DECENTRALIZED PRIVACY-PRESERVING COLLECTIVE AND MULTI-OBJECTIVE TRADING PROTOCOLS ON BLOCKCHAIN WITH ZERO-KNOWLEDGE PROOFS

by

Goshgar Can Ismayilov

B.S., Computer Engineering, Marmara University, 2016

M.S., Computer Engineering, Marmara University, 2019

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Doctor of Philosophy

Graduate Program in Computer Engineering

Boğaziçi University

2025

ACKNOWLEDGEMENTS

Finis coronat opus! (*The end crowns the work!*). I hereby declare the completion of a never-ending but evenly enlightening journey—a journey that is largely filled with learning, unlearning and relearning. I would like to use this opportunity to express my gratitude to all those who have guided, inspired and supported me along the road.

First and foremost, I would like to express my deepest gratitude to my advisor, Prof. Can Özturan, for guiding and supporting me throughout all these years. His practical solutions to the problems, insightful discussions and timely decisions have brought all of this effort to the fruition. I am still vividly remembering the day he suggested me to explore blockchain and zero-knowledge proof in his office like yesterday—a simple spark that opened the door to a fascinating world of research.

I would extend my sincere gratitude to the doctoral committee members; Prof. Öznur Özkasap, Prof. Alper Şen, Assoc. Prof. Şerif Bahtiyar and Assoc. Prof. Atay Özgövde for their gracious participation and significant feedback to contribute to the depth and quality of this work. With that respect, I would also like to thank Bogazici University for providing such an intellectually-stimulating research environment. I also owe to those who came before me. This work stands on your shoulders.

Lastly but above all, I want to warmly thank my family, Prof. Zameddin Ismailov and Feride Resulova for everything they have quietly done for me. Their unconditional support and faith have truly shaped me into the person that I am today.

ABSTRACT

DECENTRALIZED PRIVACY-PRESERVING COLLECTIVE AND MULTI-OBJECTIVE TRADING PROTOCOLS ON BLOCKCHAIN WITH ZERO-KNOWLEDGE PROOFS

In this thesis, we propose decentralized privacy-preserving cryptographic protocols on blockchain with zero-knowledge proofs for three essential problems. Firstly, the *privacy-preserving payment* problem refers to a specific group of transactions where a source address (i.e. sender) transfers a certain amount of tokens to a destination address (i.e. receiver) while still protecting the privacy of their balances and transaction details. We extend this problem to address multi-token payments as well. Secondly, the *privacy-preserving aggregation* problem refers to a multi-party computation where a group of blockchain addresses (i.e. aggregators) aggregate their data to reach global aggregation by protecting the privacy of their own data. We also extend this problem for prefix aggregation and to support for arbitrary numbers of aggregators. Thirdly, *privacy-preserving multi-token bartering-based trading* problem refers to a multi-party computation where a group of blockchain addresses (i.e. barterers) collectively exchanges a set of tokens for another set of tokens via proposing bids by protecting the privacy of their balances and bids. We extend this problem as multi-objective bartering using Bellman-Ford and pareto-domination. We propose the protocols of *PTTS* for the first problem, *PVSS* and *PRFX* for the second problem and *PMTBS* and *zkMOBF* for the third problem. We analyze the protocols in terms of their scalability (computational, communication and storage overheads) and their security (potential attacks and reduction proofs). We perform experimental study on Ethereum and Avalanche to measure gas consumption, proof generation/verification times and proof artifact size.

ÖZET

SIFIR BİLGİ İSPATLARIYLA BLOKZİNCİR ÜZERİNDE MERKEZİYETSİZ, MAHREMİYET KORUYUCU, KOLEKTİF VE ÇOK AMAÇLI PROTOKOLLER

Bu tezde, üç temel problem için sıfır bilgi ispatlarıyla desteklenen merkeziyetsiz, mahremiyet korumalı kriptografik protokoller tasarladık. İlk olarak, mahremiyet korumalı ödeme problemi, bir kaynak adresin (gönderici) bir hedef adrese (alıcı) belirli bir miktarda jeton transfer ettiği, ancak bakiye ve işlem detaylarının gizliliğinin korunduğu özel bir işlem grubunu ifade etmektedir. Bu problemi, çoklu jeton ödemelerini de kapsayacak şekilde genişlettik. İkinci olarak, mahremiyet korumalı toplama problemi ise, bir grup blokzincir adresinin (toplayıcılar) kendi bireysel verilerini gizliliklerini koruyarak nihai bir toplam değere ulaşmak amacıyla birleştirdiği güvenli çoklu taraf hesaplamasını ifade etmektedir. Bu problemi de, önek toplamayı ve rastgele sayıda toplayıcı desteğini içerecek şekilde genişlettik. Üçüncü olarak, mahremiyet korumalı çoklu jeton takasına dayalı ticaret problemi, bir grup blokzincir adresinin (takasçılar) belirli bir jeton grubunu başka bir jeton grubu karşılığında kolektif olarak değiştirdiği ve teklif vererek müzayede sürecine katıldığı güvenli çoklu taraf hesaplamasını ifade etmektedir; ki burada, bakiye ve teklif gizliliği korunmaktadır. Bu problemi de, Bellman-Ford ve pareto-dominasyonu kullanarak çok amaçlı takasları içerecek şekilde genişlettik. İlk problem için *PTTS*, ikinci problem için *PVSS* ve *PRFX* ve üçüncü problem için *PMTBS* ve *zkMOBF* protokollerini tasarladık. Bu protokolleri ölçeklenebilirlik (hesaplama, iletişim ve depolama) ve güvenlik (saldırılar ve indirgeme ispatları) açısından analiz ettik. Ethereum ve Avalanche üzerinde deneyler gerçekleştirerek, bu protokollerin blokzincir gaz tüketimini, ispat oluşturma/doğrulama sürelerini ve ispat çıktı boyutlarını ölçülmüştür.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iii
ABSTRACT iv
ÖZET v
LIST OF FIGURES x
LIST OF TABLES xvi
LIST OF SYMBOLS xix
LIST OF ACRONYMS/ABBREVIATIONS xxii
1. INTRODUCTION 1
1.1. Motivation 5
1.2. Contributions 8
2. BACKGROUND 11
2.1. On Blockchain 11
2.2. On Token Standards 16
2.3. On Hypercube Network Topology 19
2.4. On Network Flows 20
2.5. On Multi-Objective Optimization 21
2.6. On Commitment Schemes 23
2.7. On Public-Key Cryptography 25
2.8. On Zero-Knowledge Proof 29
2.9. On ZoKrates Framework 35
3. RELATED WORK 37
3.1. On Privacy-Preserving Token Transfer 37
3.2. On Privacy-Preserving Data Aggregation 40
3.3. On Token Bartering 44
4. PROTOCOL I: PRIVACY-PRESERVING PAYMENT 48
4.1. Problem Definition 48
4.2. System Architecture 52
4.2.1. Zero-Knowledge Proof Model 58

4.2.2. Smart Contract Model 60
4.2.3. Web Interface Model 63
4.3. Protocol Analysis 67
4.3.1. Scalability 67
4.3.1.1. Computational Overhead 68
4.3.1.2. Communication Overhead 68
4.3.1.3. Storage Overhead 69
4.3.2. Security 70
4.3.2.1. Reduction Proof 71
4.3.3. Limitations 72
4.4. Balance Range Disclosure Attack 74
4.4.1. Minimum Cost Flow Network 75
4.4.2. Contiguity in Balance Ranges 77
4.5. Experimental Evaluation 78
4.5.1. Experimental Setup 78
4.5.2. Requirement Verification 80
4.5.3. For Privacy-Preserving Token Transfer 81
4.5.3.1. Blockchain Gas Consumption 81
4.5.3.2. Proof Generation/Verification Times 83
4.5.3.3. Proof Artifact Sizes 84
4.5.4. For Balance Range Disclosure Attack 84
4.5.4.1. Varying Number of Addresses and Transactions 86
4.5.4.2. Varying Transaction Leakage Ratio 87
5. PROTOCOL II: PRIVACY-PRESERVING DATA AGGREGATION 90
5.1. Problem Definition 90
5.2. System Architecture 94
5.2.1. Zero-Knowledge Proof Model 103
5.2.2. Smart Contract Model 104
5.2.3. Web Interface Model 108
5.3. Extension to Incomplete Hypercube Networks 110
5.3.1. On Underdetermined System 111

5.3.1.1. Hypergraph Data Representation 113
5.3.2. Communication Techniques on Incomplete Hypercube Networks 114
5.3.2.1. Node Multiplexing 114
5.3.2.2. Topological Recursing 116
5.3.2.3. Data Splitting 118
5.4. Extension to Privacy-Preserving Prefix Aggregation 121
5.4.1. On Delegation Scheme 124
5.5. Protocol Analysis 129
5.5.1. Scalability 129
5.5.1.1. Computational Overhead 129
5.5.1.2. Communication Overhead 130
5.5.1.3. Storage Overhead 130
5.5.2. Security 132
5.5.2.1. Reduction Proof 133
5.5.3. Limitations 134
5.6. Experimental Evaluation 135
5.6.1. Experimental Setup 136
5.6.2. For Privacy-Preserving Data Aggregation 136
5.6.2.1. Requirement Verification 136
5.6.2.2. Blockchain Gas Consumption 137
5.6.2.3. Proof Generation/Verification Times 141
5.6.2.4. Proof Artifact Sizes 142
5.6.3. For Privacy-Preserving Prefix Aggregation 142
5.6.3.1. Blockchain Gas Consumption 142
5.6.3.2. Proof Generation/Verification Times 143
5.6.3.3. Proof Artifact Sizes 144
6. PROTOCOL III: PRIVACY-PRESERVING TOKEN BARTERING 146
6.1. Problem Definition 146
6.2. System Architecture 156
6.2.1. Zero-Knowledge Proof Model 166
6.2.2. Smart Contract Model 169

6.2.3. Web Interface Model 177
6.3. Extension to Privacy-Preserving Multi-Token Transfer 181
6.4. Protocol Analysis 182
6.4.1. Scalability 182
6.4.1.1. Computational Overhead 182
6.4.1.2. Communication Overhead 183
6.4.1.3. Storage Overhead 184
6.4.2. Security 185
6.4.2.1. Reduction Proof 186
6.4.3. Limitations 187
6.5. Experimental Evaluation 189
6.5.1. Experimental Setup 189
6.5.2. Requirement Verification 189
6.5.3. Blockchain Gas Consumption 191
6.5.4. Proof Generation/Verification Times 193
6.5.5. Proof Artifact Sizes 194
7. EXTENSION TO MULTI-OBJECTIVE BARTERING 196
7.1. Problem Definition 196
7.2. System Architecture 199
7.2.1. Detecting Cycles with Bellman-Ford Algorithm 200
7.2.2. Evaluating Feasible Cycles with Objectives 201
7.2.3. Ranking Feasible Solutions with Non-Dominated Sorting 202
7.2.4. Decision-Making for Final Solution with Utility Function 202
7.3. Current Limitation 203
7.4. Experimental Study 204
7.4.1. Zero-Knowledge Proof Generation/Verification Times 204
7.4.2. Proof Artifact Size 205
7.4.3. Blockchain Gas Consumption 205
8. CONCLUSION AND FUTURE WORK 207
REFERENCES 211
APPENDIX A: COPYRIGHT NOTICE 225

LIST OF FIGURES

Figure 1.1. Layered architecture of our protocols. 10

Figure 2.1. Blockchain as a chain of blocks. 14

Figure 2.2. Merkle tree for blockchain transactions. 16

Figure 2.3. Hypercube network topology in 3-dimensions. 20

Figure 2.4. Minimum cost circulation for bartering problem [60]. 21

Figure 2.5. Multi-objective optimization with two objectives: (blue): optimal solutions (gray): non-optimal solutions. 23

Figure 2.6. Commitment scheme on blockchain. 25

Figure 2.7. One-way functions with trapdoors. 27

Figure 2.8. Public key cryptography on blockchain: confidentiality. 28

Figure 2.9. Public key cryptography on blockchain: authentication. 28

Figure 2.10. Zero-Knowledge proof on blockchain. 34

Figure 2.11. Architecture of ZoKrates framework [27]. 36

Figure 3.1. Architecture of Masquerade [75] with trusted parties. 42

Figure 4.1. State-transition diagram of *PTTS* protocol. 53

Figure 4.2. Architecture of *PTTS* protocol. 54

Figure 4.3. Sequence diagram of *PTTS* protocol. 58

Figure 4.4. Proof implementation in ZoKrates [27] for depositing tokens. . . . 59

Figure 4.5. Proof implementation in ZoKrates [27] for withdrawing tokens. . . 60

Figure 4.6. Contract implementation for *requesting consent* phase. 61

Figure 4.7. Contract implementation for *granting consent* phase. 61

Figure 4.8. Contract implementation for *depositing tokens* phase. 62

Figure 4.9. Contract implementation for *withdrawing tokens* phase. 63

Figure 4.10. Web user interface of *PTTS* protocol. 64

Figure 4.11. Interface implementation for *deploying contract* phase. 65

Figure 4.12. Interface implementation for *requesting consent* phase. 65

Figure 4.13. Interface implementation for *granting consent* phase. 65

Figure 4.14. Interface implementation for *depositing tokens* phase. 66

Figure 4.15. Interface implementation for *withdrawing tokens* phase. 67

Figure 4.16. Balance range disclosure networks: (white) source, (green) sink, (blue) ordinary users, (red) user to be attacked. 77

Figure 4.17. The minimum cost flow network: (black) source and sink addresses, (orange) ordinary addresses, (red) address to be attacked. 85

Figure 4.18. Transaction graph: (black) source and sink addresses, (orange) ordinary addresses, (red) address to be attacked. 86

Figure 5.1. State-transition diagram of *PVSS* protocol. 95

Figure 5.2. Architecture of *PVSS* protocol. 95

Figure 5.3. Sequence diagram of *PVSS* protocol. 101

Figure 5.4. Aggregations of *PVSS* protocol in 3-dimensional hypercube network.102

Figure 5.5. Proof implementation in ZoKrates [27] for aggregating data. . . . 104

Figure 5.6. Contract implementation for *registering* phase. 105

Figure 5.7. Contract implementation for *submitting aggregation* phase. 106

Figure 5.8. Contract implementation for *verifying aggregation* phase. 107

Figure 5.9. Web user interface of *PVSS* protocol. 108

Figure 5.10. Interface implementation for *registering* phase. 109

Figure 5.11. Interface implementation for *submitting aggregation* phase. 110

Figure 5.12. Interface implementation for *verifying aggregation* phase. 110

Figure 5.13. Undeterminacy on *PVSS* protocol. 113

Figure 5.14. Hypergraph data representation of globally underdetermined system.114

Figure 5.15. *Node multiplexing* technique on *PVSS* protocol. 116

Figure 5.16. *Node multiplexing* technique on *PVSS* protocol. 116

Figure 5.17. *Topological recursing* technique on *PVSS* protocol. 118

Figure 5.18. *Topological recursing* technique on *PVSS* protocol. 118

Figure 5.19. *Data splitting* technique on *PVSS* protocol. 120

Figure 5.20. *Data splitting* technique on *PVSS* protocol. 120

Figure 5.21. Communication of aggregators in *PRFX* protocol. 123

Figure 5.22. Delegation of tokens in *PRFX* protocol. 124

Figure 5.23. Transformation of global directed graph with Euler our Technique. 126

Figure 5.24. Linearization of global directed graph with Euler Tour Technique. 126

Figure 5.25. Privacy-preserving prefix summation with *PRFX* protocol. 127

Figure 5.26. Proof implementation in ZoKrates [27] for aggregating prefix data. 128

Figure 6.1. Illustration for ascending auction bartering (with four bids $(\phi_0, \phi_1, \phi_2, \phi_3)$ and four tokens $(\tau_0, \tau_1, \tau_2, \tau_3)$). 153

Figure 6.2. Bid aggregations of *PMTBS* protocol in 2-dimensional hypercube networks. 155

Figure 6.3. State-transition diagram of *PMTBS* protocol. 157

Figure 6.4. Architecture of *PMTBS* protocol. 158

Figure 6.5. Bid aggregation with vector summation in *PMTBS* protocol. . . . 163

Figure 6.6. Sequence diagram of *PMTBS* protocol. 166

Figure 6.7. Proof Implementation in ZoKrates [27] for (Re)Proposing Bid. . . 167

Figure 6.8. Proof implementation in ZoKrates [27] for aggregating bids. 168

Figure 6.9. Proof implementation in ZoKrates [27] for bartering tokens. 169

Figure 6.10. Contract implementation for *proposing bid* phase. 171

Figure 6.11. Contract implementation for *submitting bid aggregation* Phase. . . 171

Figure 6.12. Contract implementation for *verifying bid aggregation* phase. . . . 173

Figure 6.13. Contract implementation for *reproposing bid* phase. 174

Figure 6.14. Contract implementation for *bartering tokens* phase. 175

Figure 6.15. Ambiguity of bid satisfaction in *PMTBS* protocol. 176

Figure 6.16. Comparison of ERC-20 token transfer with *PMTBS* token transfer. 177

Figure 6.17. Web user interface of *PMTBS* protocol. 178

Figure 6.18. Interface implementation for *proposing bid* phase. 179

Figure 6.19. Interface implementation for *reproposing bid* phase. 180

Figure 6.20. Interface implementation for *bartering tokens* phase. 180

Figure 6.21. Extension of $PTTS$ to privacy-preserving multi-token payment. . . 182

Figure 7.1. A simple illustration for multi-objective bartering with six bids $(\phi_0, \phi_1, \phi_2, \phi_3, \phi_4, \phi_5)$ 199

Figure 7.2. Multi-objective bartering implementation in ZoKrates framework [27]. 200

Figure 7.3. Negative cycle detection with Bellman-Ford 201

Figure 7.4. Transformation of solutions to objective values 201

Figure 7.5. Non-dominated ranking of solutions for POS 202

Figure 7.6. Selection of best solution from optimal solutions 203

LIST OF TABLES

Table 2.1. ERC-20 token standard interface. 17

Table 2.2. ERC-721 token standard interface. 18

Table 2.3. ERC-1155 token standard interface. 19

Table 2.4. Schnorr's protocol as sigma protocol. 31

Table 4.1. The *PTTS* protocol 57

Table 4.2. *PTTS* communication, computation and storage overheads between 2 users (i.e. sender and receiver). 69

Table 4.3. *PTTS* verification of problem requirements. 82

Table 4.4. Blockchain gas consumption of the *PTTS* protocol phases. 83

Table 4.5. *PTTS* zero-Knowledge proof artifact size. 84

Table 4.6. Network solution times and certainty levels for varying number of addresses and transactions where transaction leakage ratio is 0.5. . 87

Table 4.7. Network solution times and certainty levels for varying number of addresses and transactions where transaction leakage ratio is 0.5. . 89

Table 5.1. The *PVSS* protocol 100

Table 5.2. Comparison of communication techniques on incomplete hypercube networks. 121

Table 5.3. *PVSS* scalability as communication, computation and storage overheads. 131

Table 5.4. *PRFX* scalability as communication, computation and storage overheads. 131

Table 5.5. *PVSS* verification of problem requirements. 138

Table 5.6. Blockchain gas consumption of the *PVSS* protocol phases. 140

Table 5.7. *PVSS* zero-knowledge proof generation times (in seconds). 141

Table 5.8. *PVSS* zero-knowledge proof artifact size. 142

Table 5.9. Blockchain gas consumption of *PRFX* protocol phases. 143

Table 5.10. *PRFX* zero-knowledge proof generation times (in seconds). 144

Table 5.11. *PRFX* zero-knowledge proof artifact size. 145

Table 6.1. The *PMTBS* protocol 165

Table 6.2. *PMTBS* scalability as communication, computation and storage overheads. 184

Table 6.3. *PMTBS* verification of problem requirements. 191

Table 6.4. Blockchain gas consumption of the *PMTBS* protocol phases. . . . 193

Table 6.5. *PMTBS* zero-knowledge proof generation times (in seconds). . . . 194

Table 6.6. *PMTBS* zero-knowledge proof artifact size. 195

Table 7.1. *zkMOBF* zero-knowledge proof generation/verification times. . . . 205

Table 7.2. *zkMOBF* zero-knowledge proof artifact size. 206

Table 7.3. Blockchain gas aonsumption of *zkMOBF* approach. 206

LIST OF SYMBOLS

b_i	Proof verification of ith user (true/false)
c_i^{χ}	Commitment of ith user on a private data χ
$\mathcal{B}_i$	ith block
$\mathcal{BC}$	Blockchain
D	Decryption output
E	Encryption output
$F(.)$	Multi-objective optimization function
$\mathcal{E}$	Edges (Transactions)
$\mathcal{E}^*$	Transactions Leaked
$\mathcal{G}$	Graph
$\mathtt{h}$	Hypercube network communication step
$\mathcal{H}_i$	ith block header
$\mathtt{I(.)}$	Mutual information function
M	Number of total tokens
$\mathcal{M}$	Balance range disclosure attack algorithm
N	Number of total users
$\mathcal{P}$	Prover in zero-knowledge proof
pk_i	Public key of ith user
pok	Zero-knowledge proving key
sk_i	Secret key of ith user
$\mathcal{S}im$	Zero-knowledge proof simulator
tx_{ij}	Regular transaction between ith and jth users
tx_i^+	Minting transaction to ith user
tx_i^-	Burning transaction from ith user
Tx	Set of all transactions
u_i	ith user
$\mathcal{U}$	Set of all users
$\mathcal{V}$	Vertices (Addresses)

$\mathcal{V}$	Verifier in zero-knowledge proof
vek	Zero-knowledge verification key
view(u_i)	Set of information that ith user can access to
γ	Blockchain gas consumption of operation
Δ	Transaction amount
ϵ	Generic random masking number
η	Attack certainty level
$\theta_{i,t}$	Balance of ith user at time t
Θ	Set of all user balances
λ	Generic security parameter
π_i^{op}	Proof of ith user for operation op
$\xi_{comp}/\xi_{comp}^{\Sigma}$	Computational individual/system overhead
$\xi_{comm}/\xi_{comm}^{\Sigma}$	Communication individual/system overhead
$\xi_{mem}/\xi_{mem}^{\Sigma}$	Storage individual/system overhead
σ	Generic salting parameter
Σ	Aggregation
τ	Token type
$\phi_{i,j,t}^{-}$	jth token of ith bid to be supplied at time t
$\phi_{i,j,t}^{+}$	jth token of ith bid to be demanded at time t
ϕ_i^{cost}	cost of ith bid
Φ	Set of all bids
χ	Any secret data
X	Set of all secret data
Ψ^{op}	Computation of operation op to prove
$\|.\|_0$	ℓ_0-norm function
`Hash(.)`	Hash function
`Cm.Comm(.)`	Commitment scheme commit function
`Cm.Vfy(.)`	Commitment scheme verification function
`Pk.Dec(.)`	Public-key cryptography decryption function

`Pk.Enc(.)`	Public-key cryptography encryption function
`Pk.Setup(.)`	Public-key cryptography setup function
`Pk.Sign(.)`	Public-key cryptography signature function
`Pk.Vfy(.)`	Public-key cryptography signature verification function
`Pr(.)`	Probability function
`Zk.Gen(.)`	Zero-knowledge proof generation function
`Zk.Setup(.)`	Zero-knowledge setup function
`Zk.Vfy(.)`	Zero-knowledge proof verification function

LIST OF ACRONYMS/ABBREVIATIONS

ACE	AZTEC Cryptographic Engine
AVAX	Avalanche Native Currency
AZTEC	Anonymous Zero-Knowledge Transactions with Efficient Communication
BFT	Byzantine Fault Tolerant
BSS	Basic Secure Summation
DApp	Decentralized Application
DDPG	Deep Deterministic Policy Gradient
DPoS	Delegated Proof-of-Stake
DSS	Distributed Secure Summation
ECIES	Elliptic Curve Integrated Encryption Scheme
ERC-20	Ethereum Request for Comment-20 for Fungible Tokens
ERC-721	Ethereum Request for Comment-721 for Non-Fungible Tokens
ERC-1155	Ethereum Request for Comment-1155 for Multi-Tokens
ETH	Ether - Ethereum Native Currency
ETT	Euler Tour Technique
ESS	Encrypted Secure Summation
EVM	Ethereum Virtual Machine
Geth	Go-Ethereum
Gwei	10^{-9} of 1 Ether
HC	Hypercube Network Topology
HSS	Homomorphically-Shared Secure Summation
HTTP	Hyper-Text Transfer Protocol
HTTPS	Secure Hyper-Text Transfer Protocol
IDE	Integrated Development Environment
IoT	Internet-of-Things
MOO	Multi-Objective Optimization
MPC	Multi-Party Computation

NFT	Non-Fungible Token
OR	Operation Research
PBFT	Practical Byzantine Fault Tolerant
PlonK	Permutations over Lagrange-bases for Oecumenical Non-interactive arguments of Knowledge
PMTBS	Privacy-Preserving Multi-Token Bartering System
POF	Pareto-Optimal Front
PoS	Proof-of-Stake
POS	Pareto-Optimal Set
PoW	Proof-of-Work
PRFX	Privacy-Preserving Prefix Summation System
PTTS	Privacy-Preserving Token Transfer System
PVSS	Privacy-Preserving Value Summation System
R1CS	Rank-1 Constraint System
RSS	Randomly-Shared Secure Summation
SHA256	Secure Hash Algorithm 256
SHC	Sub-Hypercube Network Topology
SMPC	Secure Multi-Party Computation
SSS	Salted Secure Summation
SPDZ	Secure Multi-Party Computation (MPC) Protocol with Dishonest Majority and Zero-Knowledge Proofs
TSS	Two-Segment Secure Summation
UAV	Unmanned Aerial Vehicle
UTXO	Unspent Transaction Output
USD	United States Dollar
Wei	10^{-18} of 1 Ether
zkMOBF	Zero-Knowledge based Multi-Objective Bellman-Ford
zkSNARKs	Zero-Knowledge Succinct Non-Interactive Argument of Knowledge
zkSTARKs	Zero-Knowledge Scalable Transparent Argument of Knowledge

1. INTRODUCTION

Blockchain is simply a decentralized ledger technology that is shared across all the available nodes in the network. For the first time, it was proposed in the white paper of S. Nakamoto in 2008 as *Bitcoin: A Peer-to-Peer Electronic Cash System* for secure, transparent, immutable and traceable records of transactions over decentralized and fault-tolerant network architecture [1]. Bitcoin has attracted significant attention with its high potential to revolutionize a wide range of industries. However, this initial idea of blockchain as a decentralized payment and cryptocurrency system at the time specifically suffered from lack of a proper scripting language that supports a diverse set of computations (i.e. lack of Turing-completeness). To address this issue, the white paper [2] of V. Buterin in 2013 proposes *Ethereum* by extending *Bitcoin* to support programming decentralization applications over smart contracts. Not just Ethereum, but also different blockchain platforms have been emerging over time for other blockchain problems as well including *Avalanche* [3] to process transactions faster with high throughput, *IOTA* [4] to securely exchange and store data of internet-of-things on graph-based architecture; and Zerocash [5] to introduce privacy over transactions with zero-knowledge proof. Nevertheless, there still exist open blockchain issues and research challenges that are waiting to be rigorously addressed.

Blockchain has expanded not only on the level of platforms, but also on the level of real-world applications it has been promisingly applied to. These applications include *energy sector* in the work [6] proposing a decentralized and scalable dynamic grouping approach for peer-to-peer energy trading among energy prosumers (i.e. producers and consumers at the same time); *federated learning* in the work [7] addressing scalable and fault-tolerant secure multi-party machine learning algorithm that collectively trains a global model; *voting* in the work [8] implementing an e-voting mechanism via smart contracts on Ethereum to collect and count the votes to return the election winner; *trading* in the work [9] providing an autonomous bartering service for the fungible tokens where bidders propose their bids and solvers solve the resulting bartering problem

to find bartering solutions while the service focuses on the maximization of the leftovers tokens as reward; *auctions* in the work [10] holding a sealed-bid auctions where bidders now submit the homomorphic encryptions of their bids while auctioneer decrypts these encryptions to claim the winner by proving the correctness of the claim; *traffic management* in the work [11] proposing a real-time location-aware system by connecting multiple blockchain platforms to process vehicular data where vehicles moves among blockchains by validating their identities through non-interactive zero-knowledge proofs and *communication* in the work [12] building a scalable and peer-to-peer network architecture to communicate multiple unmanned aerial vehicles (i.e. UAVs) while still protecting the security of the communication links. We present only a small subset of these applications here while there are many other works available in the literature.

From the historical perspective, HTTP was once the main protocol for data transmission in the early years of the internet. Over time with the advancements of adversarial attacks to it, HTTP has been evolved to HTTPS. We believe that a similar transition has been also awaiting for traditional (i.e. transparent) blockchain platforms which often expose certain sensitive information and patterns about the nodes of the networks (e.g. their balances). This may pose serious privacy risks and attack surfaces that adversarial parties may cleverly benefit from. We refer to *privacy* here as simply the right of individuals to control their own information. However, the sheer other side of the issue (i.e. naive fully-private blockchain platforms) may be also problematic and non-applicable with the potential of disrupting transaction integrity and verifiability. This naturally-forming tension between transparency and privacy needs to be thoroughly addressed and motivates the development of privacy-preserving but also publicly-verifiable systems which simply protect the privacy of data without compromising the transaction verifiability. In this respect, we have currently been witnessing the proliferation of such systems as well [5, 13, 14].

Zero-knowledge proof is a popular cryptographic protocol to address privacy issues in general where it simply allows a prover party to prove validity of a statement (e.g. computation or secret) to a verifier party without disclosing that statement itself.

For the first time, it was proposed in the following work [15] of Goldwasser, Micali and Rackoff in 1985 as *The Knowledge Complexity of Interactive Proof-Systems* that defines an additional complexity (i.e. knowledge complexity) for systems along with time and memory complexities (i.e. how much knowledge should be transferred to prove any theorem). The same work proposes an interactive proof system that requires a certain degree of interaction between the prover and the verifier where it has evolved to non-interactive proof systems on later works [16]. A proof system is said to be zero-knowledge proof in case it can satisfy the following three main properties as: (i) *completeness* that requires the the prover to convince the verifier if the computation of the prover is correct; (ii) *soundness* that requires the the prover not to convince the verifier if the computation is not correct and (iii) *zero-knowledge* that requires the verifier to learn nothing except the validity of the statement.

The zero-knowledge proof systems have been successfully integrated into numerous real-world blockchain applications. They include (i) *energy sector* in the work [17] proposing a privacy-preserving auction mechanism for energy trading where prosumers place their bids including their prices and energy types while auctioneer as trusted party sorts these bids to select the bids to satisfy with respect to these prices; *federated learning* in the work [18] collectively training a machine learning model with the datasets of multiple parties by submitting corresponding proofs for their verifiable off-chain computations while protecting the privacy of these datasets; *internet-of-things* in the work [19] providing a level-based privacy-preserving positioning protocol for IoT devices in order to prove that a specific device is at a specific location on a specific moment at the highest level of privacy; *taxation* in the work [20] addressing tax compliance issue with respect to the available regulations where businesses validates their invoice documents without disclosing the price and tax amounts; and finally *insurance* in the work [21] proposing a novel privacy-preserving car insurance system where car owners privately authenticate and revoke their insurances. For more comprehensive collections of such applications, there also exist several survey works in the literature [22].

Custom zero-knowledge proof design and implementation is very challenging and requires a certain field expertise on mathematics and cryptography. To address this issue, several proof generation and verification frameworks have been proposed in the literature including libsnark in C++ [23], Circom in Rust [24], jsnark in Java [25], snarkjs in JavaScript [26] and ZoKrates in Rust [27]. There exist several base proof systems that these frameworks are backed by as traditional proof systems including Schnorr's protocol [28], Fiat-Shamir protocol [29] and Guillou-Quisquater protocol [30] and modern proof systems including zkSNARKs [16], zkSTARKs [31] and Bulletproofs [32]. We will technically compare these proof systems and frameworks in depth in Section 2.8. In the scope of this thesis, we specifically rely on the ZoKrates framework [27] with zkSNARKs because of its efficiency, correctness and ease of integration to Ethereum [2] for several purposes including: (i) verifying the correctness of the payment without disclosing the transaction details (e.g. balances and amounts) in our privacy-preserving payment protocol (i.e. $PTTS$), (ii) verifying the correctness of the aggregation without disclosing the data to be aggregated in our privacy-preserving data aggregation protocol (i.e. $PVSS$) and (iii) verifying the correctness of bartering multiple tokens without disclosing the bids in our privacy-preserving bartering protocol (i.e. $PMTBS$).

The problems in the literature can be classified into distinct categories with respect to the number of objectives it considers while crafting potential solution(s) as *single-objective* (i.e. only one objective) [33], *multi-objective* (i.e. up to three objectives) [34] and *many-objective* (i.e. more than three objectives) [35]. The major drawback of single-objective systems is that it neglects the other potential objectives while trying to push the limits of that single objective (e.g. neglecting cost to minimize time) where it leads to practically infeasible solutions to apply in the real-world scenarios (e.g. too high cost to pay). In return, *multi-objective* and *many-objective* systems require more complex evaluation mechanisms (e.g. pareto-domination) to rank solutions where they may cause additional overhead with the increasing number of objectives. It may worsen up if there exists a dynamism among the objectives (i.e. the number of objectives changes over time) as well [36]. In addition, extraction of relevant objectives from the perspective of the problem to be solved may require further study.

We come across the applications of multi-objective approaches in several blockchain applications as well including [37], [38] and [39]. In our thesis, we extend the privacy-preserving bartering problem into an instance of multi-objective problem to be able to find feasible bartering solutions by considering several objectives simultaneously.

1.1. Motivation

The motivations for the problems addressed in this thesis are as follows:

- For privacy-preserving payments: In a direct token payment transaction, we refer to two owners of that transaction as *sender* that deposits the transaction amount (e.g. number of tokens) and *receiver* that withdraws the same amount at most. The payments on transparent blockchain platforms are quite straightforward where they simply update the balance states (for account-based platforms) of these owners. On the other hand, as previously emphasized, this exposes very sensitive information (e.g. balances) to be adversarially used (e.g. financial surveillance or de-anonymity through transaction frequency and pattern [40]) even though there exist pseudo-anonymity (i.e. links to real identities) that these platforms provide. Though, we observe the privacy-preserving payment systems over blockchain in the literature [13], [41] and [14] where we review such systems in Section 3.1 in depth. With that motivation, we design a privacy-preserving but publicly-verifiable payment protocol with zero-knowledge proof on blockchain (i.e. $PTTS$) where transaction owners involve with secure and private transactions without revealing their balances and transaction amounts to the others.
- For balance range disclosure attack: Privacy-preserving systems may perceive cryptographic security as the ultimate security without considering real-world factors. We believe that they (regardless of how cryptographically secure they are) may potentially neglect the human attack surface which refers to vulnerabilities resulting from human behaviors and actions themselves. Adversarial parties may exploit this vulnerability by gradually collecting the privacy-preserving transactions from the transaction owners themselves (i.e. senders or receivers) in return

for certain benefits (e.g. financial rewards) or punishments. Although this may seem quite challenging for individual parties, it may be within the capabilities of large system actors including the governments and exchange services. From the perspective of adversarial parties, collecting more such transactions results in constructing a clearer global transaction graph and calculating more precise balance ranges. In the most ideal scenario where all the transactions are collected, adversarial parties can calculate all user balances. In the literature, there exist many works that collect and analyze blockchain transactions to reveal sensitive information [40, 42, 43]. Therefore, we design a novel attack (i.e. balance range disclosure attack) to the privacy-preserving payment systems to be able to reveal the bare minimum and maximum values of the user balances with the minimum cost flow networks. The strength of this attack comes from the fact that the balance of a user can be still revealed even though this user itself may not reveal their own transactions.

- For privacy-preserving aggregation: In a secure multi-party computation, several parties (i.e. aggregators) may contribute their own data to reach the global data altogether for certain purposes (e.g. decision-making). Data aggregation is prevalent in several real-world applications including healthcare [44], finance [45], voting [46] and energy [47]. However, such transparent aggregation by revealing data of all aggregators (e.g. their energy consumption) may pose potentially significant risks including discriminatory pricing, market manipulation and security vulnerabilities. This leads us to privacy-preserving data aggregation where aggregators aggregate their data without revealing data themselves and learn only the global aggregation. The naive solution to provide privacy-preserving data aggregation is to employ a trusted party to collect individual data encryptions and return only the final aggregation with the corresponding proof. However, this also suffers from several issues including trustworthiness of the trusted party itself and single-point of failure. Therefore, we design a privacy-preserving but publicly-verifiable and collective data aggregation protocols with zero-knowledge proof on blockchain (i.e. $PVSS$ and $PRFX$) where aggregators can aggregate their data without revealing data themselves to the others.

- For privacy-preserving bartering-based trading: In payment systems, the transactions have only two owners (i.e. senders and receivers) where they transfer only one type of fungible tokens. However, bartering systems involve with multiple owners (i.e. barterers) where they can barter multiple tokens (i.e. exchanging a set of tokens for another set of tokens) through their bids. We observe bartering systems on several real-world applications including resource allocation [48], wireless sensor networks [49] and food traceability [50]. But, transparent bartering by revealing the bids may leak sensitive information including market behavior, financial position, consumption patterns and partnerships among barterers. This naturally leads us to the privacy-preserving bartering where the most naive way in this respect can be the trusted parties collecting the bids, constructing the resulting global bid graph and distributing the tokens according to the solution from this graph. However, we already refer to the drawbacks of trusted parties. Therefore, we design a privacy-preserving but publicly-verifiable and collective bartering protocol with zero-knowledge proof on blockchain (i.e. *PMTBS*) where barterers barter their tokens through bids without revealing their balances and bids to the others.
- For privacy-preserving multi-objective bartering-based trading: There are two main improvement points of our privacy-preserving bartering protocol: (i) it handles the problem as a satisfaction problem by including all the bids available without considering any optimization objectives and (ii) it heavily relies on the iteratively-performing ascending auction mechanism that result in additional computational and communication overheads. The optimization objectives can be the maximization of the number of bids satisfied and the maximization of the budget after bid costs are paid. To address these issues, we design a privacy-preserving multi-objective bartering approach by integrating Bellman-Ford into zero-knowledge proof on blockchain (i.e. *zkMOBF*) where barterers can barter their tokens without revealing their balances and bids under two different objectives. So, *zkMOBF* is the first approach in the literature that addresses multi-objective optimization in the scope of zero-knowledge proof.

1.2. Contributions

The main contributions of our thesis are the followings:

(i) We propose a novel privacy-preserving but publicly-verifiable cryptographic payment protocol with zero-knowledge proof on blockchain (i.e. *PTTS*) where parties (i.e. transaction owners) can safely transfer their tokens while protecting the privacy of balances and transaction amounts. We extend the base *PTTS* protocol to support multi-token transfer as well.

(ii) We model a novel attack (i.e. balance range disclosure attack) as minimum cost flow network where adversarial parties may attempt collecting the privacy-preserving payment transactions from the transaction owners with certain incentive mechanisms, constructing global payment graphs and extracting private balances from these graphs to a certain degree. We support this attack with empirical evidence with varying degrees of addresses and transactions.

(iii) We propose a novel privacy-preserving collective and trustless data aggregation protocol on blockchain (i.e. *PVSS*) with zero-knowledge proof and hypercube network topology where parties (i.e. aggregators) can aggregate their data while protecting their privacy. We extend *PVSS* to support privacy-preserving prefix summation (i.e. *PRFX*) for delegation mechanisms with Euler Tour Technique. Moreover, we propose three novel techniques (i.e. *node multiplexing*, *topological recursing* and *data splitting*) to run *PVSS* for the arbitrary number of aggregators.

(iv) We present a novel privacy-preserving collective multi-token bartering protocol with zero-knowledge proof on blockchain (i.e. *PMTBS*) where parties (i.e. barterers) can exchange a set of tokens in return for another set of tokens while protecting their bids and balances. For this protocol, we propose a novel token transfer technique (in addition to standard ERC-20 token transfer) as well where barterers act on only their balances without affecting the balances of other barterers. We extend the *PMTBS* protocol to support multi-objective bartering as well by optimizing several objectives at the same time as the *zkMOBF* protocol. Refer to Figure 1.1 for the common layers of our protocols as the blockchain layer, the

utility layer and the application layer.

(v) We demonstrate our protocols under three main models as the zero-knowledge proof model for off-chain computation, the smart contract model for on-chain computation and the web user interface model. Furthermore, we analyze the scalability of the protocols in terms of the computational, communication and storage overheads; and the security of the protocols in terms of potential security attacks and formal reduction proofs.

(vi) We carry out extensive experimental studies over our protocols to support their theoretical foundations and measure their performance in terms of blockchain gas consumption, proof generation/verification times and proof size. These studies justify the validity and applicability of our protocols on the real-world blockchain applications. Our protocols are open for further inspection as well [51–55].

The rest of the work is organized as follows: Chapter 2 reviews the background and Chapter 3 presents the related work for privacy-preserving protocols on blockchain. Chapter 4 proposes our privacy-preserving payment protocol (i.e. *PTTS*). Chapter 5 proposes our privacy-preserving data aggregation protocols (i.e. *PVSS* and *PRFX*). Chapter 6 proposes our privacy-preserving multi-token bartering protocols with zero-knowledge proofs (i.e. *PMTBS*). Chapter 7 presents the privacy-preserving multi-objective bartering approach (i.e. *zkMOBF*). Each chapter includes its own experimental results. Finally, Chapter 8 concludes the thesis.

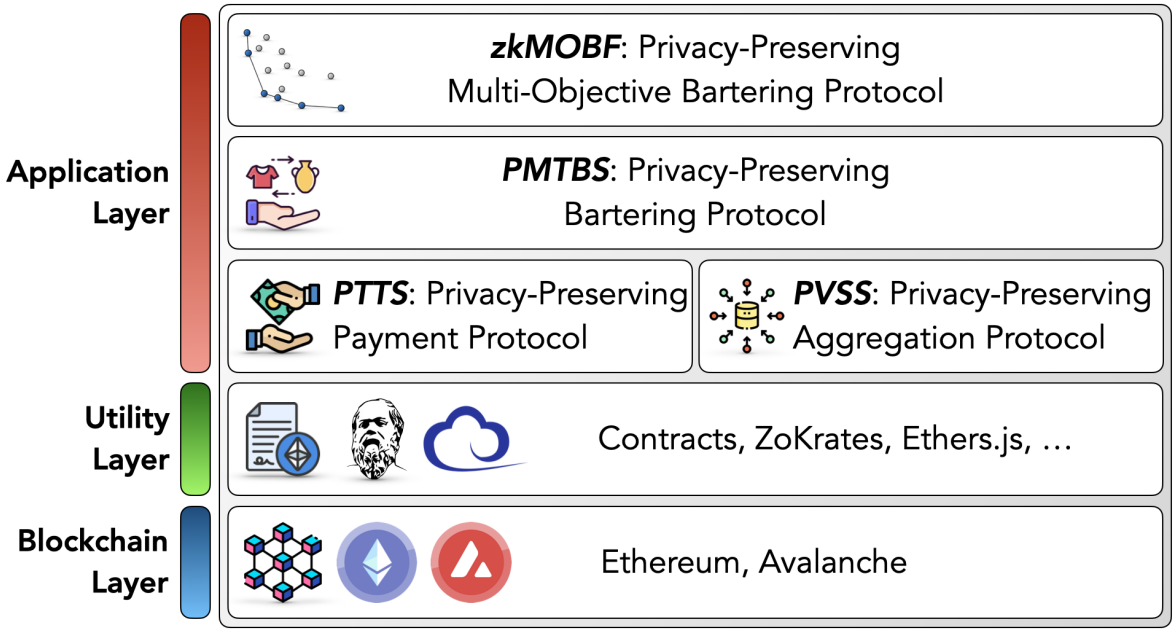


Figure 1.1. Layered architecture of our protocols.

2. BACKGROUND

In this chapter, we briefly review some important and fundamental computer science principles where we believe they will help the reader to conceptualize and contextualize what we aim to solve, how we aim to solve and how much we aim to benefit from these solutions. These principles include blockchain, token standards, hypercube network topology, network flows, multi-objective optimization, commitment schemes, public-key cryptography, zero-knowledge proof and ZoKrates [27]. To support understandability of these principles further in our scope, we purposefully present several simple depictions as well.

2.1. On Blockchain

Blockchain can be viewed as a decentralized ledger of transactions that is shared among all the nodes of the network. It was first proposed in the following seminal work *Bitcoin: A Peer-to-Peer Electronic Cash System* [1]. In the most general form, blockchain can be defined as:

- Immutable. Once a transaction is committed, it cannot be altered or deleted.
- Decentralized. Transactions are distributed across a network of nodes without any central authority.
- Secure. Data integrity and authentication are protected through a series of cryptographic primitives including public-key encryptions and hashing functions.
- Transparent. Transactions are visible to all nodes in public blockchains.
- Traceability. Transactions are time-stamped by allowing auditability of history.
- Fault Tolerant. Blockchain resiliently continues to operate under node failures.
- Anonymous. There theoretically do not exist any links among the blockchain addresses and real identities of nodes.
- Programmable. Smart contracts allow custom automated and decentralized applications (i.e. dApps) to be developed and deployed.

We define a blockchain $\mathcal{BC}$ (specifically account-based blockchain) as the set of chained blocks $\mathcal{B}$ as follows:

$$\mathcal{BC} : \langle \mathcal{B}_0, \ldots, \mathcal{B}_i, \ldots, \mathcal{B}_{|\mathcal{BC}|-1} \rangle \tag{2.1}$$

where $\mathcal{BC}$ refers to blockchain while $\mathcal{B}$ refers to an individual block to construct that blockchain altogether. With a similar fashion, a block $\mathcal{B}$ consists of a set of transactions:

$$\mathcal{B}_i : \langle tx_0, \ldots, tx_j, \ldots, tx_{|\mathcal{B}|-1}, \mathcal{H}_i \rangle \tag{2.2}$$

where tx refers to a transaction while $\mathcal{H}$ is a block header. Each block includes a certain number of transactions at most with respect to a block size. There exist multiple types of transactions including (i) the *regular* transaction tx that specifies a payment among two addresses as follows:

$$tx_{sr} : \langle \Delta, u_s, u_r \rangle \tag{2.3}$$

where Δ is the amount transferred from the source address (i.e. sender) u_s to the destination address (i.e. receiver) u_r; (ii) the *minting* transaction tx^+ and (iii) *burning* transaction tx^- as:

$$tx_r^+ : \langle \Delta, u_r \rangle \tag{2.4}$$

$$tx_s^- : \langle \Delta, u_s \rangle \tag{2.5}$$

where the amount to be minted Δ is transferred to the address u_r in tx_r^+ while the amount to be burned Δ is deducted from the address u_s in tx_s^-. There also exists the fourth type of transaction as the *contract* transaction where we leave its definition. Under this framework, we view an instance of blockchain as state while transaction as state-transition function in the following way:

$$tx : \mathcal{BC}_t \rightarrow \mathcal{BC}_{t+1} \tag{2.6}$$

where a particular transaction tx changes the state of the blockchain from $\mathcal{BC}_t$ to $\mathcal{BC}_{t+1}$.

To continue our formal definition, block header $\mathcal{H}$ is simply a metadata of a block including several descriptive information including version, block size, nonce, time, difficulty, counter, hash value of the previous block and root hash value of Merkle tree. Let's briefly define what they are with the following way:

- version: is the version of the blockchain protocol for backward compatibility.
- block size: is the size of the block in terms of the number of transactions
- nonce: is the counter to increment while solving cryptographic puzzles to meet the difficulty criteria of mining.
- time: is the time at which the block was created.
- difficulty: is the parameter to periodically adjust the target difficulty for mining with respect to the available computational power in the network.
- counter: is the total number of transactions included in the block.
- previous block hash: is the hash value of the previous block header, which ensures the immutability of the blockchain by linking the current block to the previous block. This hash value is computed as follows:

$$h_{prev} = \texttt{Hash}(\mathcal{H}_{i-1}). \tag{2.7}$$

  Note that any alteration in any previous block results in cascading changes of hash values of the blocks afterwards.
- merkle root hash: is the root hash of the Merkle tree having all the transactions in the block by allowing efficient and secure verification of data integrity. The root hash value is computed as follows:

$$h_{root} = \texttt{Hash}(\langle tx_0, tx_1, \ldots, tx_{|\mathcal{B}|-1} \rangle) \tag{2.8}$$

  where it will be further discussed in this section as well.

Blockchain with several blocks including block headers and transactions are clearly depicted in Figure 2.1 as well.

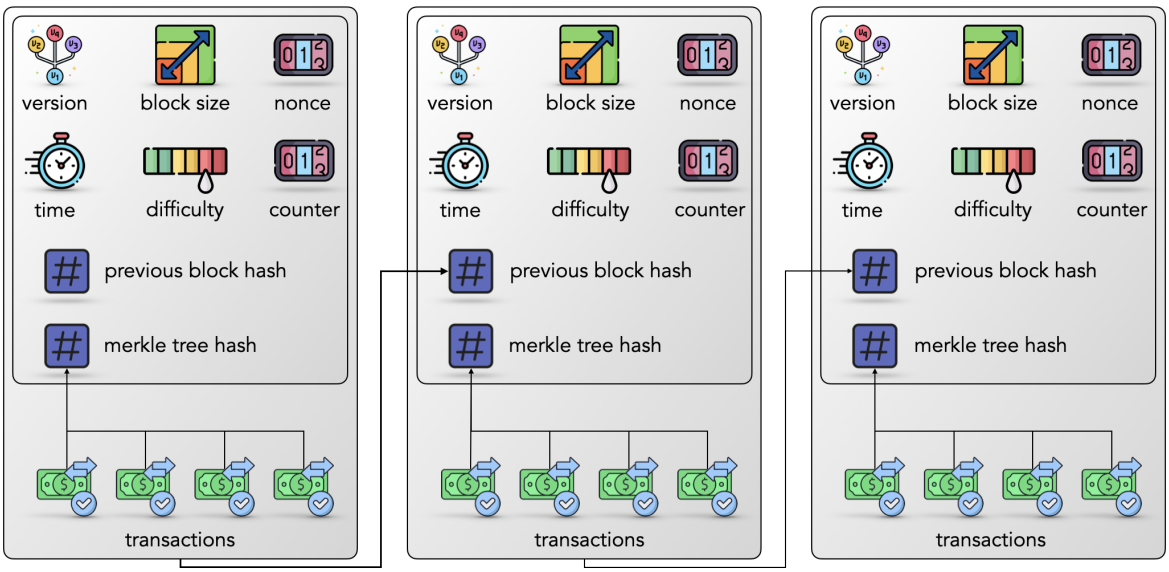


Figure 2.1. Blockchain as a chain of blocks.

Blockchain inherently needs consensus mechanisms that bring all the nodes of the network into a common agreement. There exist several such consensus mechanisms in the literature including *proof-of-work* (i.e. PoW), *proof-of-stake* (i.e. PoS) and *delegated proof-of-stake* (i.e. $DPOS$). As the most popular mechanism, PoW [1] requires the block miners to solve cryptographic puzzles (i.e. hashing) with certain difficulty to show their proof of works. The first miner solving the puzzle correctly has the right to broadcast their own block to the network. Blockchain periodically adjusts this difficulty with respect to the computational power available across the network. We formally define the total work for an entire blockchain $\mathcal{BC}$ with the following way:

$$\mathcal{BC}^W = \sum_{\mathcal{B}_i \in \mathcal{BC}} \frac{1}{\texttt{Pr}(\mathcal{B}_i)} \tag{2.9}$$

where it implies that the total work (i.e. the weight of that blockchain) $\mathcal{BC}^W$ is inversely changing with the probability to mine the blocks it has. With that respect, PoW requires all the nodes to always adopt the longest chain (i.e. the chain with the most

work) in case there exist multiple chain forks as:

$$\mathcal{BC}^{W_{max}} = \arg\max_{BC} \mathcal{BC}^{W} \tag{2.10}$$

where it simply returns the blockchain having the most weight $\mathcal{BC}^{W_{max}}$.

Merkle root (i.e. the root hash of binary *merkle tree*) provides data integrity of several transactions of the block it resides in for quick verification, fast transmission over networks and efficient storage. A merkle tree is constructed through recursively hashing the tree nodes by beginning from leaf nodes until there exists only one hash (i.e. root hash). This construction is shown in Figure 2.2 where the leaf nodes represent the blockchain transactions (with 4 transactions) and they are hashed into four hashes in *level-0*, two hashes at the *level-1* and finally the root hash at the *level-2*:

$$h_0 = \texttt{Hash}(tx_0) \tag{2.11}$$

$$h_1 = \texttt{Hash}(tx_1) \tag{2.12}$$

$$h_2 = \texttt{Hash}(tx_2) \tag{2.13}$$

$$h_3 = \texttt{Hash}(tx_3) \tag{2.14}$$

$$h_4 = \texttt{Hash}(h_0, h_1) \tag{2.15}$$

$$h_5 = \texttt{Hash}(h_2, h_3) \tag{2.16}$$

$$h_6 = \texttt{Hash}(h_4, h_5) \tag{2.17}$$

where tx_0 and tx_1 are the leaf nodes while h_6 is the root hash. On blockchain, merkle tree is useful to verify whether a certain transaction is already included into the given block by reducing the brute-force complexity of $O(n)$ into the logarithmic complexity of $O(log(N))$. Furthermore, the memory requirement for the merkle root stays always constant regardless of the number of transactions being hashed (since hash functions map varying-size inputs to fixed-size outputs (see Section 2.6)).

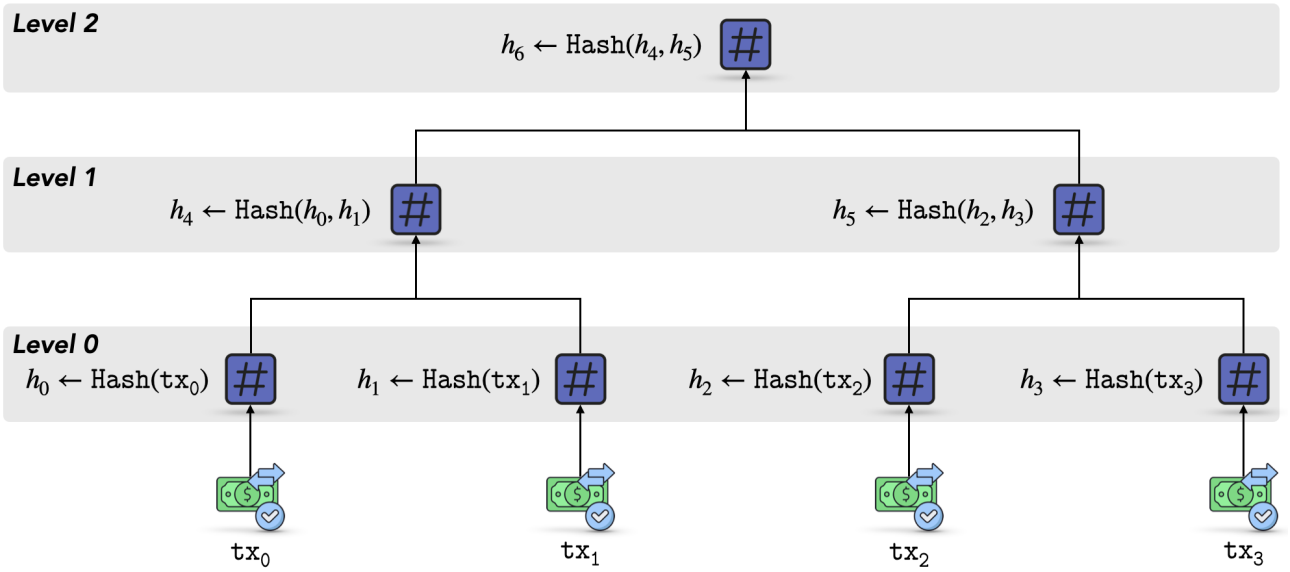


Figure 2.2. Merkle tree for blockchain transactions.

2.2. On Token Standards

Token standards are indispensable component of blockchain ecosystems that regulate the creation, management and behavior of different tokens to ensure consistency, efficiency and security. Ethereum establishes several token standards including *ERC-20* for fungible tokens, *ERC-721* for non-fungible tokens and *ERC-1155* for multi-tokens. First, the *ERC-20* token standard [56] is a widely adopted protocol on the Ethereum blockchain, which defines the set of rules to create and manage fungible tokens (i.e. non-unique and divisible tokens). The *ERC-20* interface provides several key functionalities that can be overridden with respect to the application requirements. The `name(.)` and `symbol(.)` functions set the name and abbreviation of the token while `decimals(.)` shows the total divisibility of the token. The `totalSupply(.)` function indicates the maximum supply of the token to be minted while `balanceOf(.)` retrieves the balance of a specific address. There exist two different transfer techniques in ERC-20 where `transfer(.)` directly moves tokens to another address while `transferFrom(.)` allows a third address to transfer tokens to another address. The `approve(.)` function authorizes the third address while `allowance(.)` shows the amount of tokens that is under control of the third address. The events `Transfer` and `Approval` are emitted

Table 2.1. ERC-20 token standard interface.

Type	Name	Return
function	`name()`	string
F function	`symbol()`	string
function	`decimals()`	uint8
function	`totalSupply()`	uint256
function	`balanceOf(_owner)`	uint256
function	`transfer(_from, _value)`	bool
function	`transferFrom(_from, _to, _value)`	bool
function	`approve(_spender, _value)`	bool
function	`allowance(_owner, _spender)`	uint256
event	`transfer(_from, _to, _value)`	-
event	`approval(_owner, _spender, _value)`	-

when their corresponding functions are triggered for traceability. Refer to Table 2.1 for further information about ERC-20.

The *ERC-721* token standard [57] is another popular protocol on the Ethereum blockchain to create and manage non-fungible tokens (NFTs) (i.e. unique and non-divisible tokens). The ERC-721 interface includes several functionalities that can be overridden. The `balanceOf(.)` function retrieves the number of tokens that a specific address has while `ownerOf(.)` returns the current owner that has this token. ERC-721 provides two token transfer methods as `transferFrom(.)` for standard transfers and `safeTransferFrom(.)` for more secure transfers. As in ERC-20, the `approve(.)` grants another address to transfer a specific token while `setApprovalForAll(.)` grants another address to manage all of the tokens. The following functions `getApproved(.)` and `isApprovedForAll(.)` check the approval settings related to these functions to enhance security. Finally, the events `Transfer`, `Approval` and `ApprovalForAll` are emitted when their corresponding functions are triggered for traceability again. Refer to Table 2.2 for further information about ERC-721.

The ERC-1155 token standard [58] is another Ethereum protocol that flexibly

Table 2.2. ERC-721 token standard interface.

Type	Name	Return
function	`balanceOf(_owner)`	uint256
function	`ownerOf(_tokenId)`	address
function	`safeTransferFrom(_from, _to, _tokenId)`	-
function	`safeTransferFrom(_from, _to, _tokenId, _data)`	-
function	`transferFrom(_from, _to, _tokenId)`	-
function	`approve(_to, _tokenId)`	-
function	`setApprovalForAll(_operator, _approved)`	-
function	`getApproved(_tokenId)`	address
function	`isApprovedForAll(_owner, _operator)`	bool
event	`Transfer(_from, _to, _tokenId)`	-
event	`Approval(_owner, _approved, _tokenId)`	-
event	`ApprovalForAll(_owner, _operator, _approved)`	-

supports the creation and management of both fungible and non-fungible tokens within a single contract. In this interface, `balanceOf(.)` retrieves the balance of a specific token for the given owner while `balanceOfBatch(.)` allows querying multiple balances in a single call, which reduces blockchain gas consumption to a certain extent. For token transfers, `safeTransferFrom(.)` securely moves a token between addresses while `safeBatchTransferFrom(.)` moves multiple tokens and token amounts in a single call. The function `setApprovalForAll(.)` manages the approvals while `isApprovedForAll(.)` check these approvals. The events including `TransferSingle`, `TransferBatch` and `ApprovalForAll` are emitted when their corresponding functions are triggered for traceability again. The event `URI` associates metadata URIs with tokens for dynamic token information retrieval. Refer to Table 2.3 for further information about ERC-1155. All ERC-20, ERC-721 and ERC-1155 interfaces support only public token transfer where sender addresses, receiver addresses and transaction amounts are open to inspect. In the scope of our thesis, we address the privacy issue, especially over the ERC-20 and ERC-1155 interfaces to mask the transaction amount through zero-knowledge proof techniques while still preserving their verifiability.

