S_2 \stackrel{?}{=} R - e \cdot Q$. In order to compute the full signature, each party's unlocking secrets s_1 and s_2 can be added resulting in $s = (k_1 + k_2) - (x_1 + x_2) \cdot e$ which is a valid signature for the distributed secret key $x_1 + x_2$. (Note that a similar approach can be followed to create SSSig scheme for EdDSA since it is is a variant of Schnorr's signature scheme.)

4.3.2 Polynomial Locking

In this section, we present the polynomial locking scheme, which is used to link the unlocking secrets and eventually release the secrets at the same time. We modify Shamir Secret Sharing [26] to create a polynomial locking scheme. The scheme is comprised of three algorithms PolyLock, PolyVerify and PolyRelease, and is presented in Algorithms 1, 2 & 3 respectively.

The PolyLock algorithm links all the unlocking secrets to a polynomial. It takes as input the list of unlocking secrets L_ϕ, the list of public forms of the secrets L_Φ, and the list of the order of the groups of the secrets L_λ. L_ϕ is in the form of a tuple (i, ϕ_i). L_λ is comprised of order of groups used by the signature scheme in a cryptocurrency system. For example, in Bitcoin, it is the order of base point G in secp256k1 curve. In line 1, we set the degree of polynomial k to be one less than the size of L_λ, but it could be the size of any of the inputs as they are the same. All secrets are converted to the largest group order q in the list of group orders L_λ. If q is not a prime, then the next largest prime is selected. After the group conversions, we obtain $k + 1$ values, with which we create a polynomial $f(x)$ over $\mathbb{Z}_q$ of degree k:

Algorithm 1 PolyLock$(L_\phi, L_\Phi, L_\lambda)$

input : $L_\phi =$ list of secrets, $L_\Phi =$ list of public forms of secret, $L_\lambda =$ list of order of groups of secrets

output : $L_{\text{-}x} =$ list of points in the polynomial, $L_\pi =$ list of group conversion proofs, $\pi_x =$ proof for positive values of x, $\pi_{\text{-}x} =$ proof for negative values of x

1 **Let** $L_{[\![c]\!]}, L_{\text{-}x}, L_{\kappa_c}, L_{\kappa_\phi}, L_\pi, L_{[\![\phi]\!]}$ be empty lists.
2 $k = L_\lambda.\texttt{size} - 1$
3 $q = \texttt{max}(L_\lambda)$
4 **if** q is not prime, $q = \texttt{nextLargestPrime}(q)$
5 **Let** $\mathbb{G}$ be a discrete log group of order q
6 $f(x) = \texttt{solvePolynomial}(L_\phi, q)$
7 **Let** L_c be the list of coefficeints of $f(x)$
8 **for** $c_i \in L_c$ **do**
9     $\{[\![c_i]\!], \kappa_{c_i}\} = \mathsf{com}(c_i, \mathbb{G})$
10     $L_{[\![c]\!]}.\texttt{concat}([\![c_i]\!])$
11     $L_{\kappa_c}.\texttt{concat}(\kappa_{c_i})$
12 **end**
13 **for** $i \in \{-1... - k\}$ **do**
14     $L_{\text{-}x}.\texttt{concat}(i, f(i))$
15 **end**
16 **for** $i \in \{0, 1, ...k\}$ **do**
17     $\{[\![\phi_i]\!], \kappa_i\} = \mathsf{com}(L_\phi[i], \mathbb{G})$
18   **if** $L_\Phi[i]$ *exists* $\wedge L_\Phi[i] \notin \mathbb{G}$ **then**
19         $\pi_i = \mathsf{P}^{\mathsf{GC}}_{\mathsf{zk}}(\{L_\Phi[i], [\![\phi_i]\!]\}, \{L_\phi[i], \kappa_i\})$
20   **else**
21         $\pi_i = \texttt{true\_proof}$
22   **end**
23     $L_{\kappa_\phi}.\texttt{concat}(\kappa_i)$
24     $L_\pi.\texttt{concat}(\pi_i)$
25     $L_{[\![\phi]\!]}.\texttt{concat}([\![\phi_i]\!])$
26 **end**
27 $\pi_x = \mathsf{P}_{\mathsf{zk}}(\{L_{[\![c]\!]}, L_{[\![\phi]\!]}\}, \{L_{\kappa_c}, L_{\kappa_\phi}\})$
28 $\pi_{\text{-}x} = \mathsf{P}_{\mathsf{zk}}(\{L_{[\![c]\!]}, L_{\text{-}x}\}, \{L_{\kappa_c}\})$
29 **return** $(L_{\text{-}x}, L_\pi, \pi_x, \pi_{\text{-}x})$

$$f(x) = c_0 + c_1 x + c_2 x^2 + ... + c_k x^k \mod q$$

In line 6, we call the standard `solvePolynomial` function to obtain the polynomial $f(x)$ over $\mathbb{Z}_q$ with its coefficients in L_c. Next, the coefficients are committed to generate the proofs for the point values of the polynomial. In lines 13-15, we calculate the negative point values to be used as a partial key for the other party. In lines 16-26, we generate the proofs for correct group conversions. However, if all groups are same, then we simply set it to `true_proof` that always verifies successfully. The algorithm outputs a list of points L_{-x} on the polynomial as a tuple in point-value representation $(i, f(i))$, along with a list of proofs L_π relating to the group conversion of secrets, proof π_x relating to the correctness of positive point values hiding the secrets, and proof π_{-x} relating to the correctness of negative point values.

Algorithm 2 PolyVerify$(L_\pi, \pi_x, \pi_{-x}, L_\lambda)$

input : L_π = list of group conversion proofs, π_x = proof for positive values of x, π_{-x} = proof for negative values of x, L_λ = list of order of groups of secrets
output : 1 for accept or 0 for reject

$k = L_\lambda.\texttt{size} - 1$
$q = \texttt{max}(L_\lambda)$
if q is not prime, $q = \texttt{nextLargestPrime}(q)$
Select the smallest secure elliptic curve of with a cofactor of 1 and order q or greater. $q =$
for $i \in \{0, 1, ..., k\}$ **do**
    if $L_\lambda[i] \neq q$ **then**
        if $V_{zk}^{GC}(L_\pi[i]) \neq 1$ **then**
            return 0
        end
    else
        if $V_{zk}(L_\pi[i]) \neq 1$ **then**
            return 0
        end
    end
end
if $V_{zk}(\pi_x) \vee V_{zk}(\pi_{-x}) \neq 1$ **then**
    return 0
end
return 1

The PolyVerify algorithm verifies the proofs of well-formedness of the polynomial, i.e., the polynomial hides the desired unlocking secrets and the partial key is the key to the proper polynomial. It takes the list of proofs $L_\pi, \pi_x \& \pi_{\text{-}x}$ and the list of orders L_λ as input, and it outputs 1 for acceptance or 0 for the rejection of the proofs.

Algorithm 3 PolyRelease$(L_{\text{-}x}, j, \phi_j, L_\lambda)$

input : $L_{\text{-}x}$ = list of points in the polynomial, ϕ_j = secret at position j, j = position of secret in polynomial, L_λ = list of order of groups of secrets
output : L_ϕ = list of the original secrets

$k = L_\lambda.\texttt{size} - 1$
$q = \texttt{max}(L_\lambda)$
if q is not prime, $q = \texttt{nextLargestPrime}(q)$
Let L_ϕ be an empty list.
$f(x) = \texttt{solvePolynomial}(L_{\text{-}x}.\texttt{concat}(j, \phi_j), q)$
for $i \in \{0, 1, 2, ..., k\}$ **do**
    $L_\phi.\texttt{concat}(f(i)))$
end
return L_ϕ

The PolyRelease algorithm releases the unlocking secrets locked in the polynomial. It takes the list of points $L_{\text{-}x}$ along with a unique point (j, ϕ_j) and L_λ as input. Since we already have k point values in the polynomial, with an additional point solving the polynomial is trivial. It returns the original list of secrets L_ϕ as output.

In practice, a party runs the PolyLock algorithm to get the partial key $L_{\text{-}x}$ and the proofs. This partial key along with the proofs are sent to another party willing to get the unlocking secret who first verifies the proofs by running PolyVerify and after getting a new point in the polynomial runs the PolyRelease algorithm to get the original secrets linked and locked in the polynomial.

Since we use polynomials to link and lock a party's secret values, increasing the number of secrets locked by the polynomial is simply a matter of increasing the order of the polynomial used. This property is crucial to enabling many-to-many atomic swaps as secrets for claim transaction in different blockchains can be locked in a single polynomial lock.

4.3.3 Contingency Protocol

Contingency protocol ensures that assets of each party are recoverable in case either party decides to abort before termination or acts maliciously. It is dependent upon the actual cryptocurrency system used and the functionalities available. This protocol run between two parties each with sets of blockchains works for cases where at least one supports *escape transactions*. The contingency protocol is presented as follows.

If a blockchain supports escape transactions then, the contingency protocol is straight forward. Each party $\mathcal{P}_j$ creates escape transaction $T^E_{\mathcal{B}_i^{(j)}} = (pk_i^{(j)}, \mathcal{P}_j, t_j)$ for blockchains in lists $\mathcal{L}_j$ paying to their own address which is signed using distributed signing protocol to produce signature $\sigma^E_{\mathcal{B}_i^{(j)}}$ for $j \in \{1, 2\}$ and $i \in \{1, 2, \ldots, \mathcal{L}_j.\texttt{size}\}$. Recall that the account which funds the escape transactions are one time joint accounts created using two-party key generation protocol, thus requiring a distributed signing protocol to create valid signatures. At the end of the protocol, $\mathcal{P}_1$ gets $(T^E_{\mathcal{B}_i^{(1)}}, \sigma^E_{\mathcal{B}_i^{(1)}})$ locked for time t_1 in blockchains in list $\mathcal{L}_1$ while $\mathcal{P}_2$ gets $(T^E_{\mathcal{B}_i^{(2)}}, \sigma^E_{\mathcal{B}_i^{(2)}})$ locked for time t_2 in blockchains in list $\mathcal{L}_2$ such that $t_1 >> t_2$. In case of deviation from the protocol, each party can independently post the escape transaction to the blockchain to recover their funds.

However, in cryptocurrency systems that do not support time-locked escape transactions, the contingency protocol is more involved. We describe an alternative contingency protocol for cases where at least one of the blockchains in $\mathcal{L}_1$ does not support escape transactions and all blockchains in $\mathcal{L}_2$ support escape transactions. First, we select a prime order q to be the largest order from the blockchains in $\mathcal{L}_2$. If the largest order is not prime then, we select the next largest prime. The parties jointly agree on difficulties δ_1 and δ_2 for concealed time locks such that δ_1 is much greater than δ_2, e.g. δ_2 takes about 12 hours to solve and δ_1 take about a week. This is to ensure that $\mathcal{P}_1$ can not hold their escape transactions until $\mathcal{P}_1$ submits theirs and make the puzzle a race that $\mathcal{P}_2$ is capable of winning.

Concealed Time-locked Puzzle: $\mathsf{Conceal}(m, [\![m]\!], \kappa_m, \delta)$
For party $\mathcal{P}_1$ to create a concealed time-locked puzzle Π' hiding message m with difficulty δ and prove its correctness to $\mathcal{P}_2$, parties follow the following steps:

1. $\mathcal{P}_1$ creates a Paillier Key (pk_P, sk_P) using the method from Lindell's Keygen [30] with security parameter 1^n.
2. $\mathcal{P}_1$ sends pk_P to $\mathcal{P}_2$. $\mathcal{P}_1$ and $\mathcal{P}_2$ parse N from pk_P and chooses $g_P \leftarrow\$\, \mathbb{Z}_n$
3. $\mathcal{P}_1$ sends $h_P = g_P^{2^\delta}$ to $\mathcal{P}_2$, along with an accompanying proof [49]
4. $\mathcal{P}_1$ and $\mathcal{P}_2$ agree on an integer k for the cut-and-choose size
5. $\mathcal{P}_2$ chooses $j \leftarrow\$\, \mathbb{Z}_k$ and creates $\{[\![j]\!], \chi_j\} = \mathsf{com}(j)$
6. $\mathcal{P}_2$ sends $[\![j]\!]$ to $\mathcal{P}_1$
7. For each $i \in \mathbb{Z}_k$, $\mathcal{P}_1$ creates the following set for cut-and-choose:
   (a) $r_i \leftarrow\$\, \mathbb{Z}_n$
   (b) $w_i = g_P^{r_i} \mod n$
   (c) $x_i \leftarrow \mathsf{Gen}(\{1\}^{256})$
   (d) $\{[\![x_i]\!], \kappa_{x_i}\} = \mathsf{com}(x_i)$
   (e) $e_i \leftarrow \mathsf{Enc}_{x_i}(w_i)$
   (f) $m_i \leftarrow\$\, \mathbb{Z}_q$
   (g) $\{[\![m_i]\!], \kappa_i\} = \mathsf{com}(m_i)$
   (h) $v_i = (N+1)^{m_i} \cdot h_P^{r_i \cdot N} \mod N^2$
8. $\mathcal{P}_1$ sends $\{(e_i, [\![x_i]\!], [\![m_i]\!], v_i) | i \in \mathbb{Z}_k\}$ to $\mathcal{P}_2$
9. $\mathcal{P}_2$ sends j and χ_j to $\mathcal{P}_1$.
10. $\mathcal{P}_1$ verifies $j \in \mathbb{Z}_k$ and $\mathsf{V}_{\mathsf{com}}([\![j]\!], j, \chi_j)$
11. $\mathcal{P}_1$ sends $\{(x_i, w_i, \kappa_{x_i}, r_i, m_i, \kappa_i) | i \neq j \land i \in \mathbb{Z}_k\}$ to $\mathcal{P}_2$
12. $\mathcal{P}_2$ verifies $g_P^{r_i} = \mathsf{Dec}_{x_i}(e_i)$, $\mathsf{V}_{\mathsf{com}}([\![m_i]\!], m_i, \kappa_i)$, and $\frac{(v_i \cdot h_P^{-r_i \cdot N} \mod N^2) - 1}{N} = m_i$ for all $i \neq j \land i \in \mathbb{Z}_k$
13. $\mathcal{P}_1$ sends $m' = m - m_j$ to $\mathcal{P}_2$
14. Both parties calculate $[\![m]\!]' = [\![m_j]\!] + (m')G$
15. $\mathcal{P}_1$ sends $\pi = \mathsf{P}_{\mathsf{zk}}^{\mathsf{DL}}([\![m]\!] \cdot ([\![m]\!]')^{-1}, \kappa_m - \gamma)$ to $\mathcal{P}_2$.
16. $\mathcal{P}_2$: If $\mathsf{P}_{\mathsf{zk}}^{\mathsf{DL}}([\![m]\!] \cdot ([\![m]\!]')^{-1}) \neq 1$, abort.
17. Return $\{x_j, \kappa_{x_j}\}$ to $\mathcal{P}_1$ and $\Pi' = \{v_j, e_j, m', [\![x_j]\!]\}$ to $\mathcal{P}_2$

Protocol 4.4: Concealed Time locked puzzles

$\underline{\mathsf{Contingency}_{\mathcal{P}_1}(\alpha_i^{(j)}, pk_i^{(j)}, \delta_1, \mathcal{L}_j)}$		$\underline{\mathsf{Contingency}_{\mathcal{P}_2}(\beta_i^{(j)}, pk_i^{(j)}, \delta_2, \mathcal{L}_j)}$
$m_1 \leftarrow\$ \mathbb{Z}_q$		$m_2 \leftarrow\$ \mathbb{Z}_q$
$\bar{\alpha}_i^{(1)} := \texttt{extractkey}(\alpha_i^{(1)})$		$\bar{\beta}_i^{(1)} := \texttt{extractkey}(\beta_i^{(1)})$
$(pk_1)_i^{(1)} := \texttt{publicform}(pk_i^{(1)})$		$(pk_2)_i^{(1)} := \texttt{publicform}(pk_i^{(1)})$
$L_{\bar{\alpha}^{(1)}} := [\bar{\alpha}_1^{(1)}, \bar{\alpha}_2^{(1)} ..., \bar{\alpha}_i^{(1)}, m_1]$		$L_{\bar{\beta}^{(1)}} := [\bar{\beta}_1^{(1)}, \bar{\beta}_2^{(1)} ..., \bar{\beta}_i^{(1)}, m_2]$
$\{[\![m_1]\!], \kappa_1\} = com(m_1)$		$\{[\![m_2]\!], \kappa_2\} = com(m_2)$
$L_{(pk_1)}^{(1)} := [(pk_1)_1^{(1)}, (pk_1)_2^{(1)}, ..., (pk_1)_i^{(1)}, [\![m_1]\!]]$		$L_{(pk_2)}^{(1)} := [(pk_2)_1^{(1)}, (pk_2)_2^{(1)}, ..., (pk_2)_i^{(1)}, [\![m_2]\!]]$
$L_\lambda^{(1)} := [\texttt{ord}(B_1^{(1)}), \texttt{ord}(B_2^{(1)}), ...\texttt{ord}(B_i^{(1)}), q]$		$L_\lambda^{(1)} := [\texttt{ord}(B_1^{(1)}), \texttt{ord}(B_2^{(1)}), ..., \texttt{ord}(B_i^{(1)}), q]$
$\Omega_1 := \mathsf{PolyLock}(L_{\bar{\alpha}^{(1)}}, L_{(pk_1)}^{(1)}, L_\lambda^{(1)})$	$\xrightarrow{\Omega_1}$	$\Omega_2 := \mathsf{PolyLock}(L_{\bar{\beta}^{(1)}}, L_{(pk_2)}^{(1)}, L_\lambda^{(1)})$
If $\mathsf{PolyVerify}(\Omega_2) \neq 1$, abort	$\xleftarrow{\Omega_2}$	If $\mathsf{PolyVerify}(\Omega_1) \neq 1$, abort
$\Pi_1' = \mathsf{Conceal}(m_1, x_1, \delta_1)$, parses $[\![x_1]\!]$	$\xrightarrow{\Pi_1'}$	$\Pi_2' = \mathsf{Conceal}(m_2, x_2, \delta_2)$, parses $[\![x_2]\!]$
$T^{E_1}_{\mathcal{B}_i^{(2)}} = (pk_i^{(2)}, \mathcal{P}_1, t_1)$	$\xleftarrow{\Pi_2'}$	$T^{E_2}_{\mathcal{B}_i^{(2)}} = (pk_i^{(2)}, \mathcal{P}_2, t_2)$

Jointly run corresponding PSign on each escape transaction for each blockchain $\mathcal{B}_i^{(2)}$ in List $\mathcal{L}_2$

$\{a^{E_2}_{\mathcal{B}_i^{(2)}}, A^{E_2}_{\mathcal{B}_1^{(2)}}, B^{E_2}_{\mathcal{B}_1^{(2)}}, \bar{\sigma}^{E_2}_{\mathcal{B}_1^{(2)}}\} \leftarrow \mathsf{PSign}(\alpha_i^{(2)}, pk_i^{(2)}, T^{E_2}_{\mathcal{B}_i^{(2)}})$		$\{b^{E_2}_{\mathcal{B}_i^{(2)}}, A^{E_2}_{\mathcal{B}_1^{(2)}}, B^{E_2}_{\mathcal{B}_1^{(2)}}, \bar{\sigma}^{E_2}_{\mathcal{B}_1^{(2)}}\} \leftarrow \mathsf{PSign}(\beta_i^{(2)}, pk_i^{(2)}, T^{E_2}_{\mathcal{B}_i^{(2)}})$
$\{a^{E_1}_{\mathcal{B}_i^{(2)}}, A^{E_1}_{\mathcal{B}_1^{(2)}}, B^{E_1}_{\mathcal{B}_1^{(2)}}, \bar{\sigma}^{E_1}_{\mathcal{B}_1^{(2)}}\} \leftarrow \mathsf{PSign}(\alpha_i^{(2)}, pk_i^{(2)}, T^{E_1}_{\mathcal{B}_i^{(2)}})$		$\{b^{E_1}_{\mathcal{B}_i^{(2)}}, A^{E_1}_{\mathcal{B}_1^{(2)}}, B^{E_1}_{\mathcal{B}_1^{(2)}}, \bar{\sigma}^{E_1}_{\mathcal{B}_1^{(2)}}\} \leftarrow \mathsf{PSign}(\beta_i^{(2)}, pk_i^{(2)}, T^{E_1}_{\mathcal{B}_i^{(2)}})$
$L_\lambda^{(2)} := [\texttt{ord}(B_1^{(1)}), \texttt{ord}(B_2^{(1)}), ..., \texttt{ord}(B_i^{(1)}), \texttt{ord}(x_1)]$		$L_\lambda^{(2)} := [\texttt{ord}(B_1^{(1)}), \texttt{ord}(B_2^{(1)}), ..., \texttt{ord}(B_i^{(1)}), \texttt{ord}(x_2)]$
$L_{a^{E_2}} := [a^{E_2}_{\mathcal{B}_1^{(2)}}, a^{E_2}_{\mathcal{B}_2^{(2)}}, ..., a^{E_2}_{\mathcal{B}_i^{(2)}}, x_1]$		$L_{b^{E_1}} := [b^{E_1}_{\mathcal{B}_1^{(2)}}, b^{E_1}_{\mathcal{B}_2^{(2)}}, ..., b^{E_1}_{\mathcal{B}_i^{(2)}}, x_2]$
$L_{A^{E_2}} := [A^{E_2}_{\mathcal{B}_1^{(2)}}, A^{E_2}_{\mathcal{B}_2^{(2)}}, ..., A^{E_2}_{\mathcal{B}_i^{(2)}}, [\![x_1]\!]]$		$L_{B^{E_1}} := [B^{E_1}_{\mathcal{B}_1^{(2)}}, B^{E_1}_{\mathcal{B}_2^{(2)}}, ..., B^{E_1}_{\mathcal{B}_2^{(2)}}, [\![x_2]\!]]$
$\Omega_3 := \mathsf{PolyLock}(L_{a^{E_2}}, L_{A^{E_2}}, L_\lambda^{(2)})$	$\xrightarrow{\Omega_3}$	$\Omega_4 := \mathsf{PolyLock}(L_{b^{E_1}}, L_{B^{E_1}}, L_\lambda^{(2)})$
If $\mathsf{PolyVerify}(\Omega_4) \neq 1$, abort	$\xleftarrow{\Omega_4}$	If $\mathsf{PolyVerify}(\Omega_3) \neq 1$, abort
	$\xrightarrow{a^{E_2}_{\mathcal{B}_i^{(2)}}}$	
$\sigma^{E_1}_{\mathcal{B}_i^{(2)}} \leftarrow \mathsf{Complete}\left(T^{E_1}_{\mathcal{B}_i^{(2)}}, a^{E_1}_{\mathcal{B}_i^{(2)}}, b^{E_1}_{\mathcal{B}_i^{(2)}}, \bar{\sigma}^{E_1}_{\mathcal{B}_i^{(2)}}\right)$	$\xleftarrow{b^{E_1}_{\mathcal{B}_i^{(2)}}}$	$\sigma^{E_2}_{\mathcal{B}_i^{(2)}} \leftarrow \mathsf{Complete}\left(T^{E_2}_{\mathcal{B}_i^{(2)}}, a^{E_2}_{\mathcal{B}_i^{(2)}}, b^{E_2}_{\mathcal{B}_i^{(2)}}, \bar{\sigma}^{E_2}_{\mathcal{B}_i^{(2)}}\right)$
Return $(T^{E_1}_{\mathcal{B}_i^{(2)}}, \sigma^{E_1}_{\mathcal{B}_i^{(2)}}, \Pi_2', \Omega_2, \Omega_4)$		**Return** $(T^{E_2}_{\mathcal{B}_i^{(2)}}, \sigma^{E_2}_{\mathcal{B}_i^{(2)}}, \Pi_1', \Omega_1, \Omega_3)$

Protocol 4.5: Alternative Contingency

For parties $(\mathcal{P}_1, \mathcal{P}_2)$, the protocol takes as input their list of account keys for the joint accounts $\alpha_i^{(j)}$ & $\beta_i^{(j)}$ for blockchains in $\mathcal{L}_j$, the public keys for joint accounts $pk_i^{(j)}$ for blockchains in $\mathcal{L}_j$, the difficulties for concealed time locked puzzles δ_1 & δ_2 and the list of blockchains $\mathcal{L}_j$ for $j \in \{1, 2\}$. Each party $\mathcal{P}_1$ and $\mathcal{P}_2$ first randomly chooses m_1 and m_2 from $\mathbb{Z}_q$. Each party P_j creates a polynomial lock Ω_j linking their half of the secrets, $\bar{\alpha}$ for $\mathcal{P}_1$ and $\bar{\beta}$ for $\mathcal{P}_2$ extracted from their unlocking secrets using `extractkey(.)`, for joint accounts in list $\mathcal{L}_1$ along with m_1 for $\mathcal{P}_1$ and m_2 for $\mathcal{P}_2$. For the locks each party also needs to input the public forms of their secrets which they can trivially calculate. For that, we assume a function `publicform(`pk`)`. Similarly, we assume a function `ord(.)` which returns the order of the group for its input. The outputs of PolyLock are sent to the other party which is verified by each party i.e. Ω_1 is sent to $\mathcal{P}_1$ and Ω_2 is sent to $\mathcal{P}_2$. After that, each party creates a concealed time with secret keys x_1 and x_2 and difficulty δ_1 and δ_2 concealing the values m_1 and m_2 respectively to receive Π_1' and Π_2' for $\mathcal{P}_1$ and $\mathcal{P}_2$.