Table 2.3. ERC-1155 token standard interface.

Type	Name	Return
function	`balanceOf(_owner, _id)`	uint256
function	`balanceOfBatch(_owners, _ids)`	uint256[]
function	`setApprovalForAll(_operator, _approved)`	bool
function	`isApprovedForAll(_owner, _operator)`	bool
function	`safeTransferFrom(_from, _to, _id, _amount, _data)`	-
function	`safeBatchTransferFrom(_from, _to, _ids, _amounts, _data)`	-
event	`TransferSingle(_operator, _from, _to, _id, _amount)`	-
event	`TransferBatch(_operator, _from, _to, _ids, _amounts)`	-
event	`ApprovalForAll(_owner, _operator, _approved)`	-
event	`URI(_value, _id)`	-

2.3. On Hypercube Network Topology

A network topology refers to a certain arrangement of nodes and their connections in communication network. A hypercube topology is a closed, convex and multi-dimensional network with 2^d number of nodes for d-dimension to connect every node (ranging from zero to $2^d - 1$) with other nodes with certain rules through bi-directional links. If the number of nodes simply complies to 2^d, it is called as *complete hypercube*. On the contrary, *incomplete hypercube* relaxes this strict requirement by allowing any arbitrary number of nodes where certain positions in network seem missing. The hypercube topology is mostly known for its high degree of scalability for a growing number of nodes by requiring only $log(N)$ communication for for N number of nodes. In the scope of blockchain, we formally define a hypercube network through a graph network $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V}$ is the set of vertices referring to different parties in blockchain while $\mathcal{E}$ is the set of edges referring to the direct communication links between those parties. Each party u is represented by a distinct bit string of length $log(|\mathcal{V}|)$ as $u = u_{log(|\mathcal{V}|)-1} \dots u_j \dots u_0$ where $u_j \in \{0, 1\}$. This graph is a hypercube if and only if the vertices whose bit string representations differ by exactly one bit are connected, $\mid u - u' \mid = 1$ where $\mid . \mid$ is the Hamming distance operator, which is shown in Figure 2.3. In this thesis, we integrate hypercube topology into our protocols (i.e. into

the *PVSS* and *PMTBS* protocols) as a logical layout over blockchain to build scalable communication networks for secure multi-party computation.

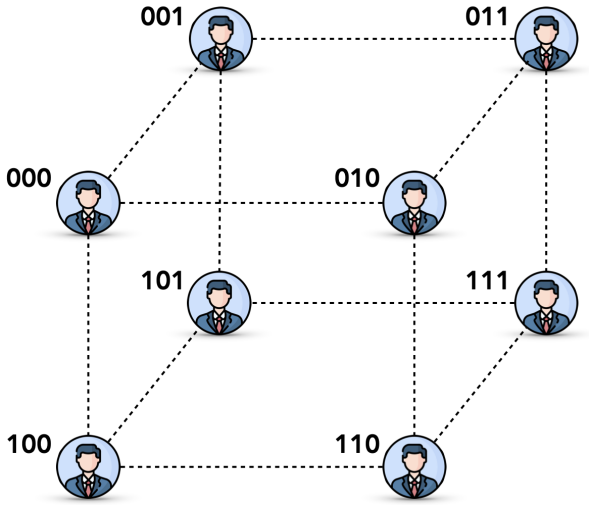


Figure 2.3. Hypercube network topology in 3-dimensions.

2.4. On Network Flows

Network flows are among fundamental concepts of graph theory and optimization to be used to express a wide variety of optimization problems. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a generic graph network with a set of vertices $\mathcal{V}$ and a set of edges $\mathcal{E}$ between these vertices. Each node $i \in \mathcal{V}$ is associated with a supply/demand value $b(i)$ where it is a supply node if $b(i) > 0$ and a demand node if $b(i) < 0$. Each edge $e_{ij} \in \mathcal{E}$ between nodes i and j is associated with a lower bound l_{ij} to represent the minimum amount of flow, an upper bound u_{ij} to represent the maximum amount of flow and a cost per flow c_{ij}. For a network flow problem (i.e. minimum cost flow network for sake of simplicity), the goal is to find feasible flows for edges to minimize the total cost:

$$min. \quad \sum_{(e_{ij} \in E)} c_{ij} \cdot x_{ij} \tag{2.18}$$

$$s.t. \quad \sum_{(e_{ij} \in E)} x_{ij} - \sum_{(e_{ji} \in E)} x_{ji} = b(i) \tag{2.19}$$

$$l_{ij} \leq x_{ij} \leq u_{ij} \tag{2.20}$$

where x_{ij} is the decision variable over the edge e_{ij}. Network flows can be used to solve different problems including shortest path, maximum flow, assignment, transportation, circulation, convex cost flow, generalized flow, multi-commodity flow, minimum spanning tree and matching problems [59]. In addition, network flows can be used to model bartering problems as minimum cost circulation problem [60], as shown in Figure 2.4 where the first numbers on edges are the capacities while the second numbers are the amount of flows. Barterers here may have several instances of the same items to barter with set of barterers as $\mathcal{U} = u_1, u_2, ..., u_q$ and set of instances as $I = i_1, i_2, ..., i_p$. Let $g(t, s)$ be the number of instances for the item i_s the barterer u_t supplies and $r(t, s)$ be the number of instances for the item i_s the barterer u_t demands. In a bipartite network, there exists an edge (u_t, i_s) if $g(t, s) > 0$ and another edge (i_s, u_t) if $r(t, s) > 0$.

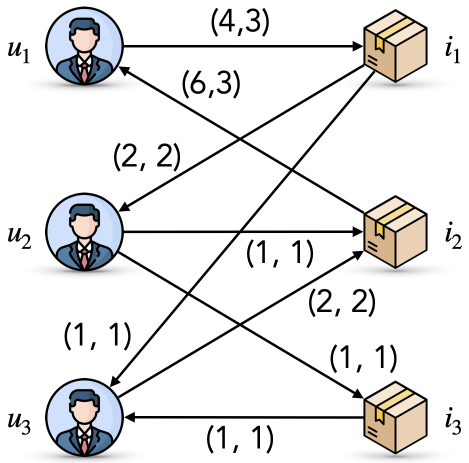


Figure 2.4. Minimum cost circulation for bartering problem [60].

2.5. On Multi-Objective Optimization

Multi-objective optimization (MOO) refers to a set of techniques to solve optimization problems consisting of multiple objectives (with at least two conflicting objectives) to be simultaneously considered [61]. Without loss of generality, a multi-objective problem can be defined for minimization with the following way:

$$min. \quad F(x) = \{F_0(x), F_1(x), ..., F_{|F|-1}(x)\}$$

$$\begin{aligned} s.t. \qquad & g_i(x) \leq 0 \\ & h_i(x) = 0 \end{aligned} \tag{2.21}$$

where x is the decision vector (i.e. solution) to minimize the objective vector F under the certain inequality constraints $g_i(x)$ and equality constraints $h_i(x)$. The decision vectors x reside in *decision space* while their objective projections $F(x)$ reside in *objective space*. In single-objective optimization, solutions can be ranked by simply sorting the values of that single objective. On the other hand, multi-objective optimization asks for a more sophisticated ranking mechanism. The easiest technique is to linearly combine the objective values through objective weights where the correct setup of these weights is a significant issue. Another technique is *pareto-domination* which is based on pairwise solution comparison where a solution x_1 is said to dominate another solution x_2 in case x_1 is no worse than x_2 for all the objectives and x_1 is strictly better than x_2 in at least one objective, $x_1 \prec x_2$. More formally for minimization:

$$\forall i \; F_i(x_1) \leq F_i(x_2) \wedge \exists j \; F_i(x_1) < F_i(x_2) \tag{2.22}$$

where the solutions like x_1 that are not dominated by any other existing solutions are called as the *pareto-optimal* solutions, $\nexists x_2 \in X : x_2 \prec x_1$. The set of pareto-optimal solutions refers to *pareto-optimal set* (i.e. POS) in decision space as $POS = \{x_1 \in X | \nexists x_2 \in X : x_2 \prec x_1\}$ and *pareto-optimal front* (i.e. POF) in objective space as $POF = \{y \in F(X) | x : POS\}$ [61]. In Figure 2.5, objective space with two different objectives is given where the blue points are the pareto-optimal solutions (i.e. non-dominated solutions) while the gray points are dominated solutions. In the scope of our thesis, we use multi-objective optimization in order to find an optimal bartering solution in privacy-preserving multi-objective bartering approach (i.e. *zkMOBF*) by integrating a multi-criteria decision-making technique (i.e. utility function). We assume that the decision and objective spaces in the problem are not dynamic (i.e. fixed with no change) over time.

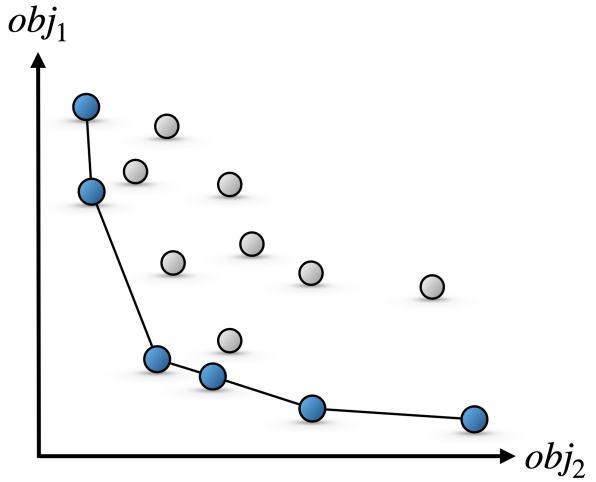


Figure 2.5. Multi-objective optimization with two objectives: (blue): optimal solutions (gray): non-optimal solutions.

2.6. On Commitment Schemes

Commitments (with domain abbreviation `Cm`) are one-way and efficient cryptographic functions that map varying-size input values into fixed-size and random output values to provide data integrity [62], (i.e. a change in input results in a devastating and unpredictable change in output) as follows:

$$\texttt{Cm} \;:\; \{0,1\}^l \rightarrow \{0,1\}^k \tag{2.23}$$

where l and k are the lengths of the input and output values where k is expected to be always constant. This allows a party to guarantee the integrity of data without disclosing that data itself to the adversary parties. In the scope of this thesis, we use two main commitment scheme functions as (i) the commit function `Cm.Comm(.)` to commit a certain private value x with a large salting parameter σ and (ii) the verification function `Cm.Vfy(.)` to check the correctness of the commitment value. We will use the following notation to represent these functions:

$$c \;\leftarrow\; \texttt{Cm.Comm}(x, \sigma) \tag{2.24}$$

$$b \leftarrow \texttt{Cm.Vfy}(c, x, \sigma) \tag{2.25}$$

where c is the resulting commitment value of the commit function while b is the resulting boolean value (i.e. true if the commitment value is correct and vice versa) of the verification function. For the sake of simplicity in mathematical notations, we omit salting parameters ($\sigma \xleftarrow{\$} \mathbb{Z}_0^+$) in equations for the rest of the thesis.

There exist several security requirements that commitment schemes need to satisfy including the *hiding* and *binding* properties [63]. The hiding property basically asks not to reveal any meaningful information about the data itself through the commitment value of that data as follows:

$$\mid \texttt{Pr}[c_1 = \texttt{Cm.Comm}(x_1, \sigma) \mid c_1] - \texttt{Pr}[c_2 = \texttt{Cm.Comm}(x_2, \sigma) \mid c_2] \mid < \epsilon \tag{2.26}$$

where it says that the probability to distinguish two given commitments (i.e. to match two given input values to two given output values) successfully must be negligibly small. Otherwise, the commitment scheme fails to satisfy the *hiding* property. On the other hand, the binding property asks not to have exactly the same commitment value for two different input values (i.e. in order to have collision resistance) as follows:

$$\texttt{Pr}[x_1 \neq x_2 \mid \texttt{Cm.Comm}(x_1, \sigma) = \texttt{Cm.Comm}(x_2, \sigma)] < \epsilon \tag{2.27}$$

where it says that having any (x_1, x_2) pair with the same commitment values must be negligibly small. Otherwise, the commitment scheme fails to satisfy the *binding* property. This would pave the way for a party to initially commit for x_1 and to later maliciously reveal x_2 since the adversary parties cannot distinguish their commitments. A commitment scheme is said to be *computationally* hiding/binding if it satisfies that property with negligible error for an adversary with limited computational power; *statistically* hiding/binding if it satisfies with negligible error for an adversary with unlimited computational power; and *perfectly* hiding/binding if it still satisfies but with no error now for an adversary with unlimited computational power [64].

Commitment scheme on blockchain is shown in Figure 2.6 where there exist two parties (as Alice and Bob). Alice uses the commitment function `Cm.Comm(.)` in order to commit the private data x by obscuring it with salting parameter σ. She submits the resulting commitment c is to the contract with `setCommitment(.)` function. For Bob to verify the correctness of the commitment, Alice needs to reveal x and σ. Bob later fetches the commitment, private data and salting parameter from the contract and uses `Cm.Vfy(.)` to verify the correctness. In our thesis, we heavily use commitment schemes to enforce users to commit certain values: (i) the amount of tokens to transfer in our privacy-preserving token transfer protocol, (ii) the private data to aggregate in our privacy-preserving data aggregation protocol and (iii) the bid to barter in our privacy-preserving multi-token bartering protocol.

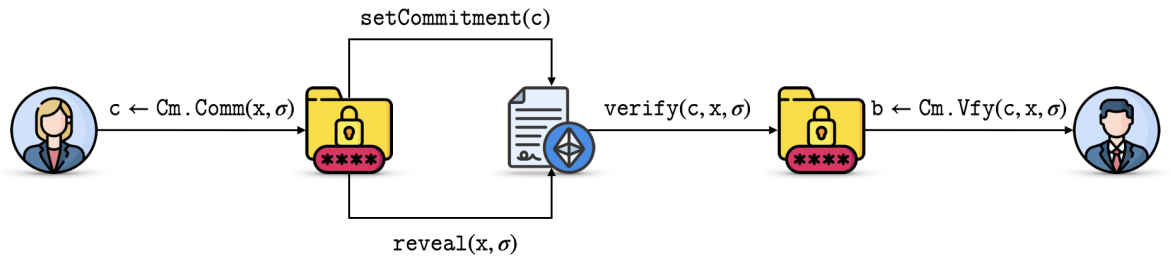


Figure 2.6. Commitment scheme on blockchain.

2.7. On Public-Key Cryptography

A *one-way* function maps values in input space to certain values in output space such that the function f itself is easy to calculate in one direction while the inverse function f^{-1} is infeasible to calculate in the other direction. The most known one-way functions are commitment schemes that are already defined in Section 2.6. We can extend the notion of one-way function into *trapdoor one-way* function f_s which is is easy to calculate in one direction but the inverse function f_s^{-1} is infeasible to calculate in the other direction unless the additional secret is known where s is the trapdoor, (see Figure 2.7). The most popular trapdoor one-way functions are public-key schemes

(with domain abbreviation Pk):

$$\texttt{Pk} \;:\; \{0,1\}^l \times \{0,1\}^n \rightarrow \{0,1\}^k \tag{2.28}$$

where l, n and k are the lengths of the input value, the key and the output value. This allows a party to guarantee confidentiality or authentication with respect to scheme design. In this thesis, we use five main public-key scheme functions (asymmetric encryption) as (i) the setup function `Pk.Setup(.)` to generate a public key pk and a private key sk, (ii) the encryption function `Pk.Enc(.)` to encrypt the given private data with the public key of the recipient for confidentiality, (iii) the decryption function `Pk.Dec(.)` to decrypt the given encryption with the private key of the recipient for confidentiality, (iv) the sign function `Pk.Sign(.)` to put a signature over a public data for authentication and finally (v) the signature verification function `Pk.Vfy(.)` to verify the correctness of this signature for authentication. We represent these functions with the following notation in our thesis:

$$(pk, sk) \leftarrow \texttt{Pk.Setup}(\lambda) \tag{2.29}$$

$$E \leftarrow \texttt{Pk.Enc}(x, pk) \tag{2.30}$$

$$x \leftarrow D \leftarrow \texttt{Pk.Dec}(E, sk) \tag{2.31}$$

where E is the encryption value of the encryption function while D is the resulting decryption value of the decryption function. It should be noted that the application of the decryption function over the encryption function must result in the same private data as $x \leftarrow \texttt{Pk.Dec}(\texttt{Pk.Enc}(x, pk), sk)$. And also:

$$E \leftarrow \texttt{Pk.Sign}(x, sk) \tag{2.32}$$

$$b \leftarrow D \leftarrow \texttt{Pk.Vfy}(E, pk) \tag{2.33}$$

where E now represents the signature together with the public data itself while b is the resulting boolean of the signature verification function.

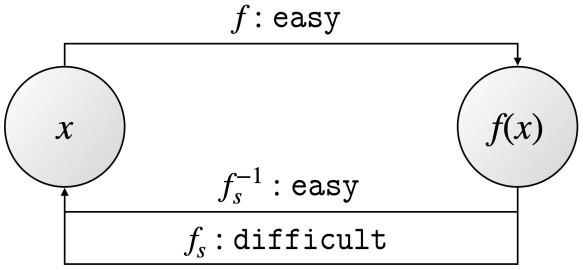


Figure 2.7. One-way functions with trapdoors.

There exist certain security requirements for public-key schemes to satisfy [63]:

- It must be efficient to generate a key pair by using Equation (2.29).
- It must be efficient to make an encryption if there exists a proper public key through Equation (2.30).
- It must be efficient to make a decryption to recover the original message if there exists a proper private key through Equation (2.31).
- It must be infeasible to recover the private key by knowing the public key.
- It must be infeasible to recover the original message by knowing the public key.
- The private and public keys can be applied in any order as:

$$x \leftarrow \texttt{Pk.Dec}(\texttt{Pk.Enc}(x, pk), sk) \tag{2.34}$$

$$b \leftarrow \texttt{Pk.Vfy}(\texttt{Pk.Sign}(x, sk), pk). \tag{2.35}$$

Public-key scheme for confidentiality on blockchain is shown Figure 2.8 where Alice is responsible for encrypting the data while Bob is responsible for decrypting it later. Alice first gets the public key of Bob from the smart contract. We assume that all users already generate their own key pairs locally via the `Pk.Setup(.)` function and submit them to the contract in advance (i.e. the contract already includes the public keys of all users). Alice first encrypts the private data x by using the public key of Bob with `Pk.Enc(.)` and submits the resulting encryption E to the contract with the `setEncryption(.)` function. Later, Bob gets that encryption with the the `getEncryption(.)` function. Now, Bob is ready to decrypt the encryption with his

own private key by using the `Pk.Dec(.)` function. In our thesis, we heavily use public-key scheme for confidentiality: (i) encryption of transaction details in our privacy-preserving token transfer protocol, (ii) encryption of data aggregations in hypercube networks in our privacy-preserving data aggregation protocol and (iii) encryption of bid aggregations in our privacy-preserving multi-token bartering protocol.

Public-key scheme for confidentiality on authentication is also shown Figure 2.9 where Alice now puts signature on data while Bob verifies the correctness of this signature. Alice first uses the sign function `Pk.Sign(.)` with her own private key and submits the resulting encryption (i.e. public data and signature together) `E` to the contract with `setEncryption(.)`. Later, Bob gets that encryption from the contract with `getEncryption(.)` as well as the public key of Alice with `getPublicKey(.)`. Now, Bob is ready to check the correctness of the signature with the verification function `Pk.Vfy(.)`. Note that this authentication process validates that it is really Alice that puts the signature (and data is still public) while the confidentiality process encrypts the data so that only Bob (holding the private key) can decrypt it.

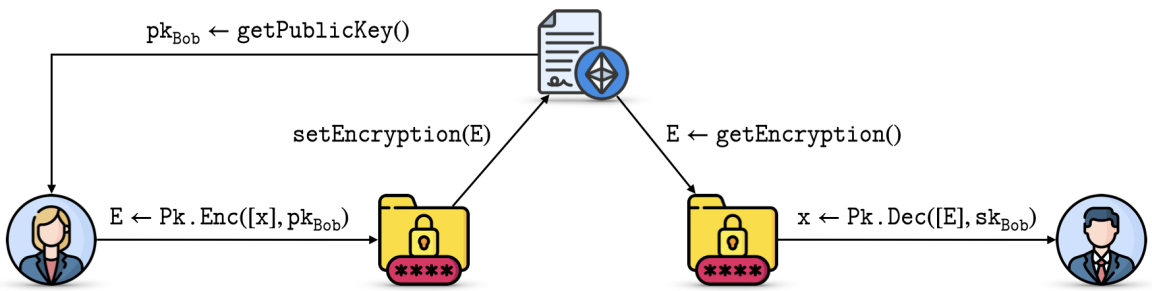


Figure 2.8. Public key cryptography on blockchain: confidentiality.

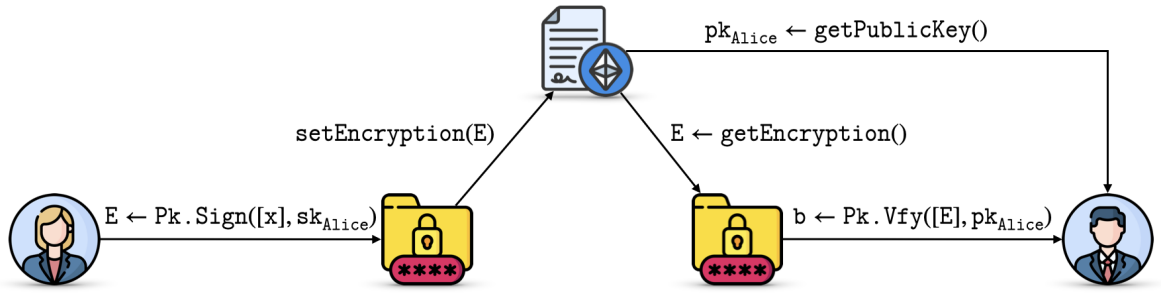


Figure 2.9. Public key cryptography on blockchain: authentication.

2.8. On Zero-Knowledge Proof

Time complexity measures the amount of time an algorithm needs to complete; and *memory* complexity measures the amount of memory it needs to complete with respect to the input size. However, *knowledge* complexity measures the amount of meaningful knowledge that needs to be revealed to verify the correctness of a statement (i.e. secret). We define *meaningful knowledge* here as only the knowledge with which the statement (or part of the statement) may be reconstructed. With that respect, zero-knowledge proof is a system with a knowledge complexity of absolutely zero where a prover party $\mathcal{P}$ aims to convince a verifier party $\mathcal{V}$ about the correctness of a statement χ without disclosing the statement itself. They are Goldwasser, Micali and Rackoff who proposed the notion of zero-knowledge proof for the first time in 1985 [15]. Historically, many researchers have contributed to the proliferation of this field afterwards. However, it is blockchain that accelerates the development of modern and complex proof systems to a great extent including zkSNARKs (i.e. Zero-Knowledge Succinct Non-Interactive Argument of Knowledge) [16], zkSTARKs (i.e. Zero-Knowledge Scalable Transparent Argument of Knowledge) [31] and Bulletproofs [32].

We can compare these systems with respect to their various parameters including proof generation complexity, proof verification complexity, proof size, requirement for trusted setup and security assumption. zkSNARKs relies on elliptic curve cryptography where proof generation is expensive while proof verification is cheap and proof size is short [16]. However, zkSNARKs requires a one-time trusted setup to generate a common reference string. The cheap proof verification and short proof make zkSNARKs suitable to be used in real-world blockchain applications. zkSTARKs relies on hash functions where proof generation is expensive and proof size is large while proof verification is cheap [31]. However, zkSTARKs do not need any trusted setup, which makes it more preferable for dynamic applications with minimal deployment overhead (e.g. Layer-2 roll-ups). Finally, Bulletproofs relies on elliptic curve cryptography where proof verification is relatively more expensive while proof size is not constant [32]. As zkSTARKs, Bulletproofs do not need trusted setup as well. In our thesis, we especially

use zkSNARKs for proof generation and verification because of its easy integration to EVM-based blockchain platforms.

There exist three main security requirements for a protocol to be a zero-knowledge proof protocol where we present their brief definitions with the following way:

- Completeness: If the statement is true, the prover can probabilistically convince the verifier about the correctness of that statement:

$$x \in L \rightarrow \texttt{Pr}[(\mathcal{P}, \mathcal{V})(x) \neq 1] < \epsilon. \tag{2.36}$$

- Soundness: If the statement is not true, the prover cannot probabilistically convince the verifier about the correctness of that statement:

$$x \notin L \rightarrow \texttt{Pr}[(\mathcal{P}, \mathcal{V})(x) = 1] < \epsilon. \tag{2.37}$$

- Zero-Knowledge: The transcription of interactions between the prover and the verifier does not yield anything except the correctness of the statement itself:

$$(\mathcal{P}, \mathcal{V})(x) \approx \mathcal{S}im(x). \tag{2.38}$$

We can classify proof systems with respect to the type of zero-knowledge property as: (i) *perfect zero-knowledge* where there exists a simulator for every verifier that generates exactly the same transcription between the prover and that verifier, (ii) *statistically zero-knowledge* where these two transcriptions are statistically indistinguishable and (iii) *computationally zero-knowledge* where these two transcriptions are computationally indistinguishable. In similar fashion, we can classify proof systems with respect to the type of soundness property as: (i) *statistical soundness* where the soundness holds correct against the computationally-unbounded prover (i.e. even such adversarial prover cannot convince the verifier) and (i) *computational soundness* where the soundness holds correct only against the computationally-bounded prover

Table 2.4. Schnorr's protocol as sigma protocol.

Schnorr's Protocol
Common Inputs:
Witness: $h \leftarrow g^x, \quad x \in \mathbb{Z}_p$
Protocol:
(1) $\mathcal{P}$: picks a random value $r \xleftarrow{\$} \mathbb{Z}_p$ and forwards $u \leftarrow g^r$ to $\mathcal{V}$
(2) $\mathcal{V}$: picks a random challenge $c \xleftarrow{\$} \{0,1\}$ and forwards it to $\mathcal{P}$.
(3) $\mathcal{P}$: computes $z \leftarrow c \cdot x + r$ and forwards z to $\mathcal{V}$.
Verification: $\mathcal{V}$ accepts if and only if $g^d \stackrel{?}{=} h^e \cdot u$.

(i.e. such adversarial prover may convince the verifier). We generally refer to the proof systems with *computational soundness* as the *argument* systems [65].

Proof systems can be also classified with respect to the interaction between parties as: (i) *interactive* where the prover exchanges several interactions with the verifier to convince him and (ii) *non-interactive* where the prover can generate proof and later the verifier can verify that proof without any interaction. The most famous *interactive* proof system is three-way *sigma* protocols as: (i) the prover first commits to a secret χ, (ii) the verifier generates a random challenge c and (iii) the prover generates a corresponding response z with respect to that challenge. There exist numerous instances of *sigma* protocols including *Schnorr's* protocol [28], *Fiat-Shamir* protocol [29] and *Guillou-Quisquater* protocol [30] where we briefly show how a simple *Schnorr's* protocol works in Table 2.4. According to that table, $\mathcal{P}$ has the secret χ that $\mathcal{V}$ does not know. Firstly, $\mathcal{P}$ picks completely a random value r and forwards the result of the discrete logarithm function $u \rightarrow g^r$ to $\mathcal{V}$. Secondly, $\mathcal{V}$ picks completely a random challenge c and forwards it to $\mathcal{P}$ where $\mathcal{P}$ does not know this challenge in advance. This compels $\mathcal{P}$ to be ready for all possible challenges by preventing him to cheat. Thirdly, $\mathcal{P}$ generates response $z \rightarrow c \cdot \chi + r$ and forwards z to $\mathcal{V}$. Note that if $\mathcal{V}$ selects the challenge $c \leftarrow 0$, $\mathcal{V}$ learns only $z \rightarrow r$. However, if $\mathcal{V}$ selects the challenge $c \leftarrow 1$, $\mathcal{V}$ learns $z \rightarrow \chi + r$ where χ is successfully masked.

Proof systems can be classified with respect to the the problems they are built upon as: *number-theoretic* where such systems rely on difficulty of problems from number theory involving integers, prime numbers, modular arithmetic and finite groups and *graph-theoretic* where such systems rely on structural complexity and combinatorial property of problems from graph theory including graphs, edges, vertices and their relations. The *number-theoretic* approaches include the discrete logarithm problem as:

$$h \leftarrow g^{\chi} \tag{2.39}$$

where $g, h \in \mathbb{G}$ are generators of cyclic group $\mathbb{G}$ while χ is the secret. As clear in Table 2.4, the *Schnorr's* protocol uses the discrete logarithm [28]. It includes quadratic residuosity problem as:

$$h \leftarrow \chi^2 \tag{2.40}$$

where h is the quadratic residue. The *Fiat-Shamir* protocol uses this problem to build the proof system [29]. Finally, it includes integer factorization problem as:

$$h \leftarrow \chi \cdot y \tag{2.41}$$

where χ and y are prime numbers where splitting h into the prime numbers is difficult. The *Guillou-Quisquater* protocol uses this problem to build the proof system [30]. On the other hand, the *graph-theoretic* approaches include (i) *graph isomorphism* problem where the prover shows isomorphism among two graphs without revealing the isomorphism itself and the (ii) *graph-3 coloring* problem where the prover shows the validity to map each vertex into one of three colors where no adjacent vertices have the same color without revealing the mapping itself [66].

In this thesis, we abstract the functionalities of the zero-knowledge proof scheme (with domain abbreviation `Zk`) with the following three representations as (i) the setup function `Zk.Setup(.)` to generate a proving key *pok* used during proof generation and

a verification key *vek* used during on-chain proof verification, (ii) the proof generation function `Zk.Gen(.)` to generate proofs off-chain based on certain computational statements, public inputs and private inputs and (iii) the proof verification function `Zk.Vfy(.)` to later verify these proofs on-chain based on only the public inputs. We will use the following notations:

$$(pok, vek) \leftarrow \texttt{Zk.Setup}(\lambda) \tag{2.42}$$

$$\pi \leftarrow \texttt{Zk.Gen}(\Psi, x, c, pok) \tag{2.43}$$

$$b \leftarrow \texttt{Zk.Vfy}(\pi, c, vek) \tag{2.44}$$

where π is the proof of the proof generation function while b is the resulting boolean value (i.e. true if the proof is correct and vice versa) of the proof verification function.

Zero-knowledge proof scheme on blockchain is shown in Figure 2.10 where there are two parties as usual including prover (i.e. Alice) who proves the correctness of a statement (i.e. computation) over certain private data and verifier (i.e. Bob) who verifies the resulting proof of the prover. In two-phase protocol, Alice first needs to make a commitment with `Cm.Comm(.)` over her private data (review Section 2.6) and submit it to the contract `setCommitment(.)` in the first phase. Later in the second phase, Alice generates a zero-knowledge proof in order to show the correctness of the computation function Ψ with private data itself (x and σ) and its public commitment C via the proof generation function `Zk.Gen`; and submits the proof to the contract with `submitProof(.)`. Afterwards, Bob verifies this proof π only with the public commitment c via the proof verification function `Zk.Vfy`. Firstly, it should be noted here that the proof generation requires both public and private inputs while the proof verification requires only the public inputs. With that way, Bob as a verifier learns nothing about the statement of Alice except the correctness of the statement itself (i.e. zero-knowledge proof property). Secondly, we do not need Bob here to trigger the `verifyProof(.)` function. Instead, Alice as a prover can trigger this function at the same time she submits the proof. It is the on-chain verifiability property of proof that allows this scheme to work even without Bob itself.

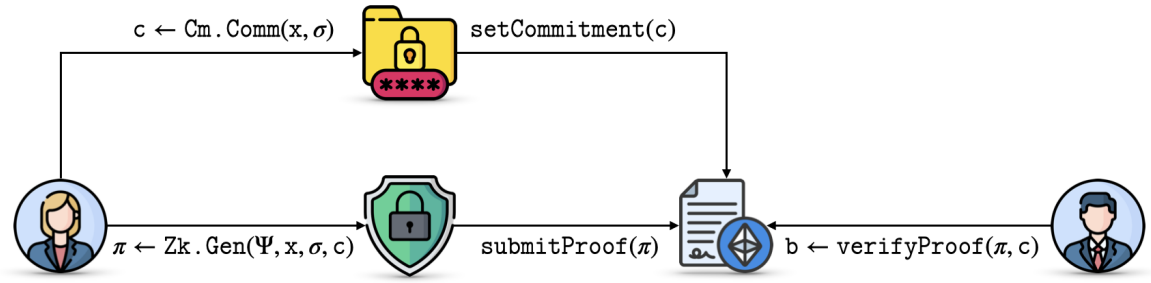


Figure 2.10. Zero-Knowledge proof on blockchain.

In our thesis, we heavily rely on zero-knowledge proof: (i) to prove that balance is sufficient to deposit certain amount of tokens without revealing the balance and the amount in our privacy-preserving payment protocol, (ii) to prove that the data is correctly aggregated based on two other data without revealing them in our privacy-preserving data aggregation protocol and (iii) to prove that the certain tokens are bartered in return for other tokens in bid without revealing the bid itself. In the scope of our thesis, we define zero-knowledge proof with the following way: "*privacy-preserving and publicly-verifiable on-chain state transition function for off-chain computation*": (i) it is *privacy-preserving* since it surely protects the privacy of the private data, (ii) it is *publicly-verifiable* since validators of blockchain can verify the correctness of this function, (iii) it is *on-chain state transition function* since it updates the state of a certain variable (e.g. commitment) with another state on-chain while ensuring the correctness of that transition and (iv) it is for *off-chain computation* since prover performs the computation, resulting in this state transition, out-of-blockchain. We mathematically represent this definition with the following way:

$$\begin{aligned} c &\leftarrow c' \cdot [\texttt{Zk.Vfy}(\pi, c)] + c \cdot [1 - \texttt{Zk.Vfy}(\pi, c)] \\ c &\leftarrow c' \cdot b + c \cdot [1 - b] \end{aligned} \tag{2.45}$$

where it changes the state by replacing the current commitment c with the next commitment c' if the proof is correctly verified (i.e. $b \leftarrow 1$) or leaves it as it is if not (i.e. $b \leftarrow 0$). For instance, the zero-knowledge proof verification scheme changes the state of the balance from one commitment to another commitment once the statement of

depositing tokens from that balance is correctly verified.

2.9. On ZoKrates Framework

ZoKrates is a scalable and privacy-preserving proof generation and verification framework that is directly compatible with Ethereum [27]. It specifically uses the zk-SNARKs (Zero-Knowledge Succinct Non-Interactive Arguments of Knowledge) proof protocol [16] which satisfies (i) *succinctness* (i.e. shortness) in terms of proof size and cheapness in verification, (ii) *non-interactivity* between prover and verifier; and (iii) the *zero-knowledge property*. There exist two advantages of the ZoKrates framework. First, it moves the expensive proof generation operation out of blockchain without compromising any security and enables cheap on-chain verification of off-chain computation where the off-chain node becomes prover while blockchain validators become verifier. This introduces a certain degree of (i) efficiency and scalability by preventing potential high blockchain gas consumption and (ii) predictability by having constant verification cost regardless of any computation. Second, it abstracts the complex mathematical and cryptographic operations of zero-knowledge proof into high-level domain-specific language implementation. In our thesis, we heavily rely on the ZoKrates framework to generate proofs off-chain and later to verify these proofs on-chain.

The architectural design of the ZoKrates framework consists of six interdependent modules as *parser*, *flattener*, *witness generator*, *R1CS*, *libsnark* and *contract generator* [27]. The *parser* and the *flattener* (together as the *compiler*) modules first transform the high-level domain-specific code into the linear code with variable definitions and assertions. The *witness generator* module executes the linear code by collecting private and public inputs and stores the variable values as the witness. The *R1CS* module transforms the linear code into the Rank-1 Constraint System to be compatible with the *libsnark* circuit generation framework. The *libsnark* module is a cryptographic zkSNARKs library written in C++ to perform one-time key generation setup (for proving and verification keys) and to generate zero-knowledge proofs. The proving key is used during off-chain proof generation while the verification key is used

during on-chain proof verification. The *contract generator* automatically generates an Ethereum-compatible verifier smart contract that is ready to be deployed for proof verification. The smart contract itself wraps the verification key for complex mathematical verification computation. The verifier contract includes *verifyTx* to be externally called through proof and public inputs to return a boolean variable where *true* indicates the correctness of proof or vice versa. The result of the *verifyTx* function can be further processed to branch the execution of the function that calls it. It should be noted that proof size remains always fixed while the public input array size can vary application to application, which may have an impact on blockchain gas consumption to a certain extent. The whole architecture of the ZoKrates framework is depicted in Figure 2.11.

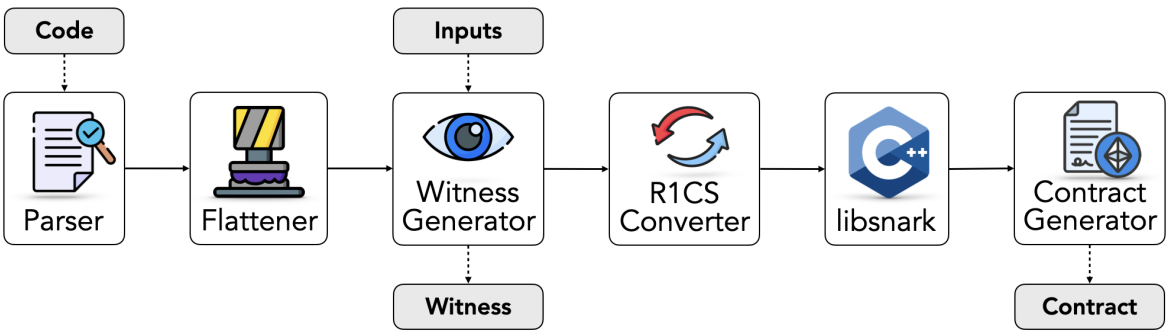


Figure 2.11. Architecture of ZoKrates framework [27].

3. RELATED WORK

In this chapter, we focus on reviewing the relevant works from the literature where we believe that they will benefit to properly position our current thesis with respect to the existing works. We generally target three specific domains (although not necessarily limited to) as privacy-preserving payment, privacy-preserving data aggregation and privacy-preserving bartering problems.

3.1. On Privacy-Preserving Token Transfer

AZTEC (i.e. Anonymous Zero-Knowledge Transactions with Efficient Communication) refers to a set of zero-knowledge proof functions on Ethereum for privacy-preserving transactions where the transaction amount is private. These functions include (i) join-split, (ii) bilateral swap, (iii) dividend proof, (iv) mint, (v) burn, (vi) private range and (vii) public range. For instance, the join-split function privately collects multiple input notes to combine them in the join phase and privately splits it into multiple output notes in the split phase while still ensuring the equality of the total input values to the total output values. In addition, the private range function is used to prove that a note is greater than another note while the public range function is used to prove that a note is greater than a public integer. AZTEC implements an interactive sigma proof protocol and a commitment scheme where it applies the Fiat-Shamir heuristic [29] to transform this proof protocol into non-interactive proof protocol. This implementation is managed through a smart contract as ACE (AZTEC Cryptographic Engine). However, the drawback of AZTEC is that it requires the values to be represented as notes in advance, which introduces additional overhead (i.e. conversion from values to notes and later from notes to values).

Zcash [13] proposes privacy-preserving transactions using zero-knowledge proof (i.e. zkSNARKs [16]) upon the existing transparent transaction scheme of Bitcoin [1]. Without loss of generality, there exists a single form of address (i.e. public address)

and a single form of transaction (i.e. public transaction) in a transparent blockchain platform. However, Zcash extends it by introducing two address types as private z-addresses and public t-addresses and four transaction types as *shielded* z-to-z transaction, *shielding* t-to-z transaction, *deshielding* z-to-t transaction and *public* t-to-t transaction. The distinction between Zcash and AZTEC lies in their architectural design and privacy guarantees. While Zcash is an independent blockchain platform, AZTEC is a privacy-preserving protocol running over contracts to enhance privacy within the existing infrastructure of Ethereum. Additionally, Zcash ensures transactional privacy not only by hiding transaction amounts but also by anonymizing the source and destination addresses. Furthermore, Zcash introduces the idea of viewing keys for transactions so that auditing organizations (e.g. governments) can selectively access transaction details without disclosing to other parties.

Zerocash proposes another privacy-preserving payment protocol with strong mathematical guarantees by just integrating an extra privacy layer through zkSNARKs over the existing Bitcoin platform [5]. It basically introduces two types of exchangeable assets as *basecoins* and *zerocoins* where transactions involving with basecoins remain fully transparent, similar to the standard Bitcoin transactions. However, the transactions with zerocoins ensures privacy by hiding the transaction source and destination addresses as well as the transaction amounts. The parties can also privately split and merge zerocoins while still preserving their total values. This results in more flexible design where the parties can freely choose the option with respect to their current preferences. Zerocash enables the transformations from basecoins to zerocoins through the *mint* transactions and from zerocoins to basecoins through the *pour* transactions while validating the correctness of these operations with zero-knowledge proof verification.

Zether [41] is another privacy-preserving payment protocol for smart contract-based blockchains including Ethereum. It utilizes non-interactive zero-knowledge protocol, public-key encryption scheme and digital signatures where it first transforms *Ether* to Zether to provide privacy. Our privacy-preserving payment protocol (i.e. *PTTS*) differs from Zether in many ways. First, Zether relies on Bulletproofs [32] for

zero-knowledge proof protocol while we use zkSNARKs [16] for our privacy-preserving payment protocol (i.e. *PTTS*) where it is known that this protocol requires relatively longer time to generate and verify proofs [67]. Second, Zether offers a locking or pending transaction mechanism in order to prevent front-running problem where a transaction may eliminate the validity of another transaction. However, there is no front-running problem in our protocol since user balances are updated immediately once the proofs are verified. Third, Zether requires +7M gas consumption for payments while our protocol needs comparably less gas consumption, (see Section 4.5.3).

There also exist several novel payment protocols in the literature. The following work [68] proposes a decentralized payment protocol (i.e. UTT) that focuses on accountable privacy. It implements a novel permissioned blockchain infrastructure with Byzantine Fault Tolerant (i.e. BFT) where parties perform privacy-preserving payments up to a certain threshold per month. Another work [69] proposes an efficient non-interactive coin mixing protocol with minimal cryptographic assumptions (i.e. Veksel) that uses the one-out-of-many zero-knowledge proof system where parties can prove their knowledge of commitments within the set of commitments. It defines a novel non-interactive zero-knowledge protocol with constant proof size and verification time. Comparably, it needs cheaper trusted setup in return for larger transaction size and slower verification time, compared to Zcash [13]. The final work [70] proposes a privacy-preserving account-based payment system (i.e. Platypus) based on zkSNARKs where it requires a central body to verify proofs and maintain a public transaction log. In addition, it imposes a global maximum holding limit for ordinary balances.

As there exist privacy-preserving payment protocols, there are also cleverly-designed attacks available in the literature to exploit vulnerabilities of these protocols. The following work [40] provides a feature-based blockchain network analysis framework to identify the statistical properties of transaction mixing services in three levels (e.g. network level, account level and transaction level) and identifies several temporal transaction motifs to train a learning model. A temporal motif is expressed through the number of nodes and transactions it involves as well as the time within which these

transactions occur. The following work [71] introduces a novel cryptographic attack to the Monero blockchain platform [72] that obfuscates the transaction owners by adding multiple decoy addresses through ring signatures that are claimed to be indistinguishable. The attack aims to reduce the anonymity of these transactions with three phases (i.e. preparation phase, setup phase and attack phase). Finally, the work [43] builds transaction network and run address clustering heuristics (e.g. multi-input heuristic, change heuristic and variable change heuristic) on the Zcash blockchain platform [13]. The attack in this thesis differs from these attacks on certain points: (i) it requires a large adversarial organization to attack, (ii) that organization should collect the privacy-preserving payment transactions from the transaction owners to build a global transaction graph and (iii) it runs the minimum cost flow network over the graph to calculate the minimum and maximum bounds for the balance it attacks to.

3.2. On Privacy-Preserving Data Aggregation

The following work [73] reviews the existing general-purpose privacy-preserving aggregation protocols in the literature and proposes three novel aggregation protocols as well. These protocols include (i) Basic Secure Summation (i.e. BSS) with random masking, (ii) Encrypted Secure Summation (i.e. ESS) with homomorphic encryption, (iii) Salted Secure Summation (i.e. SSS) with random salt masking, (iv) Randomly-shared Secure Summation (i.e. RSS) with secret splitting, (v) Distributed Secure Summation (i.e. DSS) with public-key encryption, (vi) Two-Segment Secure Summation (i.e. TSS) with homomorphic encryption and finally (vii) Homomorphically-Shared Secure Summation (i.e. HSS) with partially-homomorphic encryption. The work theoretically performs privacy and complexity analysis (in terms of computational and communication complexity) of all these protocols. It supports the theoretical findings through empirical results as well by measuring runtimes to aggregate with the increasing number of parties. This work expresses the essential ways to aggregate data while it omits the challenges of the platform over which that aggregation is performed. In our thesis, we utilize the random salt masking technique to build our protocol (i.e. *PVSS*) and the secret splitting technique to extend that protocol for incomplete hypercubes.

The work [74] proposes a novel framework (i.e. CypherChain) specifically for privacy-preserving data aggregation by integrating two different protocols as Cypher (i.e. Collaborative Cyphertext Processing) and Chain (i.e. Clustered Hypergraph Aggregation and Interaction). In the Cypher protocol, the parties first encrypt their data with their public keys and the resulting encryptions are distributed over the existing parties (similar to the secret splitting technique in the previous work). They perform homomorphic computations over all the splits they have. Later, the Chain protocol applies a global aggregation function over all these partial aggregations to compute the final global aggregation through a hypergraph coloring algorithm. Based upon the experimental results over the real-world dataset, the proposed framework achieves notable improvement over the computational costs and aggregation speed.

The work [75] proposes a general-purpose privacy-preserving data aggregation protocol (i.e. Masquerade) on public blockchains to compute certain private statistical operations including sums, averages and histograms. The work has several actors including (i) participants with certain private data and interested in the computation result, (ii) curator for collecting and aggregations the encryptions through homomorphic encryption schemes, (iii) analyst for checking the correctness of the participant proofs and decrypting the encryption aggregation to reveal the computation result and (iv) auditors for checking the correctness of encryptions and proofs. The proposed protocol there has five stages as (i) *key-generation* where the analyst generates and distributes the public keys to the participants, (ii) *encrypt-prove-commit* where the participants encrypt the private data with these public keys and generate zero-knowledge proof for the correctness of their local computations, (iii) *aggregation* where the curator verifies the correctness of the proofs to aggregate the encryptions in order to obtain the final encryption, (iv) *audit-decrypt* where the analyst receiving the final encryption decrypts it through the private key to learn the computation result and (v) *public verification* where auditors publicly check the correctness of all encryptions and proofs. This work successfully integrates homomorphic encryptions and zero-knowledge proofs for privacy-preserving data aggregation, but with sacrificing decentralization by introducing multiple trusted parties (e.g. curator and analyst) to the protocol, which

is a significant issue to consider. The general architecture of the Masquerade protocol is given in Figure 3.1 where the interactions of the actors with the bulletin board (e.g. public blockchain) are also shown.

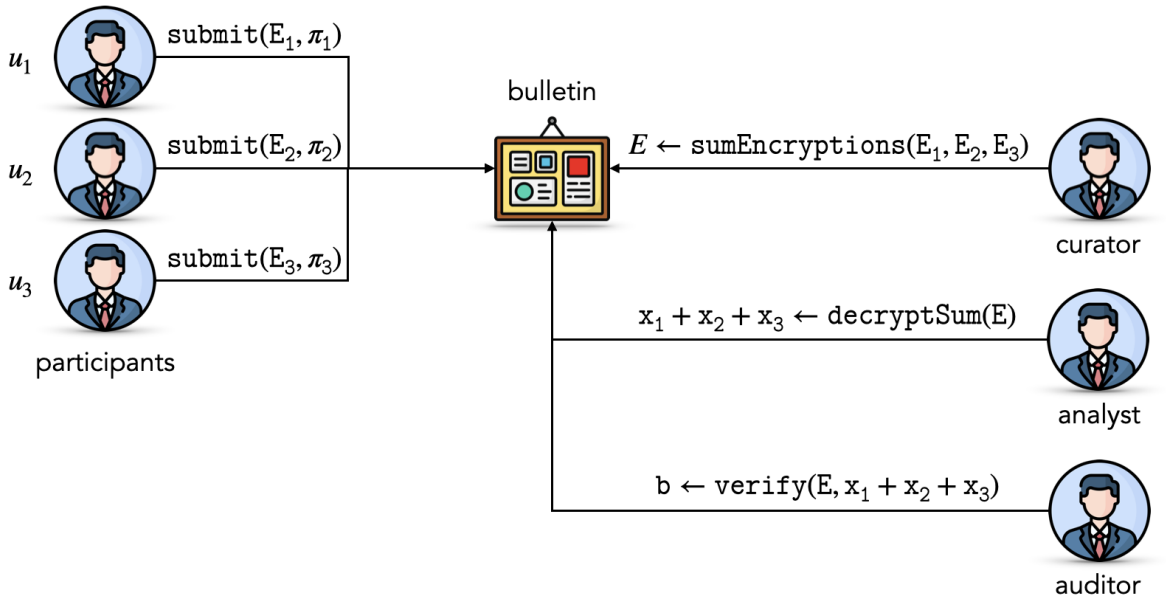


Figure 3.1. Architecture of Masquerade [75] with trusted parties.

The work by [47] proposes a novel privacy-preserving energy storage sharing protocol on blockchain by utilizing a traditional privacy-preserving data aggregation scheme (i.e. SPDZ [76]) where the parties can compute the total energy demand without revealing their individual energy demands. The work includes four phases as (i) *initialization* where the system parameters are chosen and the SPDZ preprocessing is completed, (ii) *pre-operation scheduling* where the parties aggregate their day-ahead energy demands via the SPDZ protocol, (iii) *cost-sharing operation* where the parties split the total costs of the energy storage service based on a certain cost-sharing scheme via the SPDZ protocol again and (iv) *post-operation* the parties sign individual energy storage service receipts. For most simplistic terms, SPDZ divides a private value of a party into multiple chunks and distributes each chunk to a different party over the system. The construction of the same private value consequently requires the collaboration of these parties having the chunks. The work [47] successfully integrates

this SPDZ protocol into the non-interactive zero-knowledge proof scheme for privacy-preserving energy storage sharing application without the need for trusted setup. On the other hand, the main limitation here is that the computational and the communication overheads of the protocol linearly increase (i.e. in non-scalable fashion) with the increasing number of parties where blockchain gas consumption exceeds the current limit at 25 parties. In our thesis, we propose a more scalable privacy-preserving data aggregation protocol where the overhead now logarithmically increases.

E-voting is another form of data aggregation where the votes of the individual voters must be tallied to determine the winner of the election. The following work [46] proposes a novel privacy-preserving voting protocol on blockchain (i.e. ethVote) by integrating three different smart contracts. The registration contract is governed by the registration authority that is able to register the voters to a certain election; the election factory contract is governed by the election authority to create and deploy election contract instances to blockchain for every new election; and the election contract collects and aggregates the individual votes from the voters using homomorphic encryption and zero-knowledge proof protocols. The work suffers from two main drawbacks as (i) availability of trusted parties (i.e. registration and election authorities) and (ii) lack of correct implementation of zero-knowledge proof due to some technical limitations of ZoKrates [27].

The work [77] proposes a novel privacy-preserving protocol that integrates deep learning and homomorphic encryption to provide an end-to-end data aggregation system for smart grids. The protocol consists of interconnection of three different layers as:(i) *smart meters* to collect encrypted electricity consumption of every party in a specific home area network, (ii) *aggregation providers* to decrypt, verify and then aggregate the encryptions of several home area networks and (iii) *power centers* to distribute electricity with respect to the resulting aggregations. The work analyzes the proposed protocol from the scalability and security perspectives by considering computational and communication overheads with the increasing number of smart meters. According to the experimental study, the protocol shows improvement over the exist-

ing approaches in the literature to a certain extent while it still relies on the central authorities (i.e. aggregation providers) to aggregate data.

The work [78] focuses on the privacy-preserving data aggregation and model training problem for federated learning of IoT devices. It proposes a hierarchical approach where private data are progressively aggregated from the local nodes to the cluster models, to the edge models and finally to the global models. The work also presents an aggregation node selection protocol (i.e. DDPG) in order to select the most optimal set of nodes for the aggregation. However, the work suffers from the existence of the central authorities (i.e. leaders) during the data aggregation and the result verification stages. Meanwhile, it has a lack of sufficient experiment study including blockchain gas consumption. When taking all these relevant works into consideration, it can be said that the aggregation protocol that we will propose enables large-scale and general-purpose data aggregation for summation privately, collectively, distributedly and without any central authority on the blockchain environment.

3.3. On Token Bartering

The work [79] classifies the bartering problems with respect to the number of distinct items (i.e. types) and the number of instances of these items (i.e. units) to be bartered into four main categories as: (i) single-item single-instance, (ii) single-item multi-instance, (i) multi-item single-instance, (i) multi-item multi-instance. This work specifically addresses the inclusive and exclusive multi-item multi-instance resource bartering over grids where barterers can exchange a set of resources with multiple instances with another set of resources. It aims to solve this problem through integer programming with directed hypergraphs. It generates test scenarios through four parameters as the number of barterers, the maximum number of resources a barterer can have, the maximum number of bids a barterer can propose and the maximum number of resources a barterer can demand in a single bid. The maximum number of resources a barterer can supply is naturally limited to the number of resources owned. For various configurations of these parameters, the work justifies the validity of the proposed

algorithm by presenting experimental results in terms of solution times.

The work [80] presents three different bartering models for real-world problems including domain-name, book and music bartering. These models focus on (i) single-item single-instance bartering, (ii) single-item multi-instance bartering and (iii) multi-item single-instance bartering where it solves the first two polynomial-time problems with minimum cost flow networks and the third NP-Hard problem with integer programming integrated with directed hypergraphs. The main objective of these models is to maximize the number of items to be bartered along with several additional objectives as well including maximizing the barterer priorities and maximizing the total fees collected. It performs several experiments with varying size of items, instances and barterers to measure the solution times. Although this work successfully addresses digital bartering over the internet, it differs from our work to a great extent where we mainly discuss privacy-preserving and collective bartering over a decentralized network.

The similar work [81] proposes a hybrid auction-barter solution for digital used-car bartering problem with three different bid types as: (i) *sale bids* to supply a car in auction, (ii) *purchase bids* to demand a car in auction and (iii) *barter bids* to trade a single car with another single car with differential price (i.e. difference of cars to be bartered in value). Note that this problem is in the single-item single-instance bartering category. The solution consists of two phases as *non-bidding* phase where barterers can modify their own bids (e.g. adding, deleting or updating bids) and *bidding* phase where barterers must perform the actions they commit in the previous phase (e.g. supplying or demanding cars). The objective of the work is to maximize the profit of the auction owner by solving the polynomial-time problem with minimum cost circulation network. This work suggests the notion of private bids to be integrated as well since an adversarial barterer (having the bids of all barterers) may construct and solve the global graph bid graph to modify their own bid. In parallel with that suggestion, we propose a privacy-preserving bartering protocol (i.e. *PMTBS*) in order to protect the privacy of the barterer bids along with their balances.

The following work [9] advances the notion of the digital bartering to the next level as the autonomous and decentralized bartering over blockchain, (i.e. *Barter Machine*). It specifically addresses the multi-item (i.e. multi-token) multi-instance bartering problem for both fungible (complying to the ERC-20 standard) and non-fungible (complying to the ERC-721 standard) tokens where certain blockchain addresses (i.e. barterers) initially propose their bids to the contract. Having all the bids, certain other blockchain addresses (i.e. the solution providers) solves the resulting problem to return feasible bartering solutions. This problem has certain characteristic challenges to be addressed including when to withdraw tokens to be supplied from the balances where two main approaches are proposed: (i) withdrawing directly while proposing bids or (ii) withdrawing while constructing the solutions afterwards. The contract also enables the solution providers to lock their solutions to prevent the other providers from proposing the same solution. The goal of the problem is to maximize the profit of the contract by maximizing the number of tokens that are left over once the bartering is completed. The work justifies the applicability of the proposed technique on Ethereum in terms of blockchain gas consumption with the increasing number of bids. However, the privacy of bids and balances still remains to be addressed in this work as well.

The following work [82] focuses on the digital bartering of the physical items through non-fungible tokens where it employs blockchain mainly for providing traceability and transparency over the bartering information; and verifying the digital identities of barterers. Such information include the blockchain addresses of barterers and the items to be exchanged. This work designs a fair-price heuristic to generate a global price matrix by dynamically calculating the exchange prices of the available items with respect to the current supply and demand weights. However, this work greatly lacks experimental performance of the proposed technique (e.g. blockchain gas consumption) where we can't observe the scalability of that matrix generation with respect to the increasing number of barterers or items to be exchanged. Moreover, the work mainly emphasizes the importance of transparency rather than the privacy of such design.

Swapping as direct and one-to-one exchange of items can be considered as a

more simple form of bartering. Uniswap [83] is one of the popular swapping protocols on Ethereum, which enables trading tokens without any intermediary party. Uniswap utilizes liquidity pools over smart contracts to collect tokens from the parties (i.e. swappers) and set their prices with respect to the current supply and demand values. The main problem of Uniswap is that it is fully transparent where anyone can trace transactions of swappers on these contracts (i.e. what they supply and demand), which may reveal important information for an adversarial party. Secondly, it does not inherently support multi-token exchange that leads to inefficiencies in certain scenarios. There exist other swapping protocols as well including PancakeSwap [84] and SushiSwap [85].

The work [86] proposes a generalized privacy-preserving bartering protocol with only two parties (i.e. barterers) for three types of bartering scenarios: (i) exchanging single-item, (ii) exchanging single item in return for multiple times and (iii) exchanging multiple items in return for multiple items. The protocol utilizes a homomorphic encryption scheme and assumes that these two barterers (e.g. Alice and Bob) have their own utility functions that map items to utility valuations. According to this protocol: (i) Bob submits his own valuation encryptions to Alice, (ii) Alice pairs these encryptions with her items and sorts the pairs based on her own preferences and (iii) they interact multiple times to find the best match by comparing their encryptions. However, there exist several drawbacks of this protocol. First, the protocol supports only two barterers, which hinders its adoption and scalability to a great extent. Second, it requires these two barterers to interact several times to reach the conclusion to barter. Third, the barterers need to maintain utility functions. Finally, the barterers may barter only subsets of their bids since the items to be bartered are known after the privacy-preserving comparison. The following works [87–89] propose privacy-preserving bartering protocols with homomorphic encryption schemes for only two parties while the work [90] proposes for multiple barterers by repeating the two-party protocol among multiple parties. However, in this thesis, we address decentralized, privacy-preserving, non-interactive and scalable bartering among multiple barterers on blockchain.

4. PROTOCOL I: PRIVACY-PRESERVING PAYMENT

- quis quid cui transfert?
(who transfers what to whom?)

In this chapter, we define the privacy-preserving token payment problem on blockchain by specifying the objectives, constraints and requirements. For the given problem, we propose our privacy-preserving payment protocol (i.e. *PTTS*) that integrates commitment scheme, public-key encryption and zero-knowledge proof under three main models as the zero-knowledge proof model, the smart contract model and the web interface model. Inspired by our protocol, we propose a novel attack scheme (i.e. balance range disclosure attack) to formalize the way to learn balances from privacy-preserving payments if certain criteria are satisfied. We analyze our protocol with respect to scalability (through computational, communication and storage overheads) and security (through attacks and formal reduction proofs). We also perform experiments to measure its performance with the blockchain gas consumption, zero-knowledge proof generation/verification times and zero-knowledge proof artifact sizes.