Next, each party creates and signs escape transactions $T^{E_1}_{\mathcal{B}_i^{(2)}} = (pk_i^{(2)}, \mathcal{P}_1, t_1)$ for $\mathcal{P}_1$ and $T^{E_2}_{\mathcal{B}_i^{(2)}} = (pk_i^{(2)}, \mathcal{P}_2, t_2)$ for $\mathcal{P}_2$ for blockchains in list $\mathcal{L}_2$ such that $t_1 >> t_2$ using corresponding PSign protocols. Once again with their unlocking secrets for each escape transactions $T^{E_1}_{\mathcal{B}_2}$ paying to $\mathcal{P}_2$ along with the secret key to their concealed time-locked puzzle x_1, $\mathcal{P}_1$ creates a polynomial lock Ω_3. Similarly, $\mathcal{P}_2$ creates polynomial lock Ω_4 with their unlocking secrets for each escape transaction $T^{E_2}_{\mathcal{B}_2}$ paying to $\mathcal{P}_1$ along with their secret key x_2 to concealed time-locked puzzle. The parties then send the polylock outputs Ω_3 to $\mathcal{P}_2$ and Ω_4 to $\mathcal{P}_1$ and verifies it. Once verified, the parties send all their unlocking secrets for the other party's escape transaction to each other. $\mathcal{P}_1$ sends $a^{E_2}_{\mathcal{B}_i^{(2)}}$ to $\mathcal{P}_2$ while $\mathcal{P}_2$ sends $b^{E_1}_{\mathcal{B}_i^{(2)}}$ to $\mathcal{P}_2$. With this the parties can complete the signatures for their escape transactions using corresponding `Complete` protocols that can be posted to the blockchain when required. The protocol returns each party with their signed escape transaction, concealed time locks and polynomial locks to each party. $\mathcal{P}_1$ gets $(T^{E_1}_{\mathcal{B}^{(2)}_i}, \sigma^{E_1}_{\mathcal{B}^{(2)}_i}, \Pi'_2, \Omega_2, \Omega_4)$ and $\mathcal{P}_2$ gets $(T^{E_2}_{\mathcal{B}^{(2)}_i}, \sigma^{E_2}_{\mathcal{B}^{(2)}_i}, \Pi'_1, \Omega_1, \Omega_3)$.

Escape Protocol. In the event of PolySwap failure after assets are in escrow, the mechanisms in the Contingency Protocol are used to recover the assets. When both sets of blockchains support escape transactions, this is trivial. $\mathcal{P}_1$ submits their escape transactions after t_1 has elapsed and $\mathcal{P}_2$ submits their escape transactions after t_2 has elapsed and the protocol terminates on failure with both parties getting their assets back.

Otherwise, the escape protocol is more complicated. Once t_2 elapses, $\mathcal{P}_2$ executes their escape transactions on the blockchains in L_2. $\mathcal{P}_1$ extracts the signature from this transaction and uses Ω_4 to recover x_2. $\mathcal{P}_1$ uses x_2 to reveal the concealed time-locked puzzle Π'_2, recovering Π_2. $\mathcal{P}_1$ then solves the puzzle to extract m_2 and uses m_2 with Ω_2 to extract $\mathcal{P}_2$'s private keys to the joint accounts for the blockchains in L_1. $\mathcal{P}_1$ then unilaterally creates transactions out of the joint accounts from the blockchains from L_1. If $\mathcal{P}_2$ does not execute their escape transactions before t_1 elapses, then this process is mirrored. Note that the difficulty of Π_1 is much higher than Π_2 to prevent $\mathcal{P}_2$ from using superior hardware to steal assets.

Chapter 5

PolySwap: PRIVACY-PRESERVING ATOMIC SWAP PROTOCOL

Addressing our research question, we propose PolySwap, a generic framework for *privacy-preserving multi-chain atomic swap* with the following properties:

- No trusted third party is required.
- No special features are required, such as scripting, besides the support for time-locked escape transaction.
- An outside observer cannot confirm whether or not an atomic swap occurred.
- An outside observer cannot distinguish atomic swap's transactions from normal ones.
- Supports atomic swap from any *set* of blockchains to any other *set* of blockchains.

We take advantage of the fact that all blockchains use digital signature as a common cryptographic primitive to verify transactions. We introduce a novel secret sharing signature scheme to remove the necessity of common interfaces between the blockchains in question and not limiting itself to common functionalities available on the blockchains. These secret sharing signatures allow an arbitrarily large number of signatures to be bound together, such that the release of any single transaction on one blockchain opens the remaining transactions for the other party, allowing multi-chain atomic swaps while still being indistinguishable from a standard signature. We provide construction details of SSSig for ECDSA, Schnorr, and CryptoNote-style Ring signatures. Out of the top 30 mainstream cryptocurrencies [50], the provided constructions for SSSig covers 23 based on the signature algorithm used. Additionally, we provide an alternative contingency protocol, allowing parties to exchange to and from blockchains that do not support any form of time-locked escape transactions.

5.1 PolySwap

PolySwap is a two-party protocol that enables a party to exchange any number of cryptoassets with another party without a trusted intermediary in a single run of the protocol. PolySwap is presented in Protocol 5.1.

The protocol is jointly run by two parties $\mathcal{P}_1$ and $\mathcal{P}_2$ holding assets in a list of blockchains $\mathcal{L}_1$ and $\mathcal{L}_2$ respectively. We describe the protocol for when blockchains in $\mathcal{L}_1$ supports escape transactions. However the protocol can be easily adjusted to address instances where blockchains in $\mathcal{L}_2$ support time-locked transaction and $\mathcal{L}_1$ does not, meaning the roles of P_1 and P_2 can be readily interchanged. In step 1, both parties run the KeyGen functionality for each blockchain in lists $\mathcal{L}_1$ and $\mathcal{L}_2$ to create joint accounts with a shared public key pk and corresponding secret keys as α and β respectively for parties $\mathcal{P}_1$ and $\mathcal{P}_2$. The joint accounts function as escrows where a party deposits the assets to be exchanged as a transaction from a joint account needs to be jointly signed by each parties. After creating the joint accounts, both parties jointly run the contingency protocol as described in protocol Section 4.3.3 to ensure that the assets are recoverable in case of unsuccessful termination of the protocol. In step 3, each party deposits the agreed values of assets for exchange in the respective joint accounts.

In step 4, parties $\mathcal{P}_1$ and $\mathcal{P}_2$ jointly create and sign claim transactions paying to the other party from each joint account in each blockchain in lists $\mathcal{L}_1$ and $\mathcal{L}_2$ using PSign from SSSig for respective blockchain. The claim transactions for blockchains in list $\mathcal{L}_1$ pays

PolySwap: Privacy-Preserving Multi-chain Atomic Swap Protocol
For security parameter 1^n, parties $\mathcal{P}_j$, holding assets in a list $\mathcal{L}_j$ of blockchains $\mathcal{B}_i^{(j)}$ for $j \in \{1,2\}$ representing the parties and $i \in \{1, 2, ...\mathcal{L}_j.\texttt{size}\}$ representing a blockchain in the list, where $\mathcal{L}_2$ supports time-locked escape transactions, proceed in the following steps:

1. For each blockchain $\mathcal{B}_i^{(j)}$, $\mathcal{P}_1$ and $\mathcal{P}_2$ jointly run KeyGen which returns a shared public key $pk_i^{(j)}$ and corresponding secret keys to each party, $\alpha_i^{(j)}$ to $\mathcal{P}_1$ and $\beta_i^{(j)}$ to $\mathcal{P}_2$.
2. $\mathcal{P}_1$ and $\mathcal{P}_2$ jointly run $\mathsf{Contingency}$ protocol as described in Section 4.3.3 to receive signed time-locked escape transactions to ensure fair termination of protocol.
3. $\mathcal{P}_1$ and $\mathcal{P}_2$ each post deposit transactions in their respective blockchains, depositing agreed values to the public key, joint account, created in Step 1. If not all blockchains in $\mathcal{L}_1$ support time locks, then $\mathcal{P}_2$ posts their deposit transactions first.
4. $\mathcal{P}_1$ and $\mathcal{P}_2$ create claim transactions and jointly generate secret sharing signatures for the claim transactions in each blockchain:
   (a) For each blockchain $\mathcal{B}_i^{(j)}$, $\mathcal{P}_1$ and $\mathcal{P}_2$ jointly create claim transactions $T^C_{\mathcal{B}_i^{(j)}} = (pk_i^{(j)}, \mathcal{P}_{3-j})$.
   (b) For each blockchain $\mathcal{B}_i^{(j)}$, $\mathcal{P}_1$ and $\mathcal{P}_2$ jointly run the PSign protocol on transaction $T^C_{\mathcal{B}_i^{(j)}}$ as message. Both parties receive a partial signature $\bar{\sigma}^C_{\mathcal{B}_i^{(j)}}$. $\mathcal{P}_1$ receives an unlocking secret $a^C_{\mathcal{B}_i^{(j)}}$ and $\mathcal{P}_2$ receives unlocking secret $b^C_{\mathcal{B}_i^{(j)}}$.
5. $\mathcal{P}_1$ creates a PolyLock linking their SSSig unlocking secrets:
   (a) $\mathcal{P}_1$ and $\mathcal{P}_2$ create a list of orders L_λ from the list of blockchains in $\mathcal{L}_2$ and $\mathcal{L}_1$.
   (b) $\mathcal{P}_1$ runs $\mathsf{PolyLock}(L_\phi, L_\Phi, L_\lambda)$ where $L_\phi = [a^C_{\mathcal{B}_i^{(j)}}]$ and $L_\Phi = [A^C_{\mathcal{B}_i^{(j)}}]$, to receive $\Omega = (L_{\text{-}x}, L_\pi, \pi_x, \pi_{\text{-}x})$.
   (c) $\mathcal{P}_1$ sends Ω to $\mathcal{P}_2$. $\mathcal{P}_2$ verifies by running $\mathsf{PolyVerify}(L_\pi, \pi_x, \pi_{\text{-}x}, L_\lambda)$.
6. If $\mathcal{P}_2$ accepts, $\mathcal{P}_2$ releases all their secrets $b^C_{\mathcal{B}_i^{(2)}}$ so that $\mathcal{P}_1$ can redeem their claim transactions $T^C_{\mathcal{B}_i^{(2)}}$ consequently making it possible for $\mathcal{P}_2$ to redeem their claim transactions $T^C_{\mathcal{B}_i^{(1)}}$:
   (a) For each blockchain $\mathcal{B}_i^{(2)}$ in $\mathcal{L}_2$, $\mathcal{P}_1$ computes the full signature $\sigma^C_{\mathcal{B}_i^{(2)}}$ by running $\mathsf{Complete}(m, a^C_{\mathcal{B}_i^{(2)}}, b^C_{\mathcal{B}_i^{(2)}}, \bar{\sigma}^C_{\mathcal{B}_i^{(2)}})$ and posts the transaction $(T^C_{\mathcal{B}_i^{(2)}}, \sigma^C_{\mathcal{B}_i^{(2)}})$ to $\mathcal{B}_i^{(2)}$.
   (b) $\mathcal{P}_2$ retrieves a signature from a blockchain $\mathcal{B}_i^{(2)}$ from $\mathcal{L}_2$, $\sigma^C_{\mathcal{B}_i^{(2)}}$. $\mathcal{P}_2$ runs $\mathsf{Reveal}(b^C_{\mathcal{B}_i^{(2)}}, \sigma^C_{\mathcal{B}_i^{(2)}})$ to compute $a^C_{\mathcal{B}_i^{(2)}}$.
   (c) $\mathcal{P}_2$ uses $\mathsf{PolyRelease}(L_{\text{-}x}, a^C_{\mathcal{B}_i^{(2)}}, i, L_\lambda)$ to get $\mathcal{P}_1$'s secrets, $a^C_{\mathcal{B}_i^{(1)}}$ for each blockchain $\mathcal{B}_i^{(1)}$ in $\mathcal{L}_1$.
   (d) For each blockchain $\mathcal{B}_i^{(1)}$ in $\mathcal{L}_1$, $\mathcal{P}_2$ computes the full signatures $\sigma^C_{\mathcal{B}_i^{(1)}}$ by running $\mathsf{Complete}(m, a^C_{\mathcal{B}_i^{(1)}}, b^C_{\mathcal{B}_i^{(1)}}, \bar{\sigma}^C_{\mathcal{B}_i^{(1)}})$ and posts the transaction $(T^C_{\mathcal{B}_i^{(1)}}, \sigma^C_{\mathcal{B}_i^{(1)}})$ to $\mathcal{B}_i^{(1)}$.
7. If all transactions are posted and confirmed, return *success*.

Protocol 5.1: PolySwap

to $\mathcal{P}_2$ while those in $\mathcal{L}_2$ pays to $\mathcal{P}_1$. PSign outputs unlocking secrets ϕ_1 along with their public forms Φ_1 for $\mathcal{P}_1$ and unlocking secret ϕ_2 with its public form Φ_2 for party $\mathcal{P}_2$ for each of the claim transactions in each blockchain. We represent the unlocking secret as a with its public form as A for party $\mathcal{P}_1$ and unlocking secret as b with its public form as B for party $\mathcal{P}_2$ for readability. In step 5, $\mathcal{P}_1$ creates a polynomial lock $\mathsf{PolyLock}$ with their outputs from step 4 for claim transactions $T^C_{\mathcal{B}_i^{(1)}}$ and $T^C_{\mathcal{B}_i^{(2)}}$ for blockchains in list $\mathcal{L}_1$ and $\mathcal{L}_2$ respectively. $\mathcal{P}_1$ sends the outputs from this lock to $\mathcal{P}_2$ who runs the $\mathsf{PolyVerify}$ algorithm to verify the validity of the polynomial.

After $\mathcal{P}_2$ accepts the proofs of $\mathsf{PolyLock}$, in step 6, $\mathcal{P}_2$ sends all their unlocking secrets from step 4 for claim transactions in $\mathcal{L}_2$ to $\mathcal{P}_1$. With these unlocking secrets $\mathcal{P}_1$ runs $\mathsf{Complete}$ to recover signatures for the claim transactions for each blockchains in list $\mathcal{L}_2$. $\mathcal{P}_1$ posts these signatures and transactions to the respective blockchains in list $\mathcal{L}_2$ to claim the escrowed assets. After a transaction is confirmed by a blockchain in $\mathcal{L}_2$, $\mathcal{P}_2$ recovers a full signature on any one of the claim transaction to get a unique point on the polynomial to run $\mathsf{PolyRelease}$. $\mathsf{PolyRelease}$ outputs the unlocking secrets of $\mathcal{P}_1$ for claim transactions for blockchains in $\mathcal{L}_1$ along with those in $\mathcal{L}_2$. $\mathcal{P}_2$ is only concerned with the unlocking secrets for claim transactions in list $\mathcal{L}_1$. With these unlocking secrets, $\mathcal{P}_2$ computes full signatures and posts claim transactions to blockchains in list $\mathcal{L}_1$. Once all the transactions are confirmed, $\mathcal{P}_2$ acquires the escrowed assets in joint accounts in $\mathcal{L}_1$ while $\mathcal{P}_1$ already has acquired the assets in joint accounts in $\mathcal{L}_2$, thus completing the $\mathsf{PolySwap}$ protocol returning *success*.

A step-by-step detail sequence diagram of $\mathsf{PolySwap}$ for one-to-one atomic swap between two party is shown in Figure 5.1.

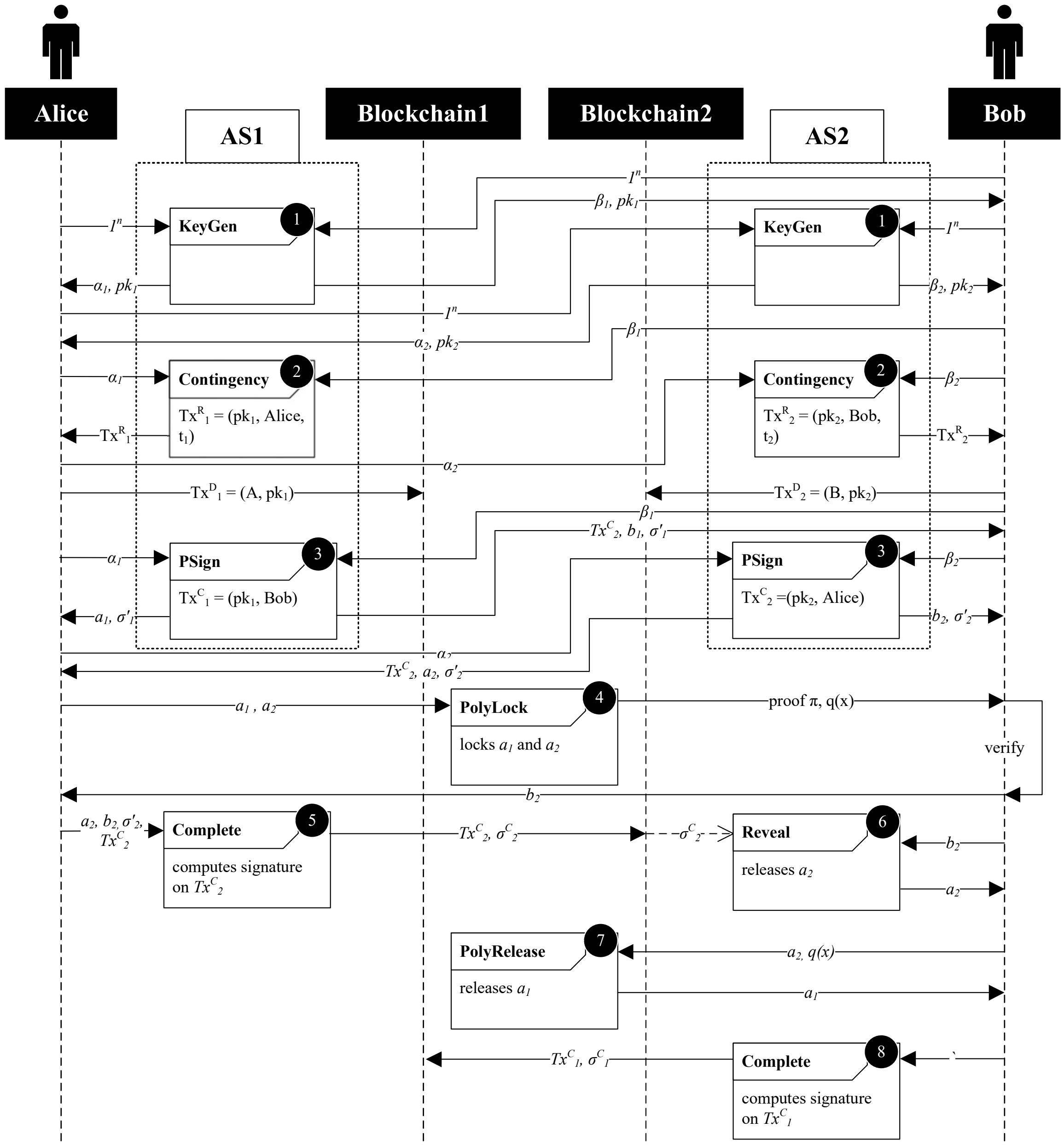


Figure 5.1: PolySwap details for atomic swap between Alice and Bob owning assets in Blockchain 1 and Blockchain 2

5.2 Discussion and Limitation

Optionality/Lockup Griefing. HTLC-based atomic swaps are asymmetric as only one party, Bob–for an atomic swap between Alice and Bob, carries the secret and the completion of the swap is dependent upon Bob's decision to release the secret or not. The party holding the secret does not have any incentive, positive or negative, to complete the swap, since after he waits for the time out, he can get his asset back. Therefore, Bob now has an *option* to either continue or abort the swap depending upon the volatile exchange rate for the asset under consideration. This provides Bob with an *inadvertent call option* [51], weakening the definition of fairness in the protocol. Also, if Bob does not go through the swap, then Alice's asset will be locked until timeout, causing a *lockup griefing* attack [44]. Our protocol also suffers from similar problems; however, techniques involving holding collateral assets in joint accounts and subsequent penalties for misbehaving parties could address these problems. Nonetheless, holding collateral reduces the usability of the protocol, and this problem is a matter of trade-offs.

Linkability due to Payment Values and Time. Although it is impossible to prove a link between transactions of an atomic swap between blockchains created by PolySwap, it may be possible to infer that an atomic swap occurred due to the values associated with the transactions (with a lower level of certainty). For example, an adversary who sees $5 moving from one account to an intermediate account then, to another account in one blockchain, and a similar structure and value, like $4.99 of value moving in another blockchain at around the same time period, they might reasonably assume that this is an atomic swap. When the values of the transactions are public, it is possible to make such an analysis unless the values transferred are random. However, we can make such an analysis less effective. When executing an atomic swap between two cryptocurrencies without hidden values, we recommend increasing the duration of the swap and treating each blockchain as two blockchains, effectively executing a 2 blockchain to 2 blockchain atomic swap (e.g. Bitcoin and Bitcoin to Litecoin and Litecoin). This makes the analysis described above significantly harder and the conclusions less probable.

Indistinguishability. Transactions created by PolySwap are indistinguishable in the sense that they look like majority of transactions in the respective blockchain network. This is valid if we consider the transaction independently which is true for stateful cryptocurrency systems like Ethereum. However, in cryptocurrency systems based on UTXOs like Bitcoin where a transaction cannot exist on its own and must refer to previous transaction outputs, an inevitable pattern exists [52] which could have adverse effects in user privacy. Nonetheless, for such cryptosystems, general recommendations for improving privacy like creating transactions with multiple inputs and outputs can be used to thwart privacy attacks using transaction pattern analysis.

Limitations. We require at least one of the sets of blockchains to support escape transactions. As a result, a direct atomic swap between two such currencies like Monero and ByteCoin is not possible *atomically* since neither blockchain supports escape transactions. However, an exchange of Monero and ByteCoin can occur with an intermediary blockchain, e.g. Bitcoin, where PolySwap is first executed between Monero and Bitcoin and then between Bitcoin and ByteCoin. Our protocol also utilizes time-locked puzzles based on repeated squaring in an RSA modulus. Such puzzles tend to be imprecise and be partially dependent on the computational power of the parties in question.

Chapter 6

EXPERIMENTAL EVALUATION

In this chapter, we discuss the experiments and results for PolySwap. First, we briefly explain the experimental test bed. Next, we present the results from the execution of PolySwap on Bitcoin and Ethereum testnets, and discuss different case studies. Finally, we discuss the experiments relating to scalability and efficiency of PolySwap.