4.1. Problem Definition

Privacy-preserving payment problem refers to a specific group of transactions where a source address (i.e. sender) transfers a certain amount of tokens to a destination address (i.e. receiver) while still protecting the privacy of their balances and transaction details. This problem has two kinds of information asymmetry: (i) each address possesses their own private balance while the other addresses need it to verify the correctness of the transaction involving with that address and (ii) two transaction-involving addresses know transaction amount while other addresses need it to verify the correctness of the transaction. For the problem, let $\mathcal{U}$ a set of blockchain addresses:

$$\mathcal{U} : \{u_1, u_2, \ldots, u_{N-1}\} \tag{4.1}$$

where u_i is the ith user while N is the maximum number of users. Each user is associated with:

$$u_i : \langle \theta_i, pk_i, sk_i \rangle \tag{4.2}$$

where θ_i is the private balance, pk_i public key and sk_i is the secret key of the ith user. We assume that all values in the set of balances Θ must be specified from the set of only positive integers including zero (i.e. no balance can be negative or real number):

$$\Theta : \{\theta_1, \theta_2, \ldots, \theta_n\}, \quad \theta_i \in \mathbb{Z}^+ \cup \{0\}. \tag{4.3}$$

In the scope of privacy-preserving payment problem, we define a payment (i.e. a regular transaction on blockchain) between two blockchain addresses with the following way:

$$tx_{sr} : \langle \Delta, u_s, u_r \rangle \tag{4.4}$$

where tx refers to a transaction and Δ is the total transaction amount that the sender address u_s transfers to the receiver address u_r. We assume that Δ can be only positive integers excluding zero as $\Delta \in \mathbb{Z}^+$, (i.e. no transaction amount can be fractional).

As defined in Section 2.1, state of a blockchain (i.e. account-based blockchains) can be expressed through a set of current balances $\mathcal{BC} : \langle \theta_0, \ldots, \theta_s, \ldots, \theta_r, \ldots, \theta_{N-1} \rangle$. Each transaction tx changes this current state into the next state by applying the following effects without loss of generality:

$$\theta'_s = \theta_s - \Delta \tag{4.5}$$

$$\theta'_r = \theta_r + \Delta \tag{4.6}$$

$$0 < \Delta \leq \theta_s \tag{4.7}$$

$$\theta_s, \theta'_s, \theta_r, \theta'_r \geq 0 \tag{4.8}$$

where θ_s and θ_r are the current and θ'_s and θ'_r are the next balances of the sender and receiver, respectively. We define these equalities (as well as inequalities) as the main constraints of a valid state transition. The constraint in Equation (4.5) deposits transaction amount Δ from the sender current balance while the constraint in Equation (4.6) withdraws this amount to the receiver current balance. The constraint in Equation (4.7) ensures that the sender current balance must be sufficient to deposit this amount while the constraint in Equation (4.8) requires all these parameters to at least equal to or to be greater than zero. We assume that total balance across all addresses must be preserved during that state transition as:

$$\sum_{i=0}^{N-1} \theta'_i = \sum_{i=0}^{N-1} \theta_i \tag{4.9}$$

where it requires the sums of all balances in the current state must be exactly equal to the sums of all balances in the next state, which is also equal to the total token supply. Note that this is valid for a regular transaction tx where more complex transactions may naturally violate this assumption by mining or burning tokens. But, this requires a more comprehensive transaction definition as $tx^+ : \langle \Delta^+, u_i \rangle$ for minting and $tx^- : \langle \Delta^-, u_i \rangle$ for burning where Δ^+ and Δ^- are the amounts of tokens to be minted and burned, respectively while still satisfying $\Delta^- \leq \theta_i$ for balance sufficiency.

The definitions so far comprise only a public payment among two addresses without considering privacy-preserving characteristics of the problem that we have previously pledged. Now, we introduce two different privacy goals as the *balance privacy* where no address must see the balance of another address and *transaction privacy* where no address (except the sender and receiver addresses) must see the transaction details of a certain transaction. For the balance privacy:

$$\forall u_j \neq u_i : \Pr[\theta_i \mid \text{view}(u_j)] = \Pr[\theta_i] \tag{4.10}$$

where $\text{view}(u_j)$ simply refers to the information that user u_j has access to. This statement implies that there isn't any address u_j whose view on the balance of the

address u_i is probabilistically equal to the view of the address u_i itself. Similarly for the transaction privacy:

$$\forall u_k \notin \{u_i, u_j\} : \Pr[tx_{ij} \mid \text{view}(u_k)] = \Pr[tx_{ij}] \tag{4.11}$$

where it implies that there isn't any address u_k whose view on the transaction tx_{ij} between the addresses u_i and u_j probabilistically equal to the views of the addresses u_i and u_j themselves. From this perspective, we can now define the main objectives of the problem (i.e. maximizing these privacy goals) with the following way:

$$\min_{u_j} \quad \mathtt{I}(\text{view}(u_j); \theta_i), \quad \forall u_j \neq \{u_i\} \tag{4.12}$$

$$\min_{u_k} \quad \mathtt{I}(\text{view}(u_k); tx_{ij}), \quad \forall u_k \neq \{u_i, u_j\} \tag{4.13}$$

where $\mathtt{I}(.)$ is the mutual information measurement function which simply quantifies the amount of information of a certain address on a certain value in our scope. We conceptually introduce this measurement just to formalize the problem objectives where the exact quantification approach of this function is out of our scope. The statement given in Equation (4.12) implies that the first goal of the problem is to minimize the amount of information the address u_j accesses to over the balance θ_i that the address u_i has, as complied to the balance privacy defined in Equation (4.10). Similarly, the statement in Equation (4.13) implies that the second goal of the problem is to minimize the amount of information the address u_k accesses to over the transaction tx_{ij} between the addresses u_i and u_j, as complied to the transaction privacy defined in Equation (4.11). In the most ideal scenario, we expect the mutual information for both objectives to be zero where this is interpreted that no information about θ_i and tx_{ij} leaks. In addition, note how the information asymmetries in the beginning is transformed into the problem objectives later.

Privacy-preservation is the most significant requirement that our problem requires to be satisfied. However, there exist several other requirements as well as follows:

- Privacy: it must protect the privacy of balances and transaction details, (i.e. users must know only their own balance).
- Confidentiality: it must ensure the secrecy of message encryption (including transaction details) between sender and receiver.
- Trustless: users must complete their payments without relying on any trusted third party.
- Public Verifiability: it must allow validators on blockchain to check the correctness of computations sender or receiver perform off-chain over private data.
- Non-Interaction: it must allow the zero-knowledge proofs to be generated without any interaction.
- Correctness: it must function correctly and properly at any given time.
- Usability: it must properly abstract the complex cryptographic operations.

Overall, our driving motivation for this problem can be stated as *how to develop a privacy-preserving but publicly-verifiable payment protocol over a decentralized network (i.e. blockchain) through a certain cryptographic primitives (i.e. commitments, public-key encryptions, zero-knowledge proof) without any assistance of trusted third parties.*

4.2. System Architecture

For the given problem, we develop a privacy-preserving token transfer protocol (i.e. *PTTS*) on blockchain with zero-knowledge proof. Our protocol for the given problem is (i) *privacy-preserving* since it hides the transaction-involving balances and amounts throughout the transfer session, (ii) *publicly-verifiable* since the validators can still verify the correctness of the transactions with proof verification, (iii) *trustless* since users can complete their payments without any trusted third party and (iv) partially *non-interactive* since it does not require rounds of interactions to verify proofs (except the interaction for initial mutual *consent*). The protocol is built upon three main actors as (i) *developer* who performs one-time setup and deploys the smart contracts on blockchain, (ii) *sender* who deposits certain amounts of tokens to the contract pool and (iii) *receiver* who later withdraw certain amounts of tokens from the contract pool

again by preserving the privacy of balances and amounts.

Our *PTTS* protocol includes five main phases as: (i) *deploying contracts* where developer deploys the main transfer contract in addition to two proof-verifying contracts for depositing and withdrawing tokens, (ii) *requesting consent* where sender needs consent for interaction with receiver before depositing tokens, (iii) *granting consent* where receiver grants that consent to approve the interaction, (iv) *depositing tokens* where sender deposits certain amount of tokens by generating corresponding proof with the intention to show the correctness of off-chain computation and (v) *withdrawing tokens* where receiver withdraws the tokens by generating another proof. The protocol employs three smart contracts as (i) main transfer contract to act as an escrow mechanism and orchestrating protocol phases and (ii) two separate proof-verifying contracts (since the off-chain computation of sender and receiver is different) to efficiently verify zero-knowledge proofs on-chain. The state transitions for these phases are shown in Figure 4.1 as well. In addition, the architecture of the *PTTS* protocol is depicted in Figure 4.2. Note that one-time mutual consent is sufficient to perform multiple payments for every pair of senders and receivers.

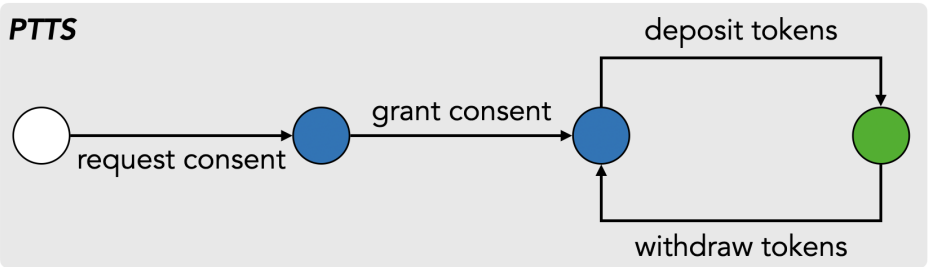


Figure 4.1. State-transition diagram of *PTTS* protocol.

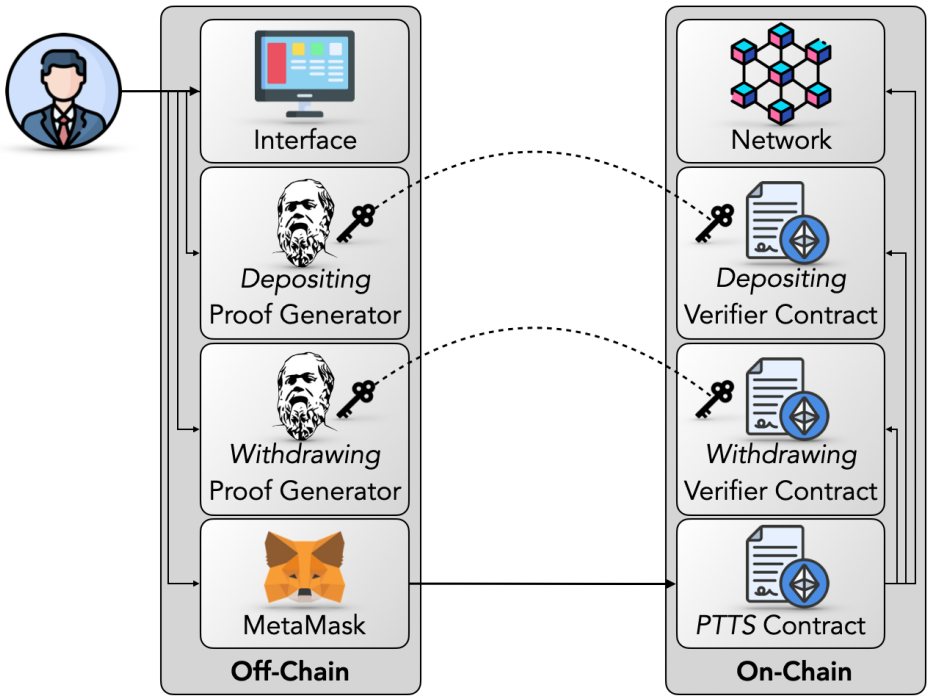


Figure 4.2. Architecture of *PTTS* protocol.

Deploying Contracts. Developer performs one-time setup in order to generate a proving key (for off-chain proof generation) and a verification key (for on-chain proof verification) with the following way:

$$(pok, vek) \leftarrow \texttt{Zk.Setup}(\lambda) \tag{4.14}$$

where *pok* denotes proving key while *vek* denotes verification key. Developer has to perform this phase separately for depositing and withdrawing. The proof-verifying contracts store the verification keys to verify proofs on-chain. Note that the corresponding proving and verification keys must be matched for proofs to be correctly verified.

Requesting and Granting Consent. In these phases, we assume that the main contract already stores the balance commitments of both sender and receiver with their corresponding salting parameters:

$$c_s^{\theta} \leftarrow \texttt{Cm.Comm}(\theta_s) \tag{4.15}$$

$$c_r^{\theta} \leftarrow \texttt{Cm.Comm}(\theta_r) \tag{4.16}$$

where θ_s and θ_r are private sender and receiver balances, respectively while c_s^{θ} and c_r^{θ} are their public commitments on the contract. For the sake of simplicity, we also assume that sender and receiver request and grant the consent before the next phases.

Depositing Tokens. Sender first selects the amount of tokens to be transferred to receiver and takes its commitment later:

$$c^{\Delta} \leftarrow \texttt{Cm.Comm}(\Delta) \tag{4.17}$$

where Δ is the amount while c^{Δ} is the commitment of that amount on the contract. We expect this amount to be less than the current sender balance as $\Delta \leq \theta_s$. Sender encrypts transaction details (including the amount and its corresponding salting parameter) through the public key of receiver who later decrypts it through their own private key with the following way:

$$E \leftarrow \texttt{Pk.Enc}([\Delta], pk_r) \tag{4.18}$$

where E denotes the resulting encryption of the public-key encryption function `Pk.Enc` while pk_r is the receiver public key. Sender has to generate a zero-knowledge proof per deposition in order to show that the off-chain computation for depositing tokens is correct by complying to all the rules with the following way:

$$\Psi^{deposit} \leftarrow \theta_s - \Delta \tag{4.19}$$

$$\pi_s \leftarrow \texttt{Zk.Gen}(\Psi^{deposit}, \theta_s, \Delta, c_s^{\theta}, c_s^{\theta}, pok) \tag{4.20}$$

$$b_s \leftarrow \texttt{Zk.Vfy}(\pi_s, c_s^{\theta}, c^{\Delta}, vek) \tag{4.21}$$

$$c_s^{\theta} \leftarrow \texttt{Cm.Comm}(\theta_s \leftarrow \theta_s - \Delta) \tag{4.22}$$

where $\Psi^{deposit}$ is the computation to which the sender is subjected (i.e. subtracting the

amounts of tokens from their current balance). While π_s is the resulting depositing proof of the proof generation function `Zk.Gen` based on the function $\Psi^{deposit}$, the current balance θ_s, the current amount Δ and their commitments c_s^θ and c^Δ, b_s is boolean result of the proof verification function `Zk.Vfy` based on the proof π_s and public commitments. If the depositing proof is verified on-chain ($b_s \leftarrow true$), the contract replaces the current sender balance with their next balance c_s^θ and shares the encryption E with the receiver.

Withdrawing Tokens. Receiver takes and decrypts the encryption E:

$$D \leftarrow \texttt{Pk.Dec}([E], sk_r) \tag{4.23}$$

where D is the resulting decryption (or transaction details) of the public-key decryption function `Pk.Dec` with the receiver private key sk_r. Receiver has to generate a zero-knowledge proof in order to show that the off-chain computation for withdrawing tokens is correct by complying to all the rules (similar to sender):

$$\Psi^{withdraw} \leftarrow \theta_r + \Delta \tag{4.24}$$

$$\pi_r \leftarrow \texttt{Zk.Gen}(\Psi^{withdraw}, \theta_r, \Delta, c_r^\theta, c^\Delta, pok) \tag{4.25}$$

$$b_r \leftarrow \texttt{Zk.Vfy}(\pi_r, c_r^\theta, c_\Delta, vek) \tag{4.26}$$

$$c_r^\theta \leftarrow \texttt{Cm.Comm}(\theta_r \leftarrow \theta_r + \Delta) \tag{4.27}$$

where $\Psi^{withdraw}$ is the computation to which the receiver is subjected (i.e. adding the amounts of tokens to their current balance). While π_r is the resulting withdrawing proof of the proof generation function `Zk.Gen` based on the function $\Psi^{withdraw}$, the current balance θ_r, the current amount Δ and their commitments c_r^θ and c^Δ, b_r is boolean result of the proof verification function `Zk.Vfy` based on the proof π_r and public commitments. If the withdrawing proof is correctly verified on-chain ($b_r \leftarrow true$), the contract replaces the current receiver balance with their next balance c_r^θ by finishing the protocol. Note that both proof generation functions use both public and private information the the proof verification functions use only the public information over the contract. The summary of the *PTTS* protocol is given in Table 4.1. The sequence diagram of the

PTTS protocol is depicted in Figure 4.3 as well by clearly demonstrating interactions among the protocol actors.

Table 4.1. The *PTTS* protocol

Protocol I: *PTTS*
1. zkSNARKs Setup
(a) Developer performs one-time setup by generating proving and verification keys as
$\texttt{(pok, vek)} \leftarrow \texttt{Zk.Setup}(\lambda)$.
(b) All user balances are immutably stored on the contract as
$c_i^{\theta} \leftarrow \texttt{Cm.Comm}(\theta_i)$.
2. Depositing of u_s
(a) User u_s deposits certain tokens Δ to the contract by generating *depositing* proof as
$c_s^{\theta} \leftarrow \texttt{Cm.Comm}(\theta_s)$
$c^{\Delta} \leftarrow \texttt{Cm.Comm}(\Delta)$
$\pi_s \leftarrow \texttt{Zk.Gen}(\Psi^{deposit}, \theta_s, \Delta, c_s^{\theta}, c^{\Delta}, pok)$.
(b) The correct verification of proof π_s in the contract as
$b_s \leftarrow \texttt{Zk.Vfy}(\pi_s, c_s^{\theta}, c^{\Delta}, vek)$ updates user balance as
$c_s^{\theta} \leftarrow \texttt{Cm.Comm}(\theta_s \leftarrow \theta_s - \Delta)$.
(c) User u_s submits encryption of that deposition to user u_r with their public key pk_r as
$E \leftarrow \texttt{Pk.Enc}([\Delta], pk^r)$.
3. Withdrawing of u_r
(a) User u_r decrypts that encryption with their private key sk_r as
$D \leftarrow \texttt{Dec}([E], sk_r)$.
(b) User u_r withdraws tokens Δ from the contract by generating *withdrawing* proof as
$\pi_r \leftarrow \texttt{Zk.Gen}(\Psi^{withdraw}, \theta_r, \Delta, c_r^{\theta}, c^{\Delta}, pok)$.
(c) The correct verification of proof π_r in the contract as
$b_r \leftarrow \texttt{Zk.Vfy}(\pi_r, c_r^{\theta}, c^{\Delta}, vek)$ updates user balance as
$c_r^{\theta} \leftarrow \texttt{Cm.Comm}(\theta_r \leftarrow \theta_r + \Delta)$.

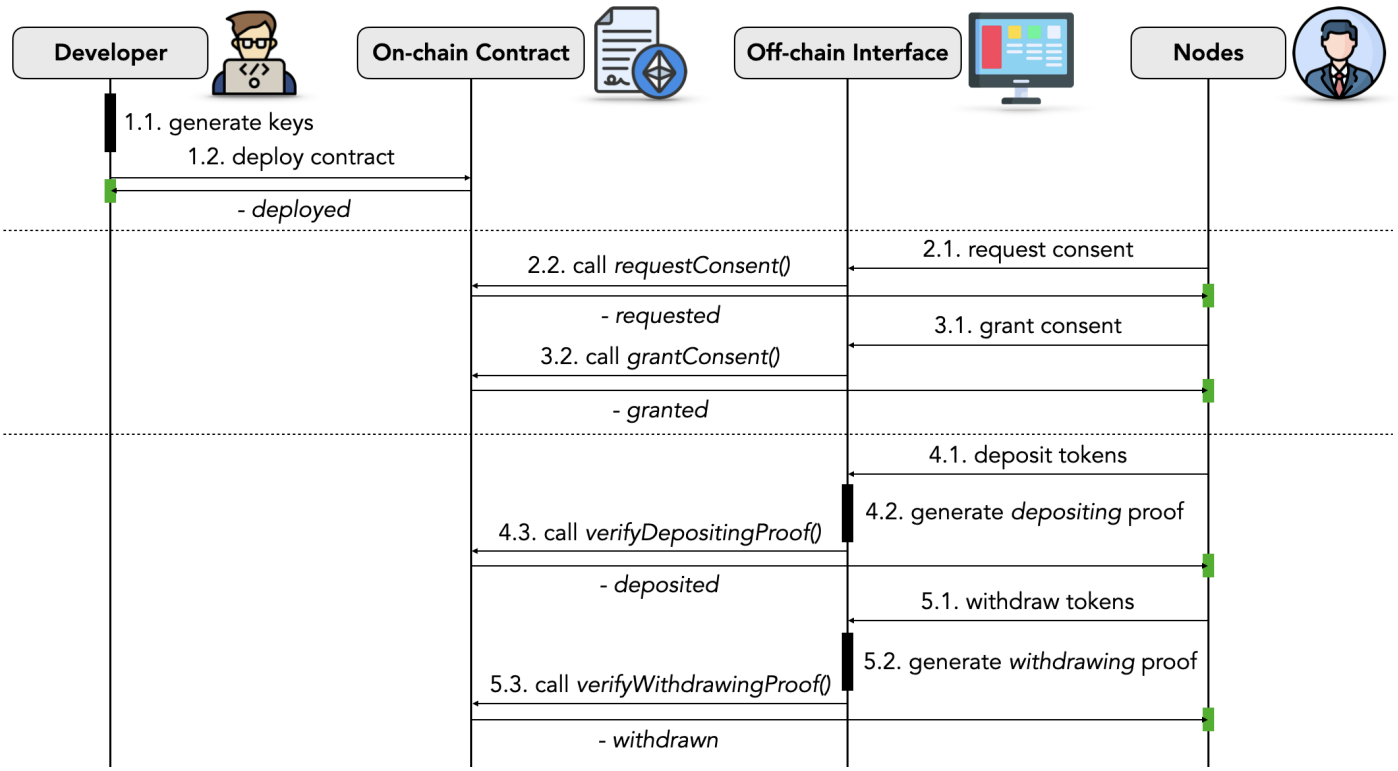


Figure 4.3. Sequence diagram of *PTTS* protocol.

4.2.1. Zero-Knowledge Proof Model

The goal of the proof model in our privacy-preserving payment protocol is to replace certain data with their corresponding commitments by still providing a publicly-verifiable computation to be able to transfer tokens from one address to another. Our proof model involves two main algorithms as *depositing tokens* for sender (see Figure 4.4) and *withdrawing tokens* for receiver (see Figure 4.5). In Figure 4.4, there exist private function inputs including sender balance, amount to deposit and their salting parameters; and public inputs including their commitment values. In Line 2-4, the commitments for the current balance, the amount and the next balance are internally computed. In Line 4, the amount to deposit is deducted from the current sender balance to get the next sender balance. In Line 5, the proof checks two criteria: as (i) the balance is sufficient to deposit and (ii) the internal commitments are equal to the input commitments. The failure in any of these conditions leads the resulting proof not to be verified on-chain (and eventually to abort the corresponding transaction).

In Figure 4.5 for *withdrawing tokens*, in Line 4, the amount once deducted from the sender balance is withdrawn to the receiver balance. In Line 5, the proof now checks only the correctness of the commitments.

There exist certain points to be noted in our zero-knowledge proof model, regarding the way it is integrated into our overall system. First, the off-chain proof generation always needs both the private and public inputs from the delegation node while the on-chain proof verification needs only the public inputs in the smart contract. This implicitly prevents a malicious actor from freely modifying private values (e.g. balance) since their commitments are internally computed and compared with the commitments from the smart contract. Second, the *binding* and *hiding* properties of the commitment scheme in use set up a common security ground both for sender and receiver parties. Third, the large salting parameters prevent an enumeration (look-up table) attack to be performed over the commitments in the smart contract, which will be thoroughly discussed later. Fourth, impacts of parties on their balances are intentionally decoupled where the successful proof verification of Figure 4.4 and Figure 4.5 update only the sender balance and only the receiver balance, respectively.

```
1: def main(private balance, private amount, private balanceSalt, private amountSalt,
   private nextBalanceSalt, public _balanceComm, public _amountComm, public
   _nextBalanceComm):
2:   balanceComm ← sha256(balanceSalt, balance)
3:   amountComm ← sha256(amountSalt, amount)
4:   nextBalanceComm ← sha256(nextBalanceSalt, balance - amount)
5:   result = if(amount ≤ balance &&
                 balanceComm == _balanceComm &&
                 amountComm == _amountComm &&
                 nextBalanceComm == _nextBalanceComm) then true else false fi
6:   return result
```

Figure 4.4. Proof implementation in ZoKrates [27] for depositing tokens.

```
1: def main(private balance, private amount, private balanceSalt, private amountSalt,
   private nextBalanceSalt, public _balanceComm, public _amountComm, public
   _nextBalanceComm):
2:   balanceComm ← sha256(balanceSalt, balance)
3:   amountComm ← sha256(amountSalt, amount)
4:   nextBalanceComm ← sha256(nextBalanceSalt, balance + amount)
5:   result = if(balanceComm == _balanceComm &&
                amountComm == _amountComm &&
                nextBalanceComm == _nextBalanceComm) then true else false fi
6:   return result
```

Figure 4.5. Proof implementation in ZoKrates [27] for withdrawing tokens.

4.2.2. Smart Contract Model

The goal of our smart contract model is to (i) manage the interactions between sender and receiver, (ii) store immutable and publicly-verifiable commitments; and (iii) verify the depositing and withdrawing proofs (see Figure 4.4 and 2) on-chain to provide a seamless privacy-preserving payment. The contract model involves with the functions for the *requesting consent* (see Figure 4.6), *granting consent* (see Figure 4.7), *depositing tokens* (see Figure 4.8) and *withdrawing tokens* (see Figure 4.9) phases of our protocol. In Figure 4.6, sender calls the *requestConsent* function by providing the receiver address to request a consent to deposit tokens. In Line 2, the corresponding entry in the mapping (i.e. dictionary) is set to true. In Figure 4.7, the receiver calls the *grantConsent* function by providing the sender address to grant a consent to that request. In Line 2, it checks whether there really exists such a request. In Line 4, it grants the request by setting the corresponding entry in the mapping to true. Although these two phases have a negative impact on the protocol non-interactivity to a certain extent, it is useful (i) to indicate the intention (non-repudiation) of the parties on privacy-preserving transfer with the other on legal accusations and (ii) to retrieve the receiver public key for the next phases.

```
1: def requestConsent(address _to) public:
2:     request[msg.sender][_to] ← true
```

Figure 4.6. Contract implementation for *requesting consent* phase.

```
1: def grantConsent(address _from, string _publicKey) public:
2:     require(request[_from][msg.sender] == true)
3:     publicKeys[msg.sender] ← _publicKey
4:     grant[_from][msg.sender] ← true
```

Figure 4.7. Contract implementation for *granting consent* phase.

In Figure 4.8 for *depositing tokens*, sender calls *depositTokens* function with the corresponding proof to deposit a certain amount of tokens to the contract pool. In Line 2, it first checks whether the receiver really grants consent. In Line 3, it verifies the proof by making an external call to *verifyTx* function in the *verifier* smart contract (review Section 2.9). If the proof is correct in Line 4, it updates (i) the sender balance with the next balance commitment (as a transition from one balance state to another) in Line 5, (ii) the allowance to track the amount to deposit and withdraw in Line 6 and (iii) the encryption including the transaction details in Line 7. In case the proof is incorrect, it commits the transaction with no update at all. There exist certain points to be noted here as well. First, the commitments for the amount and the next balance are function inputs while the commitment for current balance is not (i.e. rather automatically fetched from the contract), which prevents malicious actions on the state transitions of the balance, (i.e. otherwise sender could set any balance). Second, the off-chain encryption of transaction details through the receiver public key sets up a private communication channel between sender and receiver where only the receiver can decrypt it with their own private key to learn these details later for *withdrawing* proof generation. The harm resulting from the encryption of incorrect transaction details (which would prevent the receiver from not withdrawing the tokens) falls on the sender himself rather than the receiver, (akin to burning own tokens). This naturally incentivizes senders to adhere the protocol.

```
1: def depositTokens(address _to, uint256 amountComm, uint256 nextBalanceComm,
   string encryption, Proof proof) public:
2:     require(grant[msg.sender][_to] == true)
3:     isProofCorrect ← verifier.verifyTx(proof,
                                          balanceComm[msg.sender],
                                          amountComm,
                                          nextBalanceComm)
4:     if(isProofCorrect):
5:         balanceComm[msg.sender] ← nextBalanceComm
6:         allowance[msg.sender][_to] ← amountComm
7:         encryptions[msg.sender][_to] ← encryption
```

Figure 4.8. Contract implementation for *depositing tokens* phase.

In Figure 4.9 for *withdrawing tokens*, the receiver calls *withdrawTokens* function with the corresponding proof to withdraw tokens from the contract pool. In Line 2-3, it checks whether consent is granted and the amount of tokens to be withdrawn is correct. In Line 4, it verifies the proof with an external call to the verifier contract. If the proof is correct in Line 5, it updates the receiver balance with the next balance commitment in Line 6 and reset the allowance to prevent any replay attack to maliciously withdraw the tokens twice in Line 7. Similar to Figure 4.8, the commitment for the current balance is directly fetched from the smart contract (i.e. *balanceComm[msg.sender]*). Note that the proving key used off-chain for the proof generations and the verification key used on-chain for the proof verifications must be correctly matched, (which would lead the proofs not to be verified otherwise).

```
1: def withdrawTokens(address _from, uint256 amountComm, uint256 nextBal-
   anceComm, Proof proof) public:
2:     require(grant[_from][msg.sender] == true)
3:     require(allowance[_from][msg.sender] == amountComm)
4:     isProofCorrect ← verifier.verifyTx(proof,
                                          balanceComm[msg.sender],
                                          amountComm,
                                          nextBalanceComm)
5:     if(isProofCorrect):
6:         balanceComm[msg.sender] ← nextBalanceComm
7:         allowance[msg.sender][_to] ← 0
```

Figure 4.9. Contract implementation for *withdrawing tokens* phase.

4.2.3. Web Interface Model

The goal of our web interface model is (i) to abstract complex mathematical and cryptographic operations (for commitment, public-key encryption and zero-knowledge proof schemes) into simple actions, (ii) to provide more efficient, fast and user-friendly experience to protocol actors. Once built over client browser, it does not require any additional installation except the *MetaMask* extension [91] is already running so that the interface can connect to the smart contracts. This interface provides five main actions for (i) *deploying contract*, (ii) *requesting consent*, (iii) *granting consent*, (iv) *depositing tokens* and (v) *withdrawing tokens*. The illustrations of these actions are shown in Figure 4.10. In addition, the open-source implementation is publicly available as well in the following Github page for further inspection [51].

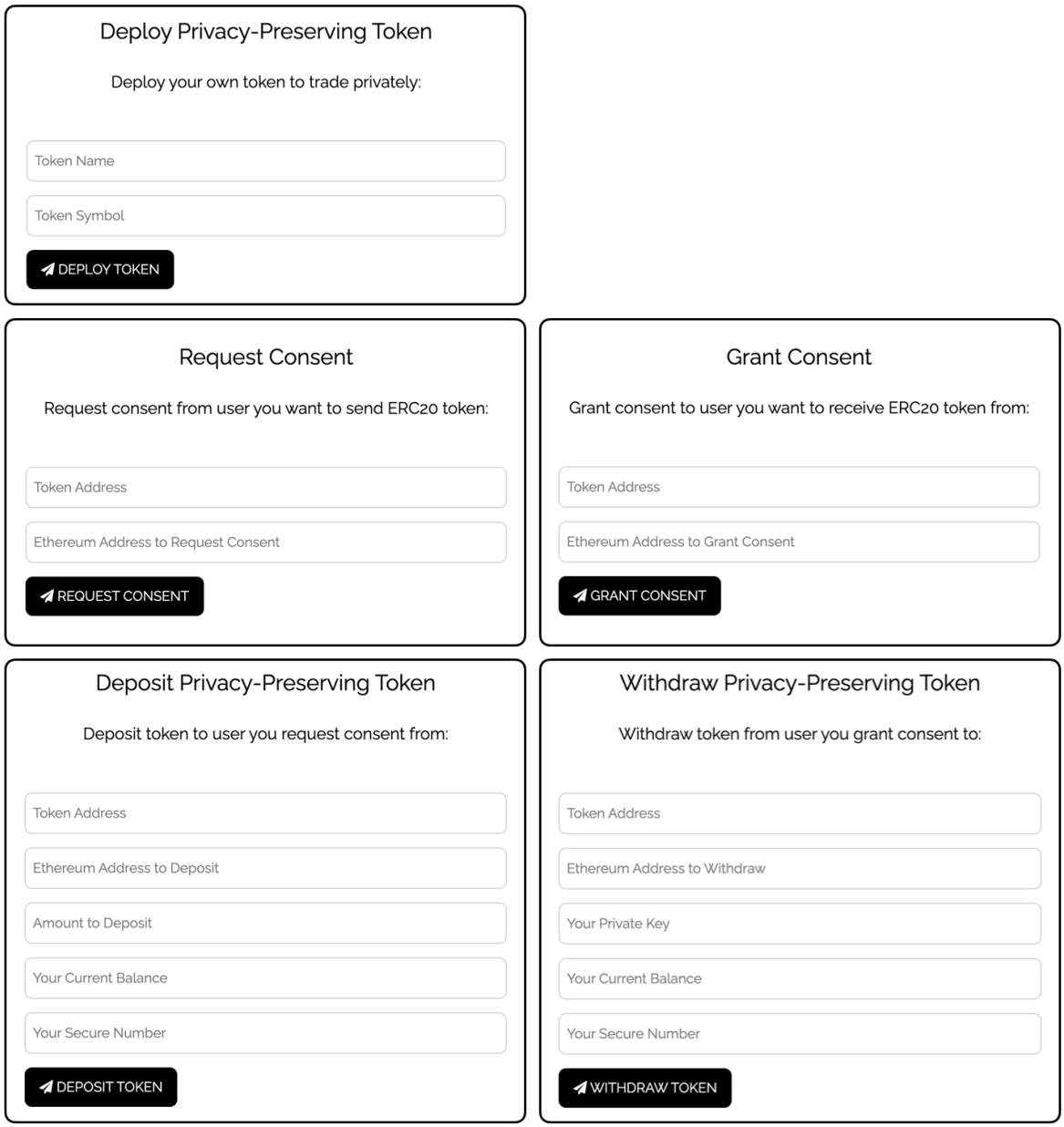


Figure 4.10. Web user interface of *PTTS* protocol.

The first action is *deploying token* where a contract owner specifies contract metadata (including token name and token symbol) to deploy a token contract instance to blockchain. The algorithm for this action is given Figure 4.11 where it gets metadata from the interface in Line 2 and deploys a contract instance in Line 3. The resulting contract address and contract creation transaction are returned to the owner. The second action is *requesting consent* where a sender specifies a receiver address over a token contract address to request consent. The algorithm for this action is given in

Figure 4.12 where it first gets necessary inputs in Line 2, creates a contract instance based on the contract address in Line 3 and calls the *requestConsent* function (see Figure 4.6) in the contract in Line 4. The third action is *granting consent* where the receiver specifies a sender address over a token contract address to grant consent. The algorithm for this action is also given in Figure 4.13 where it automatically restores the public key in Line 4 and calls the *grantConsent* function (see Figure 4.7) in the contract in Line 5.

```
1: async def deploy():
2:     metadata ← interface.getContractMetadata()
3:     [transaction, address] ← web3.deployContract(metadata)
4:     return [transaction, address]
```

Figure 4.11. Interface implementation for *deploying contract* phase.

```
1: async def requestConsent():
2:     [contractAddress, receiverAddress] ← interface.getInputs()
3:     contract ← new web3.Contract(contractAddress)
4:     transaction ← contract.requestConsent(receiverAddress) - (Figure 4.6)
5:     return transaction
```

Figure 4.12. Interface implementation for *requesting consent* phase.

```
1: async def grantConsent():
2:     [contractAddress, senderAddress] ← interface.getInputs()
3:     contract ← new web3.Contract(contractAddress)
4:     publicKey ← interface.restorePublicKey()
5:     transaction ← contract.grantConsent(senderAddress, publicKey) - Figure 4.7
6:     return transaction
```

Figure 4.13. Interface implementation for *granting consent* phase.

The fourth action is *depositing tokens* where the sender deposits certain amounts of tokens to the receiver. The algorithm for this action is given Figure 4.14 where it gets inputs in Line 2, computes commitments for current balance, next balance and amount in Line 4, encrypts transaction details in Line 5-6 and generates a zero-knowledge proof based on these public and private inputs in Line 7. It calls the *depositTokens* function (see Figure 4.8) in the contract in Line 8. Note that all the private inputs (e.g. current balance) are processed inside the interface without any network communication. Furthermore, since only the sender knows their private data, it always has to be explicitly presented to the interface. Finally, the fifth action is *withdrawing tokens* where the receiver withdraws tokens from the sender. The algorithm for this action is given in Figure 4.15 where it gets inputs in Line 2, gets and decrypts encryption in Line 4-5, computes commitments for current balance, next balance and amount in Line 6 and generates a zero-knowledge proof based on these public and private inputs in Line 7. It calls the *withdrawTokens* function (see Figure 4.9) in the contract in Line 8.

```
1: async def depositTokens():
2:     [contractAddress, receiverAddress, balance, amount] ← interface.getInputs()
3:     contract ← new web3.Contract(contractAddress)
4:     [balanceComm, nextBalanceComm, amountComm] ← interface.commit(balance,
   amount)
5:     publicKey ← contract.getPublicKey()
6:     encryption ← interface.encrypt(amount)
7:     proof ← zokrates.generate(balance, amount, balanceComm, nextBalanceComm,
   amountComm) - Figure 4.4
8:     transaction ← contract.depositTokens(receiverAddress, amountComm, nextBal-
   anceComm, encryption, proof) - Figure 4.8
9:     return transaction
```

Figure 4.14. Interface implementation for *depositing tokens* phase.

```
1: async def withdrawTokens():
2:     [contractAddress, senderAddress, balance, privateKey] ← interface.getInputs()
3:     contract ← new web3.Contract(contractAddress)
4:     encryption ← contract.getEncryption()
5:     amount ← interface.decrypt(encryption)
6:     [balanceComm, nextBalanceComm, amountComm] ← interface.commit(balance,
   amount)
7:     proof ← zokrates.generate(balance, amount, balanceComm, nextBalanceComm,
   amountComm) - Figure 4.5
8:     transaction ← contract.withdrawTokens(senderAddress, amountComm, nextBal-
   anceComm, proof) - Figure 4.9
9:     return transaction
```

Figure 4.15. Interface implementation for *withdrawing tokens* phase.

4.3. Protocol Analysis

4.3.1. Scalability

In this section, we theoretically analyze the scalability of the *PTTS* protocol with respect to the computational, communication and storage overheads. Each of these overheads gives certain insights about the efficiency of the protocol and clarifies the points to be further improved. For instance, the *computational overhead* returns how complex and resource-consuming the cryptographic and blockchain operations are, which leads us to select more optimal machines over which the off-chain operations are carried out. The *communication overhead* returns the intensity of the interactions among the users that the protocol requires to move forward where each interaction may accumulate additional network load to the system. Finally, the *storage overhead* returns how much memory the protocol needs both off-chain and on-chain where on-chain memory is certainly more expensive because of blockchain gas consumption.

4.3.1.1. Computational Overhead. It is the zero-knowledge proof generation function that drives the computational overhead the most. For each privacy-preserving payment, both sender and receiver need to generate proofs (i.e. one *depositing proof* of sender and one *withdrawing proof* of receiver). By comparison, generating *depositing proof* proof is slightly more complex than generating *withdrawing proof* proof since the latter proof does not check that the balance is sufficient to deposit, (see Figure 4.4 and Figure 4.5). By neglecting this difference for the sake of simplicity, we represent this overhead with respect to the number of proofs generated with the following way:

$$\xi_{comp} = 1 \tag{4.28}$$

$$\xi_{comp}^{\Sigma} = N \cdot \xi_{comp} \tag{4.29}$$

where ξ_{comp} and ξ_{comp}^{Σ} are the computational overheads on a party and on a system, respectively while N is the number of users. Note that since our protocol strictly requires two parties, N must be necessarily 2 here. Therefore, evaluation of scalability with the increasing number of users is not applicable to our protocol while a future extension into the privacy-preserving multi-party payment may potentially use it as a metric. On the other hand, the zero-knowledge proof verification function of our protocol is free from any computational overhead (except the blockchain gas consumption) since it runs on-chain over blockchain.

4.3.1.2. Communication Overhead. It is the interactions between sender and receiver that drives the communication overhead the most in our protocol. For each privacy-preserving payment, sender needs to request consent and later deposit tokens while submitting encryptions while receiver needs to grant consent and later withdraw tokens. We represent this overhead by counting the number of communications with the following way:

$$\xi_{comm} = 2 \tag{4.30}$$

$$\xi_{comm}^{\Sigma} = N \cdot \xi_{comm} \tag{4.31}$$

where ξ_{comm} and ξ_{comm}^{Σ} are the communication overheads on a party and on a system, respectively while N is 2. For the same reason in computational overhead, scalability analysis with the increasing number of users is not applicable here, too. Moreover, we may address the issue of these direct interactions to provide a more non-interactive system in the future.

4.3.1.3. Storage Overhead. It is the on-chain memory that drives the storage overhead the most in our protocol. For each user, the protocol needs to store mappings (i.e. dictionaries) for (i) balance commitment, (ii) amount commitments, (iii) consent requests, (iv) consent grants and (v) encryptions as well. We represents this overhead by counting the number of entries required per payment with the following way:

$$\xi_{mem} = 5 \tag{4.32}$$

$$\xi_{mem}^{\Sigma} = N \cdot \xi_{mem} \tag{4.33}$$

where ξ_{mem} and ξ_{mem}^{Σ} are the storage (i.e. memory) overheads on a party and on a system, respectively while N is 2. One more point, our protocol requires verification keys to be stored on-chain and proving keys for zero-knowledge proof, (review Section 2.8 and Section 2.9). With respect to the proof circuit complexity, verification keys stay constant (in terms of memory) while the proving keys enlarge. Fortunately, the current machines are sufficiently powerful to store proving keys without any trouble.

Table 4.2. *PTTS* communication, computation and storage overheads between 2 users (i.e. sender and receiver).

Nodes	**Communication Overhead**		**Computation Overhead**		**Storage Overhead**	
	On Node	**On System**	**On Node**	**On System**	**On Node**	**On System**
2	1	2	2	4	5	10

4.3.2. Security

In this section, we evaluate the resistance of our protocol with respect to several attacks and later analyze its security by using formal reduction proof. We consider and define the most relevant attacks in the scope of our thesis including replay attack, preimage attack, Sybil attack and collusion attack. Those definitions will be valid and used throughout the thesis. Inclusion of other attacks is also among our future works.

Replay Attack. The replay attack refers to duplication of a valid transaction in the blockchain and maliciously replaying it again. Our privacy-preserving payment system is robust against the replay attack. When the stages are individually analyzed, it is easy to see that the *requesting consent* and the *granting consent* stages of the framework are handshake stages between sender and receiver where certain values in the mappings are set to true in the contract. Even if the transactions corresponding to these stages are replicated again, the same values in the mappings are set to true. However, the transactions verifying proofs (for the *private depositing* and the *private withdrawing* stages) need the following elaboration. In our framework, the proofs are generated by considering a specific set of commitments where they cannot be verified on another set. In these stages, once the proofs are verified, it changes the current set of commitments with the next set of commitments. Therefore, they cannot be verified again with these next set.

Preimage Attack. We define preimage attack as the utilization of an enumeration (i.e. look-up) table to compare and find a corresponding commitment for user balances. This attack is especially strong in narrow preimage spaces where storing all commitments corresponding to the values in this space is not expensive. To eliminate this attack, we introduce large and random salting parameters (i.e. σ) when hashing user balances (see Equation (2.24)) and regularly replace these parameters with new parameters at every transfer. This results in different commitment values even if the balances of two distinct users are the same. For an adversarial party, storing commitments for all salting parameters has the complexity of brute-forcing the commitments.

Collusion Attack. We define the collusion attack as the partnership of more than one adversarial party through a specific way (i.e. exchanging information they have) to have more information that they cannot individually learn or to have more control over protocol. There can be two types of collusion attacks as collusion with trusted setup and collusion among peers, where the former attack is not applicable in our scope since the *PTTS* protocol (and all the other protocols in this thesis) does not rely on any trusted setup. *PTTS* is also resistant to the latter attack since each party is responsible for updating only their own balance. On the other hand, there exist certain points to be noted. First, in the scenario where the receiver deliberately reveals transactional details, the other parties learn the amounts sender deposits. However, this is also disadvantageous for the receiver himself. Second, an adversarial organization (as in our balance range disclosure attack) may learn additional information if multiple parties reveal their own privacy-preserving transactions.

4.3.2.1. Reduction Proof. In this section, we theoretically show the security of the *PTTS* protocol through the formal reduction proofs to the underlying zero-knowledge proof protocol (i.e. zkSNARKs [16]) with the following way:

Theorem. The privacy-preserving payment protocol *PTTS* is secure if and only if the zkSNARKs protocol [16] is secure.

Proof. Suppose that $\mathcal{A}$ and $\mathcal{B}$ are the adversaries that can break the *PTTS* and zkSNARKs protocols, respectively. For the following scenario between the adversary $\mathcal{A}$ and the challenger (i.e. verifier) $\mathcal{C}$:

- The challenger $\mathcal{C}$ runs the zkSNARKs trusted setup to generate the proving key pok and the verification key vek.
- The adversary $\mathcal{A}$ samples a random balance $\theta_{\mathcal{A}}$ and an amount Δ and forwards their commitments $c^{\theta}_{\mathcal{A}} \leftarrow \texttt{Cm.Comm}(\theta_{\mathcal{A}})$ and $c_{\Delta} \leftarrow \texttt{Cm.Comm}(\Delta)$ to the challenger.
- The adversary $\mathcal{A}$ samples a random falsified balance $\theta'_{\mathcal{A}}$ where $\theta'_{\mathcal{A}} > \theta_{\mathcal{A}}$ by invoking the adversary $\mathcal{B}$ to generate a false proof $\pi \leftarrow \texttt{Zk.Gen}(\Psi^{deposit}, \theta'_{\mathcal{A}}, \Delta, c^{\theta}_{\mathcal{A}}, c^{\Delta}, pok)$

with the falsified balance and forwards the proof π to the challenger.

- The challenger $\mathcal{C}$ returns true in case this false proof is correctly verified as $b \leftarrow \texttt{Zk.Vfy}(\pi, c^{\theta}_{\mathcal{A}}, c^{\Delta}, vek)$ and false otherwise.

With respect to this scenario, the challenger always returns true if and only if there really exists the adversary $\mathcal{B}$ that has ability to break zkSNARKs by forging the proof with respect to the falsified balance x'. However, this contradicts our assumption that the zero-knowledge proof scheme itself is secure. More formally:

$$\left| \Pr \left[1 \leftarrow \texttt{C}(\texttt{Zk.Vfy}(\pi, c^{\theta}_{\mathcal{A}}, c^{\Delta}, vek)) \middle| \begin{array}{l} (pok, vek) \leftarrow \texttt{Zk.Setup}(\lambda) \\ \theta_{\mathcal{A}}, \theta'_{\mathcal{A}}, \Delta \xleftarrow{\$} \mathbb{Z}_0^+, \quad \theta \\ \Psi^{deposit} \leftarrow \theta'_{\mathcal{A}} - \Delta \\ c^{\theta}_{\mathcal{A}} \leftarrow \texttt{Cm.Comm}(\theta_{\mathcal{A}}) \\ c^{\theta'}_{\mathcal{A}} \leftarrow \texttt{Cm.Comm}(\theta'_{\mathcal{A}}) \\ c^{\Delta} \leftarrow \texttt{Cm.Comm}(\Delta) \\ \pi \leftarrow \texttt{A}(\texttt{B}(\texttt{Zk.Gen}(\Psi^{deposit}, \theta'_{\mathcal{A}}, \Delta, c^{\theta}_{\mathcal{A}}, c^{\Delta}, pok))) \end{array} \right] \right| = 0 \tag{4.34}$$

where this equation implies that the probability of correct verification of the false proof must be zero from the perspective of the challenger (i.e. verifier).

4.3.3. Limitations

The *PTTS* protocol has certain limitations that we must address in the future as: (i) partial interaction, (ii) address linkability, (iii) multi-token payment, (iv) platform dependencies, (v) lack of forward secrecy and (iv) performance issues. We explain these limitations with the following way:

- Partial Interaction: The protocol allows proof generation without any interaction among users. However, it still requires a certain degree of interaction to complete

payments (e.g. requesting and granting consent). Although this enforces an additional layer of non-repudiation over the users in parallel to our intention, it unfortunately slows down the payment procedure itself.

- Address Linkability: The protocol successfully guarantees privacy only for transaction amounts while the source and destination addresses are publicly visible where an adversarial party may cleverly design an attack by utilizing patterns, frequency, timings and address links from these transactions. In the balance range disclosure attack that we design (see Section 4.4), we also use this information to build a global transaction graph.
- Multi-Token Payment: The protocol currently supports single token payment between addresses where it leads to inefficiencies in case multiple tokens are to be transferred. We address this open issue in the scope of our privacy-preserving multi-token bartering protocol (i.e. *PMTBS*).
- Platform Dependencies: We assume that the protocol consisting of several smart contracts must run over EVM-supported blockchains (e.g. Ethereum [2], Avalanche [3]). Similarly, we rely on zkSNARKs protocol [16] to generate and verify zero-knowledge proofs. On the other hand, these components of our protocol can be replaced with respect to the requirements of the application to be built.
- Lack of Forward Secrecy: We assume that the users have single pairs of public and private keys. In case the private keys of certain users are compromised, an adversarial party may decrypt the details of all the previous transactions (i.e. future compromises cannot protect the past transactions).
- Performance Issues: The protocol has certain computational, communication and storage overheads, (see Section 4.3.1). Improvement over these overheads may eventually result in further efficiency over blockchain gas consumption (i.e. cost of payment transaction sender and receiver have to cover) and zero-knowledge proof generation times. Management of user public keys as the storage overhead can be referred to under this item as well.

4.4. Balance Range Disclosure Attack

In this section, we define a novel attack (i.e. balance range disclosure attack), targeting the *privacy-preserving payment* systems in general where an adversarial organization (e.g. exchange service, government) collects the private payment transactions from the users (i.e. senders and receivers) that they deliberately leak (for specific incentives) in order to calculate the private balances of the users. The strength of this attack comes from the fact that the balance of a user can be still estimated even though that user itself may not leak their own transactions. We define the balance range disclosure attack with the following formal way:

$$\langle Tx, Tx^*, \Theta, \mathcal{M} \rangle \tag{4.35}$$

where $tx_{ij} \in Tx$ is the set of all transactions, $tx^*_{ij} \in Tx^*$ is the set of all transactions that the organization knows (i.e. public transactions now), $\theta_i \in \Theta$ is the set of all user balances and $\mathcal{M}$ is the attacking algorithm that the organization uses to attempt the attack. The goal of the organization is to calculate the bounds for a balance to be attacked as:

$$\theta_i \subseteq [\theta_i^{\min}, \theta_i^{\max}] \tag{4.36}$$

where $\theta_i^{\min}$ is the minimum potential bound while $\theta_i^{\max}$ is the maximum potential bound.

We intentionally use the keyword *calculating* here instead of *estimating* (i.e. *predicting*) since there is always a space for errors during the estimation while the calculation must result in the exact solution in our scope. This can be interpreted that the bounds the attack eventually finds must be always correct. Finally, we can define the attacking algorithm $\mathcal{M}$ on a specific user u_i with the following way:

$$\mathcal{M} : (Tx, Tx^*, \Theta, u_i) \rightarrow [\theta_i^{\min}, \theta_i^{\max}] \tag{4.37}$$

where it selects a user u_i over all the public transactions Tx^*, private transactions Tx and private balances Θ to return the minimum $\theta_i^{\min}$ and the maximum bounds $\theta_i^{\max}$. This statement reveals important properties for our attack: (i) it always needs to target a single user where it must be repeated again for another user and (ii) it employs the private transactions as well since the links between sender and receiver addresses can still give useful information about the balances of the user to be attacked.

4.4.1. Minimum Cost Flow Network

The balance range privacy attack problem can be solved by two separate executions of minimum cost flow network algorithms with certain cost assignments on the edges. Let $\mathcal{G} = (\mathcal{U}, Tx)$ be a directed graph where $\mathcal{U}$ is the set of nodes (i.e. users) and Tx is the set of edges (i.e. transactions). Tx^* is the subset of transactions that are leaked to a malicious organization in the blockchain. Each node i is associated with a supply value b_i where the source node supplies all the tokens and the sink node demands all these tokens while the rest of the nodes have b_i set to 0. Each edge between the nodes i and j is also associated with lower/upper bounds $[l_{ij}, u_{ij}]$ on the feasible flow that can pass through the edge and a cost value c_{ij} to be used in the objective function. For each edge corresponding to a leaked transaction, the lower and upper bounds are the same and equal to the transaction amount v_{ij}. For the remaining non-leaked transactions, we take l_{ij} as 0 and u_{ij} as the total token supply. Finally, the flow on the edge x_{ij} is the decision variable of the problem to be optimized.

In the first minimum cost flow network, the costs of all the edges (except the edge between the user to be attacked and the sink node) are set to zero. However, the edge between that user and the sink node is set to -1 which means that the network solver will try to flow as many tokens as possible on this edge in order to minimize the overall cost of the network. On the other hand, in the second minimum cost flow network, the edge between the user to be attacked and the sink node is set to $+1$ while all the other edges are still zero. In this case, the network flow solver will try to flow as few tokens as possible through this edge to avoid increasing the overall cost of the network. In

the formulation of these two network flow problems, the two objectives are represented with the following way:

$$min. \sum_{(i,j)\in E} -c_{ij}x_{ij} \tag{4.38}$$

$$min. \sum_{(i,j)\in E} +c_{ij}x_{ij} \tag{4.39}$$

where x_{ij} is the corresponding decision variable on the edge between the user u_i and u_j. Note that we aim to minimize both networks: (i) minimize by increasing the flow when only negative cost is available in Equation (4.38) and (ii) minimize by decreasing the flow while only positive cost is available in Equation (4.39). Both objectives are subjected to the following same constraints:

$$s.t. \quad \sum_{(i,j)\in E} x_{ij} - \sum_{(j,i)\in E} x_{ij} = b_i, \forall i \in V \tag{4.40}$$

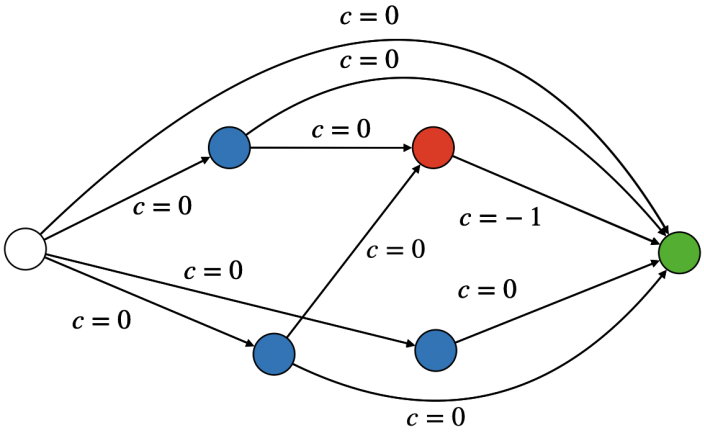
$$l_{ij} \leq x_{ij} \leq u_{ij}, \forall (i,j) \in E \tag{4.41}$$

$$l_{ij} = u_{ij} = v_{ij}, \forall (i,j) \in E^* \tag{4.42}$$

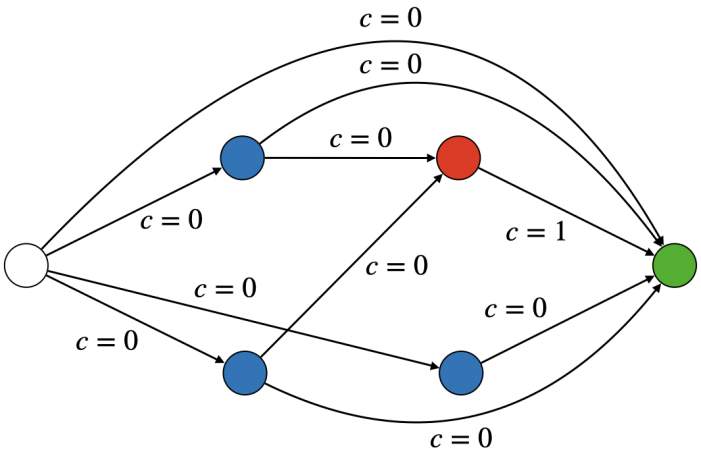
$$b_S = b_{total} \tag{4.43}$$

$$b_T = -b_{total} \tag{4.44}$$

where Equation (4.40) determines the supply of all nodes by subtracting the outgoing flows from the incoming flows while Equation (4.41) limits the flows between the lower and upper bounds for all transactions except the leaked transactions while Equation (4.42) strictly set the flows for the leaked transactions (since their flows are publicly known). Equation (4.43) and Equation (4.44) set the supply of the source and sink nodes where b_{total} is the total token supply to the blockchain. These two network graphs are illustrated in Figure 4.16 where the top is for the maximum feasible balance while the bottom is for the minimum feasible balance. Note that we assume there is only one feasible region between the minimum and the maximum feasible values found where all the in-between values are feasible too. The correctness of this assumption will be shown in the next section.



(a) For Maximum Feasible Bound



(b) For Minimum Feasible Bound

Figure 4.16. Balance range disclosure networks: (white) source, (green) sink, (blue) ordinary users, (red) user to be attacked.

4.4.2. Contiguity in Balance Ranges

By solving two separate minimum cost flow networks with +1 and -1 cost assignments on certain edges, we eventually find the minimum and the maximum possible balances that a specific user may have. However, such an approach only becomes valid as long as all the balances residing between the minimum value and the maximum value are contiguously feasible. Otherwise, we would end up with multi-piece feasible regions in-between rather than single one-piece feasible region. Based upon the issue of balance contiguity, our goal here is to show that all the balances between the minimum

and the maximum feasible balances are also feasible to be able to validate the correctness of our approach. Firstly, note that the minimum cost flow network is designed for a linear optimization problem where the constraints are in the form of $Ax \leq b$. Any linear optimization problem is also a convex problem which results in a convex feasible region. According to the definition of convexity, any convex combination $\forall \alpha \in [0, 1]$ of any two arbitrary points $x_1, x_2 \in X$ taken from the feasible region X must reside in the same feasible region again [92]:

$$Ax_3 = \alpha Ax_1 + (1 - \alpha)Ax_2 \leq \alpha b + (1 - \alpha)b = b \tag{4.45}$$

$$Ax_3 \leq b \tag{4.46}$$

where $x_3 = \alpha x_1 + (1 - \alpha)x_2$. Based on these facts, it can be said that any balance between the minimum and the maximum feasible balances is also feasible.

4.5. Experimental Evaluation

In the experimental evaluation section of this chapter, we initially present the languages, tools, libraries and test parameters in order to improve the reproducibility of our experiments. We evaluate the performance of the proposed transfer protocol in terms of (i) the requirements the problem needs, (ii) the blockchain gas consumption (as gas units, gas costs in ethers and dollars), (iii) zero-knowledge proof generation/verification times in seconds and lastly (iv) zero-knowledge proof artifacts size in memory (i.e. size of proving key, verification key, proof circuit and proof itself). For the balance range disclosure attack, we also perform several experiments by varying the total number of blockchain addresses, transactions and transaction leakage ratios.