6.1 Experiment Test Bed

We perform our experiments on two environments: 1) Intel Core i7, 3.6GHz, 16 GB RAM, Windows OS machine, and 2) 11 GB RAM, Arch Linux OS running on a virtual box in the prior machine. We execute PolySwap on Arch linux and efficiency and scalability experiments on Windows OS. We implement PolySwap in Java 1.8 using the following libraries and APIs:

- Bouncy Castle Crypto API [53] bcprov-jdk15on:1.57
- BitcoinJ [54] bitcoinj-core:0.15.6
- JSON-RPC for Java [55] jsonrpc4j:1.5.3
- Web3j [56] web3j:core:4.5.5
- Solidity [57] solc:0.5.14
- Infura [58] (API access provider for Ethereum network)
- Bitcoin Core [59] v0.17.1

6.2 PolySwap Execution on Testnet

We evaluate the correctness and privacy-preserving properties of the protocol by executing instances of PolySwap for Bitcoin and Ethereum between two parties. To achieve this, we develop a prototypical software implementation of PolySwap to swap between Bitcoin and Ethereum blockchains. We perform the experiments on the testnet of each blockchain. We setup a full node for Testnet3 to communicate with the Bitcoin test network. The size of the downloaded blockchain data was 26 GB for Testnet3. On the other hand, we planned to use the Ropsten testnet as it is based on Proof-of-Work, like Ethereum mainnet, while all other testnets for Ethereum are based on Proof-of-Authority. However, because of large size of blockchain data required to be downloaded and slower verification time, we opt for an API endpoint service instead, provided by Infura to connect to Rinkeby testnet in Ethererum. Time-locks for Bitcoin transactions are implemented using *nLockTime* field in transactions. As Ethereum transactions do not have such fields, we emulate time-locks by using Solidity smart contracts. Source code for a simple time-lock smart contract in Ethereum is included in Appendix A. We use blockchain explorers: `etherscan.io` and `blockcypher.com` to verify the acceptance of transactions into the test networks.

We successfully execute a one-to-one atomic swap between Bitcoin (Testnet3) and Ethereum (Rinkeby) using PolySwap, which took 8.3 seconds to complete (excluding confirmation time). Transactions used to execute the atomic swap using PolySwap are shown in Table 6.1.

PolySwap has two terminal cases attributing to fair exchange: on ***success***, each party should end up owning other party's assets, and on ***abort*** or ***failure***, each party should retain their own assets.

These terminal cases are tested by the following test cases and expected outcomes

Table 6.1: Transactions used to execute an atomic swap using PolySwap between Bitcoin and Ethereum testnets.

Testnet	Deposit Transaction hash	Claim Transaction hash
Bitcoin: Testnet3	ad98cbbf7169b49238cb234326bb632d5182dc3ae8210bb3f2ee501d8aea27da	f86eefdc2327fcccefe2dc144c1d04de0ae71a556491f1f4768def971f9ed58f
Ethereum: Rinkeby	0x6e95b8cb10488415caec47caff3a1da2c3f7514218b0f4893ea4e9e5f603617a	0x468d62443a3f18b188cd56feea928c67da86de0e4e6ace4e8681037ec81cdfb3

indicating successful evaluation of PolySwap:

The details of these case evaluations are shown in Table 6.2

Table 6.2: Transaction details for different test cases

Case	Ethereum	Bitcoin	Remarks
Case I	0x468d62443a3f18b188cd56feea928c67da86de0e4e6ace4e8681037ec81cdfb3	f86eefdc2327fcccefe2dc144c1d04de0ae71a556491f1f4768def971f9ed58f	Claim transaction hash
Case II	0xfad9914081d6c347df81b527c20e1892dd3029aa3fd3e817bfd1625e04b5d048	0108bcc86b2f9f555d03ea7154119dd27522b9f1bf8a66ea1fda59d1f0e564aa	Refund transaction hash
Case III	0x965737a7981ce27ebb81638ff0dc4df0012882df0dc664ac22c7275a22c17018	-	Refund transaction hash

- **Case I (Effectiveness): After party 1 receives their unlocking secret from party 2 and party 1 posts their claim transaction in blockchain 2, can party 2 recover their unlocking secret from the signature in the claim transaction in blockchain 2 to complete the signature for their claim transaction in blockchain 1? In other words, when party 1 gets party 2's assets in blockchain 2, can party 2 get party 1's assets in blockchain 1?**

  Given that both Polynomial locking scheme and SSSig are correct, party 2 should be able to complete the signature for their claim transaction and post it on blockchain

1. The expected outcome is an addition in party 1's wallet balance on blockchain 2 & party 2's wallet balance on blockchain 1, and a subtraction in party 1's wallet balance on blockchain 1 & party 1's wallet balance on blockchain 2 by the agreed upon exchange values.

In order to test this case, the setup of the experiment was as follows: Alice owning BTC in Testnet3 of Bitcoin perform atomic swap with Bob owning Ether in Rinkeby testnet of Ethereum. The account details are shown below:

- Alice's Bitcoin account (sending):
  `tb1qga285nnwwe2288aprzdw9vlt77sdmsftmh95xy`
- Bob's Ethereum account (sending):
  `0x94f3854627826c37f5ba1f227ef42751e1e973b1`
- Alice's Ethereum account (receiving):
  `0xe98c5ab4b049df18d56ebc39f4e8e7549e3b6397`
- Bob's Bitcoin account (receiving):
  `tb1q39u0p9f0x2fujz44587v2has2tz7p7ujnzkv0c`
- Joint Bitcoin account:
  `tb1q36kn9rl6lhnrntf6xsr3al7vjmwfakzd3cst70`
- Joint Ethereum account:
  `0x9578BD6464B84e3ca3143041b267cDeC7f5BDA4F`

Alice and Bob decide on swapping 0.00008 BTC for 0.0005 Ether. First, both parties post deposit transactions in respective blockchains, transferring the agreed values to the joint accounts. Blockchain explorer view of the respective deposit transactions are shown in Figure 6.1 and Figure 6.2. Note that we use higher transaction fees for faster confirmation time. In these figures, we see that Alice starts with value

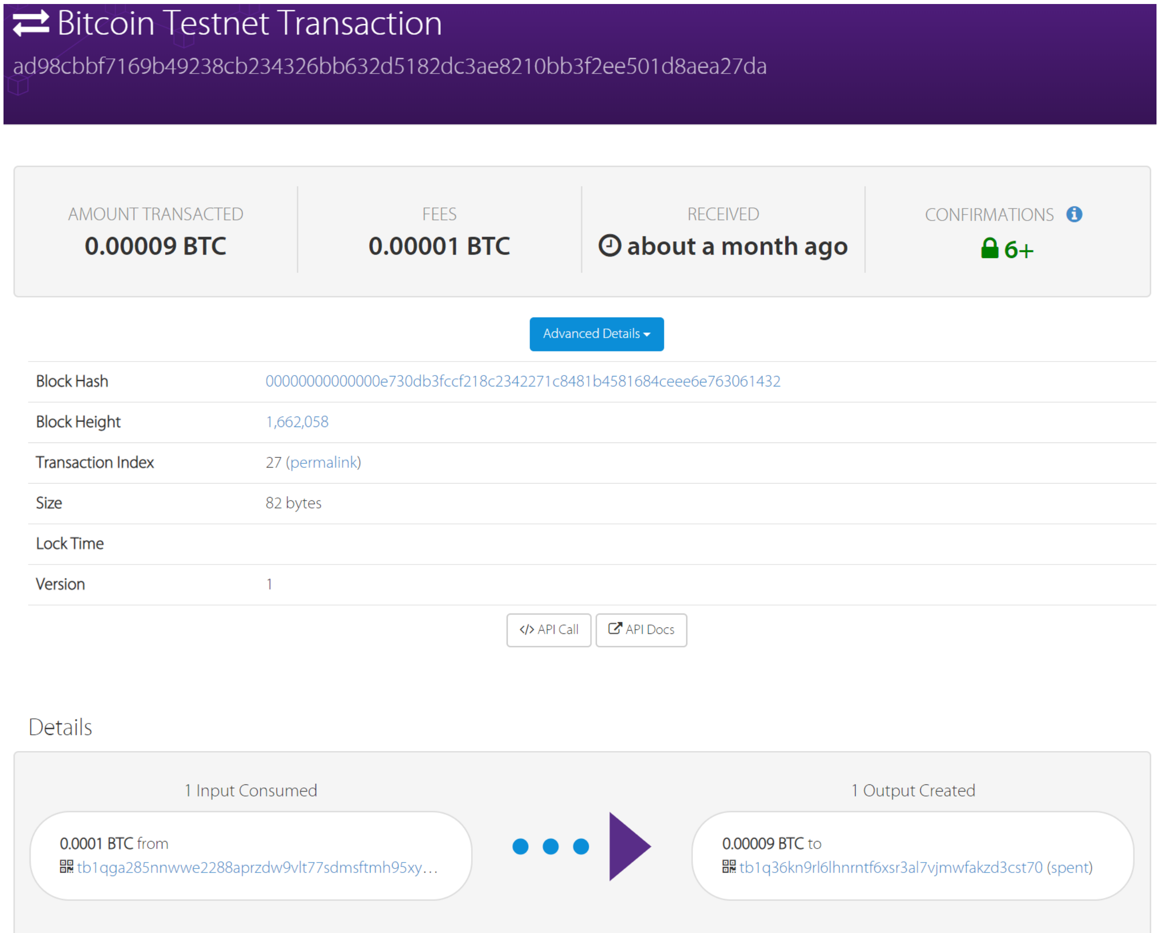


Figure 6.1: Blockchain explorer view of Alice's Deposit Transaction for Bitcoin

0.0001 BTC and pays only 0.00009 BTC to the joint account paying 0.00001 BTC as transaction fee for the deposit transaction.

Next, Alice sends her unlocking secrets for the claim transaction in Bitcoin to Bob using what Bob posted about the claim transaction to the network, as shown in Figure 6.3. By extracting the signature from this transaction, Alice calculates the signature for her claim transaction in Ethereum and posts it to the network, as shown in Figure 6.4. In order to reduce the swap time, Bob sends the transaction id (transaction hash) of his Bitcoin claim transaction to Alice; eliminating the need for Alice to

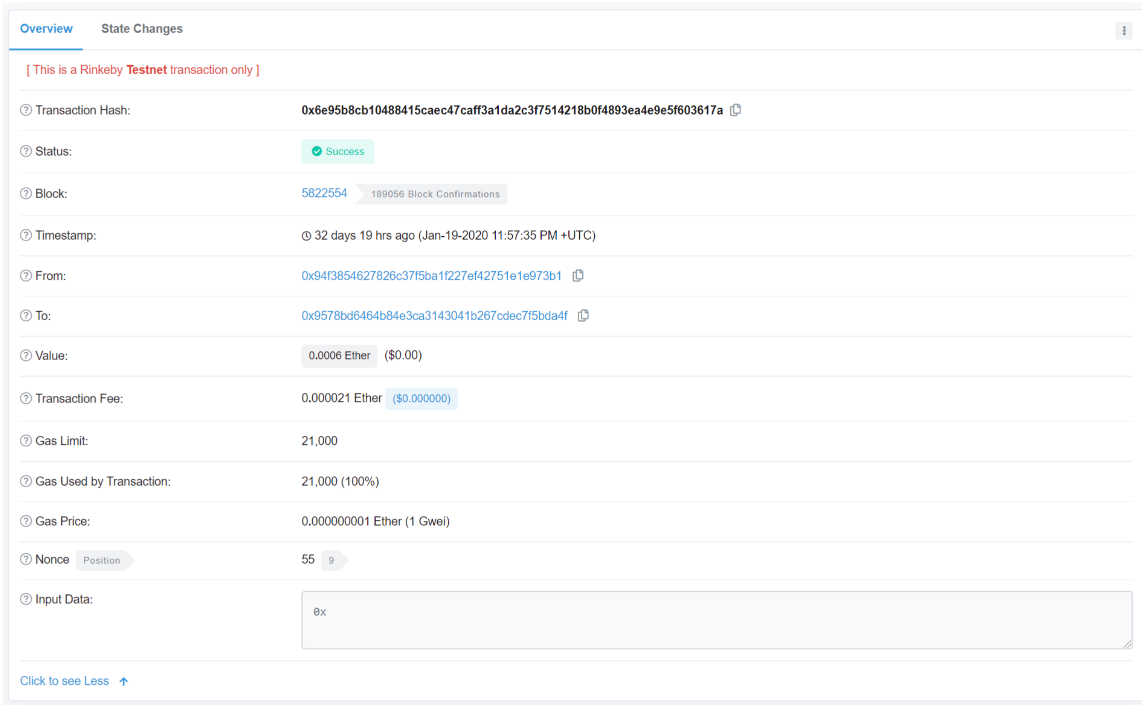


Figure 6.2: Blockchain explorer view of Bob's Deposit Transaction for Ethereum

look up Bob's claim transaction on Bitcoin's blockchain. This completes the swap as Alice's Ethereum's account and Bob's Bitcoin's account owns the respective agreed upon exchange values.

- **Case II (Fair Termination): After a party deposits their asset in the joint accounts on their blockchain and *either* party aborts, can the party recover their assets?**

  A party should be able to recover their assets by posting their escape transactions in respective blockchains from the *Contingency* protocol run by the parties before depositing their assets in the joint accounts, using a refund transaction after the expiration of lock time.

  We test this case by depositing funds to the joint accounts and then recovering those

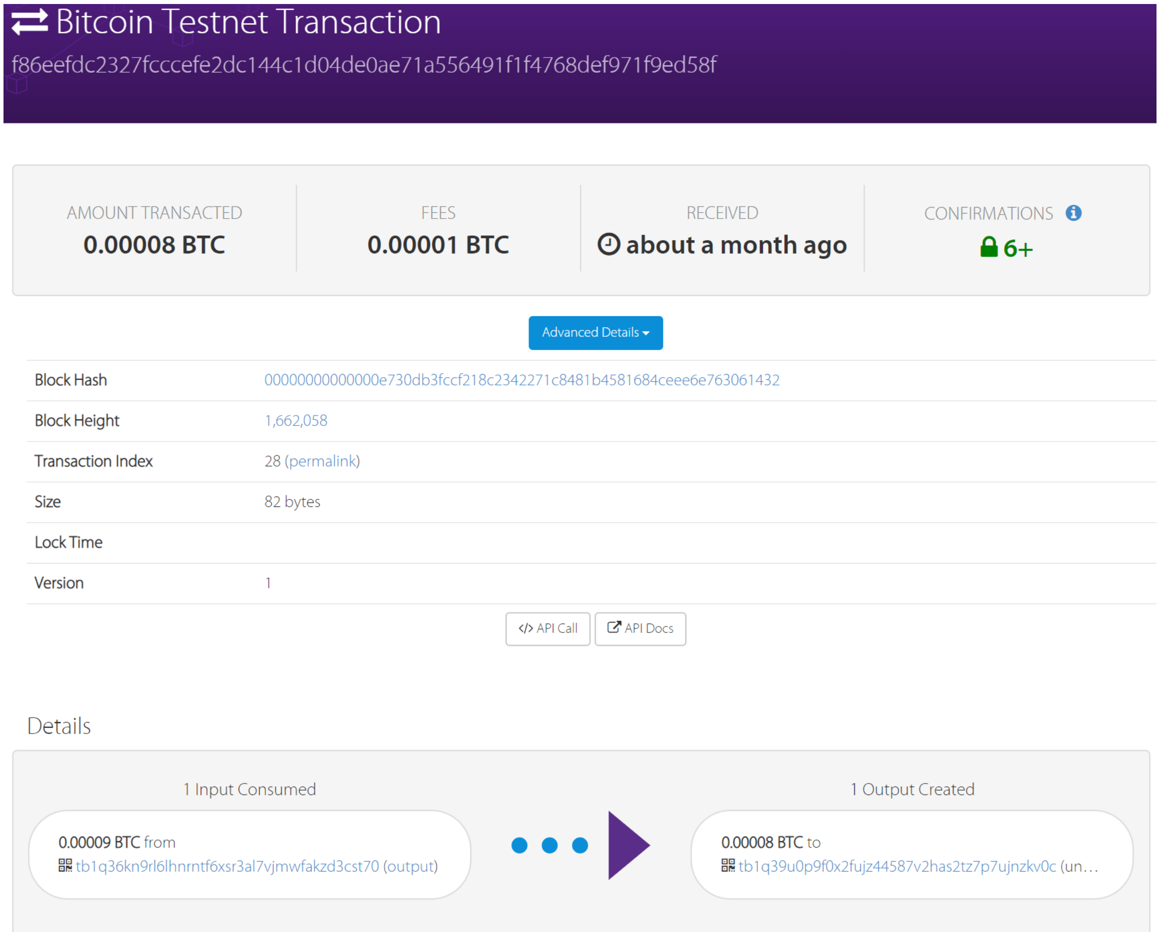


Figure 6.3: Blockchain explorer view of Bob's Claim Transaction for Bitcoin

funds by posting time-locked refund transactions on blockchain. The setup of the experiment is as follows. First, Alice and Bob owning assets in Bitcoin and Ethereum respectively create a joint account in each blockchain. Next, they jointly run the Contingency protocol to create refund transactions locked for 5 hours. Next, Bob deposits his Ether to the joint account, while Alice aborts the protocol. Next, after waiting for Alice to complete her deposit phase in Bitcoin for the 5 hours time period and not receiving deposit transaction confirmation, Bob posts the refund transaction in Ethereum network to recover his locked Ethereum.

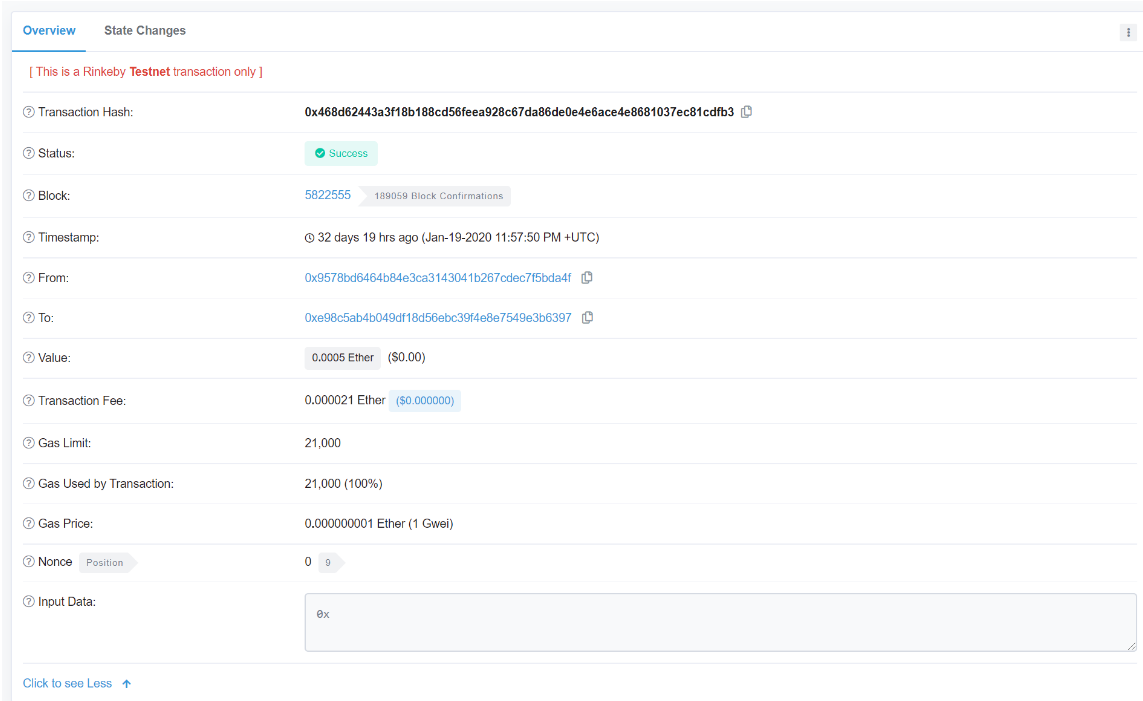


Figure 6.4: Blockchain explorer view of Alice's Claim Transaction for Ethereum

- **Case III (Fairness): Can party 1 post their claim transaction on blockchain 2 and escape transaction on blockchain 1 simultaneously to get party 2's assets on blockchain 2 and also retain their asset on blockchain 1?**

  The answer is No. That is, time locked escape transactions output from Contingency protocol should prevent this from happening.

  In the case of Bitcoin, time-locked transactions are not accepted by the network until the expiration of the time period, so the possibility of party 1 retaining their asset by posting refund transactions prematurely is highly unlikely in Bitcoin. As for Ethereum, the time-locked transaction are emulated using Ethereum smart contract, where refund is a functionality in the deployed contract. This function, even though callable via transactions before the expiration of time lock, won't execute successfully–meaning no value transfer occurs. The setup of the experiment is as follows: Alice and Bob owning assets in Bitcoin and Ethereum follow PolySwap till deposit phase. Alice sent her unlocking secret for Bob's claim transaction to Bob. Bob then posts the claim transaction to the Bitcoin blockchain along with his refund transaction on the Ethereum blockchain. The claim transaction is accepted by the Bitcoin blockchain while the Ethereum refund transaction failed to execute successfully. Using the signature on Bob's claim transaction in Bitcoin, Alice completes her claim transaction in Ethereum. Bob does not succeed in claiming Alice's bitcoins and retaining his assets in Ethereum.

6.2.1 Transaction indistinguishability and unlinkability

Due to the construction of PolySwap, transactions are indistinguishable from the normal ones and do not contain any information linking them to any other transactions in either blockchain (privacy-preserving). We empirically verify that the transactions created by PolySwap matches the majority of transactions found in respective blockchains. PolySwap requires three types of transactions based on their functionality: *Deposit* transaction, *Escape* transaction, and *Claim* transaction. Each transaction pays from an account to another spendable using normal signatures.

Transactions in Bitcoin can be distinguished based on their output scripts:

- **Pay to Public Key Hash (P2PKH)**: A transaction of this type is locked with a hash of a public key. It is spendable with a signature from the private key along with the public key corresponding to the public key hash on the transaction.

- **Pay to Public Key (P2PK)**: A transaction of this type is locked with a public key instead of a public key hash, and is spendable with the signature of the transaction for the public key.

- **Pay to Script Hash (P2SH)**: A transaction of this type is locked with the hash of an arbitrary script, e.g. multi-sig, time-locked script, and HTLC. They are spendable by fulfilling the conditions in the script using the redeem script.

- **Pay to Witness Public Key Hash (P2WPKH)**: A transaction of this type is also locked with the hash of the public key; however, it follows the segregated witness structure proposed in BIP141 [60].

- **Pay to Witness Script Hash (P2WSH)**: A transaction of this type is equivalent to P2SH except its follows the segregated witness structure.

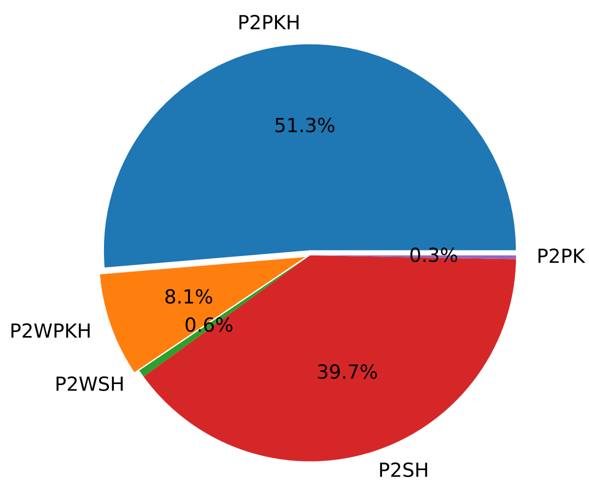


Figure 6.5: Transaction types on Bitcoin blockchain

Transactions created by PolySwap for Bitcoin are either P2PKH or P2WPKH based on whether the segregated witness proposal is followed or not. To empirically verify that P2PKH or P2WPKH belong to the majority of transactions in Bitcoin, we randomly select 100 blocks from the Bitcoin mainnet having 391,544 transaction outputs from 203,552 transactions and plot the rates of each type of transaction as shown in Figure 6.5.