4.5.1. Experimental Setup

As defined in Section 4.2, our privacy-preserving payment protocol is built upon three main models (i.e. the proof, contract and interface models) where we use the following technologies for the proof model:

- ZoKrates framework [27]: It enables developers to build zero-knowledge proof schemes over arbitrary computation easily through its own domain-specific language. Visit Section 2.9 for more information.
- ZoKrates-js library [93]: It is a JavaScript library to integrate the off-chain zero-knowledge proof generation into the existing web interface.

and the following technologies for the contract model:

- Solidity language [94]: It is a language over which the smart contracts being compatible with the EVM-based (Ethereum Virtual Machine) blockchains can be developed for various dApps (decentralized applications).
- Remix IDE [95]: It is an open-source and online IDE (Integrated Development Environment) to reliably write, compile and test smart contracts.
- Ethereum [2]: It is a popular public blockchain platform over which we securely deploy and run our smart contracts (including the main transfer contract and two proof-verifying contracts for sender and receiver).

and the following technologies for the interface model:

- HTML/CSS Language: They are popular scripting languages to create the web pages to directly interact with the protocol users and to collect necessary information to be processed later.
- JavaScript Language: It is a scripting language to add functionalities to the web page to perform certain computations off-chain (e.g. processing user information, computing commitment values, generating proofs, etc.).
- Ethers-js Library [96]: It is a JavaScript library to bridge the interactions between the web interface and the smart contracts on blockchain.
- Eccrypto-js Library [97]: It is a JavaScript library to perform public-key encryptions and decryptions using the ECIES scheme (Elliptic Curve Integrated Encryption Scheme).
- Browserify Bundler [98]: It is a bundler to transform and bundle the JavaScript

libraries to be directly used on browsers.

- Webpack Bundler [99]: It is another bundler with the same purpose of *Browserify* where it is the only known way to integrate *ZoKrates-js* into browser.
- MetaMask [91]: It is an online cryptocurrency wallet to store digital assets. It must be installed beforehand to use all of our protocols in this thesis.

For the balance range privacy attack, the addresses and transactions are randomly generated with respect to the parameters for the number of total addresses and transactions. The transactions to be disclosed to the adversarial organization are also randomly selected among the available transactions. The experimental results of this attack are computed as the mean of 20 different runs where a randomly selected address is attacked at every run. During the experiments, (i) the number of addresses changes in the range of 100 and 1,000,000, (ii) the number of transactions changes in the range of 100 and 1,000,000 and (iii) the default total token supply is 1,000,000 while the default transaction leakage ratio is 0.5. To solve the resulting minimum cost flow network with these parameters, we use the Google OR-Tools framework [100] and the Parallel Network Simplex algorithm [101]. Lastly, the experiments for both privacy-preserving payment protocol and balance range privacy attack are performed on MacBook Pro Notebook with a 2.6 GHz Intel Core i7 processor, 16 GB memory and 6 cores. Visit the following GitHub page [51] to further inspect the open-source implementations.

4.5.2. Requirement Verification

In this section, we evaluate the proposed *PTTS* protocol with respect to the problem requirements (defined in Section 4.1) including privacy, confidentiality, trustless, public-verifiability, non-interaction, correctness and usability. Our protocol satisfies these requirements by integrating certain components (including blockchain, smart contracts, commitment scheme, public-key encryption, zero-knowledge proof and web interface) into the systems with the following way:

- (R1) Privacy: It protects the privacy of balances and amounts by storing their

corresponding commitments on-chain and later verifying the correctness of the computations over these commitments with zero-knowledge proof, (see Figure 2.7. for relation between commitments and zero-knowledge proof).

- (R2) Confidentiality: It requires the sender to encrypt transaction details with the receiver public key and later the receiver to decrypt the encryption with their private key, which allows confidentiality over the encryption.
- (R3) Trustless: The protocol uses zero-knowledge proof to verify off-chain computations of users to complete their payments without trusted third party.
- (R4) Public Verifiability: Anyone can publicly verify the correctness of the zero-knowledge proofs for a particular payment on blockchain.
- (R5) Non-interaction: The protocol uses a non-interactive zero-knowledge proof protocol (i.e. zkSNARKs [16]) which generates and verifies proofs without interaction. However, the protocol still needs a certain degree of interaction to submit encryption of transaction details from sender to receiver.
- (R6) Correctness: Without a web interface, all the other modules are necessary for the protocol to guarantee the correctness of the payment.
- (R7) Usability: The protocol has a web interface that maps every complex mathematical operation into a simple user action.

4.5.3. For Privacy-Preserving Token Transfer

4.5.3.1. Blockchain Gas Consumption. We present the blockchain gas consumption of the functions in our smart contracts in Table 4.4, (measured on 27/06/2022) where the gas price is 1.5 `Gwei` and the exchange rate from Ether to USD is $1,197.66. According to the table, the most expensive operation is *deploying contract* with over +4,5 million gas units corresponding to over $8. This cost is reasonable since it includes costs of the main *transfer* contract along with two proof-verifying contracts for verifying the depositing proof (see Figure 4.4) and the withdrawing proof (see Figure 4.5). These proof-verifying contracts are complex, requiring high gas consumption. However, this is the only cost the developer (or the contract owner) has to cover just for once to deploy a contract instance on blockchain. The next expensive functions are for depositing tokens

Table 4.3. *PTTS* verification of problem requirements.

Requirement	Blockchain	Smart Contract	Commitment Scheme	Public-Key Encryption	Zero-Knowledge Proof	Web Interface
R1. Privacy			✓		✓	
R2. Confidentiality				✓		
R3. Trustless			✓	✓	✓	
R4. Public Verifiability	✓	✓			✓	
R5. Non-interaction					✓	
R6. Correctness	✓	✓	✓	✓	✓	
R7. Usability						✓

(see Figure 4.8) with +2 million gas units and for withdrawing tokens (see Figure 4.9) with +1.6 million gas units since they verify complex proofs. Note that depositing tokens requires more gas units than withdrawing tokens since it has to submit an encryption as well, (see Line 7 of Figure 4.8). Sender needs to cover the total of \$3.78 = \$0.08 + \$3.70 to eventually deposit their tokens while receiver needs to cover the total of \$3.37 = \$0.33 + \$3.04 to eventually withdraw these tokens. Although these costs (can be viewed as bank commissions) may discourage users to use our protocol to a certain extent, it should be considered that these costs always stay constant regardless of the amount of tokens to be transferred while bank commissions vary pro rata. Nevertheless, cutting down the costs is among our future work to address.

We can generalize the total gas consumption of the protocol (per transfer) from the perspective of each actor (i.e. sender and receiver) by integrating the gas consumption of its individual functions as:

$$\gamma_{sender} = \gamma_{request} + \gamma_{deposit}$$

$$\gamma_{receiver} = \gamma_{grant} + \gamma_{withdraw}$$

where requesting consent $\gamma_{request}$ and depositing tokens $\gamma_{deposit}$ constitute the cost the sender has to cover γ_{sender} while granting consent γ_{grant} and withdrawing tokens $\gamma_{withdraw}$ constitute the cost the receiver has to cover $\gamma_{receiver}$. From the perspective of the entire system:

$$\gamma_{system} = \gamma_{sender} + \gamma_{receiver} \tag{4.47}$$

where we simply sum the costs of sender and receiver.

Table 4.4. Blockchain gas consumption of the *PTTS* protocol phases.

Function	Gas Units	Gas Cost (Ether)	Gas Cost (USD)
Deploying Contract	4,557,726	0.00683658	$ 8.19
Requesting Consent	44,300	0.00006645	$ 0.08
Granting Consent	183,344	0.00027501	$ 0.33
Depositing Tokens	2,060,133	0.00309019	$ 3.70
Withdrawing Tokens	1,699,399	0.00253816	$ 3.04

4.5.3.2. Proof Generation/Verification Times. In this experiment, we measure the zero-knowledge proof generation and verification times for both *depositing tokens* and *withdrawing tokens* proof where we perform 20 independent runs and take their average as the final results. The motivation of this experiment is to reveal how long the sender (or receiver) must wait in seconds in order to deposit (or withdraw) tokens. We theoretically know that the *depositing tokens* proof (see Figure 4.4) is relatively more complex than the *withdrawing tokens* proof (Figure 4.5) since it additionally checks the sufficiency of balance to deposit. In parallel to this theoretical expectation, we observe that the *depositing tokens* proof requires approximately 147 seconds while the *withdrawing tokens* proof requires 145 seconds on average. This implies that a single payment needs nearly 5 minutes to be completed (except the phases to request and grant consent). However, the zero-knowledge proof verification that occurs on-chain is completed with respect to the current throughput of the blockchain platform itself.

4.5.3.3. Proof Artifact Sizes. In this experiment, we measure the zero-knowledge proof artifacts including (i) the proving key size, (ii) the verification key size and (iii) the proof size itself. Note that developer generates and stores proving keys off-chain to generate proofs and verification keys on-chain to verify proofs. We theoretically know that the proving key size increases with the increasing complexity of the proof implementation while the verification key and proof size remain indifferent [27]. This is beneficial for our protocol since increase in proving keys is tolerable on today's computers with large memories while increase in verification keys and proofs would lead to more blockchain gas consumption and network load. With that respect, we observe from the experiments that the proving key for the *depositing tokens* proof is slightly greater than the proving key for the *withdrawing tokens* proof while the others are equal. Relatively, the proving keys are nearly $\sim$19,000x larger than the verification keys. On the contrary, the proofs are the shortest ones with just 1KB.

Table 4.5. *PTTS* zero-Knowledge proof artifact size.

Proof	**Proving Key Size**	**Verification Key Size**	**Proof Size**
Depositing Tokens	38.9MB	2KB	1KB
Withdrawing Tokens	38.8MB	2KB	1KB

4.5.4. For Balance Range Disclosure Attack

We evaluate the success of our attack with respect to two performance metrics. The first metric is the elapsed time to solve the minimum cost flow networks by simply subtracting the start time of the attack from its finish time. The second metric is the certainty level η (which we propose) to know how much certain we are about our calculations with the following way:

$$\eta = 1 - \frac{(\theta^{max} - \theta^{min})}{SUPPLY} \tag{4.48}$$

where θ^{max} and θ^{min} are the maximum and the minimum possible balance of the target address according to our attack while $SUPPLY$ is the total supply of token and η is the resulting certainty value. This equation results in between zero and one where zero

means that it has completely no information about that address while one means that our attack exactly finds the balance of that address without any uncertainty.

Before presenting the experimental results for the attack, let's analyze a transaction graph and the resulting minimum cost flow network from this transaction graph in Figure 4.17 and Figure 4.18 where the black nodes are source and sink addresses while the red node is the address to attack. We omit the node and edge labels for the sake of simplicity. As seen in the figures, the address to attack involves two incoming transactions and three ongoing transactions. These transactions on their own don not reveal any meaningful information about the balance of that address. However, in case any other nodes in the graph deliberately reveal their own transactions to an adversarial organization, our attack can track these transactions to narrow down the balance as much as possible in order to reduce uncertainty. That's why our attack results in calculation rather than prediction or estimation.

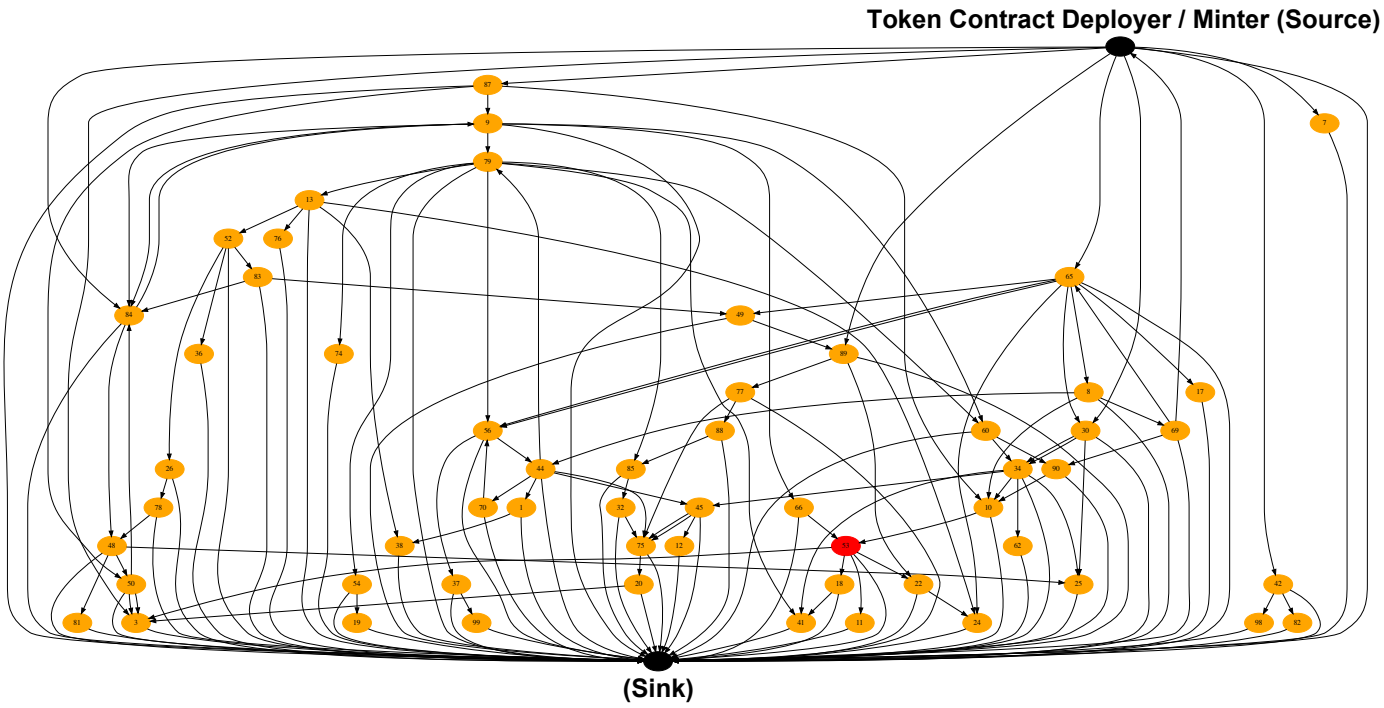


Figure 4.17. The minimum cost flow network: (black) source and sink addresses, (orange) ordinary addresses, (red) address to be attacked.

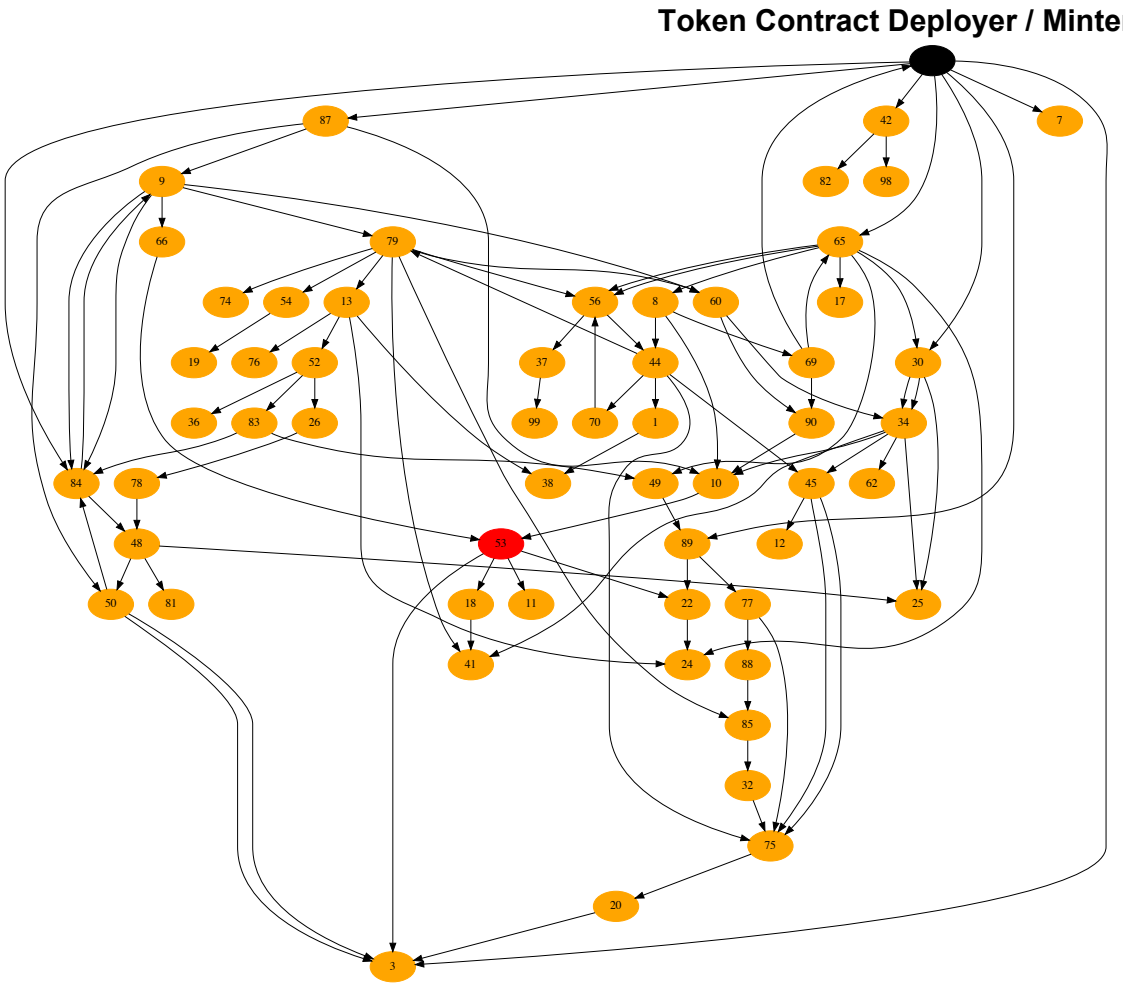


Figure 4.18. Transaction graph: (black) source and sink addresses, (orange) ordinary addresses, (red) address to be attacked.

4.5.4.1. Varying Number of Addresses and Transactions. The experimental results for the number of addresses and the number of transactions are given in Table 4.6 where total token supply is assumed to be 1,000,000 and the transaction leakage ratio is assumed to be 0.5 which means that half of all privacy-preserving transactions are leaked to the adversarial organization. We can observe from these results that the amount of time to solve the networks increases with increasing number of addresses and transactions (e.g. 100/100 case 1,000,000/1,000,000 case). However, our attack is able to solve the networks successfully in under two seconds. In the most extreme case of 1,000,000 addresses and 1,000,000 transactions, Google OR requires approximately 3 minutes

while PNS requires approximately 7 minutes. The certainty level metric uncovers the concept of transaction to address ratio which how many transactions the addresses involve with on average. In the scenarios where transaction to address ratio is higher, it is harder to trace the transactions and calculate the balance ranges correctly. We have seen this effect clearly for 100 addresses and 10,000 transactions (transaction to address ratio is 100) where the certainty level is zero. In other words, our attack fails where balances may vary from the bare minimum (i.e. zero) to bare maximum (i.e. total token supply). On the other hand, in scenarios where the users involve fewer transactions (i.e. transaction to address ratio is less than one), it is easier to have better calculation over the balance ranges.

Table 4.6. Network solution times and certainty levels for varying number of addresses and transactions where transaction leakage ratio is 0.5.

Number of Addresses	**Metrics**	**Number of Transactions**				
		100	**1,000**	**10,000**	**100,000**	**1,000,000**
100	**Time - Google OR (sec)**	<0.01	<0.01	0.01	0.10	1.05
100	**Time - PNS (sec)**	<0.01	<0.01	0.02	0.19	1.89
100	**Certainty Level**	0.89	0.06	0.00	0.00	0.00
1,000	**Time - Google OR (sec)**	<0.01	<0.01	0.02	0.13	1.41
1,000	**Time - PNS (sec)**	<0.01	<0.01	0.03	0.21	2.03
1,000	**Certainty Level**	0.94	0.90	0.01	0.00	0.00
10,000	**Time - Google OR (sec)**	0.02	0.02	0.10	0.35	2.37
10,000	**Time - PNS (sec)**	0.05	0.05	0.08	0.42	2.54
10,000	**Certainty Level**	0.98	0.98	0.91	0.03	0.00
100,000	**Time - Google OR (sec)**	0.15	0.16	0.23	2.97	9.83
100,000	**Time - PNS (sec)**	0.49	0.55	0.62	1.45	24.19
100,000	**Certainty Level**	>0.99	0.98	0.98	0.89	0.02
1,000,000	**Time - Google OR (sec)**	1.61	1.72	2.14	4.20	185.72
1,000,000	**Time - PNS (sec)**	7.75	8.14	10.11	11.96	425.82
1,000,000	**Certainty Level**	>0.99	>0.99	>0.99	>0.99	0.98

4.5.4.2. Varying Transaction Leakage Ratio. The experimental results for varying leakage ratio of transactions to adversarial organizations with respect to all available transactions in the network are given in Table 4.7 where it varies from 0.2 to 1.0. While the

ratio of 0 refers to the case with no transactions leaked while 1 refers to the case with all transactions leaked. From the table, we can firstly infer that the certainty level metric is gradually getting better with the increasing leakage ratio. This correlates with our expectations since adversarial organizations can estimate balance ranges better with more information. In the opposite scenario, they struggle to efficiently narrow down balance ranges. However, note that even if the leakage ratio is zero, balances of the addresses without involving any transactions at all can be still predictable as zero. Secondly, the elapsed time to solve the minimum cost flow networks decrease with the increasing leakage ratio since the proposed attack now runs over the network where the greater the number of edges have the same lower and upper bounds by making the network complex to solve.

Our balance range disclosure attack specifically targets the account-based platforms (e.g. Ethereum) where balance of each address is represented through a node in the network flows. However, this requires further elaboration in UTXO-based platforms (e.g. representing two node types as address node and UTXO node in the network flows) since each address now is a combination of multiple UTXOs. In addition, there exist different types of UTXO transactions (i.e. multi-signature transactions) that may pose additional complexities to be handled. Nevertheless, they share some common points: (i) source and destination addresses of UTXOs must be public and (ii) transaction amounts of UTXOs must be private. Although we hypothetically plead that our attack is also applicable for UTXO-based platforms, it needs further empirical validation and justification, which is among our future works. For mitigation against our attack, the following strategies can be adopted: (i) introducing of stealth addresses along with public addresses where [13], (ii) increasing the number of transactions per user by generating decoy transactions and (iii) further incentives for users to keep their own transactions private against adversarial organizations.

Table 4.7. Network solution times and certainty levels for varying number of addresses and transactions where transaction leakage ratio is 0.5.

Transaction Leakage Ratio	Metrics	Number of Addresses/Transactions				
		100/ 100	**1,000/ 1000**	**10,000/ 10,000**	**100,000/ 100,000**	**1,000,000 1,000,000**
0.2	**Time - Google OR (sec)**	<0.01	<0.01	0.13	2.88	135.20
0.2	**Time - PNS (sec)**	<0.01	<0.01	0.08	1.19	464.14
0.2	**Certainty Level**	0.38	0.36	0.34	0.30	0.29
0.4	**Time - Google OR (sec)**	<0.01	<0.01	0.15	3.55	142.29
0.4	**Time - PNS (sec)**	<0.01	<0.01	0.08	2.17	574.74
0.4	**Certainty Level**	0.84	0.69	0.76	0.63	0.82
0.6	**Time - Google OR (sec)**	<0.01	<0.01	0.07	1.14	26.93
0.6	**Time - PNS (sec)**	<0.01	<0.01	0.08	1.55	293.54
0.6	**Certainty Level**	0.86	0.95	>0.99	>0.99	>0.99
0.8	**Time - Google OR (sec)**	<0.01	<0.01	0.05	0.61	10.20
0.8	**Time - PNS (sec)**	<0.01	<0.01	0.07	1.14	213.31
0.8	**Certainty Level**	0.97	>0.99	>0.99	>0.99	>0.99
1.0	**Time - Google OR (sec)**	<0.01	<0.01	0.03	0.29	4.01
1.0	**Time - PNS (sec)**	<0.01	<0.01	0.07	0.91	22.60
1.0	**Certainty Level**	1.00	1.00	1.00	1.00	1.00

5. PROTOCOL II: PRIVACY-PRESERVING DATA AGGREGATION

- quis quid cum quo communicat?
(who shares what with whom?)

In this chapter, we define the privacy-preserving data aggregation problem on blockchain by specifying objectives, constraints and requirements. For the given problem, we propose our privacy-preserving data aggregation protocol (i.e. *PVSS*) that integrates commitment scheme, public-key encryption, zero-knowledge proof and hypercube networks under three main models as the zero-knowledge proof model, the smart contract model and the web interface model. We extend the base *PVSS* protocol to the *PRFX* protocol in order to be able to aggregate prefix computation data as well. We also propose multiple communication techniques (i.e. node multiplexing, topological recursing and data splitting) to extend the base *PVSS* protocol to support any arbitrary numbers of aggregators. We analyze the protocols with respect to the scalability (with computational, communication and storage overheads) and security perspectives (with potential attacks and formal reduction proofs). We also perform experiments to measure their blockchain gas consumption, zero-knowledge proof generation/verification times and zero-knowledge proof artifact sizes.

5.1. Problem Definition

Privacy-preserving data aggregation refers to a secure multi-party computation where a group of blockchain addresses (i.e. aggregators) aggregate their individual data to reach the global aggregation by still protecting the privacy of their own data. This problem (as in the *PTTS* protocol) has a specific kind of information asymmetry where each address possesses only their own data while the other aggregators not having this data need it to reach at the global aggregation by verifying the correctness of the

aggregation computation. For this problem, let $\mathcal{U}$ be a set of aggregators as:

$$\mathcal{U} = \{u_0, u_1, \ldots, u_{N-1}\} \tag{5.1}$$

where u_i is the ith aggregator while N is the maximum number of aggregators. Each aggregator is associated with:

$$u_i : \langle \chi_i, pk_i, sk_i \rangle \tag{5.2}$$

where χ_i is the private data, pk_i public key and sk_i is the secret key of the ith user. We assume that all values in the set of data values X must be specified from the set of only positive integers including zero (i.e. no data can be negative or real number):

$$X = \{\chi_0, \chi_1, \ldots, \chi_{N-1}\}, \quad \chi_i \in \mathbb{Z}^+ \cup \{0\}. \tag{5.3}$$

In the scope of privacy-preserving data aggregation problem, we define a data aggregation between two blockchain addresses with the following way:

$$f^2([\chi_0, \chi_1]) = \chi_0 + \chi_1 = \sum_{i=0}^{1} \chi_i \tag{5.4}$$

where f^2 is the generic pair-wise data aggregation function for two aggregators. Similarly, we can define a global aggregation between multiple blockchain addresses with the following way:

$$f^N([\chi_0, \chi_1, \ldots, \chi_{N-1}]) = \chi_0 + \chi_1 + \cdots + \chi_{N-1} = \sum_{i=0}^{N-1} \chi_i \tag{5.5}$$

where f^N is the generic global data aggregation function for multiple aggregators, which simply collects and transforms individual data into a global aggregation (i.e. sum). Note that in cases N complies to 2^d for any d, we can represent the function f^N

by recursively using the function f^2 as:

$$f^N = f^2(f^2(\dots,\dots), f^2(f^2([\chi_{N-4}, \chi_{N-3}]), f^2([\chi_{N-2}, \chi_{N-1}]))) \tag{5.6}$$

where the inner f^2 functions aggregates and returns the result to the immediate outer f^2 functions until the final f^2 function returns the final aggregation.

The definitions so far comprise only a public data aggregation without considering privacy-preserving characteristics of the problem that we have previously pledged. Now, we introduce two different privacy goals as the *data privacy* where no address must see the data of another address and the *intermediate aggregation privacy* where no address must see the intermediate aggregation of another address. Rather, each address must see only the final aggregation. For the data privacy:

$$\forall u_j \neq u_i : \Pr[\chi_i \mid \text{view}(u_j)] = \Pr[\chi_i] \tag{5.7}$$

where $view(u_j)$ refers to the information that user u_j has access to. This statement implies that there isn't any address u_j whose view on the data of the address u_i is probabilistically equal to the view of the address u_i itself. For the intermediate aggregation privacy:

$$\forall u_k \notin \{u_i, u_j\} : \Pr[f^2 : \chi_i + \chi_j \mid \text{view}(u_k)] = \Pr[f^2 : \chi_i + \chi_j] \tag{5.8}$$

where it implies that there isn't any address u_k whose view on the intermediate pairwise aggregation $f^2 : \chi_i + \chi_j$ the address u_i has is probabilistically equal to the views of the address u_i itself. From this perspective, we can now define the main objectives of the problem (i.e. maximizing these privacy goals) with the following way:

$$\min_{u_j} \quad \mathtt{I}(\text{view}(u_j); \chi_i), \quad \forall u_j \neq \{u_i\} \tag{5.9}$$

$$\min_{u_k} \quad \mathtt{I}(\text{view}(u_k); f^2 : \chi_i + \chi_j), \quad \forall u_k \neq \{u_i, u_j\} \tag{5.10}$$

where I(.) is the mutual information measurement function. The statement given in Equation (5.9) implies that the first goal of the problem is to minimize the amount of information the address u_j accesses to over the data χ_i that the address u_i has, as complied to the data privacy defined in Equation (5.7). Similarly, the statement in Equation (5.10) implies that the second goal of the problem is to minimize the amount of information the address u_k accesses to over the intermediate pair-wise aggregation $f^2 : \chi_i + \chi_j$ the address u_i has, as complied to the transaction privacy defined in Equation (5.8). In the most ideal scenario, we expect the mutual information for both objectives to be zero where this is interpreted as no information about χ_i and $f^2 : \chi_i + \chi_j$ leaks. Additionally, note how the information asymmetry in the beginning is transformed into the problem objectives later.

Privacy-preservation is the most significant requirement that our problem requires to be satisfied. However, there exist several other requirements as well as follows:

- Privacy: it must protect the privacy of data, (i.e. each aggregator must know only their own data).
- Confidentiality: it must ensure the secrecy of message encryptions on direct communication channels between parties.
- Trustless: aggregators must complete their aggregations without relying on any trusted third party.
- Public Verifiability: it must allow the correctness of the computations over the private data to be publicly verified.
- Authentication: it must prevent parties from being involved with the protocol if they are not known beforehand.
- Non-Interactivity: it must allow all parties to generate zero-knowledge proofs on their own without interaction.
- Scalability: it must support a growing number of parties with a tolerable degree of computational, communication and storage overheads.
- Correctness: it must function correctly and properly at any given time.
- Usability: it must provide a more user-friendly protocol.

Overall, our driving motivation for this problem can be stated as *how to develop a privacy-preserving but publicly-verifiable data aggregation protocol over a decentralized network (i.e. blockchain) through a certain cryptographic primitives (i.e. commitments, public-key encryptions, zero-knowledge proof and hypercube networks) without any assistance of trusted third parties.*

5.2. System Architecture

For the given problem in Section 5.1, we develop a privacy-preserving data aggregation protocol on blockchain with zero-knowledge proof and hypercube networks. Our protocol is (i) *privacy-preserving* since it hides the individual data of the parties throughout the entire aggregation session, (ii) *trustless* since the aggregators can complete their aggregations without relying on any trusted third party, (iii) *publicly-verifiable* since validators can still verify the correctness of the computations the parties perform off-chain for aggregation, (iv) *partially non-interactive* since it does not require interaction during proof generation (except the interactions for pair-wise aggregation itself in hypercube networks), (v) *scalable* since it provides logarithmic-scale efficiency over the increasing number of parties and (vi) *collective* since it requires collaboration of all aggregators altogether. The protocol is built upon two main actors as (i) *developer* who performs one-time setup and deploys the smart contracts on blockchain and (ii) *aggregator* who has private data and is interested in the global aggregation.

Our *PVSS* protocol has four main phases as: (i) *deploying contracts* where developer deploys the main transfer contract in addition to the proof-verifying contract for aggregating data, (ii) *registering* where parties join to a certain aggregation session with the other parties through their data commitments and public keys, (iii) *submitting aggregation* where parties submits encryptions of their data with the help of two separate hypercube networks and finally *(iv) verifying aggregation* where parties generate proofs (per network) to show the correctness of their own aggregation computations. The protocol employs two contracts in total as: (i) the main aggregation contract to maintain track hypercube networks as logical layout and store party commitments; and

(ii) the proof-verifying contract to verify the proofs at the fourth phase of the protocol. The state transitions for these phases are shown in Figure 5.1 as well. Note that single registration (i.e. the first protocol phase) proceeds to logarithmic-scale aggregation submission and verification (i.e. the second and third protocol phases). In addition, the architecture of the *PVSS* protocol is also depicted in Figure 5.2.

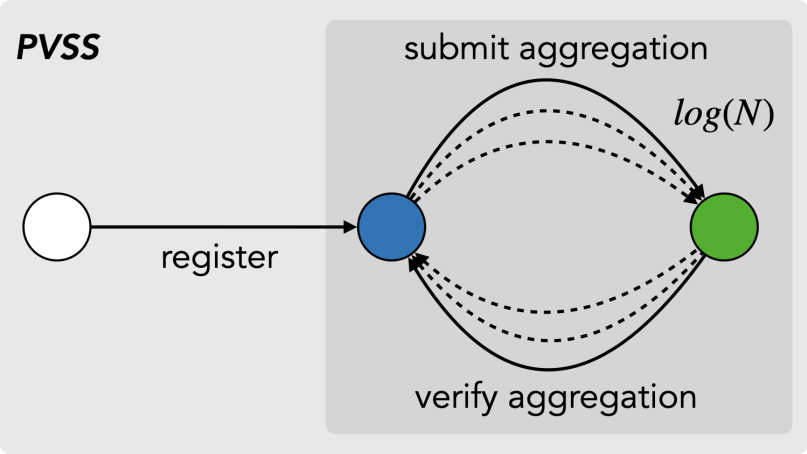


Figure 5.1. State-transition diagram of *PVSS* protocol.

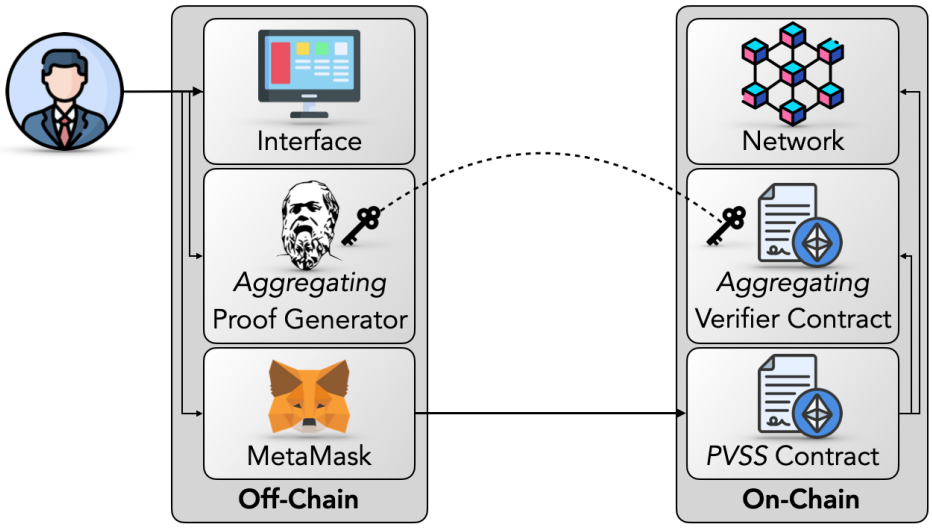


Figure 5.2. Architecture of *PVSS* protocol.

Deploying Contracts. Developer performs one-time setup to generate a proving

key (for off-chain proof generation) and a verification key (for on-chain proof verification) before deploying contracts to blockchain:

$$(pok, vek) \leftarrow \texttt{Zk.Setup}(\lambda) \tag{5.11}$$

where *pok* denotes proving key while *vek* denotes verification key. Note that the *PTTS* protocol requires two proving and verification key instances while this protocol requires only one instance. The proof-verifying contract also includes that verification key where proofs with mismatched proving keys will not be correctly verified.

Registering. Aggregator i first generates a random number to mask their individual data and maintains two aggregation instances as:

$$\Sigma_i^I \leftarrow \chi_i + \epsilon_i \tag{5.12}$$

$$\Sigma_i^{II} \leftarrow \epsilon_i \tag{5.13}$$

where χ_i and ϵ_i is the private and random data, respectively while Σ_i^I and Σ_i^{II} are these aggregation instances. Note that the aggregation at this step includes only private data of the aggregator itself. Later, the commitments of these aggregations with their corresponding salting parameters are computed:

$$c_i^I \leftarrow \texttt{Cm.Comm}(\Sigma_i^I) \tag{5.14}$$

$$c_i^{II} \leftarrow \texttt{Cm.Comm}(\Sigma_i^{II}) \tag{5.15}$$

where c_i^I and c_i^{II} are the resulting commitment values of the commitment function `Cm.Comm` based on the aggregations Σ_i^I and Σ_i^{II}. The former commitment will be used in the first hypercube network (i.e. I-Network) while the latter commitment will be used in the second hypercube network (i.e. II-Network). Aggregator completes the registration phase by generating their own keys:

$$(pk_i, sk_i) \leftarrow \texttt{PK.Setup}(\lambda) \tag{5.16}$$

where pk_i is the public key while sk_i is the secret key of the public-key setup function `Pk.Setup` based on a security parameter λ.

Submitting Aggregation. For every hypercube communication step h, the protocol pairs aggregator i with two other aggregators in two hypercube networks on-chain:

$$u_p^I \leftarrow u \oplus 2^h \tag{5.17}$$

$$u_p^{II} \leftarrow u \oplus 2^{(log(N)-h-1)} \tag{5.18}$$

where u_p^I and u_p^{II} are these peers in I-Network and II-Network, respectively where $t \in [0, log(N)-1]$ while $\oplus$ is the XOR operation. Note that these peers must be necessarily different in the initial hypercube step. Otherwise (i.e. the peers coincidentally are the same aggregator), the peer can easily learn information about the private data of aggregator as $\chi_i \leftarrow \Sigma_i^I - \Sigma_i^{II} \leftarrow (\chi_i + \epsilon_i) - \epsilon_i$. Therefore, I-Network proceeds through the hypercube dimensions as $0, 1, 2, ..., log(N) - 1$ while II-Network uses the reverse dimensions from $log(N) - 1$ to 0. II-Network may also use the shifted dimensions $1, 2, ..., log(N) - 1, 0$ as $u_{II}^p \leftarrow u \oplus 2^{(t+1) \bmod log(N)}$. Both of these methods guarantee the peers to be different at the initial communication step as well. Aggregator encrypts their first aggregation Σ_i^I through the public key of the first peer (in I-Network) and the second aggregation Σ_i^{II} through the public key of the second peer (in II-Network) with the following way:

$$E_i^I \leftarrow \texttt{Pk.Enc}([\Sigma_i^I], pk_p^I) \tag{5.19}$$

$$E_i^{II} \leftarrow \texttt{Pk.Enc}([\Sigma_i^{II}], pk_p^{II}) \tag{5.20}$$

where E_i^I and E_i^{II} are resulting encryptions of the public-key encryption function `Pk.Enc` while pk_p^I and pk_p^{II} are the public keys of the peers. We expect all the parties to perform these encryptions and share with their peers. Later, aggregator decrypts that encryption coming from their peers with their own private key sk:

$$\Sigma_p^I \leftarrow D^I \leftarrow \texttt{Pk.Dec}([E_p^I], sk_i) \tag{5.21}$$

$$\Sigma_p^{II} \leftarrow D^{II} \quad \leftarrow \quad \texttt{Pk.Dec}([E_p^{II}], sk_i) \tag{5.22}$$

where D^I and D^{II} are resulting decryptions of the public-key decryption function `Pk.Dec`. At that point, aggregator (and all the other aggregators) is ready to aggregate the aggregations (i.e. their own aggregations (Σ_i^I and Σ_i^{II}) and aggregations coming from the peers (Σ_p^I and Σ_p^{II})).

Verifying Aggregation. For every hypercube communication step h, aggregator i has to generate a zero-knowledge proof network in order to show correctness of the aggregations off-chain:

$$\Psi^I \quad \leftarrow \quad \Sigma_i^I + \Sigma_p^I \tag{5.23}$$

$$\Psi^{II} \quad \leftarrow \quad \Sigma_i^{II} + \Sigma_p^{II} \tag{5.24}$$

$$\pi_i^I \quad \leftarrow \quad \texttt{Zk.Gen}(\Psi^I, \Sigma_i^I, \Sigma_p^I, c_i^I, c_p^I, pok) \tag{5.25}$$

$$\pi_i^{II} \quad \leftarrow \quad \texttt{Zk.Gen}(\Psi^{II}, \Sigma_i^{II}, \Sigma_p^{II}, c_i^{II}, c_p^{II}, pok) \tag{5.26}$$

where Ψ^I and Ψ^{II} are the functions to which aggregator is subjected for I-Network and II-Network. While π_i^I and π_i^{II} are the resulting proofs of the the proof generation function `Zk.Gen` based on these functions (Ψ^I and Ψ^{II}), the aggregations (Σ_i^I, Σ_p^I, Σ_i^{II} and Σ_p^{II}) and their commitments (c_i^I, c_p^I, c_i^{II} and c_p^{II}), respectively for I-Network and II-Network. Aggregator later submits these proofs to on-chain contract to verify:

$$b_i^I \quad \leftarrow \quad \texttt{Zk.Vfy}(\pi_i^I, c_i^I, c_p^I, vek) \tag{5.27}$$

$$b_i^{II} \quad \leftarrow \quad \texttt{Zk.Vfy}(\pi_i^{II}, c_i^{II}, c_p^{II}, vek) \tag{5.28}$$

where b_i^I and b_i^{II} are the boolean results of the proof verification function `Zk.Vfy` based on the same functions and only the commitments (c_i^I, c_p^I, c_i^{II} and c_p^{II}). In case the proofs are correctly verified ($b_i^I \leftarrow true$ and $b_i^{II} \leftarrow true$), the contract replaces the current

aggregations with the next aggregations (i.e. aggregation of the aggregations):

$$c_i^I \leftarrow \texttt{Cm.Comm}(\Sigma_i^I \leftarrow \Sigma_i^I + \Sigma_p^I) \tag{5.29}$$

$$c_i^{II} \leftarrow \texttt{Cm.Comm}(\Sigma_i^{II} \leftarrow \Sigma_i^{II} + \Sigma_p^{II}) \tag{5.30}$$

where the contract stores these new aggregations c_i^I and c_i^{II}, respectively for I-Network and II-Network for the next hypercube communication step $h+1$. The second and third protocol phases (i.e. submitting and verifying aggregation) are repeated until the communication step is completed. After the final step, aggregator (and all the other aggregators) can compute the final aggregation by subtracting the result of the second hypercube network from the result of the first hypercube network. The summary of the *PVSS* protocol is given in Table 5.1. The sequence diagram of the *PVSS* protocol is depicted in Figure 5.3 by clearly demonstrating interactions among the protocol actors.

Table 5.1. The *PVSS* protocol

Protocol II: PVSS
1. zkSNARKs Setup
(a) Developer performs one-time setup by generating proving and verification keys as
$\texttt{(pok, vek)} \leftarrow \texttt{Zk.Setup}(\lambda)$.
(b) Aggregator data (i.e. initial aggregations) are immutably stored on the contract as
$c_i^{\chi} \leftarrow \texttt{Cm.Comm}(\chi_i)$.
2. Registering of u
(a) Aggregator u_i generates a random number to mask their private data
$\Sigma_i^I \leftarrow \chi_i + \epsilon_i$
$\Sigma_i^{II} \leftarrow \epsilon_i$.
(b) Aggregator takes the commitments of the initial aggregations
$c_i^I \leftarrow \texttt{Cm.Comm}(\Sigma_i^I)$
$c_i^{II} \leftarrow \texttt{Cm.Comm}(\Sigma_i^{II})$.
3. Aggregation Submitting of u_r
(a) Aggregator u_i submits the encryptions for two hypercube networks
$E_i^I \leftarrow \texttt{Pk.Enc}([\Sigma_i^I], pk_p^I)$
$E_i^{II} \leftarrow \texttt{Pk.Enc}([\Sigma_i^{II}], pk_p^{II})$.
(b) Once all aggregators submit their encryptions, aggregator u_i decrypts the encryptions coming from their peers
$\Sigma_p^I \leftarrow \texttt{Pk.Dec}([E_p^I], sk_i)$
$\Sigma_p^{II} \leftarrow \texttt{Pk.Dec}([E_p^{II}], sk_i)$.
3. Aggregation Verifying of u_r
(a) Aggregator u_i aggregates the aggregation (their own aggregations and aggregations coming from their peers)
$\Psi^I \leftarrow \Sigma_i^I + \Sigma_p^I$
$\Psi^{II} \leftarrow \Sigma_i^{II} + \Sigma_p^{II}$.
(b) Aggregator generates two proofs to prove their computations for two hypercube networks
$\pi_i^I \leftarrow \texttt{Zk.Gen}(\Psi^I, \Sigma_i^I, \Sigma_p^I, c_i^I, c_p^I, pok)$
$\pi_i^{II} \leftarrow \texttt{Zk.Gen}(\Psi^{II}, \Sigma_i^{II}, \Sigma_p^{II}, c_i^{II}, c_p^{II}, pok)$.
(c) The correct verification of proofs in the contract as
$b_i^I \leftarrow \texttt{Zk.Vfy}(\pi_i^I, c_i^I, c_p^I, vek)$
$b_i^{II} \leftarrow \texttt{Zk.Vfy}(\pi_i^{II}, c_i^{II}, c_p^{II}, vek)$.

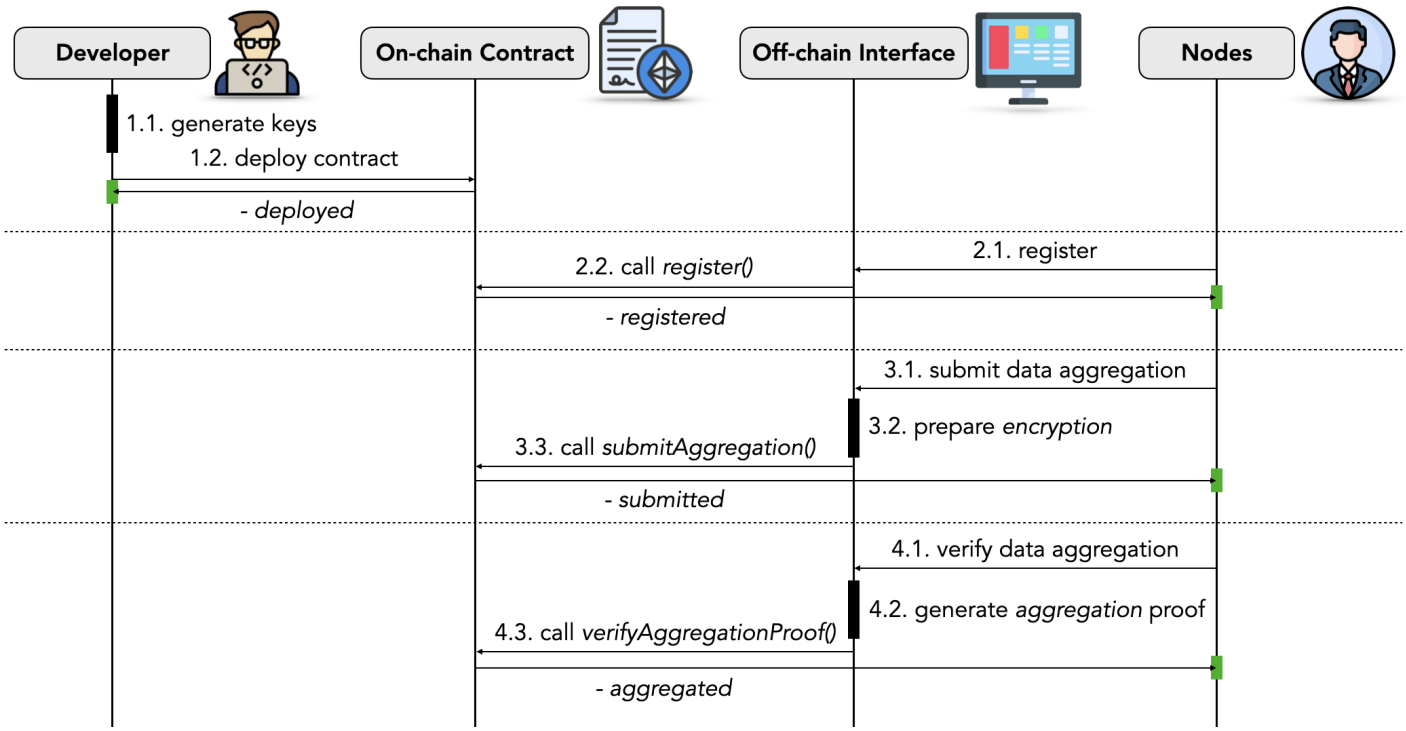


Figure 5.3. Sequence diagram of *PVSS* protocol.

We demonstrate the calculations over two hypercube networks in Figure 5.4 where there exist eight different users and consequently three hypercube steps (with simply $3 = log(8)$). In the first step of the first network, the users u_0 and u_1 exchanges the encryptions to compute Σ_1; the users u_4 and u_5 exchanges the encryptions to compute Σ_2; the users u_2 and u_3 exchanges the encryptions to compute Σ_4 and the users u_6 and u_7 exchanges the encryptions to compute Σ_4. In the second step, the users u_0, u_1, u_4 and u_5 can compute Σ_5 while the users u_2, u_3, u_6 and u_7 can compute Σ_6. In the third and final step, all the users can compute Σ_7 with the following way:

$$\Sigma_1 \leftarrow (\chi_0 + \chi_1 + e_0 + e_1)$$
$$\Sigma_2 \leftarrow (\chi_4 + \chi_5 + e_4 + e_5)$$
$$\Sigma_3 \leftarrow (\chi_2 + \chi_3 + e_2 + e_3)$$
$$\Sigma_4 \leftarrow (\chi_6 + \chi_7 + e_6 + e_7)$$
$$\Sigma_5 \leftarrow (\chi_0 + \chi_1 + \chi_4 + \chi_5 + e_0 + e_1 + e_4 + e_5)$$
$$\Sigma_6 \leftarrow (\chi_2 + \chi_3 + \chi_6 + \chi_7 + e_2 + e_3 + e_6 + e_7)$$

$$\Sigma_7 \leftarrow (\chi_0 + \chi_1 + \chi_2 + \chi_3 + \chi_4 + \chi_5 + \chi_6 + \chi_7 + e_0 + e_1 + e_2 + e_3 + e_4 + e_5 + e_6 + e_7). \tag{5.31}$$

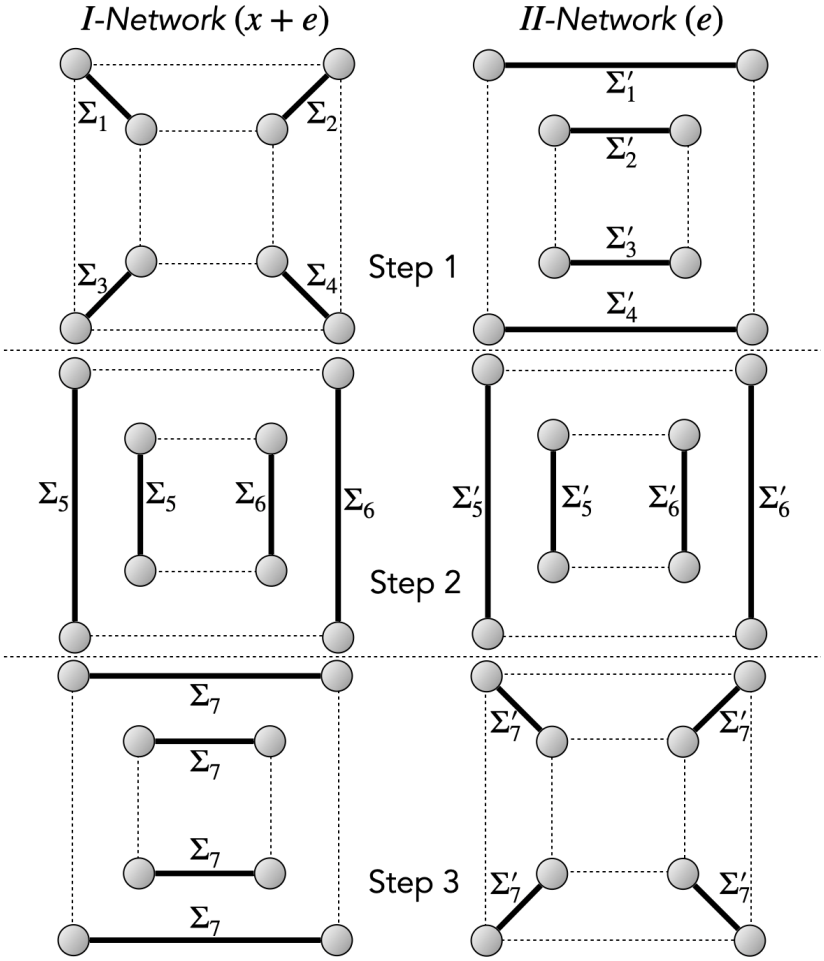


Figure 5.4. Aggregations of *PVSS* protocol in 3-dimensional hypercube network.

Similarly, in the first step of the second network, the users u_1 and u_3 exchanges the encryptions to compute Σ'_1; the users u_0 and u_2 exchanges the encryptions to compute Σ'_2; the users u_4 and u_5 exchanges the encryptions to compute Σ'_4 and the users u_5 and u_7 exchanges the encryptions to compute Σ'_4. In the second step, the users u_1, u_3, u_5 and u_7 can compute Σ'_5 while the users u_0, u_2, u_4 and u_6 can compute Σ'_6. In the third and final step, all the users can compute Σ'_7 as follows:

$$\Sigma'_1 \leftarrow (e_1 + e_3)$$

$$
\begin{aligned}
\Sigma'_2 &\leftarrow (e_0 + e_2) \\
\Sigma'_3 &\leftarrow (e_4 + e_6) \\
\Sigma'_4 &\leftarrow (e_5 + e_7) \\
\Sigma'_5 &\leftarrow (e_1 + e_3 + e_5 + e_7) \\
\Sigma'_6 &\leftarrow (e_0 + e_2 + e_4 + e_6) \\
\Sigma'_7 &\leftarrow (e_0 + e_1 + e_2 + e_3 + e_4 + e_5 + e_6 + e_7). \qquad (5.32)
\end{aligned}
$$

Once all the users have the aggregations for both networks, they now can individually subtract the aggregation of the second network from the aggregation of the first network, they can compute the final aggregation as follows:

$$
\Sigma_7 - \Sigma'_7 \leftarrow (\chi_0 + \chi_1 + \chi_2 + \chi_3 + \chi_4 + \chi_5 + \chi_6 + \chi_7). \qquad (5.33)
$$

5.2.1. Zero-Knowledge Proof Model

The goal of the proof model in our privacy-preserving data aggregation protocol is to present a publicly-verifiable computation over the commitments to be able to aggregate private values from multiple parties as a secure multi-party computation scheme. The algorithm of our proof model for *aggregating data* is given in Figure 5.5 where private function inputs include aggregation values of two parties with their salting parameters while public function inputs include their commitment values. The first aggregation value is from the aggregator running this proof while the second aggregation value is from the aggregator to which the first aggregator is paired in the hypercube networks. In Line 2-4, the commitments for the current aggregations and the final aggregation of these aggregations are internally computed. Line 4 shows how these two aggregations are summed with a new salting parameter. In Line 5, the proof checks the correctness of three internal commitments with the input commitments. In Line 6, it returns true if all the commitments are correctly given.

As in the *PVSS* protocol, this proof model uses large salting parameters to prevent an enumeration (look-up table) attack to resolve the commitments. The salting parameters are changing at each time an aggregation is performed (see Line 4) for an increasing security. Another advantage of the salting parameters is that in case the aggregations of two different parties are the same (e.g. all parties have eventually the same aggregation), their resulting commitments will be totally different. Second, off-chain proof generation needs all the inputs while the on-chain proof verification needs only the public inputs. Third, each aggregator maintains a state transition from aggregation to aggregation of aggregation until all aggregations are aggregated. Fourth, the impacts of the parties are totally decoupled by having power only their own aggregations. Finally, this proof model separately works for both *I*-Network and *II*-Network.

```
1: def main(private sum, private salt, private saltNext, private sumPair, private salt-
   Pair, public _sumComm, public _sumPairComm, public _sumNextComm):
2:     sumComm ← sha256(sum, salt)
3:     sumPairComm ← sha256(sumPair, saltPair)
4:     sumNextComm ← sha256(sum + sumPair, saltNext)
5:     result = if(_sumComm == sumComm &&
                   _sumPairComm == sumPairComm &&
                   _sumNextComm == sumNextComm) then true else false fi
6:     return result
```

Figure 5.5. Proof implementation in ZoKrates [27] for aggregating data.

5.2.2. Smart Contract Model

The goal of our smart contract model is to (i) manage the interactions between parties without any trusted party as a secure multi-party computation, (ii) store immutable and publicly-verifiable commitments for aggregations and (iii) verify the *aggregating data* proof on-chain for a seamless privacy-preserving data aggregation over *I*-Network and *II*-Network. The contract model involves with the functions for the *registering* (see Figure 5.6), *submitting aggregation* (see Figure 5.7) and *verifying ag-*

gregation (see Figure 5.8) phases of our protocol. In Figure 5.6, the parties call the *register* function to join an aggregation group by providing commitments for their private data for two networks as well as their public keys. In Line 2, it checks whether the aggregator has already joined to prevent any multiple registration attack (i.e. *Sybil* attack) to gain more control over the networks of that session (eventually learning more information about data of the other parties). Once registered, it incrementally generates a new registration identification number in Line 3 and appropriately updates the corresponding entries in the mappings in Line 4-6.

```
1: def register(uint commitmentI, uint commitmentII, string publicKey):
2:     require(bytes(publicKeys[msg.sender]).length == 0)
3:     ids[msg.sender] ← registeredUsers.length
4:     commitmentsI[msg.sender] ← commitmentI
5:     commitmentsII[msg.sender] ← commitmentII
6:     publicKeys[msg.sender] ← publicKey
7:     registeredUsers.push(msg.sender)
```

Figure 5.6. Contract implementation for *registering* phase.

In Figure 5.7 for *submitting aggregation* phase, the parties call *submitAggregation* function with two encryptions for two hypercube networks. In Line 2-5, it pairs every aggregator on-chain with two different parties at two networks by *XOR*ing the registration identification number of that aggregator with respect to the current hypercube step (i.e. $log(N)$ total steps with N number of total parties). This naturally requires the contract to constantly track the steps of the networks. In Line 6-7, it submits the encryptions to the corresponding pairs in the networks by updating the entries in the corresponding dictionaries. It should be noted that the two peers of a aggregator only at the first step in these networks have to be necessarily different, (that's why pairing calculations differ in Line 2 and 3). Otherwise, it would be possible for the peer to learn more information about the private value of that aggregator as $x \leftarrow \Sigma_I - \Sigma_{II} \leftarrow (x + e) - e$.