We observe that the majority of transactions in Bitcoin blockchain are P2PKH (51.3%) which are the type of transactions created by PolySwap for Bitcoin. Thus, PolySwap transactions for Bitcoin are indistinguishable. P2WPKH (8.1%) transactions can also be created by PolySwap, which are bound to increase with an increased adoption of BIP141. Furthermore, since the transactions only contain the hash of the public keys and the signatures for transaction verification (unlike HTLCs), these transactions are unlinkable with Ethereum transactions used in PolySwap.

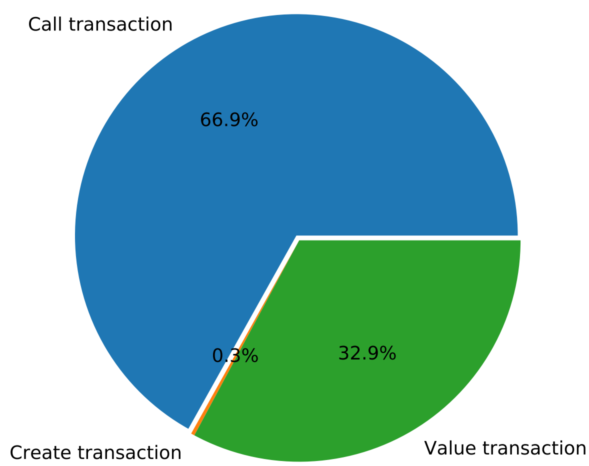


Figure 6.6: Transaction types on Ethereum blockchain

On the other hand, transactions in Ethereum are distinguished based on the purpose of the transaction:

- **Call Transaction**: Call transactions are transactions used to trigger a function call in an Ethereum smart contract. They contain a data field which specifies the function to be called and its arguments as a payload.

- **Value Transaction**: Value transactions are transactions paying from one externally owned account (EOA) to another.

- **Create Transaction**: Create transactions are transaction used to deploy smart contracts.

Transactions created by PolySwap for Ethereum are Call transactions. In order to verify that Call Transactions are the majority of transactions in Ethereum, we randomly select 100 blocks from Ethereum mainnet having 11,917 transactions and plot the rates of each type of transaction as shown in Figure 6.6

We observe that the majority of transactions in Ethereum are Call transactions (66.9%), which is the type of transactions created by PolySwap for Ethereum. Thus, PolySwap transactions for Ethereum are indistinguishable from normal transactions. Furthermore, since these transactions are verified by normal signatures and do not contain any information regarding corresponding Bitcoin transactions, these transactions are unlinkable.

6.3 Scalability

For our scalability experiment, we study the run time of PolySwap with respect to the number of blockchains involved in the swap. As PolySwap has three main protocols viz. SSSig, PolyLock and Contingency, we plot the run time of each protocol and sum them to get the total time for PolySwap, excluding the confirmation times of transactions which may vary based on specific blockchain. For our experiments, we calculate run time for different number of blockchains from 2 to 20 with an increment of 2. Run times are considered for swaps between a blockchain in the secp256k1 curve using ECDSA signature algorithm to other blockchains in the Curve25519 curve using the Cryptonote signature algorithm.

The results of the experiment are shown in Figure 6.7. From the figure, we observe that PolyLock is the most expensive protocol in PolySwap while SSSig and Contingency are

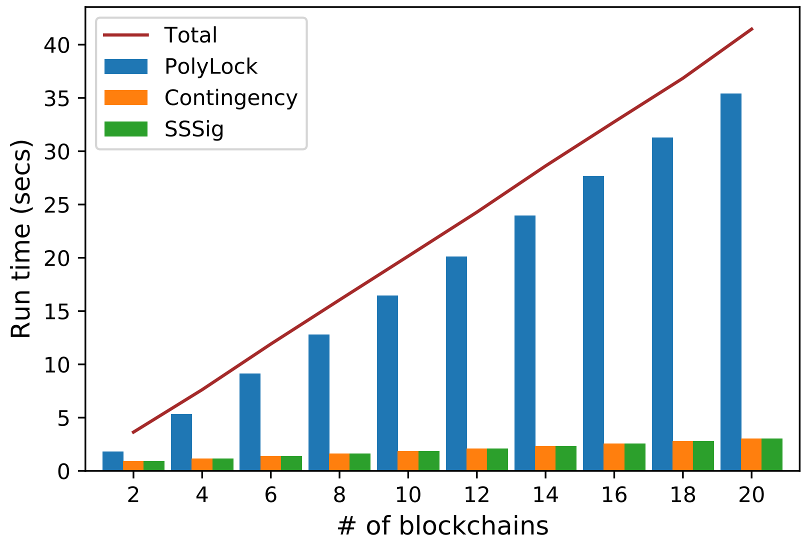
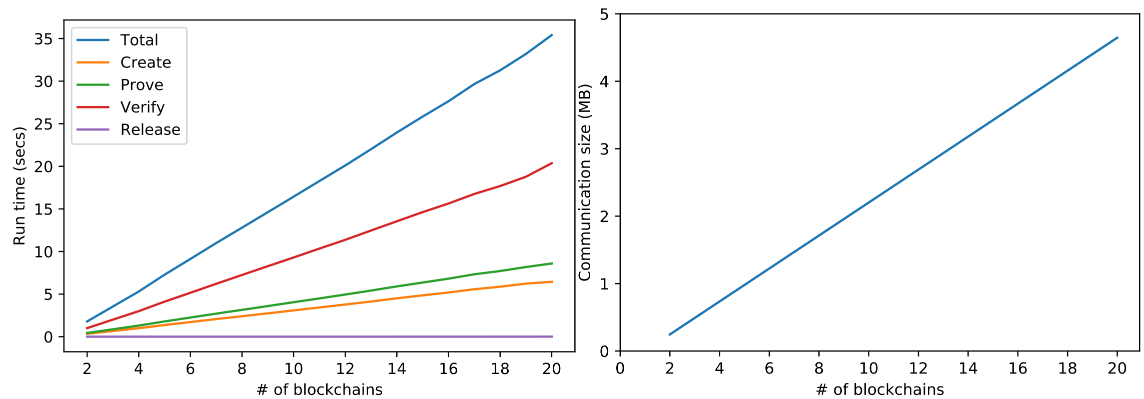


Figure 6.7: Scalability evaluation of PolySwap

negligible in comparison. We also observe that the increase in total run time with respect to the number of blockchains is linear.

Furthermore, we perform additional experiments by observing the performance of PolyLock as it is the most dominant. We study the run time and communication size for PolyLock by the changing the number of blockchains involved in the swap. We consider the worst case scenario, where unlocking secrets from each elliptic curve group used in the blockchain needs to be converted to a common group. We run the algorithm 50 times and obtain the average for a different number of blockchains ranging from 2 to 20 while observing different phases of PolyLock: Create, Prove, Verify and Release. For these experiments, we convert unlocking secrets in Curve25519 to secp256k1. For example, for 4 blockchains, 3 unlocking secrets are in Curve25519 which are converted to secp256k1 during the PolyLock execution.

Figure 6.8 shows scalability evaluation of PolyLock w.r.t run time and communication



(a) Run time (b) Communication size

Figure 6.8: Scalability evaluation of PolyLock

size. We observe that both run time and communication size grow linearly with the linear increase in the number of blockchains involved in the swap. We also observe that *Verify* is the most computationally expensive phase while *Release* is the least expensive w.r.t run time as shown in Figure 6.8a. Figure 6.8b shows that the change in communication size is also linear w.r.t the number of blockchains involved.

6.4 Efficiency

We study the efficiency of PolySwap by benchmarking SSSig. We instantiate our algorithms in secp256k1 elliptic curve group for ECDSA and Schnorr signature-based SSSig with 256 bit key size. In the ECDSA-based SSSig scheme, we use a 2048 bit key size for the Paillier public key pair. For the Cryptonote signature scheme, we instantiate the algorithm in the Curve25519 elliptic curve with 256 bit key size. We use SHA256 to model the functionality $C(x) = \{\mathsf{SHA256}(x||r) \mid r \leftarrow \{0,1\}^{|q|}\}$ as a random oracle [61] for the commitment scheme. We use Fiat-Shamir heuristic [62] for non-interactive zero-knowledge protocols. We consider KeyGen & Psign protocols and Complete & Reveal algorithms for each construction of SSSig. We run 10,000 iterations for each algorithm and compute the overall average for SSSig, which is shown in Table 6.3.

Table 6.3: Efficiency evaluation of SSSig

	ECDSA	**Schnorr**	**Cryptonote**
KeyGen	414.6676 ms 607 bytes	1.1587 ms 33 bytes	0.2914 ms 128 bytes
PSign	80.8083 ms 1165 bytes	3.2141 ms 520 bytes	11.6299 ms 1305 bytes
Complete	0.0593 ms	0.0393 ms	0.0396 ms
Reveal	0.0141ms	0.0002 ms	0.0003 ms

We observe that the KeyGen and PSign protocols for ECDSA are the most dominant. This is due to the fact that Paillier public key cryptography is used. Schnorr is about 25 times faster than ECDSA because it requires lesser computations and simpler Zero Knowledge proofs. This is further supported by the communication size where we see that the overhead for Schnorr is about half of that for ECDSA. Both Complete and Reveal algorithms for all constructions take negligible computation time as they require trivial computation and do not involve any sorts of zero knowledge proofs. Finally, we observe that PSign for Cryptonote-based SSSig has the largest communication overhead while Schnorr has the smallest.

Chapter 7

CONCLUSION AND FUTURE WORK

7.1 Conclusion

In this thesis, we present PolySwap as a generic protocol framework for achieving privacy-preserving atomic swap between two parties over multiple blockchains. PolySwap achieves secure and private swap without requiring any trusted third party and extensive scripting capability in the participating blockchain. PolySwap also provides significant user privacy benefits over HTLC-based atomic swap protocols which tend to be linkable across blockchains and easily distinguished due to the special construction of its transactions. We solve the linkability issue by delegating the atomicity requirement to a secure off-chain two-party computation protocol called PolyLock. And as for distinguishability, we present a novel cryptographic signature scheme called SSSig which enables secure two-party signing producing private outputs of unlocking secrets and public output of a partial signature. These outputs can be combined together to produce a standard signature verifiable by standard verification algorithm over which SSSig is instantiated. As concrete instantiations of SSSig, we present constructions for standard ECDSA, Schnorr and Cryptonote signature scheme which are used by many cryptocurrency systems as their signature algorithm. Because of this, we can create transactions for atomic swap which are indistinguishable from majority of transactions present in a blockchain employing cryptographic signatures for transaction verification. This enables our protocol to be privacy-preserving against a global passive observer providing unlinkability and indistinguishability. PolySwap supports any two sets of blockchains, even if they are heterogenous, given that at least one set supports time-locked escape transactions and a construction for SSSig exists for the signature scheme used in the blockchains. PolySwap does not require any scripting capabilities in the blockchain as long as it supports time-locked escape transactions. With the provided constructions for SSSig, PolySwap currently supports 23 out of the top 30 mainstream cryptocurrencies. We instantiate PolySwap for atomic swaps between two parties over Ethereum & Bitcoin protocol and evaluate it by executing atomic swaps in their respective test networks. Our experiments show that PolySwap takes about 8.3 seconds for successful completion between Testnet3 and Rinkeby without considering confirmation times for Bitcoin-Ethereum atomic swap.

7.2 Future Work

In this work, PolyLock is constructed such that unlocking secrets from each group is converted to the largest common group. Although simpler in construction, this can be less efficient. The protocol can be further optimized to require least number of group conversions by converting to the most common group instead of the largest group. Also, PolySwap uses a number of zero knowledge proofs which tend to be computationally expensive. Investigation into a newer more efficient zero knowledge proofs is required for improving efficiency of PolySwap.

Other SSSig constructions for additional signature schemes, such as EdDSA, can be created to support more existing cryptocurrency systems to increase adoption.

Finally, since PolySwap is privacy-preserving, its applicability as a mixing protocol seems to be an interesting direction for further research.

Bibliography

[1] S. Nakamoto, “Bitcoin: A peer-to-peer electronic cash system,” p. 9.

[2] F. Reid and M. Harrigan, “An Analysis of Anonymity in the Bitcoin System,” *Security and privacy in social networks*, Jul. 2011.

[3] S. Goldfeder, H. Kalodner, D. Reisman, and A. Narayanan, “When the cookie meets the blockchain: Privacy risks of web payments via cryptocurrencies,” *Proceedings on Privacy Enhancing Technologies*, vol. 2018, no. 4, pp. 179–199, Oct. 2018.

[4] E. Androulaki, G. O. Karame, M. Roeschlin, T. Scherer, and S. Capkun, “Evaluating User Privacy in Bitcoin,” in *Financial Cryptography and Data Security*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, vol. 7859, pp. 34–51.

[5] N. van Saberhagen, “CryptoNote white paper,” https://cryptonote.org/whitepaper.pdf, Oct. 2013.

[6] I. Miers, C. Garman, M. Green, and A. D. Rubin, “Zerocoin: Anonymous distributed e-cash from bitcoin,” in *2013 IEEE Symposium on Security and Privacy*. IEEE, May 2013, p. 397–411. [Online]. Available: http://ieeexplore.ieee.org/document/6547123/

[7] Litecoin. https://litecoin.org/.

[8] D. G. Wood, “Ethereum: A secure decentralised generalised transaction ledger,” p. 39, 2019.

[9] Coinmarketcap. [Online]. Available: https://coinmarketcap.com/exchanges/volume/24-hour/all/

[10] Cryptocurrency Exchange Rankings | CoinMarketCap. https://coinmarketcap.com/rankings/exchanges/. Last accessed on 2019-06-06.

[11] Mt. gox. https://en.wikipedia.org/w/index.php?title=Mt._Gox&oldid=894609192.

[12] Bitfinex hack. https://en.bitcoinwiki.org/wiki/Bitfinex_hack.

[13] M. Herlihy, “Atomic cross-chain swaps,” in *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*. ACM, 2018, pp. 245–254.

[14] H. Pagnia, H. Vogt, and F. C. Gärtner, "Fair exchange," *The Computer Journal*, vol. 46, no. 1, pp. 55–75, 2003.

[15] N. Asokan, M. Schunter, and M. Waidner, *Optimistic protocols for fair exchange*. Citeseer, 1996.

[16] H. Pagnia and F. C. Gärtner, "On the impossibility of fair exchange without a trusted third party," Technical Report TUD-BS-1999-02, Darmstadt University of Technology ..., Tech. Rep., 1999.

[17] Alt chains and atomic transfers. https://bitcointalk.org/index.php?topic=193281.msg2224949#msg2224949.

[18] J. Poon and T. Dryja, "The bitcoin lightning network," 2016.

[19] L. Gudgeon, P. Moreno-Sanchez, S. Roos, P. McCorry, and A. Gervais, "Sok: Off the chain transactions," 2019. [Online]. Available: http://eprint.iacr.org/2019/360

[20] G. Malavolta, P. Moreno-Sanchez, A. Kate, M. Maffei, and S. Ravi, "Concurrency and privacy with payment-channel networks," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 455–471.

[21] G. Malavolta, P. Moreno-Sanchez, C. Schneidewind, A. Kate, and M. Maffei, "Anonymous multi-hop locks for blockchain scalability and interoperability," in *Proceedings 2019 Network and Distributed System Security Symposium*. Internet Society, 2019. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2019/02/ndss2019_09-4_Malavolta_paper.pdf

[22] Zero Knowledge Contingent Payment - Bitcoin Wiki. https://en.bitcoin.it/wiki/Zero_Knowledge_Contingent_Payment.

[23] M. Campanelli, R. Gennaro, S. Goldfeder, and L. Nizzardo, "Zero-knowledge contingent payments revisited: Attacks and payments for services," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security - CCS '17*. ACM Press, 2017, pp. 229–243. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3133956.3134060

[24] M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek, "Fair two-party computations via bitcoin deposits," in *Financial Cryptography and Data Security*, vol. 8438. Springer Berlin Heidelberg, 2014, pp. 105–121. [Online]. Available: http://link.springer.com/10.1007/978-3-662-44774-1_8

[25] S. Dziembowski, L. Eckey, and S. Faust, "Fairswap: How to fairly exchange digital goods," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and*

Communications Security - CCS '18. ACM Press, 2018, p. 967–984. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3243734.3243857

[26] A. Shamir, "How to share a secret," *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, Nov. 1979.

[27] G. Malavolta and S. A. K. Thyagarajan, "Homomorphic Time-Lock Puzzles and Applications," in *Annual International Cryptology Conference*. Springer, 2019, pp. 620–649.

[28] T. P. Pedersen, "Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing," in *Advances in Cryptology — CRYPTO '91*, J. Feigenbaum, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, vol. 576, pp. 129–140.

[29] S. Goldwasser, S. Micali, and C. Rackoff, "The knowledge complexity of interactive proof systems," *SIAM Journal on computing*, vol. 18, no. 1, pp. 186–208, 1989.

[30] Y. Lindell, "Fast secure two-party ECDSA signing," in *Annual International Cryptology Conference*. Springer, 2017, pp. 613–644.

[31] D. Johnson, A. Menezes, and S. Vanstone, "The Elliptic Curve Digital Signature Algorithm (ECDSA)," *International Journal of Information Security*, vol. 1, no. 1, pp. 36–63, Aug. 2001.

[32] S. Goldfeder, J. Bonneau, R. Gennaro, and A. Narayanan, "Escrow protocols for cryptocurrencies: How to buy physical goods using bitcoin," in *Financial Cryptography*, 2017, pp. 321–339.

[33] W. Banasik, S. Dziembowski, and D. Malinowski, "Efficient zero-knowledge contingent payments in cryptocurrencies without scripts," in *European Symposium on Research in Computer Security*. Springer, 2016, pp. 261–280.

[34] F. Tramer, F. Zhang, H. Lin, J.-P. Hubaux, A. Juels, and E. Shi, "Sealed-glass proofs: Using transparent enclaves to prove and sell knowledge," 2016. [Online]. Available: http://eprint.iacr.org/2016/635

[35] I. Bentov and R. Kumaresan, "How to use bitcoin to design fair protocols," in *Annual Cryptology Conference*, no. 129. Springer, 2014, pp. 421–439. [Online]. Available: http://eprint.iacr.org/2014/129

[36] R. Kumaresan, V. Vaikuntanathan, and P. N. Vasudevan, "Improvements to secure computation with penalties," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS'16*. ACM Press, 2016, p. 406–417. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2976749.2978421

[37] R. Kumaresan and I. Bentov, "Amortizing secure computation with penalties," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS'16*. ACM Press, 2016, p. 418–429. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2976749.2978424

[38] A. Miller, I. Bentov, R. Kumaresan, C. Cordi, and P. McCorry, "Sprites and state channels: Payment networks that go faster than lightning." [Online]. Available: http://arxiv.org/abs/1702.05812

[39] P. McCorry, E. Heilman, and A. Miller, "Atomically trading with roger: Gambling on the success of a hardfork," in *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, J. Garcia-Alfaro, G. Navarro-Arribas, H. Hartenstein, and J. Herrera-Joancomartí, Eds. Springer International Publishing, 2017, vol. 10436, pp. 334–353. [Online]. Available: http://link.springer.com/10.1007/978-3-319-67816-0_19

[40] M. Green and I. Miers, "Bolt: Anonymous payment channels for decentralized currencies," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security - CCS '17*. ACM Press, 2017, p. 473–489. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3133956.3134093

[41] E. Heilman, L. AlShenibr, F. Baldimtsi, A. Scafuro, and S. Goldberg, "Tumblebit: An untrusted bitcoin-compatible anonymous payment hub," in *Proceedings 2017 Network and Distributed System Security Symposium*. Internet Society, 2017. [Online]. Available: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/tumblebit-untrusted-bitcoin-compatible-anonymous-payment-hub/

[42] Z. Liu, Y. Xiang, J. Shi, P. Gao, H. Wang, X. Xiao, B. Wen, and Y.-C. Hu, "HyperService: Interoperability and Programmability Across Heterogeneous Blockchains," *arXiv:1908.09343 [cs]*, Aug. 2019.

[43] A. Zamyatin, D. Harz, J. Lind, P. Panayiotou, A. Gervais, and W. J. Knottenbelt, "X CLAIM : Interoperability with Cryptocurrency-Backed Tokens ?" 2018.

[44] E. Heilman, S. Lipmann, and S. Goldberg, "The arwen trading protocols," p. 21, January 2019. [Online]. Available: https://arwen.io/whitepaper.pdf

[45] S. Thomas and E. Schwartz, "A Protocol for Interledger Payments," p. 25.

[46] P. Gazi, A. Kiayias, and D. Zindros, "Proof-of-Stake Sidechains," in *IEEE Symposium on Security and Privacy*. IEEE, 2019.

[47] S. Delgado Segura, C. Pérez-Solà, G. Navarro-Arribas, and J. Herrera-Joancomartí, "A fair protocol for data trading based on bitcoin transactions," *Future Generation Computer Systems*, 08 2017.

[48] Atomic swap - bitcoin wiki. https://en.bitcoin.it/wiki/Atomic_swap. Online; accessed 2018-12-20.

[49] W. Mao, "Timed-release cryptography," in *Selected Areas in Cryptography*, S. Vaudenay and A. M. Youssef, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 342–357.

[50] L. Wang, X. Shen, J. Li, J. Shao, and Y. Yang, "Cryptographic primitives in blockchains," *Journal of Network and Computer Applications*, vol. 127, pp. 43–58, Feb. 2019.

[51] Atomic Swaps and Distributed Exchanges: The Inadvertent Call Option. BitMEX Blog.

[52] Ferrin, "A Preliminary Field Guide for Bitcoin Transaction Patterns Danno," 2015.

[53] The Legion of the Bouncy Castle Java Cryptography APIs. https://www.bouncycastle.org/java.html.

[54] Bitcoinj. https://bitcoinj.github.io/.

[55] Briandilley/jsonrpc4j. https://github.com/briandilley/jsonrpc4j.

[56] Web3j SDK - Where Java meets the Blockchain. https://www.web3labs.com/web3j.

[57] Solidity — Solidity 0.6.3 documentation. https://solidity.readthedocs.io/en/v0.6.3/.

[58] Ethereum API — IPFS API & Gateway — ETH Nodes as a Service. https://infura.io/.

[59] Bitcoin/bitcoin. https://github.com/bitcoin/bitcoin.

[60] Segregated witness (consensus layer). https://github.com/bitcoin/bips/blob/master/bip-0141.mediawiki.

[61] M. Bellare and P. Rogaway, "Random oracles are practical: A paradigm for designing efficient protocols," in *Proceedings of the 1st ACM conference on Computer and communications security*. ACM, 1993, pp. 62–73.

[62] A. Fiat and A. Shamir, "How to prove yourself: Practical solutions to identification and signature problems," in *Conference on the Theory and Application of Cryptographic Techniques*. Springer, 1986, pp. 186–194.

Appendix A

ETHEREUM TIME-LOCK SOLIDITY SOURCE CODE

```
pragma solidity >=0.5.0;
contract TimeLock {

  address public owner;
  uint256 public unlockTime;
  uint256 public timeNow;

  modifier onlyOwner {
    require(msg.sender == owner);
    _;
  }

  constructor(address own, uint8 timePeriod) public {
    owner = own;
    unlockTime = now + (timePeriod * 1 minutes);
    timeNow = now;
    emit Deposit(timePeriod, unlockTime, timeNow, own);
  }

  function refund(address payable receiver) onlyOwner public {
    require(now >= unlockTime);
    receiver.transfer(address(this).balance);
    emit Refund(receiver, address(this).balance);
  }

  function claim(address payable receiver) onlyOwner public {
    receiver.transfer(address(this).balance);
    emit Claim(receiver, address(this).balance);
  }

```

```
    //receive any Ether sent to this contract
    function() payable external{
    }

    //log events definition in the transactions
    event Deposit(uint timePeriod, uint unlockTime, uint timeNow, address indexed owner);
    event Refund(address indexed to, uint amount);
    event Claim(address indexed payedTo, uint amount);
}
```