In Figure 5.8 for *verifying aggregation* phase, any aggregator calls *verifyAggregation* function with two proofs and the next commitments for two hypercube networks. In Line 2-5, it first tracks the peer parties and their addresses. In Line 6-7, it verifies the proofs for the networks with the current and next commitments of the aggregator and the current commitments of the peer parties. If both proofs are correct in Line 8, the number of proofs verified is incremented to track the current step of the hypercube networks in Line 9. If the number of proofs verified is equal to the number of total users in Line 10, the current step is incremented as well in Line 11. If the peers of the aggregator do not verify their proofs yet in Line 13 and 19, the next commitments of the aggregator are stored on temporary variables in Line 14 and 20, for I-Network and II-Network, respectively. Otherwise, the commitments are directly updated in Line 16-18 and 22-24, for I-Network and II-Network, respectively. It should be noted that the current commitments are automatically fetched from the smart contract, which prevents any proof verification with incorrect commitments. Second, if temporary variables were not used (i.e. updating the commitments directly), the proofs of the aggregator being submitted later would not be correctly verified since the commitments used for proof generation and the commitments used for proof verification would be totally different.

```
1: def submitAggregation(string encryptionI, string encryptionII):
2:     uint256 pairI ← ids[msg.sender] ^ (2 ** currentStage)
3:     uint256 pairII ← ids[msg.sender] ^ (2 ** (maximumStage - currentStage - 1))
4:     address pairAddressI ← registeredUsers[pairI]
5:     address pairAddressII ← registeredUsers[pairII]
6:     encryptionsI[msg.sender][pairAddressI] ← encryptionI
7:     encryptionsII[msg.sender][pairAddressII] ← encryptionII
```

Figure 5.7. Contract implementation for *submitting aggregation* phase.

```
1: function verifyAggregation(Proof proofI, Proof proofII, uint nextCommitmentI, uint
   nextCommitmentII):
2:     uint256 pairI = ids[msg.sender] ^ (2 ** currentStage)
3:     uint256 pairII = ids[msg.sender] ^ (2 ** (maximumStage - currentStage - 1))
4:     address pairAddressI = registeredUsers[pairI]
5:     address pairAddressII = registeredUsers[pairII]
6:     bool isProofCorrectI = verifier.verifyTx(proofI,
                                                commitmentsI[msg.sender],
                                                commitmentsI[pairAddressI],
                                                nextCommitmentI)
7:     bool isProofCorrectII = verifier.verifyTx(proofII,
                                                 commitmentsII[msg.sender],
                                                 commitmentsII[pairAddressII],
                                                 nextCommitmentII)
8:     if (isProofCorrectI && isProofCorrectII):
9:        numberOfProofsVerified += 1;
10:       if (numberOfProofsVerified == registeredUsers.length):
11:          currentStage += 1;
12:          numberOfProofsVerified = 0;
13:       if (tempCommitmentsI[pairAddressI] == 0):
14:          tempCommitmentsI[msg.sender] = nextCommitmentI
15:       else:
16:          commitmentsI[msg.sender] = nextCommitmentI
17:          commitmentsI[pairAddressI] = tempCommitmentsI[pairAddressI]
18:          tempCommitmentsI[pairAddressI] = 0
19:       if (tempCommitmentsII[pairAddressII] == 0):
20:          tempCommitmentsII[msg.sender] = nextCommitmentII
21:       else:
22:          commitmentsII[msg.sender] = nextCommitmentII
23:          commitmentsII[pairAddressII] = tempCommitmentsII[pairAddressII]
24:          tempCommitmentsII[pairAddressII] = 0
```

Figure 5.8. Contract implementation for *verifying aggregation* phase.

5.2.3. Web Interface Model

The web interface model in our protocol abstracts the complex zero-knowledge proof and hypercube network orchestration into simple actions by providing a user-friendly experience. It provides four main actions for (i) *deploying contract*, (ii) *registering*, (iii) *submitting aggregation* and finally (iv) *verifying aggregation*. We expect the actors to install the MetaMask extension [91] on their browsers and connect to their wallets to interact with the contracts. The illustrations of these actions are shown in Figure 5.9. The interface is also publicly available in our open-source Github page for further inspection [52].

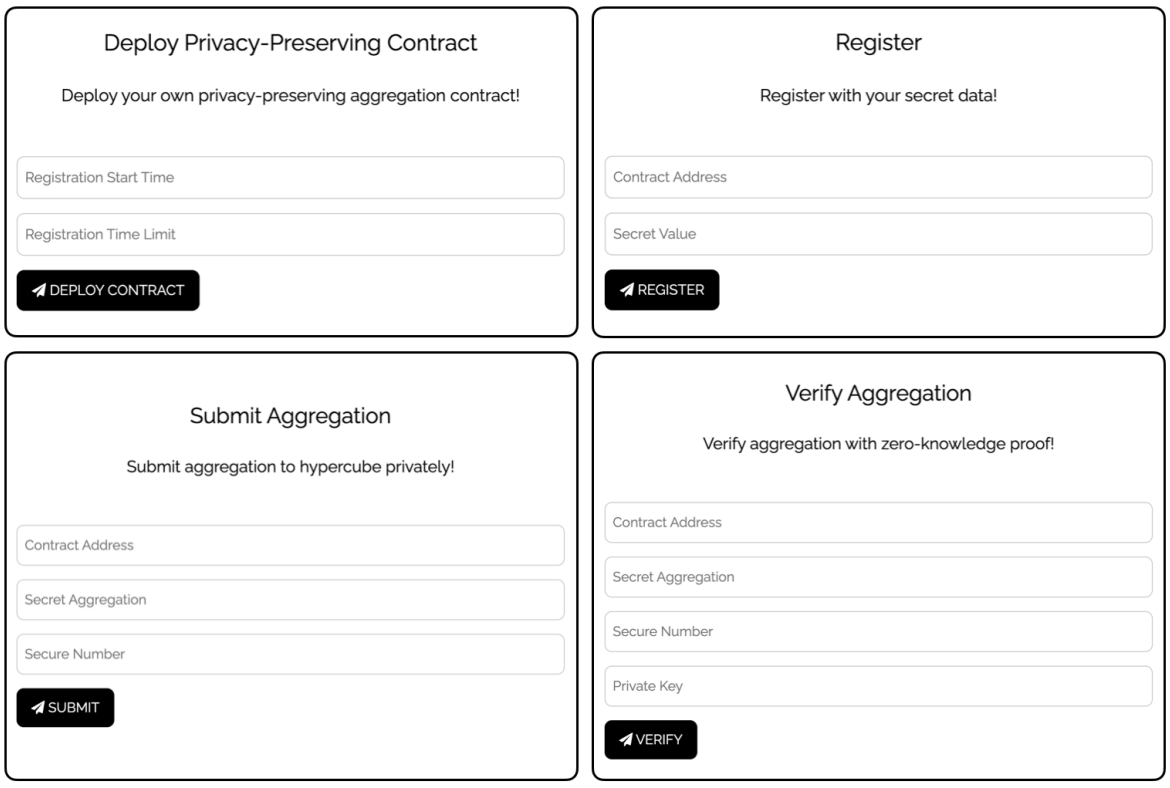


Figure 5.9. Web user interface of *PVSS* protocol.

The first action is *deploying contract* where a contract owner specifies the start time and time limit for registration to deploy a token contract instance to blockchain. Refer to Figure 4.11 for this action. The second action is *registering* where each aggre-

gator specifies their own private data to be aggregated. The algorithm for this action is given in Figure 5.10 where it restores public key in Line 4, computes hashes for the private data of the aggregator for two networks in Line 5 and calls *register* function (see Figure 5.6) in the contract in Line 6. Note that since only the aggregator knows their own private data, it has to be explicitly provided at every aggregation step.

```
1: async def register():
2:     [contractAddress, data] ← interface.getInputs()
3:     contract ← new web3.Contract(contractAddress)
4:     publicKey ← interface.restorePublicKey()
5:     [dataCommI, dataCommII] ← interface.commit(data)
6:     transaction ← contract.register(dataCommI, dataCommII, publicKey) - Figure 5.6
7:     return transaction
```

Figure 5.10. Interface implementation for *registering* phase.

The third action is *submitting aggregation* where each aggregator exchanges their aggregations with two peers over two networks. The algorithm for this action is given in Figure 5.11 where it gets public keys of their peers for two networks in Line 4, encrypts their aggregations by using these public keys in Line 5-6 and calls the *submitAggregation* function (see Figure 5.7) in the contract in Line 7. Finally, the fourth action is *verifying aggregation* where the aggregator must show the correctness of aggregation by using two aggregations (their own aggregation and aggregation coming from their peer per network). The algorithm for this action is given in Figure 5.12 where it gets encryptions from the contract in Line 4, decrypts them using their own private key in Line 5, computes necessary hashes based on all the aggregations in Line 6-7, generates two proofs with respect to these aggregations for two networks in Line 8-9 and calls the *verifyAggregation* function (see Figure 5.8) in the contract in Line 10. Note that the private key stays only inside the interface.

```
1: async def submitAggregation():
2:     [contractAddress, aggrI, aggrII] ← interface.getInputs()
3:     contract ← new web3.Contract(contractAddress)
4:     [publicKeyI, publicKeyII] ← contract.getPublicKeys()
5:     encryptionI ← interface.encrypt(aggrI, publicKeyI)
6:     encryptionII ← interface.encrypt(aggrII, publicKeyII)
7:     transaction ← contract.submit(encryptionI, encryptionII) - Figure 5.7
8:     return transaction
```

Figure 5.11. Interface implementation for *submitting aggregation* phase.

```
1: async def verifyAggregation():
2:     [contractAddress, aggrI, aggrII, privateKey] ← interface.getInputs()
3:     contract ← new web3.Contract(contractAddress)
4:     encryptions ← contract.getEncryptions()
5:     [aggrPairI, aggrPairII] ← interface.decrypt(encryptions)
6:     [aggrCommI, aggrPairCommI, nextAggrCommI] ← interface.commit(aggrI, aggrPairI)
7:     [aggrCommII, aggrPairCommII, nextAggrCommII] ← interface.commit(aggrII, aggr-
   PairII)
8:     proofI ← zokrates.generate(aggrI, aggrPairI, aggrCommI, aggrPairCommI, nextAggr-
   CommI) - Figure 5.5
9:     proofII ← zokrates.generate(aggrII, aggrPairII, aggrCommII, aggrPairCommII, nex-
   tAggrCommII) - Figure 5.5
10:    transaction ← contract.submit(proofI, proofII, nextAggrCommI, nextAggrCommII) -
   Figure 5.7
11:    return transaction
```

Figure 5.12. Interface implementation for *verifying aggregation* phase.

5.3. Extension to Incomplete Hypercube Networks

The main drawback of the *PVSS* protocol is that it heavily relies on the fully-complete hypercube networks which support only $N = 2^d$ number of aggregators in d-dimension. We observe this issue in the problem definition as well, (see Equation (5.6)) where it recursively uses the pair-wise data aggregation function f^2 to reach at the

global data aggregation. To eliminate this issue to be able to support any arbitrary number of aggregators $N = 2^d + l$ where l is residual, we need to relax this definition while still depending on the other existing definitions of the *privacy-preserving data aggregation* problem. However, we do not explicitly redefine Equation (5.6) here after this relaxation. Rather, we leave it to the techniques to be proposed in the next sections so that they can freely redefine with respect to the way they handle.

5.3.1. On Underdetermined System

The underdetermined system refers to a system of linear equations with fewer available equations than the available unknowns. This results in multiple solutions (or inability to determine a unique solution) for all unknowns. In the context of the privacy-preserving data aggregations, the unknowns are the private data of aggregating parties while the equations are the aggregations of these data using the hypercube networks. To provide a clearer explanation for this concept, we present two classifications of underdeterminacy as *global* underdeterminacy and *local* underdeterminacy. In the *global* underdeterminacy, no global solution (i.e. finding the values of all the unknowns exactly) is found while the values of certain unknowns can be still found. In contrast, in the *local* underdeterminacy, no values of any unknowns should be even found.

From that perspective, the *local* underdeterminacy is a stricter condition than the *global* underdeterminacy. A data aggregation system in the context of preserving privacy must necessarily be the *locally* underdetermined, thereby preventing finding the private data of any aggregator even when certain subsets of information are exposed. A concrete example for this issue is given in Figure 5.13 which targets for two collaboratively running 2-dimensional hypercubes (as I-Network and II-Network) for four different aggregators, (i.e. u_0, u_1, u_2, u_3). As shown in the figure, the $PVSS$ protocol [102] is *locally* underdetermined since any aggregator in the networks can obtain six aggregations (i.e. $\Sigma_1, \Sigma_1', \Sigma_4, \Sigma_4', \Sigma_6, \Sigma_6'$ for the aggregator u_1) at most from the system while there exist eight unknowns at total, (i.e. $\chi_0, \chi_1, \chi_2, \chi_3, e_0, e_1, e_2, e_3$). The

aggregation calculations for I-Network are as follows:

$$\begin{aligned}
\Sigma_0 &= \chi_0 + e_0 \\
\Sigma_1 &= \chi_1 + e_1 \\
\Sigma_2 &= \chi_2 + e_2 \\
\Sigma_3 &= \chi_3 + e_3 \\
\Sigma_4 &= \chi_0 + e_0 + \chi_1 + e_1 \\
\Sigma_5 &= \chi_2 + e_2 + \chi_3 + e_3 \\
\Sigma_6 &= \chi_0 + e_0 + \chi_1 + e_1 + \chi_2 + e_2 + \chi_3 + e_3
\end{aligned} \tag{5.34}$$

and the aggregation calculations for II-Network are as follows:

$$\begin{aligned}
\Sigma'_0 &= e_0 \\
\Sigma'_1 &= e_1 \\
\Sigma'_2 &= e_2 \\
\Sigma'_3 &= e_3 \\
\Sigma'_4 &= e_0 + e_1 \\
\Sigma'_5 &= e_2 + e_3 \\
\Sigma'_6 &= e_0 + e_1 + e_2 + e_3.
\end{aligned} \tag{5.35}$$

With this respect, we can say that $PVSS$ is not collusion-resistant since any two aggregators may collaborate (by sharing their aggregations with each other) to violate the *local* underdeterminacy of the system.

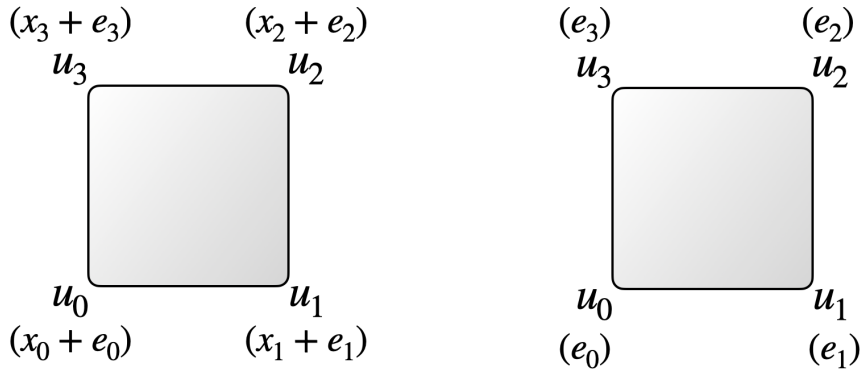


Figure 5.13. Undeterminacy on *PVSS* protocol.

5.3.1.1. Hypergraph Data Representation. The requirement of a privacy-preserving data aggregation system to be *globally* underdetermined is mentioned in the previous section. On the other hand, the requirement to be *locally* underdetermined can be expressed through the hypergraph representation. Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a hypergraph where $\mathcal{V}$ denotes the set of private data while $\mathcal{E}$ denotes the set of non-empty subsets of private data. The resulting hypergraph of a data aggregation system should not allow the element-wise difference between any two distinct linear combinations of these subsets to be less than 2. More formally:

$$|A_0 - A_1| > 1 \tag{5.36}$$

$$A_i = x_0\mathcal{E}_0 + x_1\mathcal{E}_1 + ... + x_{N-1}\mathcal{E}_{N-1} \tag{5.37}$$

$$x_i \in \{0, 1\} \tag{5.38}$$

where A_i is any linear combination of the hyperedges, x_i is the decision variable to consider the corresponding hyperedge $\mathcal{E}_i$ in that combination and N is the total number of hyperedges available in the graph. An example for the hypergraph data representation is given in Figure 5.14 where there exist five vertices and three hyperedges:

$$\begin{aligned} S_1 &= \chi_1 + \chi_2 + \chi_3 + \chi_4 + \chi_5 \\ S_2 &= \chi_1 + \chi_2 \\ S_3 &= \chi_3 + \chi_4. \end{aligned}$$

Note that the given hypergraph is not *locally* underdetermined since the value of x_5 can be found using the proper combination of the available hyperedges, $S_1 - (S_2 + S_3)$.

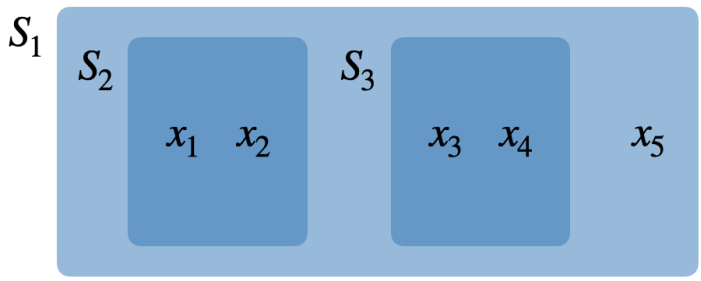


Figure 5.14. Hypergraph data representation of globally underdetermined system.

5.3.2. Communication Techniques on Incomplete Hypercube Networks

5.3.2.1. Node Multiplexing. *Node multiplexing* is a technique which takes the advantage of the existing parties that are already registered in order to fill the blanks positions in the hypercube networks. The algorithm for *node multiplexing* is given in Figure 5.16 where in Line 1-2, an arbitrary number of parties N register to the system. For each blank position available in Line 3, a random existing aggregator is selected to fill it in Line 4-5. Once all the positions are filled, the original *PVSS* protocol runs over the networks to find the final aggregation in Line 6. Note that the parties (if selected) need to proceed with their own original communications (i.e. aggregations) as well as the communications of the positions they fill in addition. At the worst case scenario (i.e. $2^d - 1$ parties in d-dimension), all the contributors except one have to fill two positions. More formally, if we redefine Equation (5.6) according to the way *node multiplexing* handles the problem:

$$f^N = f^2(\ldots, f^2(\ldots, f^2(\ldots, f^2(\chi_{N-2}, \chi_{N-1})))) \tag{5.39}$$

$$\chi_i = 0, \quad \forall i \geq 2^d \tag{5.40}$$

where it simply sets all the values to zero for all the residual positions (which are blank). The *node multiplexing* technique for three parties (i.e. u_0, u_1, u_2) is illustrated

in Figure 5.15 where u_0 is multiplexed to two positions in both networks. Note that in the figure, u'_0 in both networks is the same aggregator of u_0. Visit Section 5.2 for more information about the *PVSS* protocol itself.

Advantages. Node multiplexing have the following advantages as:

- It completely solves the main drawback of the *PVSS* protocol by supporting any arbitrary number of aggregators.
- It still uses the underlying fully-complete hypercube networks without additional topological overhead.

Disadvantages. Node multiplexing have the following disadvantages as:

- It requires the certain aggregators to proceed two different communications simultaneously to be able to fill the blank positions, which may increase the computational overhead (by generating additional zero-knowledge proofs).
- The verification of these additional proofs results in more blockchain gas consumption for certain aggregators, which may introduce unfairness.
- In case of an incorrect configuration, the multiplexed aggregator may learn additional information about data of the other contributors in the hypercube networks. This issue is shown in Figure 5.15. where u_0 at the end of the first iteration may maliciously benefit from the values from the network to compute:

$$\chi_1 = [\chi_0 + \chi_1 + e_0 + e_1] - [\chi_0 + e_0] - [e_1] \tag{5.41}$$

  where χ_1 is the private data of u_1. The frequency of this vulnerability may depend on the number of the blank positions in the network and the distribution of the available aggregators over these blank positions. Therefore, a more advanced selection technique to fill such positions (rather than the random selection) may be explored in the future (Line 4 of Figure 5.16).

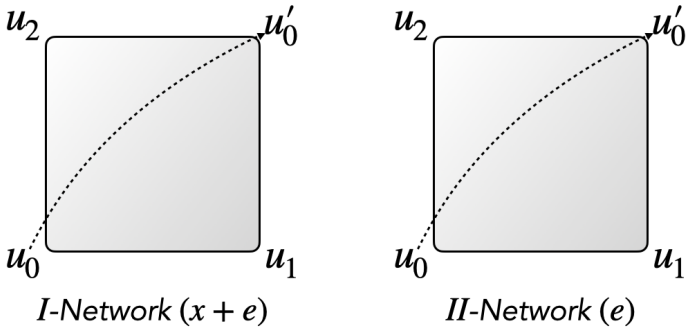


Figure 5.15. *Node multiplexing* technique on *PVSS* protocol.

```
1: for i in [0, N):
2:     HC[i] ← u_i.register(χ_i)
3: for j in [N, 2^⌈log N⌉):
4:     i ←$ HC
5:     HC[j] ← HC[i].register(0)
6: X ← PVSS(HC)
```

Figure 5.16. *Node multiplexing* technique on *PVSS* protocol.

5.3.2.2. Topological Recursing. *Topological recursing* is another technique that represents any number of the parties using a set of fully-complete hypercubes. The sub-aggregations of these complete sub-hypercube networks are recursively collapsed (i.e. aggregated) again until the final network is formed where the aggregation of this final network becomes the result of the entire system. This technique allows the composition of varying size and number of the distinct hypercubes to be formed with respect to the number of the total parties. The algorithm for *topological recursing* is given in Figure 5.18 where an arbitrary number of parties N register at first in Line 1-2. These parties are decomposed into multiple sub-hypercubes SHC in Line 3. Later, all the resulting sub-hypercubes in Line 4-5 are processed using the original *PVSS* protocol to find their own sub-aggregations in Line 6. Each sub-hypercube must randomly select a sentinel aggregator to represent that sub-hypercube in Line 7-8. This is recursively carried out in Line 9 until all the sub-hypercubes are collapsed into the single and final

aggregation in Line 10. More formally:

$$f_j^N([\chi_0, \chi_1, \ldots, \chi_{N-1}]) = \chi_0 + \chi_1 + \cdots + \chi_{N-1} = \sum_{i=0}^{N-1} \chi_i \quad (5.42)$$

$$f^N([f_0^N, f_1^N, \ldots, f_{|SHC|-1}^N]) = f_0^N + f_1^N + \cdots + f_{|SHC|-1}^N = \sum_{i=0}^{|SHC|-1} f_j^N \quad (5.43)$$

where $|SHC|$ is the number of sub-hypercubes. It first performs data aggregation at the first sub-hypercube levels in Equation (5.42) and later recursively aggregates these aggregations in Equation (5.43). The *topological recursing* is depicted in Figure 5.17 for 18 different parties (i.e. $u_0, u_1, ..., u_{17}$) where they are represented through four 2-dimensional and one 3-dimensional hypercubes. In the figure, four outermost sub-hypercubes are first collapsed into the innermost sub-hypercube where four sentinel parties there later compute the final aggregation.

Advantages. Topological recursing have the following advantages as:

- It requires fewer number of aggregators to proceed two different communications simultaneously (except the sentinel aggregators), which provides better performance than *node multiplexing* in terms of computational overheads and blockchain gas consumption.
- It offers more flexibility and scalability by supporting varying size and number of independent fully-complete hypercubes that can run in parallel.

Disadvantages. Topological recursing have the following disadvantages as:

- It requires every sub-hypercube to select a sentinel aggregator to represent that sub-hypercube in the next sub-hypercube, which results in additional orchestration and selection overhead on-chain.
- Distributing the aggregators correctly over varying size and number of hypercubes itself is a challenging task.
- It only partially solves the main drawback of the *PVSS* protocol since certain

numbers may not be expressed through a series of fully-complete sub-hypercubes.

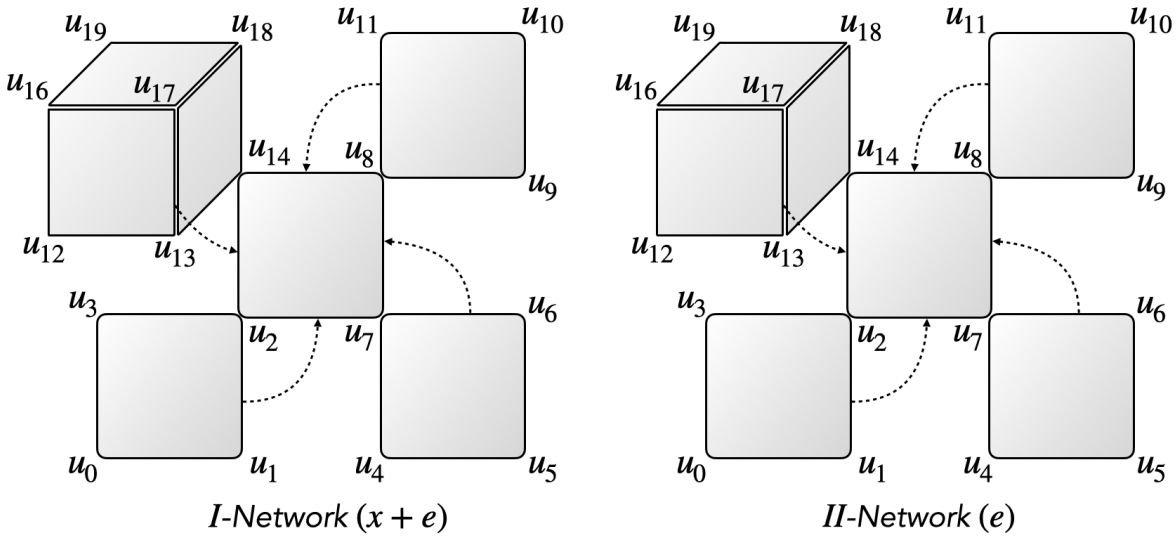


Figure 5.17. *Topological recursing* technique on *PVSS* protocol.

1: **for** i **in** $[0, N)$:
2: $\quad HC[i] \leftarrow u_i$.register$(\chi_i)$
3: $SHC = [SHC_0, SHC_1, ..., SHC_{M-1}] = HC$.decompose()
4: **while** SHC:
5: $\quad$ **for** j **in** $[0, |SHC|)$:
6: $\quad\quad X_j \leftarrow PVSS(SHC_j)$
7: $\quad\quad i \xleftarrow{\$} SHC_j$
8: $\quad\quad HC[j] \leftarrow SHC[i]$.register$(X_j)$
9: $\quad SHC = HC$.compose()
10: $X \leftarrow X_0$

Figure 5.18. *Topological recursing* technique on *PVSS* protocol.

5.3.2.3. Data Splitting. *Data splitting* is the last communication technique that we propose where the data of the aggregating parties are split into several data chunks with respect to the number of the blank positions in the network. The parties must proceed with the communications of the positions where their data chunks are used to fill. Therefore, unlike *node multiplexing*, the number of positions each aggregator

must manage may vary in this technique. However, dividing the data of all the parties except one into two chunks would be the most fair solution at the worst case scenario (i.e. $2^k - 1$ parties in a k-dimensional network). The algorithm for *data splitting* is given in Figure 5.20 where the arbitrary number of parties initially register in Line 1-2. The parties are randomly selected in Line 4 and their data are divided into two complementary chunks in Line 5. The blank positions in the network are filled in Line 6-7 until a fully-complete hypercube forms. Finally, the *PVSS* protocol runs over the resulting network to return the final aggregation in Line 8. More formally:

$$f^N = f^2(\ldots, f^2(\ldots, f^2(\ldots, f^2(\chi^2_{N-2}, \chi^2_{N-1})))) \tag{5.44}$$

$$\chi_i = \chi^1_i + \chi^2_i \tag{5.45}$$

$$\chi_i = \chi^2_i, \quad \forall i \geq 2^d \tag{5.46}$$

where it splits the data χ_i into two chunks and sets all the values to the latter chunks for all the residual positions (which are blank) in Equation (5.45). The *data splitting* is depicted in Figure 5.19 for four different parties (i.e. u_0, u_1, u_2, u_3) where the private data of u_0 is first divided into two chunks, $\chi_0 = \chi^1_0 + \chi^2_0$ and multiplexed into different positions. The same aggregator u_0 uses χ^1_0 and χ^2_0 in the first and second positions, respectively.

Advantages. Data splitting have the following advantages as:

- It solves the drawback of the *PTTS* protocol by supporting any arbitrary number of aggregators.
- Unlike *node multiplexing*, no aggregator learns additional information about data of the other aggregators in *data splitting*.

Disadvantages. Data splitting have the following disadvantages as:

- It still requires certain aggregators to proceed two different communications simultaneously, which implies additional computational overhead, blockchain gas

consumption and unfairness among aggregators.

- It requires orchestration to select certain aggregators to split their data into multiple chunks with respect to the blank positions.
- Splitting data into chunks requires the proof to show that the splitting operation itself is correctly performed, which means even more computational overhead.

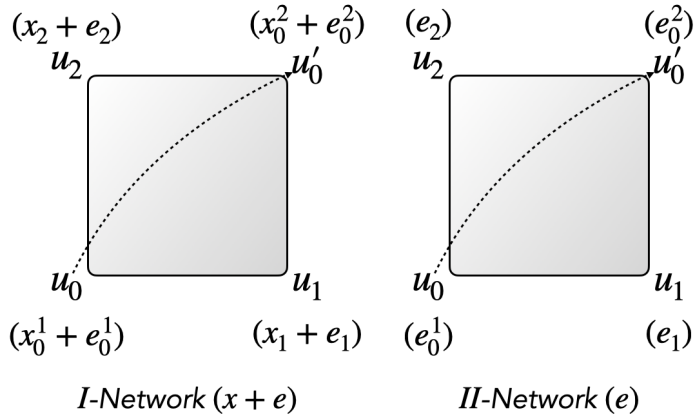


Figure 5.19. *Data splitting* technique on *PVSS* protocol.

1: **for** i **in** $[0, N)$:
2: $\quad HC[i] \leftarrow u_i$.register$(\chi_i)$
3: **for** j **in** $[N, 2^{\lceil \log N \rceil})$:
4: $\quad i \xleftarrow{\$} HC$
5: $\quad \theta_i^1, \theta_i^2 \xleftarrow{\$} \chi_i \mid \chi_i^1 + \chi_i^2 = \chi_i$
6: $\quad HC[i] \leftarrow HC[i]$.register$(\chi_i^1)$
7: $\quad HC[j] \leftarrow HC[i]$.register$(\chi_i^2)$
8: $X \leftarrow PVSS(HC)$

Figure 5.20. *Data splitting* technique on *PVSS* protocol.

We compare these techniques in Table 5.2 with respect to the following key attributes as *privacy*, *vulnerability*, *arbitrariness*, *practicality* and *interaction*. The full circle in the table shows the satisfiability of that attribute by the technique (vice versa for empty circle). We can infer from the table that *node multiplexing* may have a security vulnerability in case of an incorrect network configuration of networks by

compromising the privacy of private data χ_i. The proposed techniques successfully address the drawback of *PVSS* by supporting arbitrary numbers of aggregators except *topological recursing*. On the other hand, these techniques require additional orchestration (e.g. decomposing hypercube to sub-hypercubes in *topological recursing*) and interaction that makes *PVSS* more efficient in return.

Table 5.2. Comparison of communication techniques on incomplete hypercube networks.

Technique	Privacy	Vulnerability	Arbitrariness	Practicality	Interactivity
PVSS [102]	●	●	○	●	●
Node Multiplexing	◑	◑	●	◑	◑
Topological Recursing	●	●	◑	◑	◑
Data Splitting	●	●	●	◑	◑

5.4. Extension to Privacy-Preserving Prefix Aggregation

Privacy-preserving prefix aggregation refers to a secure multi-party computation where a group of blockchain addresses (i.e. aggregators) aggregate their individual data to reach at the prefix aggregation of their own data. While the *privacy-preserving data aggregation* problem focuses on single global aggregation, the aggregators in this problem end up with different prefix aggregation with respect to the position of the aggregator. To define this novel problem, we mostly depend upon the formal definition of the *privacy-preserving data aggregation* problem with certain modifications. Firstly, each aggregator needs to maintain two different aggregations (instead of one aggregation) for buffer values and values values with the following way:

$$f^{2b}([\chi_0, \chi_1]) = \chi_0 + \chi_1 = \sum_{i=0}^{1} \chi_i \tag{5.47}$$

where f^{2b} is the generic pair-wise buffer aggregation function for two aggregators and:

$$f_0^{2p}([\chi_0, \chi_1]) \quad = \quad \chi_0 = \sum_{i=0}^{0} \chi_i \tag{5.48}$$

$$f_1^{2p}([\chi_0, \chi_1]) \quad = \quad \chi_0 + \chi_1 = \sum_{i=0}^{1} \chi_i \tag{5.49}$$

where f_0^{2p} and f_1^{2p} are the generic pair-wise prefix aggregation functions for the aggregator on the previous position and the aggregator on the next position. In other words, the aggregator u_0 aggregates only their own data while the aggregator u_1 aggregates both data. Such separation of prefix aggregation functions results in different aggregators to eventually have different prefix values at the end. With the similar way, we can define a buffer aggregation between multiple blockchain addresses with the following way:

$$f^{Nb}([\chi_0, \chi_1, \ldots, \chi_{N-1}]) = \chi_0 + \chi_1 + \cdots + \chi_{N-1} = \sum_{i=0}^{N-1} \chi_i \tag{5.50}$$

where f^{Nb} is the generic global buffer aggregation function, which simply collects and transforms individual data into a global buffer. Similarly, we can now define a prefix aggregation between multiple blockchain addresses:

$$f^{Np}(i, [\chi_0, \chi_1, \ldots, \chi_{N-1}]) = \chi_0 + \chi_1 + \cdots + \chi_i = \sum_{i=0}^{i} \chi_i \tag{5.51}$$

where f^{Np} is the relative prefix aggregation function, which simply collects the data until the ith aggregator and transforms it into a relative prefix. Note it is the difference between Equation (5.5) and Equation (5.51) that distinguishes *privacy-preserving prefix aggregation* from *privacy-preserving data aggregation*, (see Figure 5.21). With respect to these modifications, we can now modify the privacy goals as well without further ado as follows:

$$\forall u_j \neq u_i \quad : \quad \Pr[\chi_i \mid \text{view}(u_j)] = \Pr[\chi_i] \tag{5.52}$$

$$\forall u_k \notin \{u_i, u_j\} \quad : \quad \Pr[f^{2b} : \chi_i + \chi_j \mid \text{view}(u_k)] = \Pr[f^{2b} : \chi_i + \chi_j] \tag{5.53}$$

$$\forall u_k \notin \{u_i, u_j\} \quad : \quad \Pr[f_i^{2p} : \chi_i + \chi_j \mid \text{view}(u_k)] = \Pr[f^{2b} : \chi_i + \chi_j] \tag{5.54}$$

where they are for *data privacy*, *intermediate buffer privacy* and *intermediate prefix privacy*, respectively. With respect to these goals, we can define the main objectives of the problem in the following way:

$$\min_{u_j} \quad \mathtt{I}(\text{view}(u_j); \chi_i), \quad \forall u_j \neq \{u_i\} \tag{5.55}$$

$$\min_{u_k} \quad \mathtt{I}(\text{view}(u_k); f^{2b} : \chi_i + \chi_j), \quad \forall u_k \neq \{u_i, u_j\} \tag{5.56}$$

$$\min_{u_k} \quad \mathtt{I}(\text{view}(u_k); f_j^{2p} : \chi_i + \chi_j), \quad \forall u_k \neq \{u_i, u_j\} \tag{5.57}$$

where $\mathtt{I}(.)$ is the mutual information measurement function. The objectives aim to minimize the amount of information on *data*, *intermediate buffer* and *intermediate prefix* values. In the most ideal scenario, we expect the mutual information for all objectives to be zero where this is interpreted that no information leaks. We leave the definitions of these privacy goals and objectives to the reader to avoid falling into repetition, (visit Section 5.1).

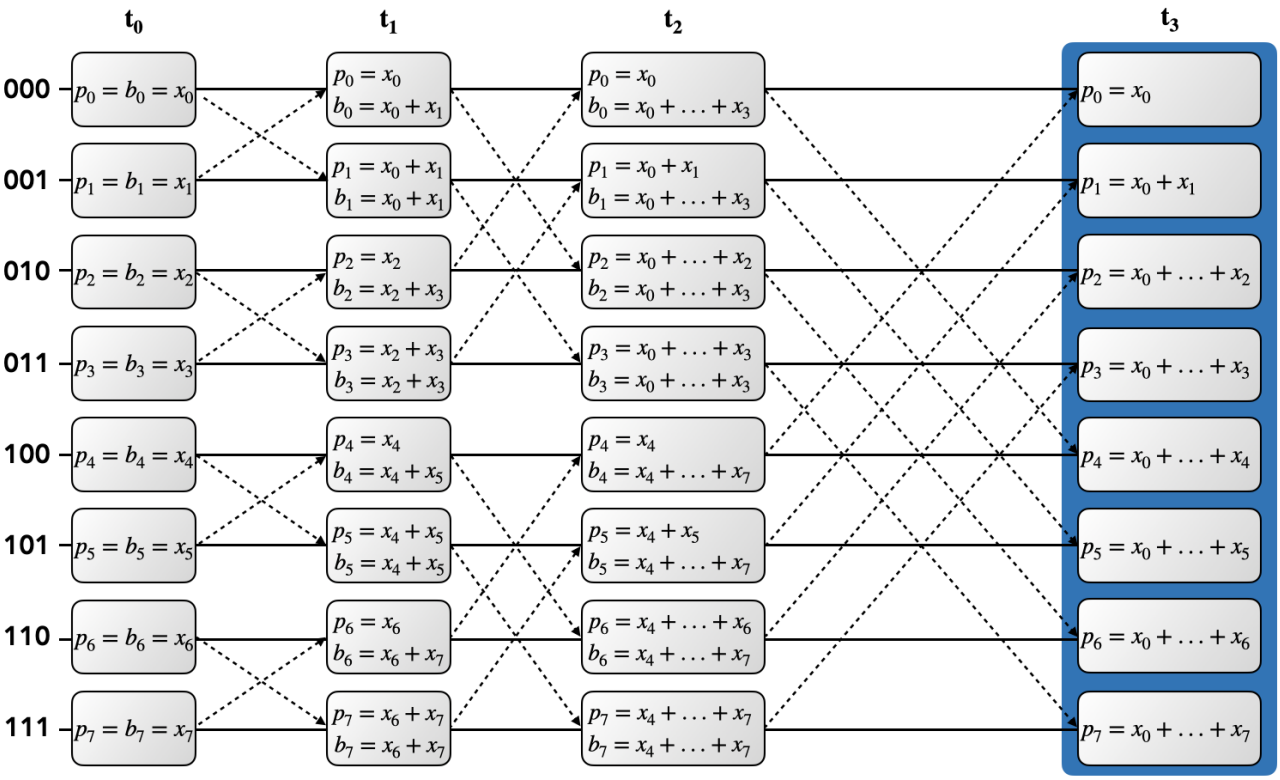


Figure 5.21. Communication of aggregators in *PRFX* protocol.

5.4.1. On Delegation Scheme

Here, we theoretically show the application of our *PRFX* protocol on the privacy-preserving delegation mechanism where the parties in blockchain network privately make delegations to the other parties with respect to the degree (i.e. weights) of these delegations and eventually learn only their cumulative sum of these delegations coming from the other parties. This mechanism can be applied to compute *(i)* the number of descendant parties per each party in the tree if the delegation weights are one or *(ii)* the voting power in the hierarchical voting mechanism if the delegation weights represent the amount of tokens the parties have. The illustration of the latter application is depicted in Figure 5.22 where the tree on the left (i.e. token tree) shows the amounts of tokens each party has while the tree on the right (i.e. delegation tree) shows the cumulative sum of each party, representing their voting power pro rata. Note that each party makes a delegation by combining the total delegation collected from the others with their own amounts of tokens (e.g. 1000 = [200 + (100 + 50)] + [150] + [250 + (50)] + [100 + (100)]). All the parties are later informed about their own accumulated token amounts so that they can use that information *(i)* to determine their proportionate voting powers and *(ii)* to choose strategies in the next delegation rounds.

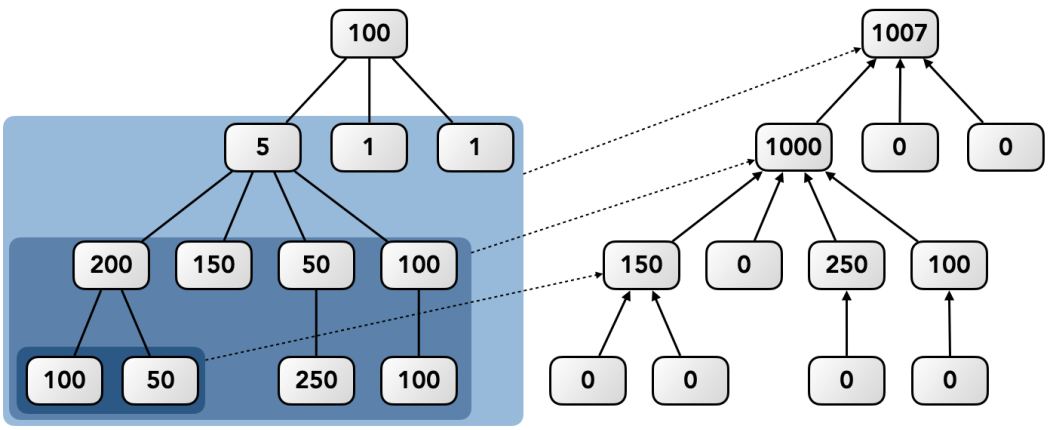


Figure 5.22. Delegation of tokens in *PRFX* protocol.

This privacy-preserving delegation problem can be formulated as the directed

graph where the nodes are the set of public blockchain addresses (i.e. parties) while the edges are the set of private delegations between these parties where their weights $w(e_{uv})$ are the amount of tokens they delegate to. The goal of our problem is to privately compute the total amount of tokens delegated to each node in the following way:

$$\mathcal{S}(v) \quad = \quad \sum_{e_{uv} \in T(v)} w(e_{uv}) + \mathcal{S}(u) \tag{5.58}$$

where $T(v)$ is the resulting sub-tree whose root is the node v and $\mathcal{S}(v)$ is the total delegation the node v eventually collects. This formula helps transforming the token tree into the delegation tree. However, the resolution of the resulting delegation tree through our protocol requires a linearization algorithm to traverse it. In this work, we use the depth-first traversal-based Euler Tour Technique, which splits each edge into two directed edges as downward (i.e. advance) and upward (i.e. retreat) edges [103]. It basically transforms the traversal of the given tree into the traversal of the singly-linked list. The first step of this transformation is shown in Figure 5.23 where for the sake of the problem, we set the weights of the advance edges from node u to v as the commitments of the amounts of tokens delegated, $w(e_{uv}) = c_{uv}$; and the weights of the retreat edges simply as zero, $w(e_{vu}) = 0$. The second step of the linearization tracks these resulting circular edges by beginning and finishing on the root (e.g. $A \rightarrow B \rightarrow C \rightarrow B \rightarrow D \rightarrow E \rightarrow D \rightarrow B \rightarrow A$), which is shown in Figure 5.24 The values at the top of the figure indicate the weights of the edges while the values at the bottom indicate their prefix summations from left to right. The linearization of the delegation tree has certain consequences to be noted as well during the prefix sum calculation: (i) single node in the graph may play multiple roles in the hypercube network, (ii) a node with more roles has more information about that round and (iii) a node may communicate with itself in the hypercube network.

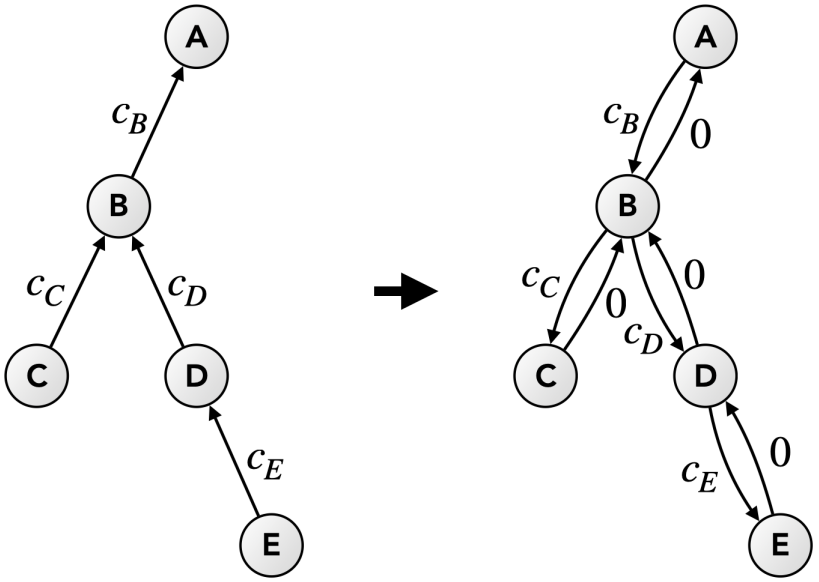


Figure 5.23. Transformation of global directed graph with Euler our Technique.

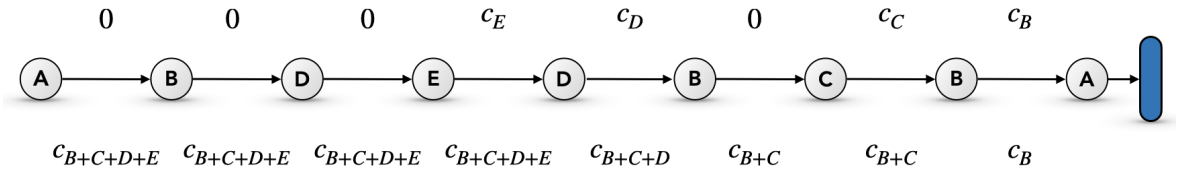


Figure 5.24. Linearization of global directed graph with Euler Tour Technique.

The *PRFX* protocol for that delegation mechanism needs the following steps to be performed: (*i*) *delegation*: the parties privately delegate tokens to the candidate parties by generating zero-knowledge proofs to prove that they have enough balances where the corresponding edges for the delegations are appended to the global directed graph right after, (*ii*) *linearization*: the global graph is traversed into single-dimensional linked list through Euler Tour Technique, (*iii*) *prefix submission*: the parties pairwise exchange their prefix summations which is shown in Figure 5.25, (*iv*) *prefix verification*: the parties verify the prefix summations with zero-knowledge proofs on-chain and finally (*v*) *delegation computation*: the parties privately learn the total amount of tokens delegated of themselves. In this final step, the parties may end up with multiple prefix values as $P : (p_0, p_1, ..., p_i)$ since they may have multiple roles, (see Figure 5.25). To

compute their total amount of tokens delegated, they must use the following way:

$$\mathcal{S}(v) = max(P) - min(P) \tag{5.59}$$

where $min(\mathcal{P})$ and $max(\mathcal{P})$ represent the minimum and maximum prefix values of the party v. So, their delegations are:

$$c_{B+C+D+E} - 0 = c_{B+C+D+E} \tag{5.60}$$
$$c_{B+C+D+E} - c_B = c_{C+D+E} \tag{5.61}$$
$$c_{B+C} - c_{B+C} = 0 \tag{5.62}$$
$$c_{B+C+D+E} - c_{B+C+D} = c_E \tag{5.63}$$
$$c_{B+C+D+E} - c_{B+C+D+E} = 0 \tag{5.64}$$

for party A, B, C, D and E, respectively. With this discussion, we justify the applicability and validity of our proposed protocol.

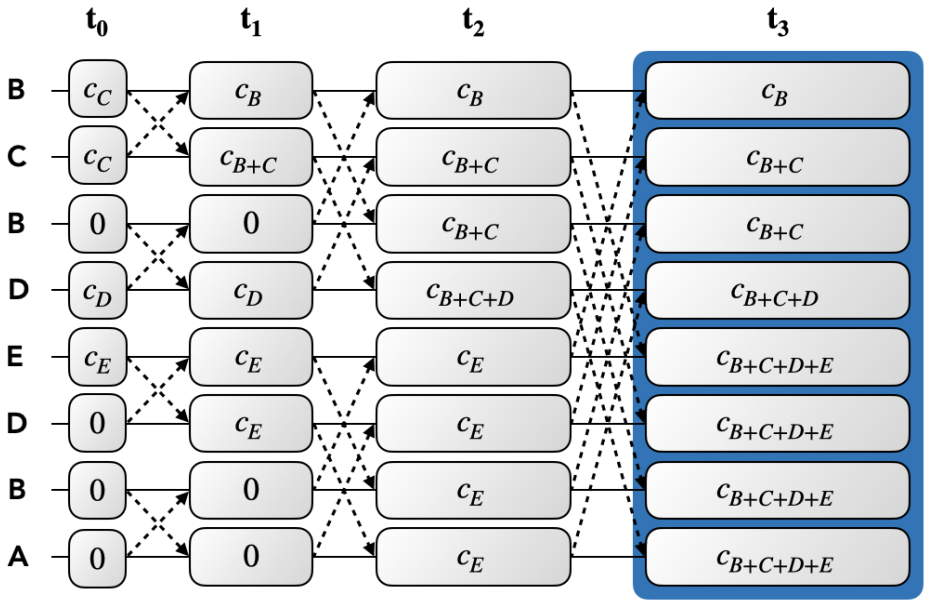


Figure 5.25. Privacy-preserving prefix summation with *PRFX* protocol.

The algorithm of our proof model for *aggregating prefix* is given in Figure 5.26

where private function inputs include prefix and buffer values of two parties with their salting parameters while public function inputs include their commitment values. In Line 2-6, the commitments for the current prefix aggregations and the final prefix aggregation of the aggregations are internally computed. Note that the buffers are always summed in Line 5 while the prefix are summed in case the party has higher order than the peer party in Line 6. In Line 7, the proof checks the correctness of the internal commitments with the input commitments. In Line 8, it returns true if all the commitments are correctly given. The main difference between this proof model and the proof model of *PVSS* is the way they aggregate data in Line 5-6.

```
1: def main(private prefix, private prefixSalt, private prefixNextSalt, private buffer,
   private bufferSalt, private bufferNextSalt, private bufferPair, private bufferPairSalt,
   public order, public orderPair, public _prefixComm, public _bufferComm, public
   _bufferPairComm, public _prefixNextComm, public _bufferNextComm):
2:     prefixComm ← sha256(prefixSalt, prefix)
3:     bufferComm ← sha256(bufferSalt, buffer)
4:     bufferPairComm ← sha256(bufferPairSalt, bufferPair)
5:     bufferNextComm ← sha256(bufferNextSalt, buffer + bufferPair)
6:     prefixNextComm ← if(order > orderPair) then
                            sha256(prefixNextSalt, prefix + bufferPair) else
                            sha256(prefixNextSalt, prefix) fi
7:     result ← if(prefixComm == _prefixComm &&
                bufferComm == _bufferComm &&
                bufferPairComm == _bufferPairComm &&
                bufferNextComm == _bufferNextComm &&
                prefixNextComm == _prefixNextComm) then true else false fi
8:     return result
```

Figure 5.26. Proof implementation in ZoKrates [27] for aggregating prefix data.

5.5. Protocol Analysis

5.5.1. Scalability

In this section, we theoretically analyze the scalability of the *PVSS* and *PRFX* protocols with respect to the computational, communication and storage overheads. Review Section 4.3.1 for further information about the importance of these overheads from the perspective of our thesis. Note that both *PVSS* and *PRFX* protocols relies on secure multi-party computation, which enables scalability analysis over the increasing number of parties. The overheads per party are more beneficial to consider from our perspective since the functionalities are completely distributed over all the parties. However, the overheads for the whole system can be safely used to estimate the system throughput in terms of the block interval since all the transactions have to be written on the blockchain.

5.5.1.1. Computational Overhead. The main factor causing the computational overhead is the zero-knowledge proof generation. For *PVSS*, each aggregator needs to generate two proofs for two hypercube networks for every hypercube network communication step $h \leftarrow log(N)$, which results in (by counting the number of proofs generated):

$$\xi_{comp} = 2 \cdot log(N) \tag{5.65}$$

$$\xi_{comp}^{\Sigma} = N \cdot \xi_{comp} \tag{5.66}$$

where ξ_{comp} and ξ_{comp}^{Σ} are the computational overheads on a party and on a system, respectively while N is the increasing number of parties. We can infer from the equations that more number of parties consequently results in higher computational overhead (i.e. with logarithmic increase on party and and super-linear increase on system). This requires the parties to generate more proofs and to spend more blockchain gas consumption to complete the protocol. Two proof generation processes for two hypercube networks might be further parallelized to alleviate temporal cost there. In addition,

the zero-knowledge proof verification on-chain is free from computational overhead. For incomplete communication techniques, *node multiplexing* needs $2 \cdot log(N)$ and *data splitting* needs $2 \cdot log(N) + 1$ with an additional proof to split the data into chunks. However, *topological recursing* may have relatively more overhead with $2 \cdot H \cdot log(N)$ where H is the number of total sub-hypercubes.

5.5.1.2. Communication Overhead. The main factor causing the communication overhead is the interactions among the parties over the hypercube networks. For *PVSS*, each aggregator needs to perform two direct communications to submit aggregation encryptions for every hypercube network communication step. We represent this overhead by counting the number of those interactions in the following way:

$$\xi_{comm} = 2 \cdot log(N) \tag{5.67}$$

$$\xi_{comm}^{\Sigma} = N \cdot \xi_{comm} \tag{5.68}$$

where ξ_{comm} and ξ_{comm}^{Σ} are the communication overheads on a party and on a system, respectively. There are two important facts to observe here. First, the number of direct communications is the natural result of the underlying network topology (i.e. hypercube). Second, the computational and communication overheads go as the same since the protocol requires a proof per encryption. For incomplete communication techniques, *node multiplexing* and *data splitting* requires $2 \cdot log(N)$ number of interactions while *topological recursing* needs $H \cdot log(N)$ for H number of sub-hypercubes.

5.5.1.3. Storage Overhead. The main factor causing the off-chain storage overhead is the proving key while the on-chain storage overhead is the mappings (i.e. dictionaries) to store (i) registrations, (ii) two aggregation commitments, (iii) two temporary aggregation commitments (see Figure 5.8), (iv) public keys and (v) two encryptions among parties. We count the number of entries in these mappings and lists to quantify the

Table 5.3. *PVSS* scalability as communication, computation and storage overheads.

#Nodes	Communication Overhead		Computation Overhead		Storage Overhead	
	On Node	On System	On Node	On System	On Node	On System
2	2	4	2	4	8	16
8	6	48	6	48	12	96
32	10	320	10	320	16	512
128	14	1792	14	1792	20	2560
1,000,000	40	40M	40	40M	26	26M

Table 5.4. *PRFX* scalability as communication, computation and storage overheads.

#Nodes	Communication Overhead		Computation Overhead		Storage Overhead	
	On Node	On System	On Node	On System	On Node	On System
2	1	2	1	2	8	16
8	3	24	3	24	10	80
32	5	160	5	160	12	384
128	7	896	7	896	14	1792
1,000,000	20	20M	20	20M	27	27M

storage overhead as follows:

$$\xi_{mem} = 6 + 2 \cdot log(N) \tag{5.69}$$

$$\xi_{mem}^{\Sigma} = N \cdot \xi_{mem} \tag{5.70}$$

where ξ_{mem} and ξ_{mem}^{Σ} are the storage (i.e. memory) overheads on a party and on a system, respectively. Be aware that the first component of Equation (5.69) (i.e. 6) represents the first four mappings while the second component (i.e. $2 \cdot log(N)$) represents the fifth mapping. We do not include the memory requirement of the verification key since its size stays always constant regardless of the protocol implementation. The scalability quantification of the *PVSS* protocol is hypothetically given in Table 5.3 with the increasing number of parties (from 2 to 1,000,000).

5.5.2. Security

In this section, we evaluate the security of the *PVSS* protocol with respect to potential attacks on blockchain and theoretically show the correctness by using the formal reduction proof. The attacks we consider include *replay* attack, *preimage* attack, *Sybil* attack and *collusion* attack. The security of the protocol can be further analyzed based on more variety of attacks, which is among our future works. Refer to Section 4.3.2 for the definitions of these attacks in the scope of our thesis.

Replay Attack. The protocol is resistant to the replay attack since the corresponding commitments for aggregations are immediately updated once zero-knowledge proofs are correctly verified. For instance, aggregator balance is replaced with a new balance commitment on-chain once the *aggregating* proof is correctly submitted and verified in the contract. In case the same transaction is attempted to be replayed, that proof would not be verified based on these new commitments since the set of commitments during proof generation and verification must be exactly matched.

Preimage Attack. The protocol handles this attack by introducing large and random salting values σ while computing commitment and regularly replacing these salting values with new values, (see Section 2.6). We use salting values to mask aggregator balances. This precaution results in distinct commitment values even if the aggregations of two distinct aggregators are numerically the same. This requires the attacker to store a large enumeration table to compare the commitment values, which is considered to be infeasible.

Sybil Attack. The protocol simply prevents the same aggregator to register to the system by constantly checking the addresses that are already registered. This also prevents an aggregator to be in multiple positions on hypercube networks with the goal to learn more information about the other aggregators. Note that an aggregator may register through different blockchain addresses where its detection and prevention may require further mechanisms which are not in the scope of this thesis.

Collusion Attack. Although the protocol prevents collusion attack to a certain extent, it can be open to this attack on specific scenarios where two aggregators having direct communications with the same aggregator in the first hypercube communication step during the *submitting aggregation* phase may learn additional information about that aggregator.

5.5.2.1. Reduction Proof. In this section, we theoretically show the security of the *PVSS* protocol through the formal reduction proofs to the underlying zero-knowledge proof protocol (i.e. zkSNARKs [16]) as follows:

Theorem. The privacy-preserving data aggregation protocol *PVSS* is secure if and only if the zkSNARKs protocol [16] is secure.

Proof. Suppose that $\mathcal{A}$ and $\mathcal{B}$ are the adversaries that can break the *PVSS* and zkSNARKs protocols, respectively. For the following scenario between the adversary $\mathcal{A}$ and the challenger (i.e. verifier) $\mathcal{C}$:

- The challenger $\mathcal{C}$ runs the zkSNARKs trusted setup to generate the proving key pok and the verification key vek.
- The adversary $\mathcal{A}$ samples a random data aggregation $\Sigma_{\mathcal{A}}$ and forwards the commitment $c_{\mathcal{A}} \leftarrow \texttt{Cm.Comm}(\Sigma_{\mathcal{A}})$ to the challenger.
- The adversary $\mathcal{A}$ samples a random but falsified data aggregation $\Sigma'_{\mathcal{A}}$ where $\Sigma_{\mathcal{A}} \neq \Sigma'_{\mathcal{A}}$ and computes the commitment $c'_{\mathcal{A}} \leftarrow \texttt{Cm.Comm}(\Sigma'_{\mathcal{A}})$.
- The adversary $\mathcal{A}$ collects the data aggregation from their peer Σ_p and invokes the adversary $\mathcal{B}$ to generate a false proof $\pi \leftarrow \texttt{Zk.Gen}(\Psi, \Sigma'_{\mathcal{A}}, \Sigma_p, c_{\mathcal{A}}, c_p, pok)$.
- The challenger $\mathcal{C}$ returns true in case this false proof is correctly verified as $b_{\mathcal{A}} \leftarrow \texttt{Zk.Vfy}(\pi, c_{\mathcal{A}}, c_p, vek)$ and false otherwise.

Based on this scenario, the challenger $\mathcal{C}$ returns true at every time if and only if there really exists the adversary $\mathcal{B}$ that has ability to break zkSNARKs by forging the proof with respect to the falsified aggregation Σ'. However, this contradicts our assumption

that the zero-knowledge proof scheme itself is secure. More formally:

$$\left|\Pr\left[1 \leftarrow \mathtt{C}(\mathtt{Zk.Vfy}(\pi, c_{\mathcal{A}}, c_p, vek)) \;\middle|\; \begin{array}{l} (pok, vek) \leftarrow \mathtt{Zk.Setup}(\lambda) \\ \Sigma_{\mathcal{A}}, \Sigma'_{\mathcal{A}} \xleftarrow{\$} \mathbb{Z}_0^+, \quad \Sigma_{\mathcal{A}} \neq \Sigma'_{\mathcal{A}} \\ c_{\mathcal{A}} \leftarrow \mathtt{Cm.Comm}(\Sigma_{\mathcal{A}}) \\ c'_{\mathcal{A}} \leftarrow \mathtt{Cm.Comm}(\Sigma'_{\mathcal{A}}) \\ \Psi \leftarrow \Sigma_{\mathcal{A}} + \Sigma_p \\ \pi \leftarrow \mathtt{A}(\mathtt{B}(\mathtt{Zk.Gen}(\Psi, \Sigma'_{\mathcal{A}}, \Sigma_p, c_{\mathcal{A}}, c_p, pok))) \end{array}\right]\right| = 0 \tag{5.71}$$

where this equation implies that the probability of correct verification of the false proof must be zero from the perspective of the challenger (i.e. verifier).

5.5.3. Limitations

The *PVSS* and *PRFX* protocols have certain and common limitations that we must address in the future as: (i) partial interaction, (ii) lack of dynamic networks, (iii) dependency to rigid topology, (iv) aggregator liveness, (v) complex orchestration, (vi) platform dependencies and (vii) performance issues. We explain these limitations as follows:

- Partial Interaction: Both protocols allow the *aggregating data* proof to be generated and verified without any interaction among aggregators. On the other hand, they require the interactions on data submissions over two hypercube networks due to characteristics of the protocols themselves.
- Lack of Dynamic Networks: Both protocols need aggregators to be ready and fixed before starting the aggregations where no further aggregator can join or leave after that point. This creates the liveness issue for aggregators as well, which will be discussed as another item.
- Dependency to Rigid Topology: Both protocols rely on the hypercube network topology which does not support any arbitrary number of aggregators. Although

we propose three novel techniques (see Section 5.3) to resolve this issue, these techniques require empirical justification. The flexibility of the protocols to work on any topology can be addressed as a future work.

- Aggregator Liveness: Both protocols require all the aggregators to be live and ready until the aggregation is completed. Therefore, we require the aggregators to be honest-but-curious actors where even a single fully-adversarial aggregator may deliberately delay or cancel that aggregation session.
- Complex Orchestration: The *PRFX* protocol puts additional complexity and overhead while extending the *PVSS* protocol while supporting any arbitrary number of aggregators. This further orchestration may result in the increasing blockchain gas consumption, unfairness among aggregators (e.g. certain aggregators must fill two positions) and the increasing proof generation times. This issue can be alleviated with careful implementation to a certain extent.
- Platform Dependencies: As in *PTTS*, both protocols require EVM-dependent blockchain platforms and the zkSNARKs protocol, (see Section 4.3.3).
- Performance Issues: Both protocols have certain computational, communication and storage overheads to be addressed, (see Section 4.3.1).

5.6. Experimental Evaluation

In this section, we present the tools and frameworks to build the privacy-preserving data and prefix protocols as well as the test parameters to improve reproducibility of our experiments. We evaluate the the performance of the protocols with respect to (i) the problem requirements, (ii) the blockchain gas consumption, (iii) zero-knowledge proof generation/verification times and (iv) zero-knowledge proof artifact sizes for proving key, verification key, proof circuit and proof itself. We also justify the scalability of our protocols with respect to the increasing number of parties in our experiments.

5.6.1. Experimental Setup

As defined in Section 5.2, our aggregation protocols have three models as the zero-knowledge proof, smart contract and web interface models where the first model requires (i) *ZoKrates framework* [27] and (ii) *ZoKrates-js library* [93]; the second model requires (i) *Solidity language* [94], (ii) *Remix IDE* [95] and (iii) *Ethereum* [2] while the third model requires (i) *HTML/CSS/Java Languages*, (ii) *Ethers-js Library* [96], (iii) *Eccrypto-js Library* [97], (iv) *Browserify Bundler* [98], (v) *Webpack Bundler* [99] and (vi) *MetaMask* [91]. Visit Section 4.5.1 for further information about these libraries. Finally, the experiments targeting these protocols are performed on MacBook Pro Notebook with a 2.6 GHz Intel Core i7 processor, 16 GB memory and 6 cores. The open-source implementations are also publicly available in the websites [52] and [53] for *PVSS* and *PRFX*, respectively.

5.6.2. For Privacy-Preserving Data Aggregation

5.6.2.1. Requirement Verification. In this section, we evaluate the proposed protocol with respect to the problem requirements (defined in Section 5.1) including privacy, confidentiality, public-verifiability, authentication, non-interaction, scalability, correctness and finally usability. Our protocol satisfies these requirements by specifically integrating certain components into the systems as given below:

- (R1) Privacy: It protects the privacy of data to be aggregated by representing it through a corresponding commitment value on-chain and verifying the correctness of the computations over it through zero-knowledge proof.
- (R2) Confidentiality: It uses the public-key encryption scheme for confidentiality of messages between the parties in hypercube networks by encrypting the aggregations via public keys and later decrypting the encryptions via private keys.
- (R3) Trustless: It uses zero-knowledge proof to verify off-chain computations of aggregators on hypercube networks to complete their aggregations without trusted third party.

- (R4) Public Verifiability: It publicly verifies the correctness of the zero-knowledge proofs on-chain with *verifyTx* function of the proof-verifying contract on blockchain.
- (R5) Authentication: It requires every aggregator to register to be able to involve the privacy-preserving data aggregation where it uses this registration information for authentication later.
- (R6) Non-interaction: The zero-knowledge proof scheme it uses is based on a non-interactive zero-knowledge proof protocol (i.e. zkSNARKs [16]). However, our protocol still needs a certain degree of interaction (see Section 5.5.1) to have the final aggregation.
- (R7) Scalability: The hypercube topology (as a logical layout scheme) over the blockchain platform (as a physical layout scheme) allows the increasing number of parties to be involved with the protocol (with a logarithmic-scale overhead). Review Section 5.5.1 for more information about the protocol overheads.
- (R8) Correctness: It needs all components to guarantee the protocol correctness except the hypercube networks and the web interface. Without these two, parties can still perform aggregation with another network by calculating complex cryptographic operations on their own.
- (R9) Usability: It has a web interface that abstracts all these complexities into simple actions to improve user-friendliness of the system.

Table 5.5 briefly summarizes how our protocol satisfies these requirements with the components it has.

5.6.2.2. Blockchain Gas Consumption. In this experiment, we measure the blockchain gas consumption of the smart contract functions where we use Avalanche Fuji [3] with 4 aggregators and Geth [104] with a varying number of aggregators, (2, 8, 32 and 128 aggregators). We present the experimental results in Table 5.6 in terms of gas units, gas cost in Avax/Ether and gas cost in USD. At the time we perform the measurements (i.e. 30/12/2022), the base gas price per gas unit is 25 nAvax and the exchange rate from Avax to USD is $22.8. Note that smart contract functions are called multiple times during the privacy-preserving aggregation where we present only their averages in

Table 5.5. *PVSS* verification of problem requirements.

Requirement	Blockchain	Smart Contract	Commitment Scheme	Public-Key Encryption	Zero-Knowledge Proof	Hypercube Networks	Web Interface
R1. Privacy			✓		✓		
R2. Confidentiality				✓			
R3. Trustless			✓	✓	✓	✓	
R4. Public Verifiability	✓	✓			✓		
R5. Authentication		✓					
R6. Non-interaction					✓		
R7. Scalability	✓					✓	
R8. Correctness	✓	✓	✓	✓	✓		
R9. Usability							✓

the table. Accordingly, the higher number of aggregators leads to more transactions for a certain function to be accumulated. The maximum number of aggregators is 128 in our experiments where it takes nearly 24 hours. However, the actual time *PVSS* needs is expected to be considerably less in the real-world scenarios because the computations are distributed over all the aggregators.

According to the given table, the most expensive function is *deploying contract* with over >3.3 million gas units, which corresponds to more than $2.12. The reason for this cost is it includes the costs of the main *aggregating* contract along with one proof-verifying contract with complex verification operations. Luckily, it is the developer that covers up this cost just for once to deploy an aggregation instance. The second most expensive is *verifying aggregation* with over >2.0 million gas units, which corresponds to $1.25. This cost results from the external function call from the main contract to the proof-verifying contract to verify the given proof. On the other hand, the least expensive function is *registering* with approximately >0.25 million gas units since it

updates only certain mapping entries. An aggregator has to pay \$0.17 to register once and $\$1.73 = \$1.25 + \$0.47$ to perform a single aggregation where the number of aggregations logarithmically changes with respect to the total number of aggregators. For instance, each aggregator needs to pay $\$7.09 = \$0.17 + \$1.73 \cdot log(16)$. In addition, we expect gas consumption to be platform-independent as long as it is EVM-based while different EVM versions may change it over time. This is also observed from our table where the consumption across Avalanche and Ethereum is negligible.

With the increasing number of parties in Geth, although the gas consumption of the contract deployments is unaffected, there exists a noticeable decrease in the gas consumption of the user registration. It has been observed that the initial transactions for that function tend to use more gas units while the following transactions neutralize this effect due to the averaging operation. The more transactions there are with the increasing number of parties involved, the more neutralized it eventually becomes. The similar phenomenon is also observed for the proof verification function where the certain transactions at the end of each hypercube dimension use more gas units (Line 10 of Figure 5.8). Assuming N parties with $log(N)$ dimensions in total, there are $log(N)$ transactions with extra gas units (γ^+) and $N - log(N)$ transactions with normal gas units (γ). Based on the fact $\gamma < \gamma^+$, the average of these transactions is:

$$\frac{log(N) \cdot \gamma^+ + (N - log(N)) \cdot \gamma}{N} \tag{5.72}$$

where we can make further simplification to obtain the following:

$$\gamma + \frac{log(N)}{N} \cdot (\gamma^+ - \gamma) \tag{5.73}$$

where note that $(\gamma^+ - \gamma)$ is always positive and when N approaches infinity (i.e. there exists an infinite number of parties involving with the protocol) as:

$$\lim_{N \to \infty} \left[\gamma + \frac{log(N)}{N} \cdot (\gamma^+ - \gamma) \right] = \gamma \tag{5.74}$$

it converges into the normal blockchain gas consumption γ.

We can also generalize the total gas consumption of the protocol from the perspective of each party γ_{party} by integrating the gas consumption of its individual functions:

$$\gamma_{party} = \gamma_{fixed} + log(N) \cdot \gamma_{dynamic} \tag{5.75}$$

$$\gamma_{fixed} = \gamma_{registration} \tag{5.76}$$

$$\gamma_{dynamic} = \gamma_{submission} + \gamma_{verification} \tag{5.77}$$

where the registration cost $\gamma_{registration}$ constitutes the constant-time fixed cost γ_{fixed} while the data submission $\gamma_{submission}$ and verification $\gamma_{verification}$ costs constitute the logarithmic-time dynamic cost $\gamma_{dynamic}$. From the perspective of the system γ_{system} as:

$$\gamma_{system} = N \cdot \gamma_{fixed} + 2 \cdot N \cdot log(N) \cdot \gamma_{dynamic} \tag{5.78}$$

where we just sum the costs of all parties, $\gamma_{system} = N \cdot \gamma_{party}$.

On the other hand, the experimental study for three communication techniques on incomplete hypercubes (i.e. *node multiplexing*, *topological recursing* and *data splitting*) is among our future works. Nevertheless, we can theoretically say that when we consider the the maximum gas consumption of the *PVSS* protocol as less than 4M gas units (where the current gas limit in Ethereum is 30M), their possible consumptions are expected to be likely under that limit.

Table 5.6. Blockchain gas consumption of the *PVSS* protocol phases.

Function	**Gas Units (Fuji 4)**	**Gas Units (Geth 8)**	**Gas Units (Geth 32)**	**Gas Units (Geth 128)**	**Gas Cost (Avax)**	**Gas Cost (USD)**
Deploying Contract	3,372,418	3,412,242	3,412,242	3,412,242	0.0927415	2.12 $
Registering	253,118	259,025	257,014	256,360	0.0075556	0.17 $
Submitting Aggregation	763,692	763,086	763,401	763,537	0.0205294	0.47 $
Verifying Aggregation	2,030,995	1,998,132	1,996,382	1,996,063	0.0547150	1.25 $

5.6.2.3. Proof Generation/Verification Times. The zero-knowledge proof generation times have been calculated through 20 independent runs. The individual measurements are presented in Table 5.7 where we can infer that the average proof generation time is approximately 88.3 seconds while the median time is 87.4 seconds. The maximum proof generation time is 95.8 seconds where it is for the first run because of the cold start effect. On the other hand, the minimum proof generation time is 87.1 seconds. As understood from the given statistics, the distribution of the proof generation times are quite uneven and positively skewed. Unlike proof generation, proof verification is performed on-chain in the contract without any time-based costs.

Table 5.7. *PVSS* zero-knowledge proof generation times (in seconds).

Run	**Verify Aggregation**
1	95.812
2	91.022
3	87.475
4	91.622
5	87.333
6	88.207
7	87.289
8	87.485
9	87.274
10	87.433
11	87.292
12	89.328
13	87.522
14	87.167
15	87.429
16	87.342
17	87.302
18	87.234
19	87.180
20	87.252

5.6.2.4. Proof Artifact Sizes. In this experiment, we measure the zero-knowledge proof artifacts including (i) the proving key size, (ii) the verification key size and (iii) the proof size. For our theoretical expectation in this experiment, refer to Section 4.5.3. We present the experimental results in Table 5.8 where we observe that the proving key is nearly ∼30,000x larger than the verification key while the proof size is just 1KB. This clearly shows that the zkSNARKs system [16] that we use can successfully generate succinct proofs. With comparison to the proving key and verification key size in Section 4.5.3, we observe that the proving key for the *aggregating data* proof is larger than the proving keys for the *depositing tokens* and *withdrawing tokens* proof since *aggregating data* is computationally more complex while the verification keys and proof sizes remains the same.

Table 5.8. *PVSS* zero-knowledge proof artifact size.

Proof	**Proving Key Size**	**Verification Key Size**	**Proof Size**
Aggregating Data	60.5MB	2KB	1KB

5.6.3. For Privacy-Preserving Prefix Aggregation

5.6.3.1. Blockchain Gas Consumption. In this experiment, we measure the blockchain gas consumption of our protocol on the *Ethereum Sepolia* blockchain and present the results in Table 5.9. The results are given in terms of the gas units, the gas cost in Ether and the gas cost in USD. At the time we perform the measurements (i.e. 25/11/2023), the base gas price per gas unit is 1.5 Gwei and the exchange rate from Ether to USD is $2074,65. We clearly see that the most expensive function is *contract deployment* itself with +2M gas units since the main *aggregating prefix* contract includes another proof-verifying contract as well. The second most expensive function is *verifying prefix* with +1.4M gas units where it makes an external call to the proof-verifying contract to verify the proof on-chain. This shows that the computations for zero-knowledge proof are the major parameter for blockchain gas consumption. On the other hand, the least expensive function is *registering* with +250K gas units where it just updates certain mapping entries over the contract.

Table 5.9. Blockchain gas consumption of *PRFX* protocol phases.

Function	Gas Units	Gas Cost (Sepolia)	Gas Cost (USD)
Deploying Contract	2,099,791	0.00554550	11.50
Registering	251,692	0.00041469	0.86
Submitting Prefix	393,707	0.00063854	1.32
Verifying Prefix	1,486,103	0.00236184	4.90

From the perspective of each aggregator, the total gas consumption of *PRFX* can be generalized with the same way for *PVSS* as follows:

$$\gamma_{party} = \gamma_{fixed} + log(N) \cdot \gamma_{dynamic} \tag{5.79}$$

$$\gamma_{fixed} = \gamma_{registration} \tag{5.80}$$

$$\gamma_{dynamic} = \gamma_{submission} + \gamma_{verification} \tag{5.81}$$

where the sum of all costs for all the aggregators results in the total gas consumption for the system as $\gamma_{system} = N \cdot \gamma_{party}$. We omit further explanations here to avoid falling into repetition.

5.6.3.2. Proof Generation/Verification Times. In this experiment, we measure the zero-knowledge proof generation and verification times for the *aggregating prefix data* proof (see Figure 5.26) where we perform 20 independent runs and take their average as the final results. The individual measurements are presented in Table 5.10 where we can infer that the minimum and maximum proof generation times are 76.8 and 84.2 seconds while the mean and median proof generation times are 80.6 and 80.2 seconds, respectively. On the other hand, zero-knowledge proof verification which is performed on-chain, is time-independent but gas-dependent.

Table 5.10. *PRFX* zero-knowledge proof generation times (in seconds).

Run	Verify Aggregation
1	79.946
2	81.809
3	79.310
4	84.234
5	78.710
6	81.152
7	81.459
8	81.740
9	74.351
10	76.814
11	81.072
12	83.584
13	79.906
14	80.063
15	83.247
16	77.978
17	80.276
18	79.732
19	79.264
20	73.518

5.6.3.3. Proof Artifact Sizes. In this experiment, we measure the zero-knowledge proof artifacts including (i) the proving key size, (ii) the verification key size and (iii) the proof size. For our theoretical expectation in this experiment, refer to Section 4.5.3. We present the experimental results in Table 5.11 where we observe that the proving key is nearly ∼60,000x larger than the verification key while the proof size is just 1KB. In comparison with the *aggregating data* proof in Section 5.6.2, we see that the proving key for the *aggregating prefix* proof is much larger since the proving key varies with respect to the proof complexity, (see Figure 5.5 and Figure 5.26). In addition, the verification key for the *aggregating prefix* proof is also larger (i.e. 3KB against 2KB) since the verification key includes the public parameters where more number of public

parameters results in this increase, (see Figure 5.5 and Figure 5.26). On the other hand, the proof size always remains constant across all our protocols.

Table 5.11. *PRFX* zero-knowledge proof artifact size.

Proof	**Proving Key Size**	**Verification Key Size**	**Proof Size**
Aggregating Prefix	120.5MB	3KB	1KB

All these experiments justify the validity and applicability of our aggregation protocols. Note that these protocols are designed with easily-replaceable components with their counterparts. For instance, since prefix computation can implemented as a normal hypercube algorithm, the hypercube topology can be replaced with another topology (e.g. shuffle-exchange topology and De Bruijn topology [105]) if necessary without affecting the other components of the protocols. The zkSNARKs proof system [16] can be also replaced as long as the new proof system satisfies certain conditions including non-interaction, off-chain proof generation, on-chain proof verification and compatibility to EVM. This makes our protocols more resilient with respect to the blocks they are built upon. Nevertheless, performance of the protocols after such changes (e.g. in terms of proof generation times) may naturally need further experimental elaboration.

6. PROTOCOL III: PRIVACY-PRESERVING TOKEN BARTERING

- quis quid cum quo commutat?
(who exchanges what with whom?)

In this chapter, we define the privacy-preserving multi-token bartering problem on blockchain by specifying objectives, constraints and requirements. For the given problem, we propose our privacy-preserving multi-token bartering protocol (i.e. *PMTBS*) that integrates commitment scheme, public-key encryption, zero-knowledge proof and hypercube networks under three main models as the zero-knowledge proof model, the smart contract model and the web interface model. We analyze the protocol with respect to the scalability (with computational, communication and storage overheads) and security perspectives (with potential attacks and formal reduction proofs). We also perform experiments for *PMTBS* to measure blockchain gas consumption, zero-knowledge proof generation/verification times and zero-knowledge proof artifact sizes.

6.1. Problem Definition

Privacy-preserving multi-token bartering-based trading refers to a secure multi-party computation where a group of blockchain addresses (i.e. barterers) collectively exchanges a set of tokens in return for another set of tokens through proposing bids by still protecting the privacy of their balances and bids. This problem (as in the *PTTS* and *PVSS* protocols) has two kinds of information asymmetry: (i) each address possesses their own private balance while the other addresses need it to verify the correctness of the bartering and (i) each address knows only their own bid while the other barterers not having this bid need it to exchange tokens. For the scope of this problem, let $\mathcal{U}$ be a set of blockchain addresses as follows:

$$\mathcal{U} = \{u_0, u_1, \ldots, u_{N-1}\} \tag{6.1}$$

where u_i is the ith barterer while N is the maximum number of users and let $\mathcal{T}$ be a set of token types as follows:

$$\mathcal{T} = \{\tau_0, \tau_1, ..., \tau_j, ..., \tau_{M-1}\} \tag{6.2}$$

where τ_j is the jth token type while M is the maximum number of token types. Each barterer is associated with:

$$u_i : \langle \theta_i, \phi_i, pk_i, sk_i \rangle \tag{6.3}$$

where θ_i is the private composite balance, ϕ_i is the private bid, pk_i public key and sk_i is the secret key of the ith barterer. The composite balance refers to a M-sized vector including balances for each token type as:

$$\theta_i = \langle \theta_{i,0}, \theta_{i,1}, \ldots, \theta_{i,M-1} \rangle \tag{6.4}$$

where $\theta_{i,j}$ is the balance for jth token type of the ith barterer. We assume that all the values in such vectors must be specified from the set of only positive integers including zero as $\theta_{i,j} \in \mathbb{Z}^+ \cup \{0\}$, (i.e. no balance can be negative or real number). Similarly, the bid consists of two M-sized vectors as:

$$\phi_i : \langle \phi_i^-, \phi_i^+ \rangle \tag{6.5}$$

$$\phi_i^- : \langle \phi_{i,0}^-, \phi_{i,1}^-, \ldots, \phi_{i,M-1}^- \rangle \tag{6.6}$$

$$\phi_i^+ : \langle \phi_{i,0}^+, \phi_{i,1}^+, \ldots, \phi_{i,M-1}^+ \rangle \tag{6.7}$$

where ϕ_i^- refers to the former vector including the number of tokens per token types to be supplied (i.e. negative changes) while ϕ_i^+ refers to the latter vector including the number of tokens per token types to be eventually demanded (i.e. positive changes). We assume that each barterer can propose only one bid at a time.

In the scope of privacy-preserving multi-token bartering problem, we define a valid bartering with multiple barterers in the following way:

$$\Phi : \langle \phi_0, \phi_1, \ldots, \phi_i, \ldots, \phi_{N-1} \rangle \tag{6.8}$$

where we can simply infer that a valid bartering needs to satisfy all the bids, rather than selecting only a subset of bids under certain criteria. This constraint results in an additional pressure over the protocol to be proposed since the available bids do not necessarily form a valid bartering solution every time. More formally, a bartering solution is valid if and only if:

$$\sum_{i=0}^{N-1} \phi_{i,j}^{-} \geq \sum_{i=0}^{N-1} \phi_{i,j}^{+}, \quad \forall j \in \{0, 1, \ldots, M-1\} \tag{6.9}$$

where it implies that the total number of tokens supplied from all the bids must be at least equal to or greater than the total number of tokens demanded to all the bids per token type. Otherwise (e.g. tokens supplied are not enough to meet demands even for one token type), the problem counts the solution as infeasible. However, the problem relaxes this constraint by allowing barterers to replace their bids at certain intervals (i.e. reproposing bids) if they comply with the following constraints:

$$\phi_{i,j,t+1}^{-} \geq \phi_{i,j,t}^{-}, \quad \forall i, j \tag{6.10}$$

$$\phi_{i,j,t+1}^{+} \leq \phi_{i,j,t}^{+}, \quad \forall i, j \tag{6.11}$$

$$\theta_{i,j} \geq \phi_{i,j,t+1}^{-}, \quad \forall i, j \tag{6.12}$$

where the first constraint refers to the *over-supply* requirement where it requires the number of tokens supplied in the next bid $\phi_{i,j,t+1}^{-}$ must be greater than the number of tokens supplied in the current bid $\phi_{i,j,t}^{-}$ per token type, which results in more number of tokens to be eventually collected. The second constraint refers to the *under-demand* requirement where it requires the number of tokens demanded in the next bid $\phi_{i,j,t+1}^{+}$ must be less than the number of tokens demanded in the current bid $\phi_{i,j,t}^{+}$, which

results in less number of tokens to be eventually distributed. These two constraints define why our problem needs to be based on ascending auctions. Lastly, the third constraint refers to the *balance* requirement where it requires the balance of every barterer to be sufficient to deposit the tokens owned.

Note that we define a data aggregation in the scope of privacy-preserving data aggregation problem for any scalar data χ. Now, we define a bid aggregation in the scope of privacy-preserving multi-token bartering problem:

$$f^2([\phi_0, \phi_1]) = \phi_0 + \phi_1 = \sum_{i=0}^{1} \phi_i \tag{6.13}$$

where f^2 is the generic pair-wise bid aggregation function for two barterers. Similarly, we can define a global bid aggregation between multiple barterers in the following way:

$$f^N([\phi_0, \phi_1, \ldots, \phi_{N-1}]) = \phi_0 + \phi_1 + \cdots + \phi_{N-1} = \sum_{i=0}^{N-1} \phi_i \tag{6.14}$$

where f^N is the generic global data aggregation function for multiple aggregators, which simply collects and transforms individual data into a global aggregation (i.e. sum). If N complies to 2^d for any d, we can represent the function f^N by recursively using the function f^2 as:

$$f^N = f^2(f^2(\ldots, \ldots), f^2(f^2([\phi_{N-4}, \phi_{N-3}]), f^2([\phi_{N-2}, \phi_{N-1}]))) \tag{6.15}$$

where the inner f^2 functions aggregates and returns the result to the immediate outer f^2 functions until the final f^2 function returns the final aggregation.

The definitions so far comprise only a public multi-token bartering among multiple barterers without considering privacy-preserving characteristics of the problem that we have previously pledged. Now, we introduce three different privacy goals as the *balance privacy* where no address must see the balance of another address, the *bid*

privacy where no address must see the bid of another address and the *intermediate bid aggregation privacy* where no address must see the intermediate vector-based bid aggregation of another address. For the balance privacy:

$$\forall u_j \neq u_i : \Pr[\theta_i \mid \text{view}(u_j)] = \Pr[\theta_i] \tag{6.16}$$

where view(u_j) simply refers to the information that user u_j has access to. This statement implies that there is not any address u_j whose view on the balance of the address u_i is probabilistically equal to the view of the address u_i itself. Similarly for the bid privacy:

$$\forall u_j \neq u_i : \Pr[\phi_i \mid \text{view}(u_j)] = \Pr[\phi_i] \tag{6.17}$$

where we can infer from the statement that there isn't any address u_j whose view on the bid of the address u_i is probabilistically equal to the view of the address u_i itself. Lastly, for the intermediate bid aggregation privacy:

$$\forall u_k \notin \{u_i, u_j\} : \Pr[f^2 : \phi_i + \phi_j \mid \text{view}(u_k)] = \Pr[f^2 : \phi_i + \phi_j] \tag{6.18}$$

where it implies that there isn't any address u_k whose view on the intermediate pair-wise bid aggregation $f^2 : \phi_i + \phi_j$ the address u_i has is probabilistically equal to the views of the address u_i itself. From this perspective, we can now define three main objectives of our problem (i.e. maximizing these privacy goals) as follows:

$$\min_{u_j} \quad \mathtt{I}(\text{view}(u_j); \theta_i), \quad \forall u_j \neq \{u_i\} \tag{6.19}$$

$$\min_{u_j} \quad \mathtt{I}(\text{view}(u_j); \phi_i), \quad \forall u_j \neq \{u_i\} \tag{6.20}$$

$$\min_{u_k} \quad \mathtt{I}(\text{view}(u_k); f^2 : \phi_i + \phi_j), \quad \forall u_k \neq \{u_i, u_j\} \tag{6.21}$$

where $\mathtt{I}(.)$ is the mutual information measurement function which simply quantifies the amount of information of a certain address on a certain value in our scope. The statement given in Equation (6.19) implies that the first goal of the problem is to

minimize the amount of information the address u_j accesses to over the balance θ_i that the address u_i has, as complied to the balance privacy defined in Equation (6.16). Similarly, the statement in Equation (6.20) implies that the second goal of the problem is to minimize the amount of information the address u_j accesses to over the bid θ_i that the address u_i proposes, as complied to the balance privacy defined in Equation (6.17). Lastly, the statement in Equation (6.20) implies that the third goal of the problem is to minimize the amount of information the address u_k accesses to over the intermediate pair-wise bid aggregation $f^2 : \phi_i + \phi_j$ the address u_i has, as complied to the transaction privacy defined in Equation (6.18). In the most ideal scenario, we expect the mutual information for all these objectives to be zero where this means that no information about θ_i, ϕ_i and $f^2 : \phi_i + \phi_j$ leaks. Note how the information asymmetries in the beginning is transformed into the problem objectives later.

Privacy-preservation is the most significant requirement that our problem requires to be satisfied. However, there exist several other requirements as well as follows:

- Privacy: it must protect the privacy of bids and multi-token balances, (i.e. each barterer must know only their own bid and balance).
- Confidentiality: it must ensure the secrecy of message encryptions on direct communication channels between parties.
- Trustless: barterers must complete their token exchanges without relying on any trusted third party.
- Public Verifiability: it must allow the correctness of the computations over the private data (i.e. bids and balances) to be publicly verified.
- Authentication: it must prevent parties to involve with protocol if they are not known beforehand.
- Non-Interactivity: it must allow all parties to generate zero-knowledge proofs on their own without interaction.
- Scalability: it must support a growing number of parties with a tolerable degree of computational, communication and storage overheads.
- Correctness: it must function correctly and properly at any given time.

- Usability: it must encapsulate the mathematical complexity to provide a more user-friendly protocol.

Overall, our driving motivation for this problem can be stated as *how to develop a privacy-preserving but publicly-verifiable multi-token bartering protocol over a decentralized network (i.e. blockchain) through a certain cryptographic primitives (i.e. commitments, public-key encryptions, zero-knowledge proof and hypercube networks) without any assistance of trusted third parties.*

We also present a numeric example for ascending auction-based bartering in Figure 6.1 with four different bids and four different tokens where there are three bidding rounds as initial bid proposing, the first bid reproposing and the second bid reproposing. In the initial iteration, the aggregations of tokens supplied ($[5, 2, 7, 3]$) is not component-wise equal to or greater than the aggregation of tokens demanded ($[2, 7, 4, 5]$) since $2 \ngeq 7$. The similar situation is available in the first iteration as well although the users, by following the requirements of the problem, increase the amounts of tokens supplied (i.e. the first bid increases supplies from ($[1, 2, 0, 1]$) to ($[1, 4, 0, 1]$)) and decrease the amounts of tokens demanded (i.e. the third bid decreases demands from ($[0, 2, 0, 3]$) to ($[0, 1, 0, 3]$)). The problem is eventually resolved in the second rebidding where the aggregation of tokens supplied ($[5, 5, 7, 4]$) is now component-wise equal to or greater than the aggregation of tokens demanded ($[2, 5, 4, 4]$). At that point, a feasible bartering solution is considered to be found and the barterers start bartering.

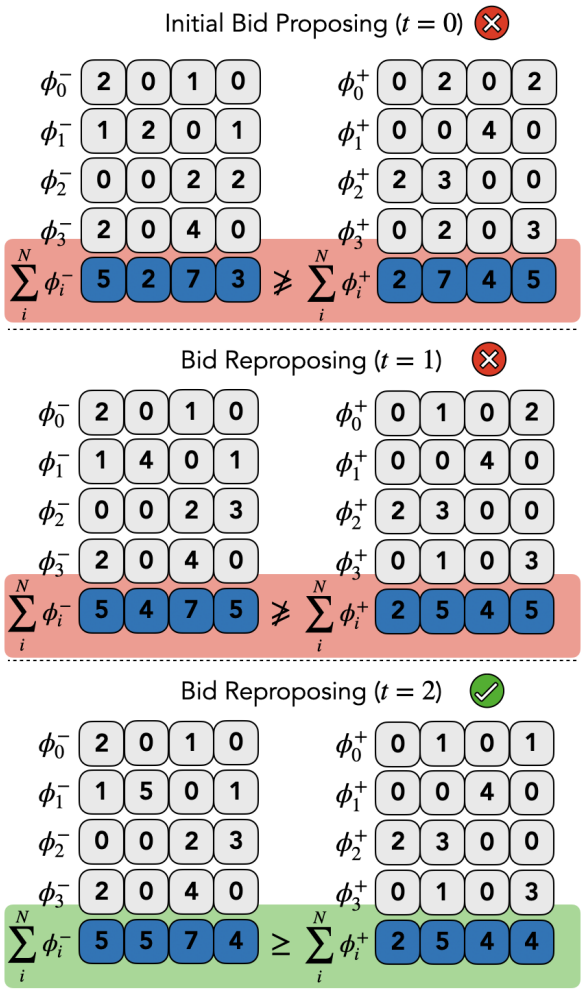


Figure 6.1. Illustration for ascending auction bartering (with four bids $(\phi_0, \phi_1, \phi_2, \phi_3)$ and four tokens $(\tau_0, \tau_1, \tau_2, \tau_3)$).

The bartering problem in Figure 6.1 can be represented through the hypercube networks as well in Figure 6.2. In $t = 0$, the users u_0 and u_1; and u_2 and u_3 aggregate their bid aggregations at the first step:

$$\begin{aligned}
\Sigma_1 &= \phi_0 + \epsilon_0 + \phi_1 + \epsilon_1 = \langle\langle 3, 2, 1, 1\rangle, \langle 0, 2, 4, 2\rangle\rangle + \epsilon_0 + \epsilon_1 \\
\Sigma_2 &= \phi_2 + \epsilon_2 + \phi_3 + \epsilon_3 = \langle\langle 2, 0, 6, 2\rangle, \langle 2, 5, 0, 3\rangle\rangle + \epsilon_2 + \epsilon_3 \\
\Sigma_1' &= \epsilon_0 + \epsilon_1 \\
\Sigma_2' &= \epsilon_2 + \epsilon_3
\end{aligned} \tag{6.22}$$

where Σ_1 and Σ_2 are the aggregations of the first network while Σ_1' and Σ_2' are the aggregations of the second network. Later, the users u_0 and u_2; and u_1 and u_3 starts aggregating their bid aggregations at the second and final step of two networks as:

$$\begin{aligned} \Sigma_3 &= \Sigma_1 + \Sigma_2 = \langle\langle 5,2,7,3\rangle, \langle 2,7,4,5\rangle\rangle + \epsilon_0 + \epsilon_1 + \epsilon_2 + \epsilon_3 \\ \Sigma_3' &= \epsilon_0 + \epsilon_1 + \epsilon_2 + \epsilon_3 \end{aligned} \tag{6.23}$$

where Σ_3 and Σ_3' are the global aggregations of the first and second networks, respectively. When we eliminate Σ_3' from Σ_3, we get :

$$\Sigma_3 - \Sigma_3' = \langle\langle 5,2,7,3\rangle, \langle 2,7,4,5\rangle\rangle \tag{6.24}$$

where we see that $2 \not\geqslant 7$ for the first token type. Therefore, barterers changes their bids as in $t = 1$ of Figure 6.1 and start aggregating again as follows:

$$\begin{aligned} \Sigma_1 &= \phi_0 + \epsilon_0 + \phi_1 + \epsilon_1 = \langle\langle 3,4,1,1\rangle, \langle 0,1,4,2\rangle\rangle + \epsilon_0 + \epsilon_1 \\ \Sigma_2 &= \phi_2 + \epsilon_2 + \phi_3 + \epsilon_3 = \langle\langle 2,0,6,3\rangle, \langle 2,4,0,3\rangle\rangle + \epsilon_2 + \epsilon_3 \\ \Sigma_1' &= \epsilon_0 + \epsilon_1 \\ \Sigma_2' &= \epsilon_2 + \epsilon_3 \\ \Sigma_3 &= \Sigma_1 + \Sigma_2 = \langle\langle 5,4,7,4\rangle, \langle 2,5,4,5\rangle\rangle + \epsilon_0 + \epsilon_1 + \epsilon_2 + \epsilon_3 \\ \Sigma_3' &= \epsilon_0 + \epsilon_1 + \epsilon_2 + \epsilon_3. \end{aligned} \tag{6.25}$$

We eliminate Σ_3' from Σ_3 as follows:

$$\Sigma_3 - \Sigma_3' = \langle\langle 5,4,7,4\rangle, \langle 2,5,4,5\rangle\rangle \tag{6.26}$$

where we see that $4 \not\geqslant 5$ for the first token type again. Therefore, barterers changes their bids last time as in $t = 2$ of Figure 6.1 and start aggregating again as follows:

$$\Sigma_1 = \phi_0 + \epsilon_0 + \phi_1 + \epsilon_1 = \langle\langle 3,5,1,1\rangle, \langle 0,1,4,1\rangle\rangle + \epsilon_0 + \epsilon_1$$

$$
\begin{aligned}
\Sigma_2 &= \phi_2 + \epsilon_2 + \phi_3 + \epsilon_3 = \langle\langle 2,0,6,3\rangle, \langle 2,4,0,3\rangle\rangle + \epsilon_2 + \epsilon_3 \\
\Sigma_1' &= \epsilon_0 + \epsilon_1 \\
\Sigma_2' &= \epsilon_2 + \epsilon_3 \\
\Sigma_3 &= \Sigma_1 + \Sigma_2 = \langle\langle 5,5,7,4\rangle, \langle 2,5,4,4\rangle\rangle + \epsilon_0 + \epsilon_1 + \epsilon_2 + \epsilon_3 \\
\Sigma_3' &= \epsilon_0 + \epsilon_1 + \epsilon_2 + \epsilon_3. \qquad (6.27)
\end{aligned}
$$

We eliminate Σ_3' from Σ_3 again as follows:

$$
\Sigma_3 - \Sigma_3' = \langle\langle 5,5,7,4\rangle, \langle 2,5,4,4\rangle\rangle \qquad (6.28)
$$

where we now see that the supply is greater than (or at least equal to) the demand for every token type. This series of computations shows all the intermediary bid aggregations clearly.

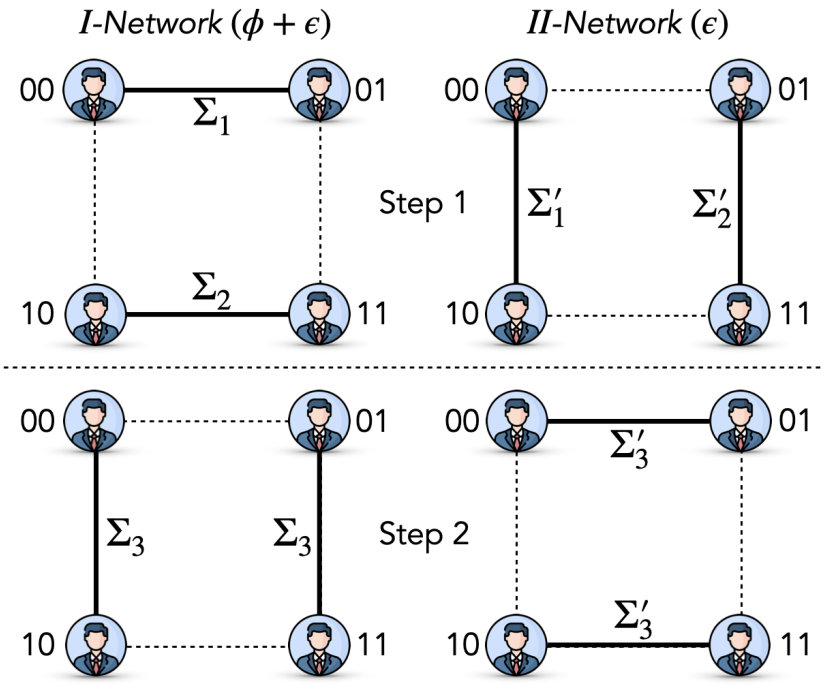


Figure 6.2. Bid aggregations of *PMTBS* protocol in 2-dimensional hypercube networks.

6.2. System Architecture

For the given problem in Section 6.1, we develop a privacy-preserving multi-token bartering protocol on blockchain with zero-knowledge proof by integrating *PVSS* as sub-protocol. Our protocol is (i) *privacy-preserving* since it hides the balance and bid information of barterers, (ii) *ascending auction-based* since barterers are expected to supply more number of tokens and to demand less number of tokens at each bidding round, (iii) *multi-token* since multiple tokens can be exchanged per auction, (iv) *trust-less* since barterers can complete their token exchanges without relying on any trusted third party, (v) *publicly-verifiable* since the validators can still verify the correctness of the bartering although they could not learn who exchanges which tokens with whom, (vi) partially *non-interactive* since the protocol does not enforce any interaction to generate proofs (except the interactions for bid aggregations in *PVSS*) and finally (vii) *scalable* since the protocol overhead increases in logarithmic-scale with the increasing number of barterers. The protocol is built upon two main actors as (i) *developer* who performs one-time setup and deploys the smart contracts on blockchain and (ii) *barterer* who has private balance and bid information with the intention of privately exchanging certain tokens in return for other tokens.

Our *PMTBS* protocol has six main phases as: (i) *deploying contracts* where developer deploys the main bartering and multi-token contracts in addition to five proof-verifying contracts (for multi-token depositing, multi-token withdrawing, (re)proposing bid, aggregating bids and bartering tokens), (ii) *proposing bid* where barterers propose their initial bids by registering to a certain bartering session, (iii) *submitting bid aggregation* where barterers submits encryptions of their bids with the help of two separate hypercube networks (as in *PVSS*), (iv) *verifying bid aggregation* where barterers generate proofs (per network) to show the correctness of their own bid aggregation computations (as in *PVSS*), (v) *(re)proposing bid* where barterers repropose their bids in the next bidding round if no feasible bartering solution is formed in the current bidding round and (vi) *bartering tokens* where barterers exchange the tokens they supply in return for the tokens they demand with respect to the bartering solution formed. The

protocol employs seven contracts at total as: (i) main multi-token contract to support depositing and withdrawing multiple tokens at once (as an extension to *PTTS*), (ii) main bartering contract to track hypercube networks, to orchestrate the interactions among barterers and to store balance and bid commitments, (iii) five proof-verifying contracts to verify the proofs for depositing tokens, withdrawing tokens, (re)proposing bid, aggregating bids and finally bartering tokens. The state transitions for these phases are shown in Figure 6.3 as well. In addition, the architecture of the *PTMBS* protocol is also depicted in Figure 6.4. Note that we integrate *PVSS* as an underlying protocol into *PMTBS* for bid aggregation.

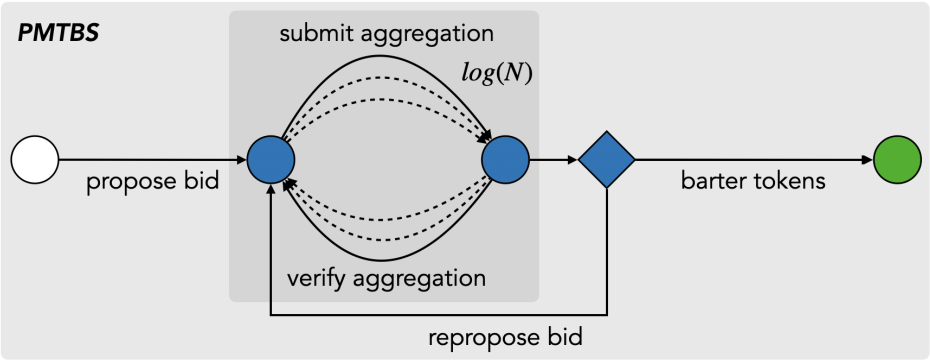


Figure 6.3. State-transition diagram of *PMTBS* protocol.

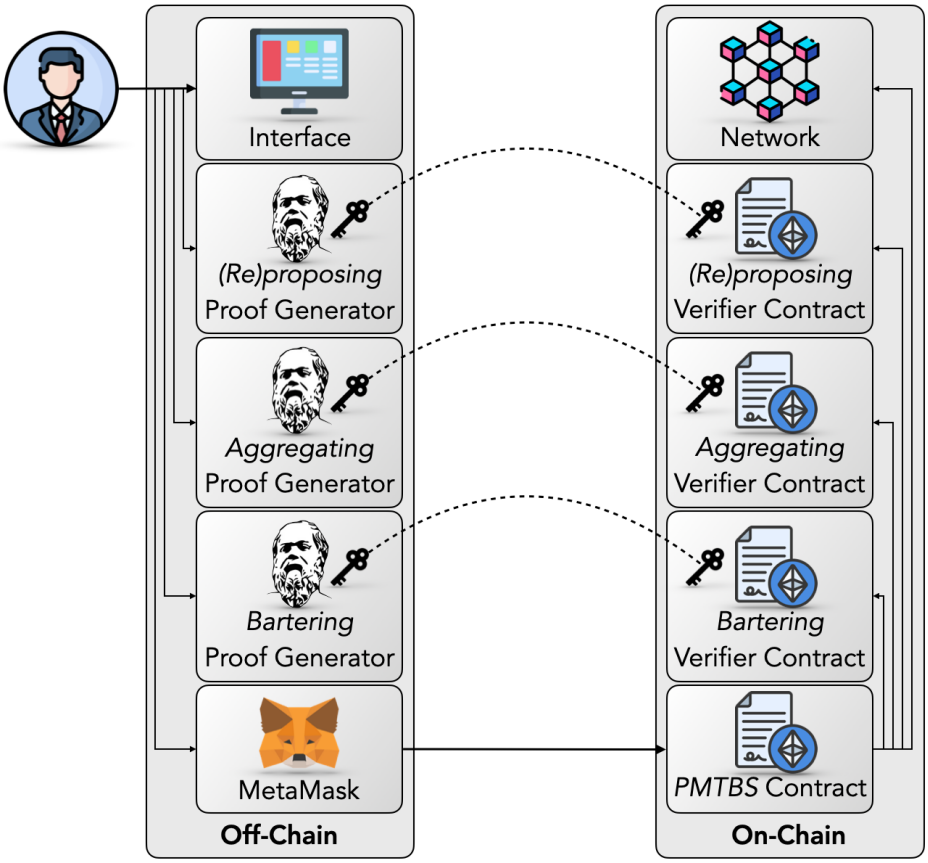


Figure 6.4. Architecture of *PMTBS* protocol.

Deploying Contracts. Developer performs a one-time setup to generate proving keys (for off-chain proof generation) and verification keys (for on-chain proof verification). In this scope of bartering, developer needs to generate three key pair instances for (re)proposing bid, aggregating bids and bartering tokens (and two more key pair instances for transferring as well) as follows:

$$(pok, vek) \leftarrow \texttt{Zk.Setup}(\lambda) \tag{6.29}$$

where *pok* denotes proving key while *vek* denotes verification key. The proof-verifying contracts also include these verification keys on-chain while the proving keys can be locally stored off-chain.

Proposing Bid. In this phase, we assume that the multi-token contract already

has the balance commitments of barterers with their corresponding salting parameters:

$$c_i^{\theta} \leftarrow \texttt{Cm.Comm}(\theta_i) \tag{6.30}$$

where θ_i represents the balance while c_i^{θ} the resulting balance commitment of the commitment function `Cm.Comm`. Barterer needs to generate a series of random numbers (per token type) in order to obscure their bid as:

$$\Sigma_i^{I} \leftarrow \phi_i + \epsilon_i \tag{6.31}$$

$$\Sigma_i^{II} \leftarrow \epsilon_i \tag{6.32}$$

where ϕ_i and ϵ_i are the private bid and random number, respectively while Σ_i^{I} and Σ_i^{II} are the bid aggregation instances. Later, the commitments of these bid aggregations are stored on-chain:

$$c_i^{\phi} \leftarrow \texttt{Cm.Comm}(\phi_i) \tag{6.33}$$

$$c_i^{I} \leftarrow \texttt{Cm.Comm}(\Sigma_i^{I}) \tag{6.34}$$

$$c_i^{II} \leftarrow \texttt{Cm.Comm}(\Sigma_i^{II}) \tag{6.35}$$

where c_i^{I} and c_i^{II} are the resulting commitment values of the commitment function `Cm.Comm` based on the bid aggregations Σ_i^{I} and Σ_i^{II}. As in *PVSS*, these two commitments are for two hypercube networks (i.e. I-Network and II-Network). Barterer also need to generate their own keys to be used in the *submitting bid aggregation* phase as:

$$(pk_i, sk_i) \leftarrow \texttt{Pk.Setup}(\lambda) \tag{6.36}$$

where pk_i is the public key while sk_i is the secret key of the public-key setup function `Pk.Setup` based on a security parameter λ. To propose bid, barterer has to generate a

zero-knowledge proof (per hypercube network):

$$\Psi^{propose} \leftarrow \theta_i - \phi_i^- \tag{6.37}$$

$$\pi_i \leftarrow \texttt{Zk.Gen}(\Psi^{propose}, \theta_i, \phi_i, c_i^\theta, c_i^\phi, c_i^I, c_i^{II}, pok) \tag{6.38}$$

where $\Psi^{propose}$ is the function to which the barterer is subjected for proposing a bid. This function directly deducts the supplied tokens from the barterer balance. While π_i is the resulting proof of the proof generation function `Zk.Gen` based on the function $\Psi^{propose}$, the balance θ_i, the bid ϕ_i and their commitments (c_i^θ, c_i^θ, c_i^I and c_i^{II}) for I-Network and II-Network. Barterer later submits this proof for verification:

$$b_i \leftarrow \texttt{Zk.Vfy}(\pi_i^{propose}, c_i^\theta, c_i^\phi, c_i^I, c_i^{II}, vek) \tag{6.39}$$

$$c_i^\theta \leftarrow \texttt{Cm.Comm}(\theta_i \leftarrow \theta_i - \phi_i^-) \tag{6.40}$$

where b_i is the boolean results of the proof verification function `Zk.Vfy` based on the same function and the commitments ($c^\theta, c_i^\phi, c_i^I$ and c_i^{II}). If the proof is correctly verified ($b_i \leftarrow true$), the contract sets the initial bid and updates the current balance c_i^θ.

Submitting Bid Aggregation. Similar to the *submitting aggregation* phase of *PVSS*, this phase pairs barterer with two other peer barterers in two hypercube networks for every hypercube communication step h:

$$u_p^I \leftarrow u \oplus 2^h \tag{6.41}$$

$$u_p^{II} \leftarrow u \oplus 2^{(log(N)-h-1)} \tag{6.42}$$

where u_p^I and u_p^{II} are these peers in I-Network and II-Network. For the same reason in *PVSS*, the peer barterers must be necessarily different in the first hypercube communication step. Barterer encrypts the first bid aggregation Σ_i^I through the public key of the first peer (in I-Network) and the second bid aggregation Σ_i^{II} through the public

key of the second peer (in II-Network):

$$E_i^I \leftarrow \texttt{Pk.Enc}([\Sigma_i^I], pk_p^I) \tag{6.43}$$

$$E_i^{II} \leftarrow \texttt{Pk.Enc}([\Sigma_i^{II}], pk_p^{II}) \tag{6.44}$$

where E_i^I and E_i^{II} are resulting encryptions of the public-key encryption function `Pk.Enc` while pk_p^I and pk_p^{II} are peer public keys. Later, barterer decrypts that encryption coming from their peers with their own private key sk_i:

$$\Sigma_p^I \leftarrow D^I \leftarrow \texttt{Pk.Dec}([E_p^I], sk_i) \tag{6.45}$$

$$\Sigma_p^{II} \leftarrow D^{II} \leftarrow \texttt{Pk.Dec}([E_p^{II}], sk_i) \tag{6.46}$$

where D^I and D^{II} are resulting decryptions of the public-key decryption function `Pk.Dec`. At that point, barterer (and all the other barterers, too) is ready to aggregate the vector-based bid aggregations (i.e. their own bid vector aggregations (Σ_i^I and Σ_i^{II}) and bid vector aggregations coming from the peers (Σ_p^I and Σ_p^{II}):

Verifying Bid Aggregation. Similar to the *verifying aggregation* phase of *PVSS*, this phase asks barterer to generate a zero-knowledge proof (per network) in order to ensure that the vector bid aggregations are correctly aggregated for every hypercube communication step h:

$$\Psi^I : \Sigma_i^I + \Sigma_p^I \tag{6.47}$$

$$\Psi^{II} : \Sigma_i^{II} + \Sigma_p^{II} \tag{6.48}$$

$$\pi_i^I \leftarrow \texttt{Zk.Gen}(\Psi^I, \Sigma_i^I, \Sigma_p^I, c_i^I, c_p^I, pok) \tag{6.49}$$

$$\pi_i^{II} \leftarrow \texttt{Zk.Gen}(\Psi^{II}, \Sigma_i^{II}, \Sigma_p^{II}, c_i^{II}, c_p^{II}, pok) \tag{6.50}$$

where Ψ^I and Ψ^{II} are the functions to which barterer is subjected for I-Network and II-Network. Moreover, π_i^I and π_i^{II} are the resulting proofs of the the proof generation function `Zk.Gen` based on these functions (Ψ^I and Ψ^{II}), the bid aggregations (Σ_i^I, Σ_p^I,

Σ_i^{II} and Σ_p^{II}) and their commitments (c_i^I, c_p^I, c_i^{II} and c_p^{II}), respectively for I-Network and II-Network. Barterer later submits these proofs to on-chain contract to verify:

$$b_i^I \leftarrow \texttt{Zk.Vfy}(\pi_i^I, c_i^I, c_p^I, vek) \tag{6.51}$$

$$b_i^{II} \leftarrow \texttt{Zk.Vfy}(\pi_i^{II}, c_i^{II}, c_p^{II}, vek) \tag{6.52}$$

where b_i^I and b_i^{II} are the boolean results of the proof verification function `Zk.Vfy` based on the same functions and only the commitments (c_i^I, c_i^{II}, c_p^I and c_p^{II}). In case the proofs are correctly verified ($b_i^I \leftarrow true$ and $b_i^{II} \leftarrow true$), the contract replaces the current vector bid aggregations with the next vector bid aggregations (i.e. bid aggregation of the bid aggregations):

$$c_i^I \leftarrow \texttt{Cm.Comm}(\Sigma_i^I \leftarrow \Sigma_i^I + \Sigma_p^I) \tag{6.53}$$

$$c_i^{II} \leftarrow \texttt{Cm.Comm}(\Sigma_{II} \leftarrow \Sigma_{II} + \Sigma_p^{II}) \tag{6.54}$$

where the contract stores these new bid aggregations c_i^I and c_i^{II}, respectively for I-Network and II-Network for the next communication step $h+1$. The third and fourth protocol phases (i.e. submitting and verifying bid aggregation) are repeated until all the communication steps are completed. After that final communication step, barterer (and all the other barterers) can compute the final vector-based bid aggregation by subtracting the result of the second hypercube network from the result of the first hypercube network. Note that the aggregation technique in *PMTBS* is the extended version of the aggregation technique of *PTTS* (i.e. extension from scalar-bid to vector-bid aggregation), which is depicted in Figure 6.5.

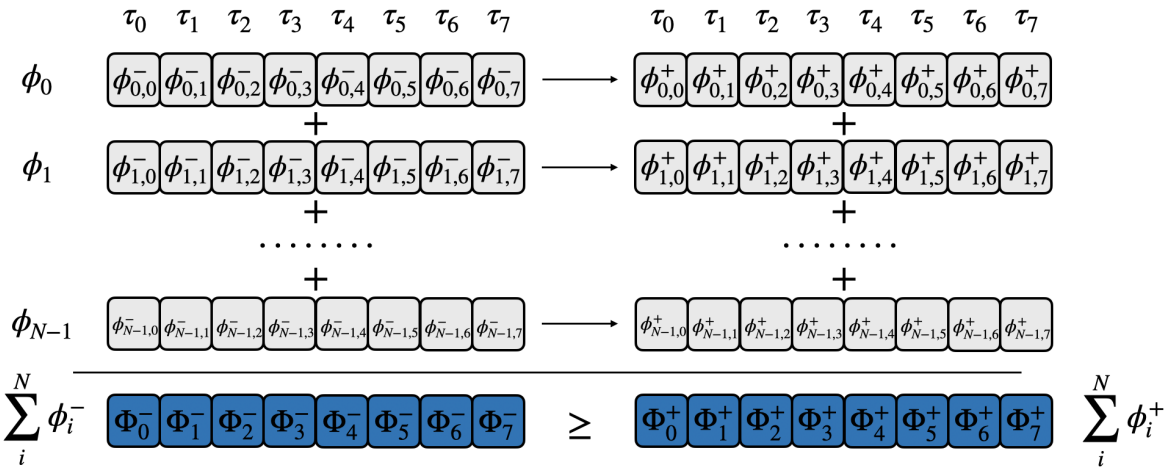


Figure 6.5. Bid aggregation with vector summation in *PMTBS* protocol.

Reproposing Bid. In this phase, the smart contracts store the commitments for the barterer balance and initial bids. In case no feasible bartering solution is found (see Equation (6.9)) with respect to the resulting bid aggregation (i.e. the number of tokens supplied is not greater than the number of tokens demanded for all token types), the protocol asks barterers to re-iterate the third and fourth phases with reproposed bids. In this phase, barterers are expected to supply more tokens (see Equation (6.10)) and demand less tokens (see Equation (6.11)) in their next bids:

$$\Psi^{repropose} \leftarrow \theta_i + \phi_i^- - \phi_i^{-\prime} \tag{6.55}$$

$$\pi_i \leftarrow \texttt{Zk.Gen}(\Psi^{repropose}, \theta_i, \phi_i, \phi_i', c_i^\theta, c_i^I, c_i^{II}, c_i^{I'}, c_i^{II'}, pok) \tag{6.56}$$

where $\Psi^{repropose}$ is the function to which the barterer is subjected for reproposing bids where the supplied tokens in their previous bid are returned back while the supplied tokens in their next bid are deducted now. Similar to the earlier proof generations, π_i is resulting proof of the proof generation function `Zk.Gen` based on that function, the current balance θ_i, the previous bid ϕ_i, the next bid ϕ_i' and their commitments ($c_i^\theta, c_i^I, c_i^{II}, c_i^{I'}$ and $c_i^{II'}$). Barterer submits that proof to the main bartering contract to verify:

$$b_i \leftarrow \texttt{Zk.Vfy}(\pi_i, c_i^\theta, c_i^I, c_i^{II}, c_i^{I'}, c_i^{II'}, vek) \tag{6.57}$$

$$c_i^{\theta} \leftarrow \texttt{Cm.Comm}(\theta_i \leftarrow \theta_i + \phi_i^{-} - \phi_i^{-'}) \tag{6.58}$$

where the proof verification function `Zk.Vfy` sets b to true ($b \leftarrow true$) if the proof is correct based on just the public commitments on-chain. If this is the case, the contract updates commitments of the current bid, current balance and current aggregations for two networks. The aggregation phases are repeated with the updated commitments.

Bartering Tokens. The protocol repeats the *reproposing bid* phase if: (i) there is no feasible bartering solution found in the previous *bidding* round or (ii) the maximum bidding round is not yet reached. On the other hand, barterer starts bartering tokens if a feasible solution is found with respect to Equation (6.9) with the following way:

$$\Psi^{barter} \leftarrow \theta_i + \phi_i^{+} \tag{6.59}$$

$$\pi_i \leftarrow \texttt{Zk.Gen}(\Psi^{barter}, \theta_i, \phi_i, c_i^{\theta}, c_i^{I}, c_i^{I}, pok) \tag{6.60}$$

where Ψ^{barter} is the function to which the barterer is subjected for bartering tokens where the tokens demanded are now given to their balance. The proof π_i is the result of the proof generation function `Zk.Gen` based on that function, the current balance θ_i, the current bid ϕ_i and their commitments (c_i^{θ}, c_i^{I} and c_i^{II}). Barterer (and all the other barterers) finally completes that bartering session after that proof is verified on-chain:

$$b_i \leftarrow \texttt{Zk.Vfy}(\pi_i, c_i^{\theta}, c_i^{I}, c_i^{II}, vek) \tag{6.61}$$

$$c_i^{\theta} \leftarrow \texttt{Cm.Comm}(\theta_i \leftarrow \theta_i + \phi_i^{+}) \tag{6.62}$$

where b_i is the result of the proof verification function `Zk.Vfy` based on just the public commitments. The function sets b_i to true ($b_i \leftarrow true$) if the proof is true and the corresponding contract function updates the barterer final balance with the next balance commitment. In case a barterer does not generate proof for that phase, the chance to get the tokens demanded is lost, which incentivizes participation. The summary of the *PMTBS* protocol is given in Table 6.1. The sequence diagram of the *PMTBS* protocol is depicted in Figure 6.6 as well by showing interactions between the actors.

Table 6.1. The *PMTBS* protocol

Protocol III: PMTBS
1. zkSNARKs Setup
(a) Developer performs one-time setup by generating proving and verification keys as
$(\texttt{pok, vek}) \leftarrow \texttt{Zk.Setup}(\lambda)$.
(b) All barterer balances are immutably stored on the contract as
$c_i^{\theta} \leftarrow \texttt{Cm.Comm}(\theta_i)$.
2. Bid Proposing of u_i
(a) Barterer u_i proposes bid ϕ_i by generating *proposing* proof
$c_i^{\phi} \leftarrow \texttt{Cm.Comm}(\phi_i)$
$c_i^{I} \leftarrow \texttt{Cm.Comm}(\phi_i + \epsilon_i)$
$c_i^{II} \leftarrow \texttt{Cm.Comm}(\epsilon_i)$
$\pi_i \leftarrow \texttt{Zk.Gen}(\Psi^{propose}, \theta_i, \phi_i, c_i^{\theta}, c_i^{\phi}, c_i^{I}, c_i^{II}, pok)$
(b) The correct verification of proof π_i in the contract as
$b_i \leftarrow \texttt{Zk.Vfy}(\pi_i, c_i^{\theta}, c_i^{\phi}, c_i^{I}, c_i^{II}, vek)$ updates barterer balance $c_i^{\theta} \leftarrow \texttt{Cm.Comm}(\theta_i \leftarrow \theta_i - \phi_i^{-})$.
(c) Barterer u_i generates a pair of public key and secret key $(pk_i, sk_i) \leftarrow \texttt{Pk.Setup}(\lambda)$
3. Bid Aggregation Submitting of u_i
(a) Barterer u_i submits the encryptions for two hypercube networks
$E_i^{I} \leftarrow \texttt{Pk.Enc}([\Sigma_i^{I}], pk_p^{I})$
$E_i^{II} \leftarrow \texttt{Pk.Enc}([\Sigma_i^{II}], pk_p^{II})$.
(b) Once all barterers submit their encryptions, barterer u_i decrypts the encryptions coming from their peers:
$\Sigma_p^{I} \leftarrow \texttt{Pk.Dec}([E_p^{I}], sk_i)$
$\Sigma_p^{II} \leftarrow \texttt{Pk.Dec}([E_p^{II}], sk_i)$.
4. Bid Aggregation Verifying of u_i
(a) Barterer u_i aggregates the bid aggregations by generating *aggregation* proof
$\Psi^{I} : \Sigma_i^{I} + \Sigma_p^{I}$, $\Psi^{II} : \Sigma_i^{II} + \Sigma_p^{II}$
$\pi_i^{I} \leftarrow \texttt{Zk.Gen}(\Psi^{I}, \Sigma_i^{I}, \Sigma_p^{I}, c_i^{I}, c_p^{I}, pok)$
$\pi_i^{II} \leftarrow \texttt{Zk.Gen}(\Psi^{II}, \Sigma_i^{II}, \Sigma_p^{II}, c_i^{II}, c_p^{II}, pok)$.
(b) The correct verification of proof π_i^{I} and π_i^{II} in the contract as
$b_i^{I} \leftarrow \texttt{Zk.Vfy}(\pi_i^{I}, c_i^{I}, c_p^{I}, vek)$
$b_i^{II} \leftarrow \texttt{Zk.Vfy}(\pi_i^{II}, c_i^{II}, c_p^{II}, vek)$ updates bid aggregations
$c_i^{I} \leftarrow \texttt{Cm.Comm}(\Sigma_i^{I} \leftarrow \Sigma_i^{I} + \Sigma_p^{I})$
$c_i^{II} \leftarrow \texttt{Cm.Comm}(\Sigma_{II} \leftarrow \Sigma_{II} + \Sigma_p^{II})$.
5. Bid Reproposing of u_i
(a) Barterer u_i reproposes bid ϕ_i' by generating *reproposing* proof
$\Psi^{repropose} \leftarrow \theta_i + \phi_i^{-} - \phi_i^{-'}$
$\pi_i \leftarrow \texttt{Zk.Gen}(\Psi^{repropose}, \theta_i, \phi_i, \phi_i', c_i^{\theta}, c_i^{I}, c_i^{II}, c_i^{I'}, c_i^{II'}, pok)$.
(b) The correct verification of proof π_i in the contract as
$b_i \leftarrow \texttt{Zk.Vfy}(\pi_i, c_i^{\theta}, c_i^{I}, c_i^{II}, c_i^{I'}, c_i^{II'}, vek)$ updates barterer balance
$c_i^{\theta} \leftarrow \texttt{Cm.Comm}(\theta_i \leftarrow \theta_i + \phi_i^{-} - \phi_i^{-'})$.
6. Token Bartering of u_i
(a) Barterer u_i barter tokens by generating *bartering* proof
$\Psi^{barter} \leftarrow \theta_i + \phi_i^{+}$
$\pi_i \leftarrow \texttt{Zk.Gen}(\Psi^{barter}, \theta_i, \phi_i, c_i^{\theta}, c_i^{I}, c_i^{II}, pok)$
(b) The correct verification of proof π_i in the contract as
$b_i \leftarrow \texttt{Zk.Vfy}(\pi_i, c_i^{\theta}, c_i^{I}, c_i^{II}, vek)$ updates barterer balance $c_i^{\theta} \leftarrow \texttt{Cm.Comm}(\theta_i \leftarrow \theta_i + \phi_i^{+})$.

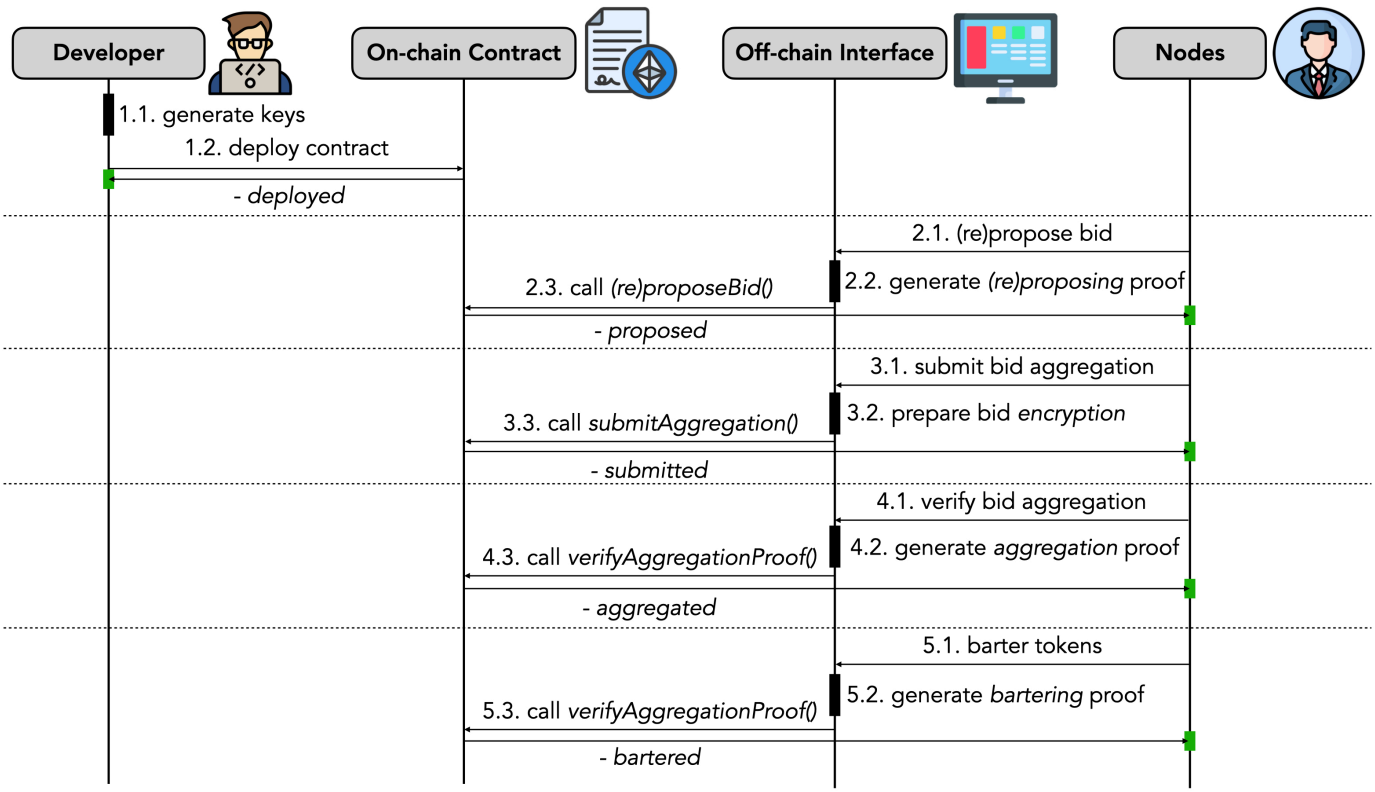


Figure 6.6. Sequence diagram of *PMTBS* protocol.

6.2.1. Zero-Knowledge Proof Model

The goal of the proof model in our privacy-preserving multi-token bartering protocol is to design a publicly-verifiable computation over the bid and balance commitments to be able to barter certain tokens in return for other tokens by still preserving their privacy. The algorithm for *(re)proposing bid* is given in Figure 6.7 where private function inputs include the current and next bids, the balance and their salting parameters while public function inputs include their commitment values. In Line 2-6, the current and next bid commitments for two hypercube networks as well as the balance commitment are internally computed. In Line 7, the tokens supplied in the bid are withdrawn from the current balance. In Line 8-9, more number of tokens must be supplied (i.e. over-supply requirement) and less number of tokens must be demanded (i.e. under-demand requirement) in the next bid with respect to the current bid. In Line 11, the proof finally checks the requirements and the correctness of the commitments

to return true. It should be noted that the impacts of the parties are clearly decoupled here as well where they have power only on their own bids and balances.

```
1: def main(private bidLeftI, private bidRightI, private bidLeftII, private bidRightII,
   private nextBidLeftI, private nextBidRightI, private nextBidLeftII, private
   nextBidRightII, private balance, public _bidCommI, public _bidCommII, pub-
   lic _nextBidCommI, public _nextBidCommII, public _balanceComm, public
   _nextBalanceComm):
2:   bidCommI ← sha256(bidLeftI, bidRightI)
3:   bidCommII ← sha256(bidLeftII, bidRightII)
4:   nextBidCommI ← sha256(nextBidLeftI, nextBidRightI)
5:   nextBidCommII ← sha256(nextBidLeftII, nextBidRightII)
6:   balanceComm ← sha256(balance)
7:   balance ← balance + (bidLeftI - bidLeftII) - (nextBidLeftI - nextBidLeftII)
8:   isIncreased ← (nextBidLeftI - nextBidLeftII) ≥ (bidLeftI - bidLeftII) ? true : false
9:   isDecreased ← (nextBidRightI - nextBidRightII) ≤ (bidRightI - bidRightII) ? true :
   false
10:  nextBalanceComm ← sha256(balance)
11:  result ← (bidCommI == _bidCommI &&
                bidCommII == _bidCommII &&
                nextBidCommI == _nextBidCommI &&
                nextBidCommII == _nextidCommII &&
                balanceComm == _balanceComm &&
                nextBalanceComm == _nextBalanceComm &&
                isIncreased == true &&
                isDecreased == true &&) ? true : false
12:  return result
```

Figure 6.7. Proof Implementation in ZoKrates [27] for (Re)Proposing Bid.

The algorithm for *aggregating bids* is given in Figure 6.8, which is the extended version (from scalar data to vector data aggregation) of the data aggregation proof of the *PVSS* protocol (see Figure 5.5). The algorithm includes the private inputs as bid aggregation of the barterer running this proof and bid aggregation of the peer the barterer is paired with and their salting parameters; and the public inputs as their

commitments. In Line 2-3, the commitments of the bid aggregations are internally computed. In Line 4-5, the left and right sides of the bid aggregations are separately aggregated. In Line 6, the commitment of the final aggregation of the aggregations is internally computed. In Line 7, it checks the correctness of these three commitments to return true in Line 8. If this proof is correctly verified on-chain, we expect the zero-knowledge proof verification function (as the state-transition function) changes the current aggregation commitment stored over the contract with the next aggregation commitment until all the aggregations are aggregated. Recall that *PMTBS* employs two simultaneous hypercube networks, which requires two such proofs to be verified to proceed the protocol forward.

```
1: def main(private bidLeft, private bidRight, private pairBidLeft, private pairBidRight,
   public _bidComm, public _pairBidComm, public _aggregatedBidComm):
2:   bidComm ← sha256(bidLeft, bidRight)
3:   pairBidComm ← sha256(pairBidLeft, pairBidRight)
4:   bidLeft ← bidLeft + pairBidLeft
5:   bidRight ← bidRight + pairBidRight
6:   aggregatedBidComm ← sha256(bidLeft, bidRight)
7:   result ← (bidComm == _bidComm &&
                pairBidComm == _pairBidComm &&
                aggregatedBidComm == _aggregatedBidComm) ? true : false
8:   return result
```

Figure 6.8. Proof implementation in ZoKrates [27] for aggregating bids.

The algorithm for *bartering tokens* is given in Figure 6.9 where the private inputs include the bid and balance information of the barterer, the final bid aggregation and their corresponding salting parameters while the public inputs include their commitments. In Line 2-6, the commitments of the bid, balance and final bid aggregation are internally computed. In Line 7, it checks whether there really exists a bartering solution. If no solution is found, the barterer cannot barter tokens since the resulting proof cannot be correctly verified on-chain. Otherwise, it returns the tokens the barterer demands in their bids to their balance in Line 8 by taking the commitment

of the next balance in Line 9. Finally, it checks the requirements and the correctness of all commitments in Line 10 to generate proof. Note that the tokens supplied are already withdrawn in the *(re)proposing bid* phase.

```
1: def main(private balance, private bidLeftI, private bidRightI, private bidLeftII,
   private bidRightII, private aggregatedBidLeftI, private aggregatedBidRightI, pri-
   vate aggregatedBidLeftII, private aggregatedBidRightII, public _balanceComm,
   public _bidCommI, public _bidCommII, public _aggregatedBidCommI, public
   _aggregatedBidCommII, public _nextBalanceComm):
2:   balanceComm ← sha256(balance)
3:   bidCommI ← sha256(bidLeftI, bidRightI)
4:   bidCommII ← sha256(bidLeftII, bidRightII)
5:   aggregatedBidCommI ← sha256(aggregatedBidLeftI, aggregatedBidRightI)
6:   aggregatedBidCommII ← sha256(aggregatedBidLeftII, aggregatedBidRightII)
7:   hasSolution ← (aggregatedBidLeftI - aggregatedBidLeftI) ≥
                    (aggregatedBidRightI - aggregatedBidRightI) ? true : false
8:   balance ← balance + (bidRightI - bidRightII)
9:   nextBalanceComm ← sha256(balance, balanceSalt)
10:  result ← (balanceComm == _balanceComm &&
                bidCommA == _bidCommA &&
                bidCommB == _bidCommB &&
                aggregatedBidCommA == _aggregatedBidCommA &&
                aggregatedBidCommB == _aggregatedBidCommB &&
                nextBalanceComm == _nextBalanceComm &&
                (hasSolution == true) ? true : false
11:  return result
```

Figure 6.9. Proof implementation in ZoKrates [27] for bartering tokens.

6.2.2. Smart Contract Model

The goal of the smart contract model in our *PMTBS* protocol is (i) to manage the protocol phases and the interactions among barterers in these phases, (ii) to store immutable and publicly-verifiable commitments for the barterer balance and bid information and (iii) to allow barterers to privately aggregate the bids and finally barter their

tokens using zero-knowledge proofs. The contract model involves with the functions for the *proposing bid* (see Figure 6.10), *submitting bid aggregation* (see Figure 6.11), verifying bid aggregation (see Figure 6.12), *reproposing bid* (see Figure 6.13) and finally *bartering tokens* (see Figure 6.14) phases of the protocol. In Figure 6.10, each barterer calls *proposeBid* function to propose their initial bid (as well as registering to that bartering session) to the contract by providing the corresponding *proposing bid* proof, two bid commitments for I-Network and II-Network, the commitment for their next balance (since the tokens supplied are deducted in this phase) and the public key for encryptions over the networks. In Line 2, it verifies the proof based on the public commitments where the contract provides the current commitments (e.g. *balanceCommitmentOf(msg.sender)*) so that barterer cannot freely change their balance. If the proof is correct in Line 3, the function changes the states of certain variables (e.g. for balance, bid and aggregation information) in Line 4-10. It returns a new registration identification number to that barterer as well in Line 11. In case the number of barterers reaches the maximum number, it moves the protocol forward in Line 13.

In Figure 6.11 for *submitting bid aggregation* phase, barterer calls *submitAggregation* function with two encryptions for two hypercube networks. In Line 2-5, it pairs every barterer on-chain with two different barterers at two networks by XORing the registration identification number of that barterer with respect to the current hypercube step (i.e. $log(N)$ total steps with N number of total barterers). This requires the contract to track the steps of both networks. In Line 6-7, it submits the encryptions to the corresponding peers in the networks. Similar to the $PTTS$ protocol, the two peers of a barterer only at the first step in these networks have to be necessarily different. Otherwise, a barterer may learn more information about the private value of another barterer as $\phi \leftarrow \Sigma_I - \Sigma_{II} \leftarrow \phi + e) - e$. Note that the way we handle this phase is the same as the $PVSS$ protocol (despite the extension from scalar to vector-based aggregation) since our proof model yields only the commitments of the computations to be stored on-chain. Therefore, changes in the computations from protocol to protocol does not require any modification on the contracts.

```
1: def proposeBid(Proof proof, uint bidCommI, uint bidCommII, uint nextBalanceComm,
   string publicKey) statusInProposal:
2:     bool isProofCorrect = verifier.verifyTx(proof,
                                               initialBidCommI,
                                               initialBidCommII,
                                               bidCommI,
                                               bidCommII,
                                               tokenContract.balanceCommOf(msg.sender),
                                               nextBalanceComm)
3:     if(isProofCorrect):
4:         tokenContract.setBalanceComm(msg.sender, nextBalanceComm)
5:         bidCommsI[msg.sender] = bidCommI
6:         aggregatedBidCommsI[msg.sender] = bidCommI
7:         bidCommsII[msg.sender] = bidCommII
8:         aggregatedBidCommsII[msg.sender] = bidCommII
9:         aggregatedBidCommsII[msg.sender] = bidCommII
10:        publicKeys[msg.sender] = publicKey
11:        ids[msg.sender] = registeredUsers.length
12:        registeredUsers.push(msg.sender)
13:        contractStatus = (registeredUsers.length == maximumUsers) ?
                                Status.Hypercube : Status.Proposal
```

Figure 6.10. Contract implementation for *proposing bid* phase.

```
1: function submitAggregation(string encryptionI, string encryptionII) statusInHypercube:
2:     uint256 pairI ← ids[msg.sender] ^ (2 ** currentStep)
3:     uint256 pairII ← ids[msg.sender] ^ (2 ** (maximumStep - currentStep - 1))
4:     address pairAddressI ← registeredUsers[pairI]
5:     address pairAddressII ← registeredUsers[pairII]
6:     encryptionsI[msg.sender][pairAddressI] ← encryptionI
7:     encryptionsII[msg.sender][pairAddressII] ← encryptionII
```

Figure 6.11. Contract implementation for *submitting bid aggregation* Phase.

In Figure 6.12 for *verifying bid aggregation* phase, barterer calls *verifyAggregation*

function two proofs and the next commitments for two hypercube networks. In Line 2-5, it first tracks the peer barterers and their addresses. In Line 6-7, it verifies these proofs for the networks with the current and next commitments of the barterer and the current commitments of the peer barterers. Note that it does not need the next commitments of the peers since the impacts the barterers have are only bound to only their own variables. If both proofs are correct in Line 8, the number of proofs verified is incremented to track the current step of the hypercube networks in Line 9. If the number of proofs verified is equal to the number of total barterers in Line 10, the current hypercube communication step is incremented as well in Line 11. In case all the bid aggregations are aggregated in Line 13, the current bidding round is incremented as well in Line 14 and the contract status is set either to the *reproposing bid* or *bartering* phase in Line 16-18. If the peers of barterer don't verify their proofs yet in Line 19 and 25, the next commitments of barterer are stored on temporary variables in Line 20 and 26, for I-Network and II-Network, respectively. Otherwise, the commitments are directly updated in Line 22-24 and 28-30, for I-Network and II-Network, respectively.

```
1: def verifyAggregation(Proof proofI, Proof proofII, uint nextCommI, uint nextCommII)
   statusInHypercube {
2:     uint256 pairI ← ids[msg.sender] ^ (2 ** currentStep)
3:     uint256 pairII ← ids[msg.sender] ^ (2 ** (maximumStep - currentStep - 1))
4:     address pairAddressI ← registeredUsers[pairI]
5:     address pairAddressII ← registeredUsers[pairII]
6:     bool isProofCorrectI ← verifier.verifyTx(proof,
                                                aggregatedBidCommsI[msg.sender],
                                                aggregatedBidCommsI[pairAddressI],
                                                newAggregatedBidCommI])
7:     bool isProofCorrectII ← verifier.verifyTx(proof,
                                                 aggregatedBidCommsII[msg.sender],
                                                 aggregatedBidCommsII[pairAddressII],
                                                 newAggregatedBidCommII)
8:     if (isProofCorrectI && isProofCorrectII):
9:         numberOfAggregationProofsVerified ← +1
10:        if (numberOfAggregationProofsVerified ← registeredUsers.length):
11:            currentHypercubeStage ← +1
12:            numberOfAggregationProofsVerified ← 0
13:            if (currentHypercubeDimension == maximumHypercubeDimension):
14:                currentBidReproposalCycle ← +1
15:                currentHypercubeDimension ← 0
16:                contractStatus ← Status.Reproposal
17:                if (currentBidReproposalCycle == maximumBiddingRound):
18:                    contractStatus ← Status.Barter
19:        if (tempCommI[pairAddressI] == 0):
20:            tempCommsI[msg.sender] ← newAggregatedBidCommI
21:        else:
22:            aggregatedBidCommsI[msg.sender] ← newAggregatedBidCommI;
23:            aggregatedBidCommsI[pairAddressI] ← tempCommsI[pairAddressI]
24:            tempCommI[pairAddressI] ← 0
25:        if (tempCommII[pairAddressII] == 0):
26:            tempCommII[msg.sender] ← newAggregatedBidCommII
27:        else:
28:            aggregatedBidCommII[msg.sender] ← newAggregatedBidCommII
29:            aggregatedBidCommII[pairAddressII] ← tempCommII[pairAddressII]
30:            tempCommII[pairAddressII] ← 0
```

Figure 6.12. Contract implementation for *verifying bid aggregation* phase.

In Figure 6.13 for the *reproposing bid* phase, barterer needs to propose bid again by satisfying the *over-supply* and *under-demand* problem requirements (see Equations (6.2.) and (6.3.)). Barterer calls the *reproposeBid* function which verifies the *reproposing bid* proof (see Figure 6.7). If the proof is correct in Line 3, it updates the balance commitment by returning the tokens supplied in the previous bid and deducting the tokens supplied in the current bid in Line 4. It also updates the bid and bid aggregation information in Line 5-8. It increments the number of proofs verified in Line 9 where it changes the contract status once all the users complete reproposing again in Line 11. There exists a subtle difference between the contract functions for the *proposing bid* and *reproposing bid* phases where the former function does not expect the *over-supply* and *under-demand* requirements to be satisfied since there is actually no previous bid at that phase to compare the next bid. Therefore, the former function uses a dummy previous bid where every next bid will certainly pass these requirements.

```
1: def reproposeBid(Proof proof, uint bidCommI, uint bidCommII, uint nextBal-
   anceComm) statusInReproposal:
2:     bool isProofCorrect ← verifier.verifyTx(proof,
                                               bidCommsI[msg.sender],
                                               bidCommsII[msg.sender],
                                               bidCommI,
                                               bidCommII,
                                               tokenContract.balanceOf(msg.sender),
                                               nextBalanceComm)
3:     if(isProofCorrect):
4:         tokenContract.setBalanceComm(msg.sender, nextBalanceComm)
5:         bidCommsI[msg.sender] ← bidCommI
6:         aggregatedBidCommsI[msg.sender] ← bidCommI
7:         bidCommsII[msg.sender] ← bidCommII
8:         aggregatedBidCommsII[msg.sender] ← bidCommII
9:         numberOfReproposalProofsVerified ← +1
10:        if (numberOfReproposalProofsVerified == registeredUsers.length):
11:            contractStatus ← Status.Hypercube
12:            numberOfReproposalProofsVerified ← 0
```

Figure 6.13. Contract implementation for *reproposing bid* phase.

In Figure 6.14 for the *bartering token* phase, barterer calls *barterTokens* function with the *bartering* proof (see Figure 6.9) and the commitment for their next and final balance. It verifies the correctness of the proof in Line 2 by using the commitments already stored over the contract and the next balance commitment. Since the barterer cannot change the existing commitment, the state transition rule the protocol enforces must be obeyed. If the proof is correct in Line 3, the barterer balance is updated with that next commitment in Line 4 and all the other bid and aggregation-related information is reset in Line 5-7. In case all barterers barter their tokens in Line 8, the contract for that bartering session self-destructs itself in Line 9. Barterer may burn their tokens by not generating this proof, which harms only himself.

```
1: def barterTokens(Proof proof, uint nextBalanceComm) statusInBarter {
2:     bool isProofCorrect ← verifier.verifyTx(proof,
                                                tokenContract.balanceOf(msg.sender),
                                                nextBalanceComm,
                                                aggregatedBidCommsI[msg.sender],
                                                aggregatedBidCommsII[msg.sender],
                                                bidCommsI[msg.sender],
                                                bidCommsII[msg.sender]);
3:     if(isProofCorrect):
4:         tokenContract.setBalanceComm(msg.sender, nextBalanceComm);
5:         bidCommsI[msg.sender] ← 0;
6:         bidCommsII[msg.sender] ← 0;
7:         numberOfBarteringProofsVerified ← +1;
8:         if(numberOfBarteringProofsVerified == registeredUsers.length):
9:             selfdestruct(payable(address(owner)));
```

Figure 6.14. Contract implementation for *bartering tokens* phase.

In our problem with collective and distributed nature, it would be possible to satisfy the right-hand side of a bid (i.e. tokens to be demanded) through multiple (but ambiguous) combinations from the other available bids. For instance, there exist three 2-token bids where the third bid demands 20 τ_0-tokens while the first and the second bids supply 10 and 15 τ_0-tokens in Figure 6.15 where there are multiple combination

to satisfy that demand as $(x, 20-x)$ with varying x. However, the challenge here is to have a collective agreement among barterers exactly on the same combination. In order to resolve this issue, we propose a novel token transfer approach in our thesis where its difference from the traditional ERC-20 standard is illustrated in Figure 6.16. According to ERC-20 shown on the top figure, sender transfers certain amount of tokens to receiver (through `transfer(_from, _value)` function) which updates their own balance along with receiver balance. On the other hand, in our *PMTBS* protocol, the barterers transfer tokens (in the proposing bids or bartering tokens phases) by updating only their own balances. According to the bids on the bottom figure, barterers deposit tokens to be supplied from only their own balances while proposing bids and later withdraw tokens to be demanded to only their own balances if a bartering solution is found. In other words, our novel transfer approach decouples the impacts of barterers on the balances of the other barterers.

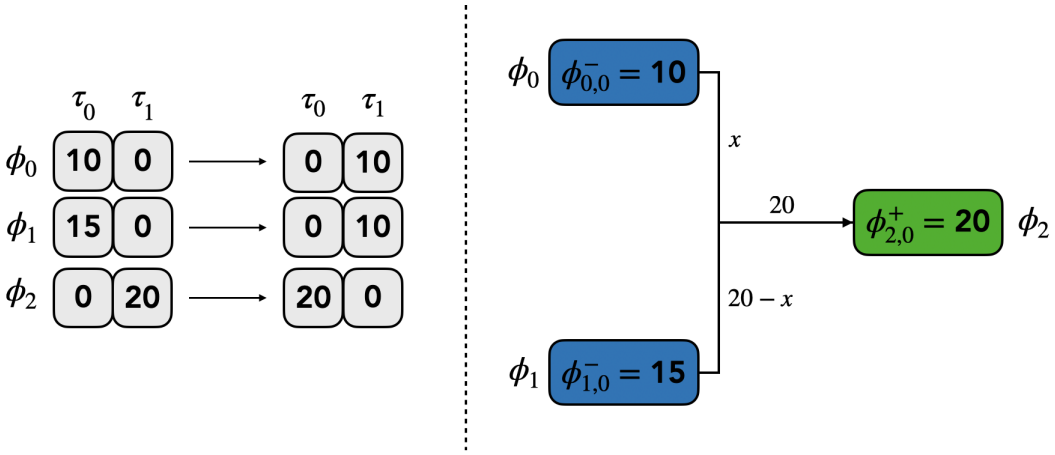


Figure 6.15. Ambiguity of bid satisfaction in *PMTBS* protocol.

In the proposed token transfer approach, the tokens to be given are immediately deducted from the user balance once the user proposes the initial bid. At each subsequent bid *reproposing* iteration, the difference in the tokens to be given between the current bid and the next bid (originated by the nature of our ascending auction system) is also deducted. On the other hand, the system does not force the user to collect the tokens to be received in the bid in case the bartering solution is found. Unless they

are not collected, they cannot be used for any other purpose including transferring to another address. Therefore, such a loss of tokens effectively creates a strong incentive for the user to collect them.

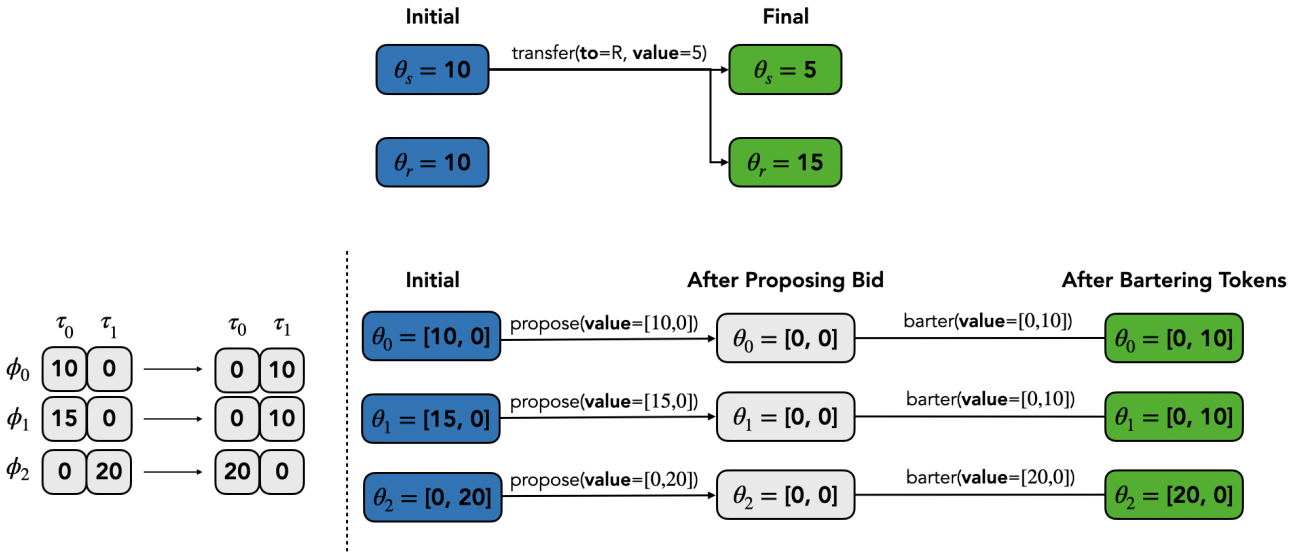


Figure 6.16. Comparison of ERC-20 token transfer with *PMTBS* token transfer.

6.2.3. Web Interface Model

The web interface model for the *PMTBS* protocol abstracts all these mathematical complexities into simple actions so that barterers can barter their tokens efficiently and swiftly. This interface mainly provides six actions for (i) *deploying contract*, (ii) *proposing bid*, (iii) *submitting bid aggregation*, (iv) *verifying bid aggregation*, (v) *(re)proposing bid* if necessary and (vi) *bartering tokens*. As in the web interfaces of our other protocols, we still expect barterers to install MetaMask extension [91] onto their browsers. The illustrations of these actions are shown in Figure 6.17. Our open-source implementation is also publicly available for further inspection [54].

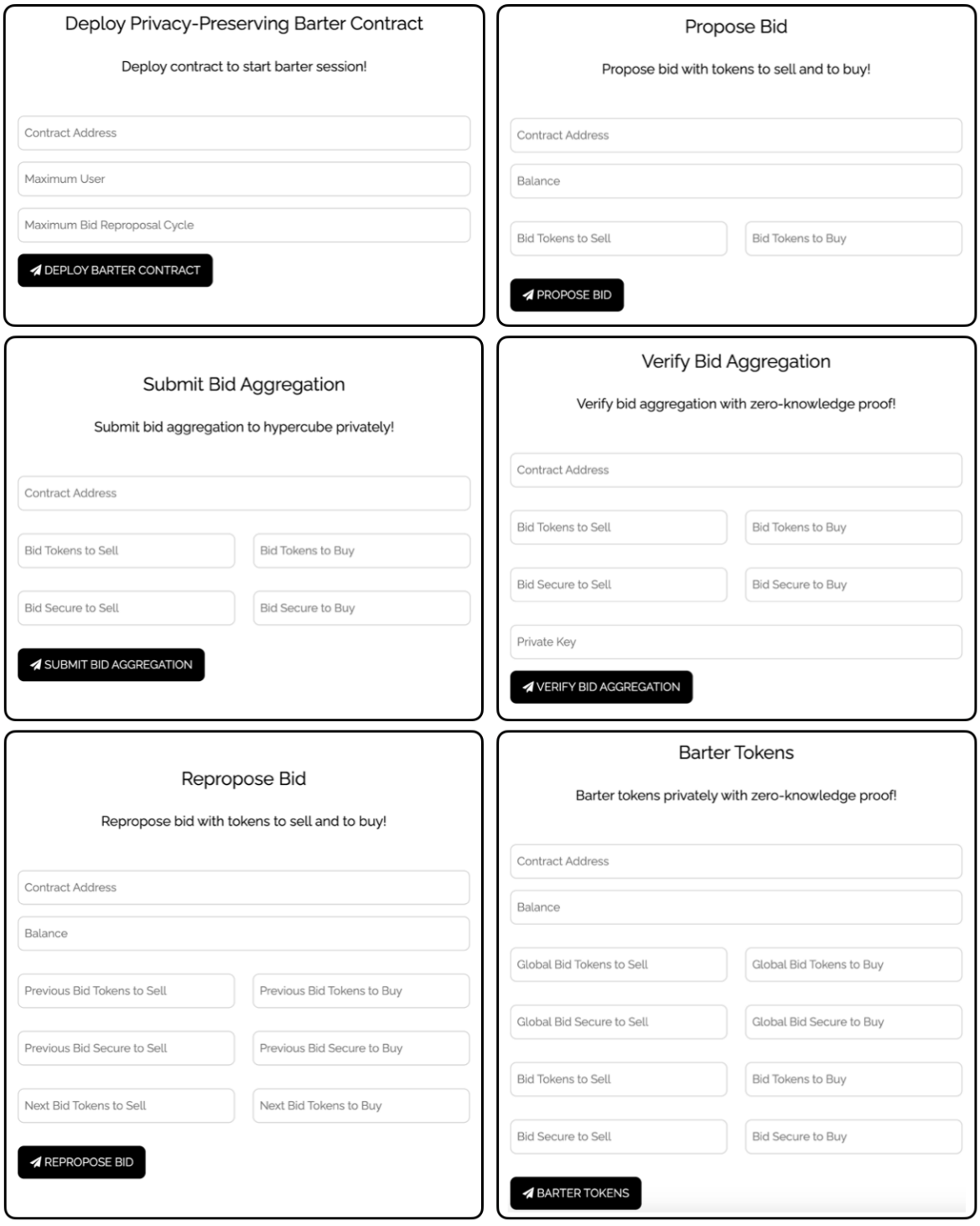


Figure 6.17. Web user interface of *PMTBS* protocol.

The first action is *deploying barter contract* where a contract owner must specify the maximum number of barterers at most and the maximum bidding round. The owner later shares the resulting contract address with the potential barterers. Refer

to Figure 4.11 for interface implementation of *deploying barter contract*. The second action is *proposing bid* where the barterer proposes the tokens to supply and the tokens to demand. The algorithm for this action is given in Figure 6.18 where it restores public key in Line 4, computes hashes for balance, next balance and bid in Line 5, generates proof with respect to these public and private values in Line 6 and calls the *proposeBid* function (see Figure 6.10) in the contract in Line 7. The third action is *submitting bid aggregation* where each barterer exchanges their own bid aggregations with two peers over two hypercube networks by calling the *submitAggregation* function in the contract (see Figure 6.11). The fourth action is *verifying bid aggregation* where each barterer must show the correctness of off-chain bid aggregation with two aggregations (their own aggregation and aggregation coming from their peer per network) by calling the *verifyAggregation* function in the contract (see Figure 6.12). Refer to Figure 5.11 and Figure 5.12 for interface implementation of the last two actions.

```
1: async def proposeBid():
2:     [contractAddress, balance, bid] ← interface.getInputs()
3:     contract ← new web3.Contract(contractAddress)
4:     publicKey ← interface.restorePublicKey()
5:     [balanceComm, nextBalanceComm, bidCommI, bidCommII] ← interface.commit(balance, bid)
6:     proof ← zokrates.generate(balance, bid, balanceComm, nextBalanceComm, bidCommI, bidCommII) - Figure 6.7
7:     transaction ← contract.proposeBid(proof, bidCommI, bidCommII, nextBalanceComm, publicKey) - Figure 6.10
8:     return transaction
```

Figure 6.18. Interface implementation for *proposing bid* phase.

The fifth action is optional *reproposing bid* where the barterer reproposes the tokens to supply and the tokens to demand with respect to the previous bid. The algorithm for this action is given in Figure 6.19 where it computes necessary hashes in Line 4, generates a corresponding zero-knowledge proof with respect to these public and private values in Line 5 by comparing the the next bid with the current bid to

check the problem requirements and calls the *reproposeBid* function (see Figure 6.13) in the contract in Line 6. Finally, the six action is *bartering tokens* where the barterers barter their tokens with respect to the bartering solution. The algorithm for this action is given in Figure 6.9 where it computes necessary hashes in Line 4, generates a corresponding zero-knowledge proof to show the correctness of bartering with respect to the global bartering solution in Line 5 and calls the *barterTokens* function (see Figure 6.14) in the contract in Line 6.

```
1: async def reproposeBid():
2:     [contractAddress, balance, currentBid, nextBid] ← interface.getInputs()
3:     contract ← new web3.Contract(contractAddress)
4:     [balanceComm, nextBalanceComm, currentBidCommI, currentBidCommII, nextBid-
   CommI, nextBidCommII] ← interface.commit(balance, currentBid, nextBid)
5:     proof ← zokrates.generate(balance, currentBid, nextBid, balanceComm, nextBal-
   anceComm, currentBidCommI, currentBidCommII, nextBidCommI, nextBidCommII) -
   Figure 6.7
6:     transaction ← contract.reproposeBid(proof, nextBidCommI, nextBidCommII) - Fig-
   ure 6.13
7:     return transaction
```

Figure 6.19. Interface implementation for *reproposing bid* phase.

```
1: async def barterTokens():
2:     [contractAddress, balance, bid, aggregation] ← interface.getInputs()
3:     contract ← new web3.Contract(contractAddress)
4:     [balanceComm, nextBalanceComm, bidCommI, bidCommII, aggrCommI, aggrCommII]
   ← interface.commit(balance, bid, aggregation)
5:     proof ← zokrates.generate(balance, bid, aggregation, balanceComm, nextBal-
   anceComm, bidCommI, bidCommII, aggrCommI, aggrCommII) - Figure 6.9
6:     transaction ← contract.barterTokens(proof, nextBalanceComm) - Figure 6.14
7:     return transaction
```

Figure 6.20. Interface implementation for *bartering tokens* phase.

6.3. Extension to Privacy-Preserving Multi-Token Transfer

As previously said in Section 6.2, the privacy-preserving multi-token bartering protocol has the main *multi-token* contract to manage the barterer balances. This contract also extends the *PTTS* protocol by supporting multi-token depositing and withdrawing. To compare, the *PTTS* protocol treats the balances and transactions amounts as scalar values while the *PMTBS* considers composite balance vectors as collection of multiple balances and composite transaction amount vectors as collection of multiple amounts as:

$$\theta_s \quad : \quad \langle \theta_{s,0}, \theta_{s,1}, \ldots, \theta_{s,M-1} \rangle \tag{6.63}$$

$$\Delta \quad : \quad \langle \Delta_0, \Delta_1, \ldots, \Delta_{M-1} \rangle \tag{6.64}$$

where θ_s refers to the balance vector of the sender u_s while Δ refers to the transaction amount between the sender u_s and receiver u_r for M different token types. When depositing and withdrawing tokens as:

$$\theta'_{s,i} \quad \leftarrow \quad \theta_{s,i} - \Delta_i, \quad \forall i \tag{6.65}$$

$$\theta'_{r,i} \quad \leftarrow \quad \theta_{r,i} + \Delta_i, \quad \forall i \tag{6.66}$$

where the protocol simply deposits amounts from the sender balance for each token type and withdraws the same amounts to the receiver balance for each token type again. To avoid repetitions across the protocols in the thesis, we will omit the common definitions. Refer to Section 4.1 for the problem definition and 4.2. for the system architecture. Furthermore, we still rely on the same zero-knowledge proof and smart contract models except certain implementation aspects (e.g. Line 4 in both Figure 4.4 and Figure 4.5 is performed for every token type through for loop). Note that this may lead the experimental results to be varied between the *PTTS* and *PMTBS* protocols.

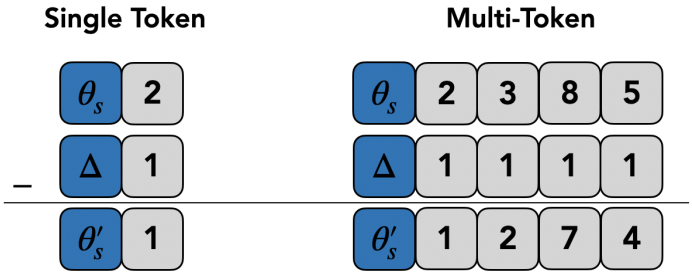


Figure 6.21. Extension of *PTTS* to privacy-preserving multi-token payment.

6.4. Protocol Analysis

6.4.1. Scalability

In this section, we theoretically analyze the scalability of the *PMTBS* protocol with respect to the computational, communication and storage overheads. Review Section 4.3.1 for further information about the importance of these overheads. We consider the increasing number of barterers to evaluate the scalability of the protocol as a secure-multi-party computation, (unlike the *PTTS* protocol).

6.4.1.1. Computational Overhead. As in our other privacy-preserving protocols, the zero-knowledge proof generation function constitutes the primary source of the computational overhead. The *PTTS* protocol relies on two proof schemes (see Figure 4.4 for *depositing tokens* and see Figure 4.5 for *withdrawing tokens*) while the *PVSS* and *PRFX* protocols rely on only one proof scheme (see Figure 5.5 for *aggregating data*). Now, the *PMTBS* protocol uses three different proof schemes for (re)proposing bid in Figure 6.7, aggregating bids in Figure 6.8 and bartering tokens in Figure 6.9. For each privacy-preserving multi-token bartering session, the barterer needs to generate one proof to (re)propose a bid, two proofs to aggregate bids for every hypercube network communication step h and one proof to barter tokens. Note that reproposing bids and later aggregating bids might be re-iterated (in case no bartering solution is found) with respect to the maximum bidding round T, (see Figure 6.1). We count the number of

proofs generated through all these phases with the following way:

$$\xi_{comp} = T \cdot (1 + 2 \cdot log(N)) + 1 \quad (6.67)$$

$$\xi_{comp}^{\Sigma} = N \cdot \xi_{comp} \quad (6.68)$$

where ξ_{comp} and ξ_{comp}^{Σ} are the computational overheads on a barterer and on a system, respectively while N is the increasing number of barterers. We can observe that the computational complexity of the *PTTS* protocol (as a sub-protocol of the *PMTBS* protocol) constitutes the larger portion of the entire complexity in Equation (6.67) as $T \cdot 2 \cdot log(N)$. Therefore, greater number of barterers results in higher blockchain gas to be consumed and more proofs to be generated. On the other hand, the zero-knowledge proof verification does not lead to computational overhead since the proofs are verified on-chain in the contract.

6.4.1.2. Communication Overhead. The direct interactions among the barterers are the main source of the communication overhead. In our protocol, each barterer needs to perform two communications for two networks for every hypercube network communication step and for every bidding round. We represent this overhead at the worst-case scenario by counting the number of interactions with the following way:

$$\xi_{comm} = 2 \cdot T \cdot log(N)) \quad (6.69)$$

$$\xi_{comm}^{\Sigma} = N \cdot \xi_{comm} \quad (6.70)$$

where ξ_{comm} and ξ_{comm}^{Σ} are the communication overheads on a party and on a system, respectively while T is the maximum number of bidding rounds. Note that these overheads result from the *submitting bid aggregation* phase on hypercube networks while the other phases do not require any sorts of direct interactions. Moreover, this overhead increases with the increasing number of barterers available in the system.

6.4.1.3. Storage Overhead. The data on the contracts is the main source of the on-chain storage while off-chain storage is relatively cheaper. The on-chain data includes mappings (i.e. dictionaries) for (i) registrations, (ii) public keys, (iii) balance commitments, (iv) two bid commitments for two networks, (v) two bid aggregation commitments , (vi) two temporary bid aggregation commitments for two networks (refer Section 5.2 for more information about temporary commitment values), (vii) two encryptions. We count the number of entries in these mappings and lists to count the storage overhead as follows:

$$\xi_{strg} = 9 + 2 \cdot log(N)) \tag{6.71}$$

$$\xi_{strg}^{\Sigma} = N \cdot \xi_{strg} \tag{6.72}$$

where ξ_{strg} and ξ_{strg}^{Σ} are the storage overheads on a party and on a system, respectively. The first component of Equation (6.71) (i.e. 9) is the first six mappings while the second component (i.e. $2 \cdot log(N)$) is the seventh mapping alone. Note that to simplify this analysis, we disregard variations in the sizes of different data entries (e.g. a public key entry and an encryption entry may have different sizes) and uniformly treat each entry as a single unit. In addition, we disregard the memory consumption of the verification key since it is constant. The scalability of the *PMTBS* protocol is theoretically given in Table 6.2 with the increasing number of parties (from 2 to 1,000,000).

Table 6.2. *PMTBS* scalability as communication, computation and storage overheads.

#Nodes	Communication Overhead		Computation Overhead		Storage Overhead	
	On Node	On System	On Node	On System	On Node	On System
2	3	6	2	4	11	22
8	8	64	6	48	15	120
32	12	384	10	320	19	608
128	16	2048	14	1792	23	2944
1,000,000	42	42M	40	40M	49	49M

6.4.2. Security

In this section, we evaluate the security of the *PMTBS* protocol with respect to potential attacks on blockchain and theoretically show the correctness by using the formal reduction proof. The attacks we consider include *replay* attack, *preimage* attack, *Sybil* attack and *collusion* attack. The security of the protocol can be further analyzed based on more variety of attacks, which is among our future works. Refer to Section 4.3.2 for the definitions of these attacks in the scope of our thesis.

Replay Attack. The protocol is resistant to the replay attack since the corresponding commitments for balances, bids and bid aggregations are immediately updated once zero-knowledge proofs are correctly verified. For instance, barterer balance is replaced with a new balance commitment on-chain once *proposing bid* proof is correctly submitted and verified in the contract. In case the same transaction is attempted to be replayed, that proof would not be verified based on these new commitments since the set of commitments during proof generation and verification must be exactly matched.

Preimage Attack. The protocol handles this attack by introducing large and random salting values σ while computing commitment and regularly replacing these salting values with new values, (see Section 2.6). We use salting values to mask barterer balances, bids and bid aggregations. This precaution results in distinct commitment values even if the balances of two different barterers are numerically the same. This requires the attacker to store a large enumeration table to compare the commitment values, which is considered to be infeasible.

Sybil Attack. The protocol simply prevents the same barterer (i.e. blockchain address) to register (i.e. propose multiple bids) to the system by constantly checking the addresses that are already registered. A barterer that proposes a bid cannot repropose another bid again until the next bidding round is open. This also prevents that barterer to be in multiple positions on hypercube networks with the goal to learn more information about the other barterers. However, it should be noted that a barterer may

propose multiple bids through different blockchain addresses where its detection and prevention may require further mechanisms which are not in the scope of this thesis.

Collusion Attack. Although the protocol prevents collusion attack to a certain extent, it can be open to this attack on specific scenarios where two barterers having direct communications with the same barterer in the first hypercube communication step during the *submitting bid aggregation* phase may learn additional information about that barterer. This issue is propagated into this protocol from the underlying *PVSS* protocol for bid aggregations. For more information about this issue, refer to Section 5.5.2. Additional precautions for this issue are among our future works as well.

6.4.2.1. Reduction Proof. In this section, we theoretically show the security of the *PMTBS* protocol through the formal reduction proofs to the underlying zero-knowledge proof protocol (i.e. zkSNARKs [16]) with the following way:

Theorem. The privacy-preserving data aggregation protocol *PMTBS* is secure if and only if the zkSNARKs protocol [16] is secure.

Proof. Suppose that $\mathcal{A}$ and $\mathcal{B}$ are the adversaries that can break the *PMTBS* and zkSNARKs protocols, respectively. For the following scenario between the adversary $\mathcal{A}$ and the challenger (i.e. verifier) $\mathcal{C}$:

- The challenger $\mathcal{C}$ runs the zkSNARKs trusted setup to generate the proving key *pok* and the verification key *vek*.
- The adversary $\mathcal{A}$ proposes a bid $\phi_{\mathcal{A}}$ and forwards the commitment $c_{\mathcal{A}}^{\phi} \leftarrow \texttt{Cm.Comm}(\phi_{\mathcal{A}})$ to the challenger.
- The adversary $\mathcal{A}$ samples a falsified bid exceeding their balance $\phi'_{\mathcal{A}} > \theta_{\mathcal{A}}$ and computes the commitment $c_{\mathcal{A}}^{\phi'} \leftarrow \texttt{Cm.Comm}(\phi')$.
- The adversary $\mathcal{A}$ invokes the adversary $\mathcal{B}$ to generate a false proof based on falsified bid $\pi_{\mathcal{A}} \leftarrow \texttt{Zk.Gen}(\Psi^{propose}, \theta_{\mathcal{A}}, \phi'_{\mathcal{A}}, c_{\mathcal{A}}^{\theta}, c_{\mathcal{A}}^{\phi'}, c_{\mathcal{A}}^{I}, c_{\mathcal{A}}^{II}, pok)$.
- The challenger $\mathcal{C}$ returns true in case this false proof is correctly verified as $b_{\mathcal{A}} \leftarrow$

$\texttt{Zk.Vfy}(\pi_{\mathcal{A}}, c^{\theta}_{\mathcal{A}}, c^{\phi}_{\mathcal{A}}, c^{I}_{\mathcal{A}}, c^{II}_{\mathcal{A}}, vek)$ and false otherwise.

This scenario requires the challenger $\mathcal{C}$ to return true if and only if there really exists the adversary $\mathcal{B}$ that has ability to break zkSNARKs by forging the proof with respect to the falsified bid ϕ'. However, this contradicts our assumption that the zero-knowledge proof scheme itself is secure. More formally:

$$\left|\Pr\left[1 \leftarrow \texttt{Zk.Vfy}(\pi_{\mathcal{A}}, c^{\theta}_{\mathcal{A}}, c^{\phi}_{\mathcal{A}}, vek) \;\middle|\; \begin{array}{l} (pok, vek) \leftarrow \texttt{Zk.Setup}(\lambda) \\ \phi_{\mathcal{A}}, \phi'_{\mathcal{A}} \xleftarrow{\$} \{\mathbb{Z}_0^+\}^M, \quad \phi'_{\mathcal{A}} > \theta_{\mathcal{A}} \\ \Psi^{propose} \leftarrow \theta_{\mathcal{A}} - \phi^{-}_{\mathcal{A}} \\ c^{\phi}_{\mathcal{A}} \leftarrow \texttt{Cm.Comm}(\phi_{\mathcal{A}}) \\ c^{\phi'}_{\mathcal{A}} \leftarrow \texttt{Cm.Comm}(\phi'_{\mathcal{A}}) \\ c^{\theta}_{\mathcal{A}} \leftarrow \texttt{Cm.Comm}(\theta_{\mathcal{A}}) \\ \pi \leftarrow \texttt{A}(\texttt{B}(\texttt{Zk.Gen}(\Psi^{propose}, \theta_{\mathcal{A}}, \phi'_{\mathcal{A}}, c^{\theta}_{\mathcal{A}}, c^{\phi'}_{\mathcal{A}}, pok)) \end{array}\right]\right| = 0 \tag{6.73}$$

where this equation implies that the probability of correct verification of the false proof must be zero from the perspective of the challenger (i.e. verifier).

6.4.3. Limitations

The *PMTBS* protocol has certain drawbacks that are needed to be thoroughly addressed in the future: (i) limit on token size, (ii) limit on barterer size, (iii) limit on bidding rounds, (iv) drawbacks of *PVSS*, (v) platform dependencies and (vi) performance issues. We refer these limitations as follows:

- Limit on Token Size: The protocol supports only fixed number of token types (i.e. M tokens) for both balances θ_i and bids ϕ_i. This limitation results from the ZoKrates framework which supports only static arrays whose length must be known at compile-time [27]. Therefore, we divide 256-bit strings of balances

into eight equally-sized sub-strings where they are allocated to store balances for seven tokens and one salting parameter to mask balances for enumeration attacks. Similarly, we divide 256-bit strings for bids into eight equally-sized sub-strings to represent their left (i.e. tokens to be supplied) and right (i.e. tokens to be demanded) parts with the corresponding salting parameters. This approach theoretically supports any number of token types as long as it is known at compile-time. Having a dynamically-changing number of tokens (e.g. with respect to the contract owner inputs) is among our future works.

- Limit on Barterer Size: The protocol relies on the complete hypercube network to aggregate bids where it supports only $N = 2^d$ number of barterers in d-dimension. For this issue, we develop three communication techniques (see Section 5.3) while their adoption to this protocol requires additional effort and justification.
- Limit on Bidding Rounds: The bidding round refer to the number of the times the barterers have to repropose their bids under certain requirements in case there does not exists any feasible bartering solution. If a solution cannot be reached even in the last bidding round, that bartering session is terminated. Note that increasing the bidding rounds poses a greater chance to find a solution while it puts additional communications to be performed among the barterers, (proportional to $N \cdot log(N)$ per round with N number of barterers). Therefore, decision to set the bidding round is a trade-off where the right balance requires careful consideration of certain factors including the number of barterers and success ratio of the session.
- Drawbacks of *PVSS*: The protocol uses the extended version of the *PVSS* protocol to aggregate vector-based bids. Therefore, *PMTBS* suffers from all the issues *PVSS* has. Refer to Section 5.5.3 for further information about these issues.
- Platform Dependencies: As in `PTTSS` and *PVSS*, this protocol requires EVM-dependent blockchain platforms and the zkSNARKs protocol, (see Section 4.3.3).
- Performance Issues: The protocol has certain computational, communication and storage overheads to be addressed, (see Section 4.3.1).

6.5. Experimental Evaluation

In this section, we outline the frameworks that we use in order to build the privacy-preserving multi-token bartering protocol and the test parameters we use during our experimental evaluation for reproducibility. The performance of the protocol is assessed through (i) the problem requirements, (ii) the blockchain gas consumption, (iii) zero-knowledge proof generation/verification times and (iv) zero-knowledge proof artifact sizes. The experiments especially regard the increasing number of parties to measure scalability; and generalizes the findings for an arbitrary number of parties.

6.5.1. Experimental Setup

The *PMTBS* protocol uses (i) ZoKrates framework [27] and (ii) ZoKrates-js library [93] for the zero-knowledge proof model; (i) Solidity language [94] and (ii) Remix IDE [95] for the smart contract model and; (i) HTML/CSS/Java Languages, (ii) Ethers-js Library [96], (iii) Eccrypto-js Library [97], (iv) Browserify Bundler [98], (v) Webpack Bundler [99] and (vi) MetaMask [91] for the web interface model. Refer to Section 4.5.1 for further information about these libraries. The protocol has been tested upon both Avalanche Fuji Test network [3] and the local Ethereum network where *Geth* (i.e. Go-Ethereum) [104] is used to perform a full simulation over multiple parties. The experiments are performed on MacBook Pro Notebook with a 2.6 GHz Intel Core i7 processor, 16 GB memory and 6 cores. The open-source implementations are also publicly available in our GitHub repository [54], respectively.

6.5.2. Requirement Verification

In this section, we evaluate the proposed *PMTBS* protocol with respect to the problem requirements (defined in Section 6.1) including privacy, confidentiality, public-verifiability, authentication, non-interaction, scalability, correctness and usability. The protocol aims to satisfy these requirements by specifically integrating certain components including blockchain, smart contract, commitment scheme, public-key encryp-

tion, zero-knowledge proof, hypercube networks and web interface as follows:

- (R1) Privacy: It protects the privacy of bids and balances by having their corresponding commitment values on-chain and later verifying the correctness of the computations over these values through zero-knowledge proof, (see Figure 2.7. for further information).
- (R2) Confidentiality: It uses the public-key encryption scheme for confidentiality to encrypt the bid aggregation information via public keys and later decrypting these encryptions via private keys (see Figure 2.5. for further information).
- (R3) Trustless: It uses zero-knowledge proof to verify off-chain computations of barterers on hypercube networks to complete their token exchanges without trusted third party.
- (R4) Public Verifiability: It publicly verifies the correctness of the zero-knowledge proofs on-chain by using several proof-verifying contracts on blockchain.
- (R5) Authentication: It requires every barterer to register while proposing their initial bids where it prevents barterers to join later as an authentication scheme.
- (R6) Non-interaction: It relies on the underlying zkSNARKs protocol [16] which is a popular non-interactive zero-knowledge proof protocol in the literature. However, as in the *PVSS* protocol, the *PMTBS* protocol also requires a certain degree of interaction to get the final bid aggregation.
- (R7) Scalability: It integrates hypercube topology as a logical layout layer over blockchain platform as a physical layout layer to improve the protocol scalability, (see Section 6.4.1 for more information about scalability and overheads).
- (R8) Correctness: Except for two components (hypercube networks and web interface), all the other components are necessary for the protocol to guarantee the correctness of the bartering among barterers.
- (R9) Usability: It integrates a web interface to decouple the complex mathematical operations from the barterer actions to improve user-friendliness.

Table 6.3 briefly presents how the *PMTBS* protocol satisfies these requirements with the components it has.

Table 6.3. *PMTBS* verification of problem requirements.

Requirement	Blockchain	Smart Contract	Commitment Scheme	Asymmetric Encryption	Zero-Knowledge Proof	Hypercube Topology	Web User Interface
R1. Privacy			✓		✓		
R2. Confidentiality				✓			
R3. Trustless			✓	✓	✓	✓	
R4. Public Verifiability	✓	✓			✓		
R5. Authentication		✓					
R6. Non-interaction					✓		
R7. Scalability	✓					✓	
R8. Correctness	✓	✓	✓	✓	✓		
R9. Usability							✓

6.5.3. Blockchain Gas Consumption

We present the blockchain gas consumption of the smart contact functions in Table 6.4 where we use the Avalanche Fuji test network [3] and the local Ethereum blockchain with 16 barterers. The results are measured on 12/06/2023 with the gas price per gas unit of 26.5 `nAvax` and the exchange rate from Avax to USD of $22.7. From the table, we observe that the most expensive operation is *deploying contract* with over +5 million gas units corresponding to $3.26. This is within the expected range since it includes the costs of the main bartering contract along with three proof-verifying contracts for proposing bids (see Figure 6.7), verifying bid aggregation (see Figure 6.8) and bartering tokens (see Figure 6.9). As in our other protocols, these three contracts include their own verification keys and complex verification operations. Fortunately, this is the only cost that the developer (or bartering owner) needs to start bartering. The second most expensive function is *proposing bid* (see Figure 6.10) with >1.7M gas units corresponding to $1.04 where it verifies the proof for proposing bid based on the

public commitments and updates the balance and bid and bid aggregation information properly. The other functions requiring the proof verification (e.g. *depositing tokens*, *withdrawing tokens*, *verifying bid aggregation*, *reproposing bid*, *bartering tokens*) spend at least >1.3M gas units as well. Note that the function for *proposing bid* spends more gas than the function *reproposing bid* since it involves a certain additional operations as well (e.g. public key management). On the other hand, the least expensive function is *submitting bid aggregation* with less than 1.0M gas units corresponding to \$0.57 since it only involves a encryption management without any proof verification. From these results, we can infer that proof verification is the source for blockchain gas consumption to a large extent. For 16 different barterers at the best-case scenario (i.e. a bartering solution is found in the first bidding round), each barterer needs to cover the minimum of $\$8.11 = \$1.04 + log(16) \cdot \$0.57 + log(16) \cdot \$0.98 + \$0.87$. Since our protocol supports multi-token bartering with 7 token types at the current implementation, it approximately results in $\tilde{\$}1$ per to barter a token type. Nevertheless, cutting down this is among our future work to address.

We generalize the total blockchain gas consumption from the perspectives of both each individual party and the whole system. For an individual party by integrating the gas consumption of the contract functions:

$$\gamma_{party} = \gamma_{fixed} + log(N) \cdot \gamma_{dynamic} \tag{6.74}$$

$$\gamma_{fixed} = T \cdot \gamma_{(re)propose} + \gamma_{barter} \tag{6.75}$$

$$\gamma_{dynamic} = T \cdot \gamma_{submit} + T \cdot \gamma_{verify} \tag{6.76}$$

where the (re)proposing $\gamma_{(re)propose}$ and the bartering costs γ_{barter} constitute the constant-time fixed cost γ_{fixed} while the bid submission γ_{submit} and verification costs γ_{verify} constitute logarithmic-time dynamic cost $\gamma_{dynamic}$ with respect to the number of total parties in the system. However, note that all the phases except the bartering phase can be re-iterated T times if no bartering solution is found, (see Section 6.2). For the

entire system with N number of parties:

$$\gamma_{system} = N \cdot \gamma_{fixed} + N \cdot log(N) \cdot \gamma_{dynamic} \tag{6.77}$$

$$\gamma_{barter} = N \cdot \Big(T \cdot \Big(\gamma_{(re)propose} + log(N) \cdot \gamma_{submit} + log(N) \cdot \gamma_{verify}\Big) + \gamma_{barter}\Big) \tag{6.78}$$

where we just sum the costs of all parties as the system cost, $\gamma_{system} = N \cdot \gamma_{party}$.

Table 6.4. Blockchain gas consumption of the *PMTBS* protocol phases.

Function	Gas Units (Avalanche)	Gas Units (Local Geth)	Gas Cost (Avax)	Gas Cost (USD)
Deploying Multi-Token Contract	3,172,699	3,172,699	0.087249	1.99
Deploying Barter Contract	5,197,558	5,197,558	0.142932	3.26
Depositing Tokens	1,683,626	1,683,650	0.044616	1.02
Withdrawing Tokens	1,300,382	1,295,630	0.034460	0.79
Proposing Bid	1,724,275	1,687,189	0.045693	1.04
Submitting Bid Aggregation	941,585	602,125	0.024952	0.57
Verifying Bid Aggregation	1,624,108	1,592,715	0.043038	0.98
Reproposing Bid	1,449,488	1,430,550	0.038411	0.88
Bartering Tokens	1,442,438	1,403,262	0.038224	0.87

6.5.4. Proof Generation/Verification Times

The zero-knowledge proof generation times for five zero-knowledge proof implementations are found through the average of 20 independent runs where the individual measurements are given in Table 6.5. From the table, we observe that the minimum, maximum and mean measurements in seconds are (i) 44.5, 46.1 and 45.2 for *(re)proposing bids* proof, (ii) 43.5, 44.1 and 43.8 for *aggregating bids* proof, (iii) 90.6, 94.7 and 92.2 for *bartering tokens* proof, (iv) 43.4, 44.1 and 43.8 for *depositing tokens* proof and finally (v) 88.8, 91.8 and 89.8 for *withdrawing tokens* proof. The proof generations for *(re)proposing bid* and *bartering token* require the most time since they are the most complex with the highest number of commitments (i.e. six different commitments). Whereas, the proof generations for *verifying bid aggregation* and *withdrawing tokens* require the least time since they compute only three different commitments. Unlike the proof generations, the proof verification that occurs on-chain is completed

with respect to the current throughput of the blockchain platform itself.

Table 6.5. *PMTBS* zero-knowledge proof generation times (in seconds).

Run	Depositing Tokens	Withdrawing Tokens	(Re)propose Bid	Verify Aggregation	Barter Tokens
1	45.001	44.562	98.051	45.731	95.475
2	44.804	43.849	94.550	44.022	89.261
3	44.581	43.832	91.196	43.609	89.453
4	46.100	43.738	91.607	43.802	89.766
5	45.269	43.808	92.552	44.164	89.550
6	44.874	43.585	91.448	45.485	88.986
7	44.938	43.933	91.049	43.992	89.128
8	45.570	43.851	94.973	43.449	90.911
9	45.246	43.744	93.799	43.807	90.600
10	45.127	43.626	92.963	43.520	89.181
11	45.119	44.012	91.456	43.753	89.098
12	45.902	44.135	94.289	44.149	90.436
13	44.751	43.623	92.904	44.017	89.382
14	44.807	43.788	91.855	43.876	91.165
15	45.165	43.594	91.862	43.419	88.921
16	46.153	43.633	91.543	44.004	89.709
17	44.841	43.599	90.844	43.991	88.889
18	45.572	43.930	90.692	43.812	91.133
19	45.520	44.087	91.714	44.038	91.813
20	44.889	43.875	91.545	43.803	89.024

6.5.5. Proof Artifact Sizes

In this experiment, we measure the zero-knowledge proof artifacts including (i) the proving key size, (ii) the verification key size and (iii) the proof size. For our theoretical expectation in this experiment, refer to Section 4.5.3. We present the experimental results in Table 6.6 that the proving keys are larger than the verification keys and proofs themselves for all proof implementations. The proof size remains constant regardless of the proof circuit complexity since the proofs are always represented through three elliptic curves. The verification keys remain stable between 2-3KB where *(re)proposing bid* has the biggest verification key since it takes more number of public input parameters. On the other hand, the proving keys are constantly changing with respect to the implementation complexity where we observe that *(re)proposing*

bid is the most complex proof with 69.2MB while *verifying bid aggregation* is the least complex with 34.5MB. These results are tolerable from perspective since the off-chain proving key storage is not expensive while the on-chain verification keys and proofs are already short.

Table 6.6. *PMTBS* zero-knowledge proof artifact size.

Proof	**Proving Key Size**	**Verification Key Size**	**Proof Size**
Depositing Multi-Tokens	34.0MB	2KB	1KB
Withdrawing Multi-Tokens	33.9MB	2KB	1KB
(Re)proposing Bid	69.2MB	3KB	1KB
Verifying Bid Aggregation	34.5MB	2KB	1KB
Bartering Multi-Tokens	68.9MB	3KB	1KB

7. EXTENSION TO MULTI-OBJECTIVE BARTERING

In this chapter, we extend the previous bartering problem into the privacy-preserving multi-objective bartering problem by specifying objectives, constraints and requirements. For the given problem, we propose our novel privacy-preserving multi-objective bartering approach (i.e. *zkMOBF*) that considers two different objectives at the same time through four phases: (i) detecting negative cycles (i.e. solutions) with Bellman-Ford, (ii) evaluating feasible solutions with the objectives, (iii) ranking the feasible solutions with non-dominated sorting and (iv) decision-making with utility function. In our empirical study, we measure proof generation/verification times, proof artifact sizes and blockchain gas consumption for three bartering scenarios. The study justifies the validity and applicability of our approach.

7.1. Problem Definition

Privacy-preserving multi-objective bartering-based trading refers to a secure multi-party computation where a group of blockchain addresses (i.e. barterers) collectively exchanges a token in return for another token by considering several optimization objectives simultaneously while still protecting the privacy of their balances and bids. Note that this problem considers single-token single-instance bartering where a barterer can propose a single instance of a single token to supply and a single instance of a single token to demand. The problem (as in *PMTBS* protocol) has two kinds of information asymmetry: (i) each address possesses their own private balance while the other addresses need it to verify the correctness of the bartering and (i) each address knows only their own bid while the other barterers not having this bid need it to exchange tokens. To define this problem, we mostly rely on the definitions of the previous bartering problem. We extend the definition of bid by integrating a cost as:

$$\phi_i \quad : \quad \langle \phi_i^-, \phi_i^+, \phi_i^{cost} \rangle \tag{7.1}$$

where $\phi_i^- \in \mathcal{T}$ is the token to be supplied, $\phi_i^+ \in \mathcal{T}$ is the token to be demanded and ϕ_i^{cost} is the cost of the bid ϕ_i while Φ refers to the set of all the bids. We assume that each bidder u_i can propose one bid ϕ_i at most at a time.

However, the main difference from the previous bartering problem lies in the definition of valid bartering solution, (see Section 6.1). While the previous problem enforces a very strict condition to satisfy all the bids available by allowing barterers to repropose their bids with complying certain constraints (e.g. *under-demand*, *over-supply* requirements), the current problem relaxes this condition where a subset of bids can be selected under multiple objectives as long as they together form a complete cycle. This implicitly changes the type of problem from the satisfaction to the optimization problem. We define the privacy goals of this problem with the following way:

$$\forall u_j \neq u_i \quad : \quad \Pr[\theta_i \mid \text{view}(u_j)] = \Pr[\theta_i] \tag{7.2}$$

$$\forall u_j \neq u_i \quad : \quad \Pr[\phi_i \mid \text{view}(u_j)] = \Pr[\phi_i] \tag{7.3}$$

for *balance* privacy and *bid* privacy respectively; and our privacy objectives are:

$$\min_{u_j} \quad \mathtt{I}(\text{view}(u_j); \theta_i), \quad \forall u_j \neq \{u_i\} \tag{7.4}$$

$$\min_{u_j} \quad \mathtt{I}(\text{view}(u_j); \phi_i), \quad \forall u_j \neq \{u_i\} \tag{7.5}$$

for balance and bid, respectively.

The multi-objective modeling of this problem simply evaluates the set of feasible solutions with several objectives simultaneously by mapping each solution x_j : $\langle x_{j0}, x_{j1}, \ldots, x_{ji}, \ldots, x_{j,N-1} \rangle$ and $x_{ji} \in \{0, 1\}$ from the decision space to a point in the objective space with the following way:

$$max. \qquad F(x_j) = \{F_0(x_j), F_1(x_j)\} \tag{7.6}$$

$$F_0(x_j) = \sum_{i=0}^{N-1} x_{ji} \tag{7.7}$$

$$F_1(x_j) = B - \sum_{i=0}^{N-1} x_{ji} \cdot \phi_i^{cost} \tag{7.8}$$

$$s.t. \qquad \sum_{i=0}^{N-1} \phi_i^{-} \cdot x_{ji} \geq \sum_{i=0}^{N-1} \phi_i^{+} \cdot x_{ji} \tag{7.9}$$

where F refers to the set of objectives and B is the total budget (which is globally shared among bidders). Equation (7.6) refers to the multi-objective optimization of two objectives where Equation (7.7) defines the first objective as the maximization of the number of total bids included in the solution while Equation (7.8) defines the second objective as the maximization of the total budget left after the costs of the bids included are covered. Finally, the constraint in Equation (7.9) implies that the total number of tokens supplied must be at least equal to or greater than the total number of tokens demanded in the solution. A solution x_j is considered as feasible if it satisfies this condition. Furthermore, each solution is assumed to have only a single simple cycle. In the problem definition, there is no explicit supply constraint since we assume that the bidders already have sufficient balances to propose bids. Note that the objectives are conflicting since satisfying more numbers of bids leads more bid costs to be accumulated. This results in a pareto-frontier with only optimal solutions to be formed. Refer to Section 2.5 for further information.

We present a simple bartering illustration in Figure 7.1 where there exist six bids at total as $\Phi : \{\phi_0, \phi_1, \phi_2, \phi_3, \phi_4, \phi_5\}$. Out of these bids, there are two different cycles (i.e. bartering solutions) as $x_0, x_1 \in S$ where S is the set of feasible solutions. The first objective of x_0 is the number of bids satisfied as 3 while the second objective is the budget left as $10 - (1 + 1 + 1) = 7$ where the total budget is taken as 10. We can compute the objectives of x_1 in similar fashion as $10 - (3 + 3) = 4$. The objective space of (F_1, F_2) with the points corresponding to these two solutions shows that x_0 is non-dominated by outperforming x_1 at every objective. Note that the set of non-dominated solutions constitutes pareto-optimal set, POS [61].

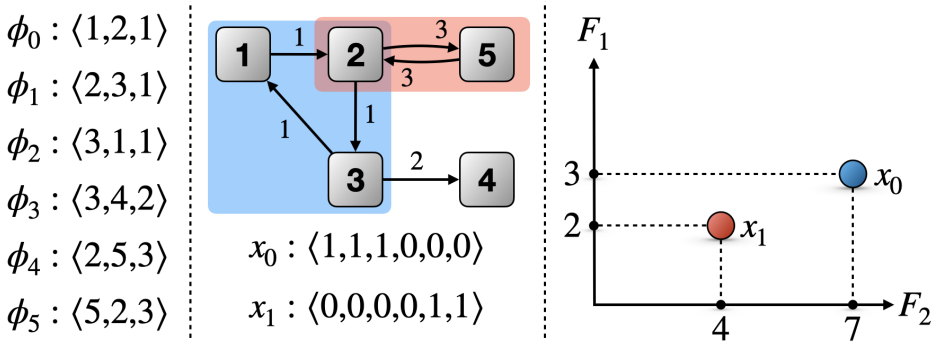


Figure 7.1. A simple illustration for multi-objective bartering with six bids $(\phi_0, \phi_1, \phi_2, \phi_3, \phi_4, \phi_5)$

7.2. System Architecture

For the given problem in Section 7.1, we develop a privacy-preserving multi-objective bartering approach on blockchain with zero-knowledge proof (i.e. *zkMOBF*). The approach has four phases as: (i) detecting cycles on bid graph with the Bellman-Ford algorithm, (ii) evaluating feasible cycles (i.e. solutions) with the objectives, (iii) ranking feasible solutions with pareto-domination and (iv) decision-making with utility function for the final solution. This approach is: (i) *privacy-preserving* since it protects the privacy of bids throughout the calculations, (ii) *multi-objective* since it considers the optimization of two objectives at the same time, (iii) *publicly-verifiable* since the off-chain proof generation for the calculations can be verified on-chain, (iv) *non-interactive* since it does not require communication during proof generation or verification. The algorithm for our approach is shown in Figure 7.2.

1: Find cycles (i.e. solutions x_j) with Bellman-Ford algorithm
2: **for** each solution x_j **do**
3: Add x_j to set of feasible solutions S as: $S \leftarrow S \cup x_j$
4: Evaluate x_j with the first objective F_1 as: $F_1(x_j)$
5: Evaluate x_j with the second objective F_2 as: $F_2(x_j)$
6: **for** each solution $x_j \in S$ **do**
7: **for** each solution $x'_j \in S$ that is $x_j \neq x'_j$ **do**
8: Compare x_j and x'_j through pareto-domination
9: Add x_j to pareto-optimal set POS if it is non-dominated
10: **for** each solution $x_j \in POS$ **do**
11: Apply utility function over x_j to compute the final score
12: **if** x_j has better score than the best score **then**
13: Select x_j as the best solution as x^*
14: **return** the best solution found x^*

Figure 7.2. Multi-objective bartering implementation in ZoKrates framework [27].

7.2.1. Detecting Cycles with Bellman-Ford Algorithm

The bids of the bidders constitute the global bid graph altogether. The constraint given in Equation (7.9) implicitly limits the valid bartering solutions over this graph to be complete cycles. More intuitively, the token demanded in the last bid in the cycle must be supplied from the first bid in the same cycle. This lets us to use a cycle detection algorithm (i.e. Bellman-Ford) to find the cycles available in the graph. The calculations for this phase correspond to Line 1 in Fig. (7.2) where it traverses every node in the graph as start node by setting distances to infinity except the start node itself. Then, it iteratively relaxes all the edges to find the shortest paths by updating the distances. After its completion, in case it could not find any further improvement over the distances, it means there is no cycle. Otherwise, it simply backtracks the previous nodes to construct a complete cycle. This cycle is included into the set of feasible solutions in Line 3. This can be seen in Fig. (7.3) where the cycles are highlighted.

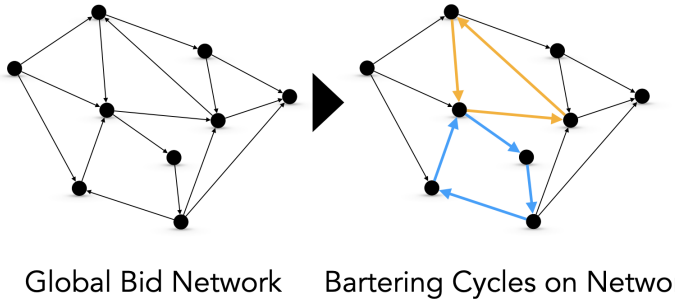


Figure 7.3. Negative cycle detection with Bellman-Ford

7.2.2. Evaluating Feasible Cycles with Objectives

The feasible cycles (i.e. feasible solutions) found in the given global bid graph must be evaluated with respect to the existing objectives of the multi-objective optimization to find their qualities. We already mathematically define our two objectives in Equation (7.7) and Equation (7.8). The calculations for this phase correspond to Lines 4-5 in Fig. (7.2). For the first objective, it simply counts the number of the bids satisfied for every feasible solution. For the second objective, it sums the costs of the bids and later subtracts the total cost from the available budget B for every feasible solution. This phase is especially important to transform the solutions (as collection of decision variables) in the decision space to the points in the objective space. This transformation can be seen in Fig. (7.4).

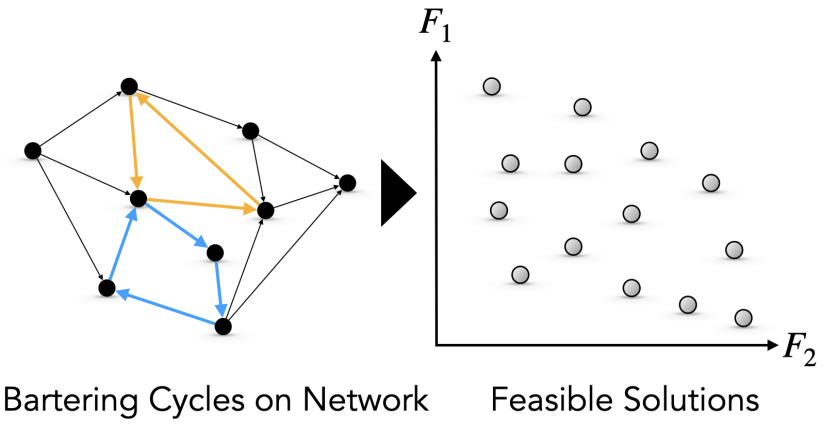


Figure 7.4. Transformation of solutions to objective values

7.2.3. Ranking Feasible Solutions with Non-Dominated Sorting

In single-objective optimization, the solutions can be ranked by simply sorting their values on that objective ascendingly or descendingly with respect to the type of the problem (i.e. minimization or maximization). However, multi-objective optimization requires a more sophisticated ranking mechanism where we apply *pareto-domination* in this work. This technique is based on pairwise solution comparison where a solution x_1 dominates another solution x_2 in case x_1 is no worse than x_2 for all the objectives and x_1 is strictly better than x_2 in at least one objective, $x_2 \prec x_1$. The calculations for this phase correspond to Lines 6-9 in Fig. (7.2) where it compares every solution x_j with all the other feasible solutions and marks x_j as non-dominated if there is not another solution that dominates x_j. The set of all non-dominated solutions constitutes POS in Line 9. This can be seen in Fig. (7.5) where the blue solutions are non-dominated.

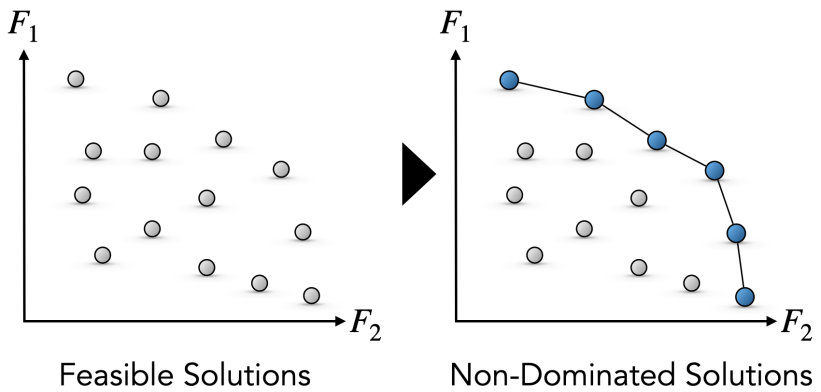


Figure 7.5. Non-dominated ranking of solutions for POS

7.2.4. Decision-Making for Final Solution with Utility Function

In our work, the third phase generates a pareto-optimal set (i.e. POS) that consists of only the non-dominated solutions by cleansing the useless solutions for our problem. The selection of the final solution to be applied over the bids from this set requires an additional phase. For this phase, we incorporate a utility function A that represents a group of probabilities to measure user preferences. These preferences

simply show how important an objective is with respect to the other objective(s). Since we have two objectives in our problem, we can represent this function as $A : \langle \alpha_1, \alpha_2 \rangle$ where α_1 and α_2 are the probabilities for the first and second objectives, respectively by satisfying the condition of $\alpha_1 + \alpha_2 = 1$. The calculations for this phase correspond to Lines 10-13 in Fig. (7.2) where it applies the preferences over the objectives for every optimal solution. It iterates over all the optimal solutions to find the single best solution with the best score in Line 13. This is seen in Fig. (7.6) where the green solution is shown as the best.

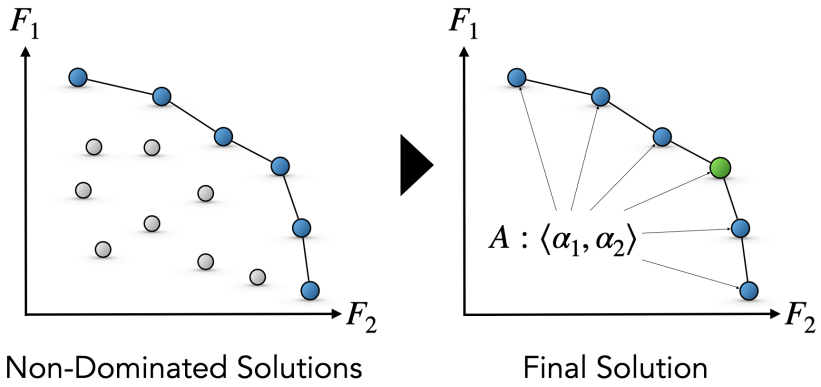


Figure 7.6. Selection of best solution from optimal solutions

7.3. Current Limitation

In this section, we evaluate potential limitations of the *zkMOBF* approach including: (i) bartering type, (ii) lack of complete system and (iii) performance issues. We aim to address these limitations in the future. We refer the limitations as follows:

- Bartering Type: The *zkMOBF* approach supports only single-instance single-token bartering where it limits the number of tokens barterers can trade at once.
- Lack of Complete System: Note that the *zkMOBF* approach includes only the implementation of the zero-knowledge proof model while omitting the other components including the smart contract and web interface models. Hence, it does not address how barterers register to the system, how they propose bids and how

these bids are properly aggregated.

- Performance Issues: The *zkMOBF* approach has high computational complexity where it generates large proving keys to be stored and processed, which will be further discussed in the experimental study.

7.4. Experimental Study

We use the *ZoKrates* framework [27] to implement *zkMOBF* over zero-knowledge proof. The proof generations and verifications are performed off-chain on the command line. The smart contracts corresponding to the proofs are automatically generated by the framework itself and ready to be deployed on Ethereum Sepolia only for proof verification. During our experiments, we consider three increasingly-complex graphs as small (with 10 bids), medium (with 30 bids) and large (with 50 bids). The open-source implementation of *zkMOBF* is available in our website [55] for further inspection and experimental reproducibility. The experiments are carried out on MacBook Air with M2 chip, 8 GB memory and 8 cores.

7.4.1. Zero-Knowledge Proof Generation/Verification Times

In this experiment, we measure the proof generation and verification times on the command line with ten independent runs. The results of the experiment are given in Table 7.1 for the small, medium and large bid graphs. From the table, we simply observe that the time to generate proof increases with the increasing graph complexity (e.g. from 23.67 seconds for small to 188.75 seconds for large). But, the standard deviations between runs are quite low (e.g. 0.01, 1.2 and 1.8 seconds for small, medium and large, respectively). On the other hand, we observe that the proof verification remains constant regardless of the graph complexity. This implies that the proof generation is computationally more expensive than the proof verification, especially for complex instances. Hence, *ZoKrates* strategically moves proof generation out of blockchain to external nodes while still keeping proof verification on-chain.

Table 7.1. *zkMOBF* zero-knowledge proof generation/verification times.

	Proof Generation			Proof Verification		
Run	**Small**	**Medium**	**Large**	**Small**	**Medium**	**Large**
1	23.5	90.1	189.6	0.013	0.013	0.013
2	23.7	91.1	188.4	0.016	0.012	0.012
3	23.7	92.7	188.9	0.013	0.015	0.012
4	23.6	92.3	187.8	0.013	0.012	0.012
5	23.8	90.2	189.7	0.012	0.013	0.013
6	23.7	88.9	188.5	0.012	0.013	0.012
7	23.8	90.4	189.0	0.012	0.014	0.012
8	23.6	89.1	191.4	0.013	0.013	0.013
9	23.6	89.9	184.4	0.015	0.013	0.013
10	23.7	90.5	189.8	0.013	0.012	0.016

7.4.2. Proof Artifact Size

In this experiment, we measure the size of several proof artifacts including the proof circuit, the proving key, the verification key and the proof itself. The results of the experiment are given in Table 7.2 for the small, medium and large bid graphs. From the table, we observe that the number of constraints in circuits and the proving key size increase with the increasing graph complexity (e.g. from +1.5M for small to +8.5M for large in circuits and from 0.63GB for small to 3.78GB for large in proving key). On the other hand, verification keys and proofs remain constant all the time. This is beneficial for blockchain applications since verification keys and proofs are processed on-chain while proving keys are stored off-chain. For the large graph, the proving key is x2.7M larger than the verification key while the proof is the shortest.

7.4.3. Blockchain Gas Consumption

In this experiment, we simply deploy the smart contracts that *ZoKrates* automatically generates to blockchain. The structures of these contracts for small, medium

Table 7.2. *zkMOBF* zero-knowledge proof artifact size.

Scenario	Circuit	Proving Key	Verification Key	Proof
Small	1,557,109	0.63GB	1.4KB	0.8KB
Medium	4,959,472	2.11GB	1.4KB	0.8KB
Large	8,515,531	3.78GB	1.4KB	0.8KB

and large graphs are basically the same except the verification keys. These keys result from the one-time setup where they are perfectly matched with their corresponding proving keys. According to our experiments, the smart contracts need approximately 1,069,137, 1,069,101 and 1,069,149 gas units (i.e. approximately $3.51) to be deployed on Ethereum Sepolia for the small, medium and large bartering instances, respectively. The results of the experiment are given in Table 7.3.

Table 7.3. Blockchain gas aonsumption of *zkMOBF* approach.

Function	Small	Medium	Large
Deploying Contract	1,069,137	1,069,101	1,069,149

8. CONCLUSION AND FUTURE WORK

In this thesis, we propose the following privacy-preserving protocols on blockchain by using zero-knowledge proof schemes: (i) *PTTS* which is a privacy-preserving payment protocol by enabling two different parties privately transfer tokens on a decentralized network which keeps their balances and transaction details private, (ii) *PVSS* which is a privacy-preserving data aggregation protocol that allows multiple parties to compute the global aggregation (i.e. sums) by privately collecting their individual data using hypercube networks, (iii) *PRFX* which is an extended version of the privacy-preserving data aggregation protocol that specializes in computing prefix aggregations (i.e. sums) by using the underlying *PVSS* technique, (iv) *PMTBS* which is a privacy-preserving multi-token bartering protocol by enabling barterers to exchange a set of their own tokens in return for another set of tokens while keeping their balances and bids private and finally (v) *zkMOBF* which is a more general multi-objective bartering protocol that extends *PMTBS*.

More specifically, *PTTS* is a (i) privacy-preserving, (ii) decentralized, (iii) publicly-verifiable, (iv) trustless and (v) partially non-interactive payment protocol where it supports three system actors as developer to deploy contracts, sender to deposit tokens and receiver to withdraw tokens. The protocol is built upon three inter-connecting layers including (i) zero-knowledge proof model to generate and verify proofs, (ii) smart contract model to serve as escrow mechanism to store payments and (iii) web interface model to provide this payment service to parties by abstracting the protocol complexities. The protocol runs over five phases as (i) *deploying contracts* to deploy contracts, (ii) *requesting consent* to ask consent from receiver, (iii) *granting consent* to grant consent to sender if asked, (iv) *depositing tokens* to deposit tokens to the contract pool to temporarily store and (v) *withdrawing tokens* to later withdraw the tokens from the contract pool. *PTTS* uses three contracts as the main *payment* contract along with two proof-verifying contracts. Inspired from this payment protocol, we propose a novel balance range disclosure attack to the privacy-preserving payment systems where we

model these payment transactions as minimum cost flow networks to be able to find lower and upper bounds blockchain addresses may have even if they are not public. We perform a series of experiments by varying the number of addresses and transactions and by varying transaction leakage ratio to justify its applicability in case an adversarial organization would collect a certain portion of privacy-preserving payments.

PVSS is a (i) privacy-preserving, (ii) decentralized, (iii) collective (iv) publicly-verifiable, (v) trustless, (vi) partially non-interactive and (vii) scalable data aggregation protocol where it supports only two system actors as developer to deploy contracts and aggregators to aggregate their individual data. The protocol uses (i) zero-knowledge proof model to generate aggregation proofs over two hypercube networks, (ii) smart contract model to orchestrate the hypercube communication steps iteratively by verifying the proofs and (iii) web interface model to provide this service to aggregators. The protocol runs over four phases as (i) *deploying contracts* to deploy contracts, (ii) *registering* to initially commit their data, (iii) *submitting aggregation* to submit their current aggregations over two hypercube networks and (iv) *verifying aggregation* to verify the correctness of their off-chain aggregations based on their submissions from the previous phase. *PVSS* uses two contracts as the main *aggregation* contract and one proof-verifying contract. The main drawback of *PVSS* is that it heavily relies on hypercube network topology for submitting aggregations, which limits the number of aggregators to be supported to a certain extent (i.e. only 2^d number of aggregators in d-dimension). Therefore, we extend *PVSS* to be able to support any arbitrary number of aggregators by proposing three communication techniques over incomplete hypercube networks as (i) *node multiplexing* to fill the blank positions with the existing aggregators in the networks, (ii) *topological recursing* to recursively collapse varying size of number of complete sub-hypercube networks until final sub-hypercube is formed, (iii) *data splitting* to split data into multiple chunks and distribute these chunks into blank positions. In addition to the first extension, we make the second extension of *PVSS* to support prefix aggregations as well in the name-space of *PRFX* where we apply it to the privacy-preserving delegation scheme by integrating Euler Tour Technique.

PMTBS is a (i) privacy-preserving, (ii) decentralized, (iii) collective, (iv) ascending auction-based, (v) multi-token, (vi) publicly-verifiable, (vii) trustless, (viii) partially non-interactive and (ix) scalable bartering protocol where it supports two system actors as developer to deploy contracts and barterers to barter tokens. The protocol uses (i) zero-knowledge proof model to generate proofs for proposing bids, aggregating these bids and later bartering tokens with respect to the bartering solution, (ii) smart contract model to manage barterer balances and bids throughout the protocol phases and (iii) web interface model to provide this service to barterers. The protocol runs over six phases as (i) *deploying contracts* to deploy contracts on-chain, (ii) *proposing bid* to propose initial bids by depositing the set of tokens supplied in bids, (iii) *submitting bid aggregation* to submit current vector-based bid aggregations over two hypercube networks, (iv) *verifying bid aggregation* to verify the correctness of off-chain bid aggregations based on the bid submissions from the previous phase, (v) *reproposing bid* to propose bid again by following certain protocol rules if no bartering solution is found and (vi) *bartering tokens* to withdraw the set of tokens demanded in bids. *PMTBS* uses seven contracts as the main *payment* and *barter* contracts along with five proof-verifying contracts. The main drawback of *PMTBS* is that it requires the several bidding rounds to reach a bartering solution since the problem requires all the bids to be satisfied. Therefore, we extend *PMTBS* to *zkMOBF* to address this issue by only presenting the zero-knowledge proof model. This model has four phases as (i) detecting cycles (i.e. feasible solutions) using Bellman-Ford in global bid graph, (ii) evaluating feasible solutions with objectives, (iii) ranking solutions with pareto-domination and (iv) decision-making with utility function to select the best solution.

We evaluate these protocols with respect to their scalability in terms of the computational, communication and storage overheads, their security in terms of the potential attacks and the formal reduction proofs and their major limitations. For scalability, we count the number of proofs generated (regardless of proof type) to quantify computational overhead, the number of direct communication links (regardless of amount of data transmitted over these links) to quantify communication overhead and the number of entries in mappings and lists (regardless of amount of data stored on-chain). For

security, we mainly consider replay attack, preimage attack, Sybil attack and collusion attack while the reduction proofs reduces the security of the protocols into the security of zkSNARKs [16]. For experimental evaluation, we use two different blockchains as Ethereum and Avalanche Fuji test networks where both platforms support EVM-based smart contract execution. The experiments are performed on MacBook Pro Notebook where the protocol implementations are openly accessible for further inspection. To measure their performance, we use certain performance measures throughout the thesis including problem requirement verification analysis, blockchain gas consumption, zero-knowledge proof generation and verification times and zero-knowledge proof artifact size. Through these experiments, we justify the applicability of our protocols on several real-world problems.

The following future works can be done to enrich our work:

- Integration of more efficient hash functions. In the current protocols, we employ the SHA256 hash function while generating commitments and zero-knowledge proof. However, there exist novel and more efficient hash functions as Poseidon [106] that is specifically designed for zkSNARKs [16]. This integration may benefit faster proof generation.
- Integration of other proof generation frameworks. We heavily use the ZoKrates framework [27] to generate and verify zero-knowledge proofs. In the literature, there exist other frameworks as well including Circom [24] which provides more flexibility and control over low-level operations for more complex computations.
- Cross-Chain Bartering. Our *PMTBS* and *zkMOBF* protocols consider bartering only for single blockchain, which may hinder their mass adoption by limiting the number of potential barterers. Hence, our protocols can be enhanced to support bartering across several blockchains. However, there are certain challenges waiting to be addressed: (i) proof system compatibility across chains, (ii) lack of token standardization across chains, (iii) fairness and atomicity of bartering and (iv) design of proper relay mechanism for cross-chain communication.

REFERENCES

1. Nakamoto, S., "Bitcoin: A Peer-to-Peer Electronic Cash System", `https://assets.pubpub.org/d8wct41f/31611263538139.pdf`, 2008, accessed on May 8, 2025.

2. Buterin, V. and G. Wood, "Ethereum White Paper", *GitHub repository*, Vol. 1, pp. 22–23, 2013.

3. Avalanche, "Fuji Blockchain", `https://www.avax.network`, 2020, accessed on May 8, 2025.

4. Silvano, W. F. and R. Marcelino, "Iota Tangle: A Cryptocurrency to Communicate Internet-of-Things Data", *Future generation computer systems*, Vol. 112, pp. 307–319, 2020.

5. Sasson, E. B., A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer and M. Virza, "Zerocash: Decentralized Anonymous Payments from Bitcoin", *IEEE symposium on security and privacy, California, USA*, pp. 459–474, 2014.

6. Ali, F. S., O. Bouachir, Ö. Özkasap and M. Aloqaily, "Synergychain: Blockchain-assisted Adaptive Cyber-Physical P2P Energy Trading", *IEEE Transactions on Industrial Informatics*, Vol. 17, No. 8, pp. 5769–5778, 2020.

7. Shayan, M., C. Fung, C. J. Yoon and I. Beschastnikh, "Biscotti: A Blockchain System for Private and Secure Federated Learning", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 32, No. 7, pp. 1513–1525, 2020.

8. Yavuz, E., A. K. Koç, U. C. Çabuk and G. Dalkılıç, "Towards Secure E-voting using Ethereum Blockchain", *6th International Symposium on Digital Forensic and Security (ISDFS), Antalya, Turkey*, pp. 1–7, 2018.

9. Ozturan, C., "Barter Machine: An Autonomous, Distributed Barter Exchange on the Ethereum Blockchain", *Ledger*, Vol. 5, pp. 20–35, 2020.

10. Galal, H. S. and A. M. Youssef, "Verifiable Sealed-bid Auction on the Ethereum Blockchain", *Financial Cryptography and Data Security: International Workshops, BITCOIN, VOTING, and WTSC, Nieuwpoort, Curacao*, pp. 265–278, 2019.

11. Li, W., H. Guo, M. Nejad and C.-C. Shen, "Privacy-Preserving Traffic Management: A Blockchain and Zero-Knowledge Proof Inspired Approach", *IEEE access*, Vol. 8, pp. 181733–181743, 2020.

12. Ghribi, E., T. T. Khoei, H. T. Gorji, P. Ranganathan and N. Kaabouch, "A Secure Blockchain-based Communication Approach for Uav Networks", *IEEE international conference on electro information technology (EIT), Illinois, USA*, pp. 411–415, 2020.

13. Hopwood, D., S. Bowe, T. Hornby and N. Wilcox, "Zcash Protocol Specification", *GitHub: San Francisco, CA, USA*, Vol. 4, No. 220, p. 32, 2016.

14. Williamson, Z. J., "The Aztec Protocol", `https://github.com/AztecProtocol/AZTEC`, 2018, accessed on May 8, 2025.

15. Goldwasser, S., S. Micali and C. Rackoff, "The Knowledge Complexity of Interactive Proof-Systems", *SIAM Journal on computing*, Vol. 18, No. 1, pp. 186–208, 1989.

16. Ben-Sasson, E., A. Chiesa, E. Tromer and M. Virza, "Succinct {Non-interactive} Zero Knowledge for a Von Neumann Architecture", *23rd USENIX Security Symposium (USENIX Security 14), California, USA*, pp. 781–796, 2014.

17. Hou, D., J. Zhang, S. Huang, Z. Peng, J. Ma and X. Zhu, "Privacy-Preserving Energy Trading using Blockchain and Zero Knowledge Proof", *IEEE international*

conference on blockchain (blockchain), Espoo, Finland, pp. 412–418, 2022.

18. Heiss, J., E. Grünewald, S. Tai, N. Haimerl and S. Schulte, "Advancing Blockchain-based Federated Learning through Verifiable Off-Chain Computations", *IEEE International Conference on Blockchain (Blockchain), Espoo, Finland*, pp. 194–201, 2022.

19. Wu, W., E. Liu, X. Gong and R. Wang, "Blockchain based Zero-knowledge Proof of Location in IoT", *IEEE International Conference on Communications (ICC), Dublin, Ireland*, pp. 1–7, 2020.

20. Fatz, F., P. Hake and P. Fettke, "Confidentiality-Preserving Validation of Tax Documents on the Blockchain", *Wirtschaftsinformatik (Zentrale Tracks), Potsdam, Germany*, pp. 1262–1277, 2020.

21. Qiu, Z., Z. Xie, X. Jiang, C. Ran and K. Chen, "Novel Blockchain and Zero-Knowledge Proof Technology-Driven Car Insurance", *Electronics*, Vol. 12, No. 18, p. 3869, 2023.

22. Sun, X., F. R. Yu, P. Zhang, Z. Sun, W. Xie and X. Peng, "A Survey on Zero-knowledge Proof in Blockchain", *IEEE network*, Vol. 35, No. 4, pp. 198–205, 2021.

23. Scipr-Lab, "libsnark", `https://github.com/scipr-lab/libsnark`, 2014, accessed on May 8, 2025.

24. iden3, "circom", `https://github.com/iden3/circom`, 2018, accessed on May 8, 2025.

25. Kosba, A., C. Papamanthou and E. Shi, "xJsnark: A Framework for Efficient Verifiable Computation", *IEEE Symposium on Security and Privacy (SP), California, USA*, pp. 944–961, 2018.

26. iden3, "snarkjs", `https://github.com/iden3/snarkjs`, 2018, accessed on May 8, 2025.

27. Eberhardt, J. and S. Tai, "Zokrates-scalable Privacy-preserving Off-chain Computations", *IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), Halifax, Canada*, pp. 1084–1091, 2018.

28. Schnorr, C.-P., "Efficient Identification and Signatures for Smart Cards", *Advances in Cryptology—CRYPTO'89 Proceedings 9, California, USA*, pp. 239–252, 1990.

29. Fiat, A. and A. Shamir, "How to Prove Yourself: Practical Solutions to Identification and Signature Problems", *Conference on the theory and application of cryptographic techniques, California, USA*, pp. 186–194, 1986.

30. Guillou, L. C. and J.-J. Quisquater, "A Paradoxical Indentity-based Signature Scheme Resulting from Zero-knowledge", *Advances in Cryptology—CRYPTO'88: Proceedings 8, California, USA*, pp. 216–231, 1990.

31. Ben-Sasson, E., I. Bentov, Y. Horesh and M. Riabzev, "Scalable, Transparent, and Post-Quantum Secure Computational Integrity", *Cryptology ePrint Archive*, Vol. 046, pp. 1–83, 2018.

32. Bünz, B., J. Bootle, D. Boneh, A. Poelstra, P. Wuille and G. Maxwell, "Bulletproofs: Short Proofs for Confidential Transactions and More", *IEEE symposium on security and privacy (SP), California, USA*, pp. 315–334, 2018.

33. Bonyadi, M. R. and Z. Michalewicz, "Particle Swarm Optimization for Single Objective Continuous Space Problems: A Review", *Evolutionary computation*, Vol. 25, No. 1, pp. 1–54, 2017.

34. Deb, K., K. Sindhya and J. Hakanen, "Multi-Objective Optimization", *Decision sciences*, pp. 161–200, CRC Press, Florida, USA, 2016.

35. Ishibuchi, H., N. Tsukamoto and Y. Nojima, "Evolutionary Many-objective Optimization: A Short Review", *IEEE congress on evolutionary computation (IEEE world congress on computational intelligence), Hong Kong, China*, pp. 2419–2426, 2008.

36. Ismayilov, G. and H. R. Topcuoglu, "Neural Network Based Multi-objective Evolutionary Algorithm for Dynamic Workflow Scheduling in Cloud Computing", *Future Generation computer systems*, Vol. 102, pp. 307–322, 2020.

37. Singh, N. and M. Vardhan, "Multi-objective Optimization of Block Size Based on Cpu Power and Network Bandwidth for Blockchain Applications", *Proceedings of the Fourth International Conference on Microelectronics, Computing and Communication Systems, Ranchi, India*, pp. 69–78, 2021.

38. Oudani, M., "A Combined Multi-objective Multi Criteria Approach for Blockchain-based Synchromodal Transportation", *Computers & Industrial Engineering*, Vol. 176, p. 108996, 2023.

39. Zanbouri, K., M. Darbandi, M. Nassr, A. Heidari, N. J. Navimipour and S. Yalcın, "A Gso-based Multi-Objective Technique for Performance Optimization of Blockchain-based Industrial Internet Of Things", *International Journal of Communication Systems*, Vol. 37, No. 15, p. e5886, 2024.

40. Wu, J., J. Liu, W. Chen, H. Huang, Z. Zheng and Y. Zhang, "Detecting Mixing Services via Mining Bitcoin Transaction Network with Hybrid Motifs", *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, Vol. 52, No. 4, pp. 2237–2249, 2021.

41. Bünz, B., S. Agrawal, M. Zamani and D. Boneh, "Zether: Towards Privacy in

a Smart Contract World", *International Conference on Financial Cryptography and Data Security, Sabah, Malaysia*, pp. 423–443, 2020.

42. Yousaf, H., G. Kappos and S. Meiklejohn, "Tracing Transactions Across Cryptocurrency Ledgers", *28th USENIX Security Symposium (USENIX Security 19), California, USA*, pp. 837–850, 2019.

43. Zhang, Z., W. Li, H. Liu and J. Liu, "A Refined Analysis of Zcash Anonymity", *IEEE Access*, Vol. 8, pp. 31845–31853, 2020.

44. Almalki, F. A. and B. O. Soufiene, "Eppda: An Efficient and Privacy-preserving Data Aggregation Scheme with Authentication and Authorization for Iot-based Healthcare Applications", *Wireless Communications and Mobile Computing*, Vol. 2021, No. 1, p. 5594159, 2021.

45. Fan, H., Y. Liu and Z. Zeng, "Decentralized Privacy-preserving Data Aggregation Scheme for Smart Grid Based on Blockchain", *Sensors*, Vol. 20, No. 18, p. 5282, 2020.

46. Mols, J. and E. Vasilomanolakis, "Ethvote: Towards Secure Voting with Distributed Ledgers", *International Conference on Cyber Security and Protection of Digital Services (Cyber Security), Dublin, Ireland*, pp. 1–8, 2020.

47. Wang, N., S. C.-K. Chau and Y. Zhou, "Privacy-preserving Energy Storage Sharing with Blockchain and Secure Multi-party Computation", *ACM SIGENERGY Energy Informatics Review*, Vol. 1, No. 1, pp. 32–50, 2021.

48. Liang, H. and W. Zhang, "A Barter and Combinatorial Auction based Hierarchical Resource Trade Mechanism for Cybertwin Network", *3rd International Conference on Hot Information-Centric Networking (HotICN), Hefei, China*, pp. 84–89, 2020.

49. Sevindik, V., "Blockchain based Resource Tokenization for Crowdfunding of Wire-

less Network Investment", *11th IFIP international conference on new technologies, mobility and security (NTMS), Paris, France*, pp. 1–5, 2021.

50. Kim, M., B. Hilton, Z. Burks and J. Reyes, "Integrating Blockchain, Smart Contract-tokens, and Iot to Design a Food Traceability Solution", *IEEE 9th annual information technology, electronics and mobile communication conference (IEMCON), Vancouver, Canada*, pp. 335–340, 2018.

51. Ismayilov, G. C., "PTTS", `https://github.com/GoshgarIsmayilov/PTTS`, 2022, accessed on May 8, 2025.

52. Ismayilov, G. C., "PVSS", `https://github.com/GoshgarIsmayilov/PVSS`, 2023, accessed on May 8, 2025.

53. Ismayilov, G. C., "PRFX", `https://github.com/GoshgarIsmayilov/PRFX`, 2023, accessed on May 8, 2025.

54. Ismayilov, G. C., "PMTBS", `https://github.com/GoshgarIsmayilov/PMTBS`, 2023, accessed on May 8, 2025.

55. Ismayilov, G. C., "zkMOBF", `https://github.com/GoshgarIsmayilov/zkMOBF`, 2025, accessed on May 8, 2025.

56. Foundation, E., "The ERC-20 Token Standard", `https://ethereum.org/en/developers/docs/standards/tokens/erc-20/`, 2015, accessed on May 8, 2025.

57. Foundation, E., "The ERC-721 Token Standard", `https://ethereum.org/en/developers/docs/standards/tokens/erc-721/`, 2018, accessed on May 8, 2025.

58. Foundation, E., "The ERC-1155 Token Standard", `https://ethereum.org/en/developers/docs/standards/tokens/erc-1155/`,

2018, accessed on May 8, 2025.

59. Ahuja, R. R., T. L. Magnanti and J. B. Orlin, *Network Flows: Theory, Algorithms, and Applications*, Prentice Hall, New Jersey, USA, 1993.

60. Bang-Jensen, J. and G. Z. Gutin, *Digraphs: Theory, Algorithms and Applications*, Springer Science & Business Media, Berlin, Germany, 2008.

61. Branke, J., *Multiobjective Optimization: Interactive and Evolutionary Approaches*, Vol. 5252, Springer Science & Business Media, Berlin, Germany, 2008.

62. Menezes, A. J., P. C. Van Oorschot and S. A. Vanstone, *Handbook of Applied Cryptography*, CRC press, Florida, USA, 2018.

63. Stallings, W., *Cryptography and Network Security, 4/E*, Prentice Hall, New Jersey, USA, 2006.

64. Günsay, E., *Zero-knowledge Range Proofs and Applications on Decentralized Constructions*, Master's Thesis, Middle East Technical University, Ankara, Turkey, 2021.

65. Goldreich[1], O., "A Taxonomy of Proof Systems", *Complexity Theory: Retrospective II*, Vol. 2, p. 109, 1997.

66. Goldreich, O., S. Micali and A. Wigderson, "Proofs that Yield Nothing But Their Validity or All Languages in NP have Zero-knowledge Proof Systems", *Journal of the ACM (JACM)*, Vol. 38, No. 3, pp. 690–728, 1991.

67. Sánchez Ortiz, E., *Zero-Knowledge Proofs Applied to Finance*, Master's Thesis, University of Twente, Enschede, Netherlands, 2020.

68. Tomescu, A., A. Bhat, B. Applebaum, I. Abraham, G. Gueta, B. Pinkas and A. Yanai, "Utt: Decentralized Ecash with Accountable Privacy", *Cryptology*

ePrint Archive, Vol. 452, pp. 1–66, 2022.

69. Campanelli, M. and M. Hall-Andersen, "Veksel: Simple, Efficient, Anonymous Payments with Large Anonymity Sets from Well-studied Assumptions", *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security, Nagasaki, Japan*, pp. 652–666, 2022.

70. Wüst, K., K. Kostiainen, N. Delius and S. Capkun, "Platypus: A Central Bank Digital Currency with Unlinkable Transactions and Privacy-preserving Regulation", *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, California, USA*, pp. 2947–2960, 2022.

71. Wijaya, D. A., J. Liu, R. Steinfeld and D. Liu, "Monero Ring Attack: Recreating Zero Mixin Transaction Effect", *17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE), New York, USA*, pp. 1196–1201, 2018.

72. Monero, "Monero", `https://www.getmonero.org`, 2014, accessed on May 8, 2025.

73. Ranbaduge, T., D. Vatsalan and P. Christen, "Secure Multi-party Summation Protocols: Are They Secure Enough Under Collusion", *Trans. Data Priv.*, Vol. 13, No. 1, pp. 25–60, 2020.

74. Karumba, S., V. Dedeoglu, R. Jurdak and S. S. Kanhere, "CypherChain: A Privacy-Preserving Data Aggregation Framework for Blockchain-Based DR Programs", *IEEE International Conference on Blockchain and Cryptocurrency (ICBC), Dublin, Ireland*, pp. 362–364, 2024.

75. Mouris, D. and N. Tsoutsos, "Masquerade: Verifiable Multi-party Aggregation with Secure Multiplicative Commitments", *ACM Transactions on Internet Tech-*

nology, Vol. 25, No. 1, pp. 1–31, 2021.

76. Damgård, I., V. Pastro, N. Smart and S. Zakarias, "Multiparty Computation from Somewhat Homomorphic Encryption", *Annual Cryptology Conference, California, USA*, pp. 643–662, 2012.

77. Singh, P., M. Masud, M. S. Hossain and A. Kaur, "Blockchain and Homomorphic Encryption-based Privacy-Preserving Data Aggregation Model in Smart Grid", *Computers & Electrical Engineering*, Vol. 93, p. 107209, 2021.

78. Liu, Z., K. Zheng, L. Hou, H. Yang and K. Yang, "A Novel Blockchain-assisted Aggregation Scheme for Federated Learning in IoT Networks", *IEEE Internet of Things Journal*, Vol. 10, No. 19, pp. 17544–17556, 2023.

79. Özturan, C., "Resource Bartering in Data Grids", *Scientific Programming*, Vol. 12, No. 3, pp. 155–168, 2004.

80. Ozturan, C., "Network Flow Models for Electronic Barter Exchanges", *Journal of Organizational Computing and Electronic Commerce*, Vol. 14, No. 3, pp. 175–194, 2004.

81. Özturan, C., "Used Car Salesman Problem: A Differential Auction-Barter Market", *Annals of Mathematics and Artificial Intelligence*, Vol. 44, pp. 255–267, 2005.

82. Qian, J. and N. Harnpornchai, "Online Barter Trade Model Based on Traceability by Blockchain Technology", *IEEE 4th International Conference on Electronic Communications, Internet of Things and Big Data (ICEIB), New Taipei City, Taiwan*, pp. 544–549, 2024.

83. Adams, H., N. Zinsmeister, M. Salem, R. Keefer and D. Robinson, "Uniswap v3 Core", `https://app.uniswap.org/whitepaper-v3.pdf`, 2021, accessed on May 8, 2025.

84. PancakeSwap, "PancakeSwap", `https://pancakeswap.finance`, 2020, accessed on May 8, 2025.

85. SushiSwap, "SushiSwap", `https://www.sushi.com/ethereum/swap`, 2020, accessed on May 8, 2025.

86. Frikken, K. and L. Opyrchal, "PBS: Private Bartering Systems", *International Conference on Financial Cryptography and Data Security, Cozumel, Mexico*, pp. 113–127, 2008.

87. Förg, F., D. Mayer, S. Wetzel, S. Wüller and U. Meyer, "A Secure Two-party Bartering Protocol using Privacy-Preserving Interval Operations", *12th Annual International Conference on Privacy, Security and Trust, Ontario, Canada*, pp. 57–66, 2014.

88. Foerg, F., S. Wetzel and U. Meyer, "Efficient Commodity Matching for Privacy-Preserving Two-Party Bartering", *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy, Arizona, USA*, pp. 107–114, 2017.

89. Wüller, S., W. Pessin, U. Meyer and S. Wetzel, "Privacy-preserving Two-party Bartering Secure against Active Adversaries", *14th Annual Conference on Privacy, Security and Trust (PST), Auckland, New Zealand*, pp. 229–238, 2016.

90. Wüller, S., U. Meyer and S. Wetzel, "Towards Privacy-preserving Multi-party Bartering", *Financial Cryptography and Data Security: International Workshops, WAHC, BITCOIN, VOTING, WTSC, and TA, Sliema, Malta*, pp. 19–34, 2017.

91. MetaMask, "MetaMask", `https://metamask.io/`, 2015, `https://metamask.io/`, accessed on May 8, 2025.

92. Boyd, S. and L. Vandenberghe, *Convex Optimization*, Cambridge university press, Cambridge, United Kingdom, 2004.

93. ZoKratesJS, "ZoKratesJS", `https://www.npmjs.com/package/zokrates-js`, 2015, accessed on May 8, 2025.

94. Foundation, E., "Solidity", `https://docs.soliditylang.org/`, 2014, accessed on May 8, 2025.

95. Foundation, E., "Remix Compiler", `https://remix.ethereum.org`, 2015, accessed on May 8, 2025.

96. EthersJS, "EthersJS Library", `https://docs.ethers.io/v5/`, 2019, accessed on May 8, 2025.

97. EccryptoJS, "EccryptoJS Library", `https://www.npmjs.com/package/eccrypto`, 2015, accessed on May 8, 2025.

98. Browserify, "Browserify Bundler", `https://browserify.org`, 2011, accessed on May 8, 2025.

99. Webpack, "Webpack Bundler", `https://webpack.js.org`, 2014, accessed on May 8, 2025.

100. Google, "OR-Tools", `https://developers.google.com/optimization`, 2015, accessed on May 8, 2025.

101. Kara, G. and C. Özturan, "Parallel Network Simplex Algorithm for the Minimum Cost Flow Problem", *Concurrency and Computation: Practice and Experience*, Vol. 34, No. 4, p. e6659, 2022.

102. Ismayilov, G. C. and C. Özturan, "Trustless Privacy-preserving Data Aggregation on Ethereum with Hypercube Network Topology", *Computer Communications*, Vol. 230, p. 108009, 2025.

103. Reif, J. H., *Synthesis of Parallel Algorithms*, Morgan Kaufmann, California, USA,

1993.

104. Foundation, E., "Geth (Go-Ethereum)", `https://geth.ethereum.org`, 2015, accessed on May 8, 2025.

105. Leighton, F. T., *Introduction to parallel algorithms and architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann, California, USA, 2014.

106. Grassi, L., D. Khovratovich, C. Rechberger, A. Roy and M. Schofnegger, "Poseidon: A new hash function for {Zero-Knowledge} proof systems", *30th USENIX Security Symposium (USENIX Security 21), online*, pp. 519–535, 2021.

107. Li, H. and W. Xue, "A Blockchain-Based Sealed-Bid e-Auction Scheme with Smart Contract and Zero-Knowledge Proof", *Security and Communication Networks*, Vol. 2021, No. 1, p. 5523394, 2021.

108. Gabizon, A., Z. J. Williamson and O. Ciobotaru, "Plonk: Permutations over Lagrange-Bases for Cecumenical Noninteractive Arguments of Knowledge", *Cryptology ePrint Archive*, Vol. 953, pp. 1–34, 2019.

109. Katseff, H. P., "Incomplete Hypercubes", *IEEE transactions on computers*, Vol. 37, No. 5, pp. 604–608, 1988.

110. Özer, A. H. and C. Özturan, "A Direct Barter Model for Course Add/Drop Process", *Discrete Applied Mathematics*, Vol. 159, No. 8, pp. 812–825, 2011.

111. Ismayilov, G. and C. Özturan, "PTTS: Zero-Knowledge Proof-based Private Token Transfer System on Ethereum Blockchain and Its Network Flow Based Balance Range Privacy Attack Analysis", *Journal of Network and Computer Applications*, Vol. 233, p. 104045, 2025.

112. Ismayilov, G. C. and C. Ozturan, "Incomplete Hypercube-based Algorithms for Privacy-Preserving Data Aggregation Using Zero-Knowledge Proofs", *6th Inter-*

national Conference on Blockchain Computing and Applications (BCCA), Dubai, UAE, pp. 519–524, 2024.

113. Gilbert, H. and H. Handschuh, "Security Analysis of SHA-256 and Sisters", *International workshop on selected areas in cryptography, Ontario, Canada*, pp. 175–193, 2003.

114. Ismayilov, G. C. and C. Ozturan, "PRFX: A Privacy-Preserving Prefix Summation Protocol on Blockchain with Zero-Knowledge Proof", *IEEE International Conference on Blockchain (Blockchain), Copenhagen, Denmark*, pp. 362–369, 2024.

115. iden3, "circom", `https://iden3.io`, 2018, accessed on May 8, 2025.

116. Ismayilov, G. C. and C. Ozturan, "PMTBS: A Blockchain-Based Privacy-Preserving Multi-Token Ascending Barter Auction System with Zero-Knowledge Proofs", Journal of Parallel and Distributed Computing (under review).

APPENDIX A: COPYRIGHT NOTICE

All visual content in the scope of this thesis is produced by the author from his own publications [102, 111, 112, 114, 116] and then reused in this thesis. This visual content whose copyrights are transferred to the publisher, are used in accordance with the current policy of the publisher on the reuse of text and visual content that is found in the web site of the publisher itself.