Privacy-Preserving Protocols on Blockchain

Hisham Galal

A Thesis

in

The Concordia Institute

for

Information Systems Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy (Information and Systems Engineering) at
Concordia University
Montréal, Québec, Canada

February 2022

Concordia University

School of Graduate Studies

This is to certify that the thesis prepared

By: **Hisham Shehata Galal Elsayed**

Entitled: **Privacy-Preserving Protocols on Blockchain**

and submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Information and Systems Engineering)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

_____ Chair
Dr. Bruno Lee

_____ External Examiner
Dr. Abdelhakim Senhaji Hafid

_____ External to Program
Dr. Wahab Hamou-Lhadj

_____ Examiner
Dr. Jeremy Clark

_____ Examiner
Dr. Mohammad Mannan

_____ Supervisor
Dr. Amr M. Youssef

Approved by _____
Dr. Mohammad Mannan, Graduate Program Director

February 9^{th}, 2022
Date of Defence

_____
Dr. Mourad Debbabi, Dean
Gina Cody School of Engineering and Computer Science

Abstract

Privacy-Preserving Protocols on Blockchain

Hisham Galal, Ph.D.

Concordia University, 2022

Blockchain is an evolving technology with the potential to reshape various industries. It is an immutable append-only distributed ledger that maintains the integrity and availability of its transactions. With blockchain, mutually distrusting parties can finally make transactions without relying on a trusted third party. Nevertheless, many organizations are reluctant to adopt it due to several issues such as privacy. More precisely, the inherent transparency of transactions in blockchain comes at the cost of privacy despite the use of pseudonymous identities. We design cryptographic protocols to improve the privacy of a set of decentralized applications utilizing blockchain.

The rapidly growing number of digital assets deployed over blockchain requires a convenient trading mechanism. Sealed-bid auctions are powerful trading tools due to their privacy advantages compared to their open-cry counterparts. However, the inherent transparency on the blockchain makes designing a sealed-bid auction a challenging task. We propose three protocols utilizing zero-knowledge proofs, trusted execution environments, and smart contracts to publicly verify the correctness of the auction winner while maintaining users' privacy.

In the first protocol, the auctioneer utilizes zero-knowledge proof of interval membership to prove the correctness of the auction winner without revealing the losing bids.

However, this protocol is expensive in verification cost and scales linearly with the number of users. To reduce the verification cost, we design a second protocol where the auctioneer utilizes an advanced zero-knowledge proving system with a constant verification complexity. Both protocols offer partial privacy as the auctioneer gets to know the actual values of bids. The third protocol provides complete privacy by utilizing a trusted execution environment to determine the auction winner without revealing the losing bids to any party. Furthermore, since this protocol relies on simple cryptographic primitives, it achieves the lowest verification cost with a constant complexity regardless of the number of bids.

Extending the work on sealed-bid auctions, we tackle a privacy problem in lit markets where all the information about bids and offers in the order book is visible to the public. While transparency helps the price discovery, it hurts financial institutions that trade large bulk orders. Therefore, we design a privacy-preserving periodic auction that hides limit-orders during the submission phase while preventing front-running and ensuring the correctness of market-clearing prices.

Next, we target a privacy problem in inter-bank payment systems. Banks transfer money and securities instantaneously on a gross basis by utilizing Real-Time Gross Settlement (RTGS) system. Central banks operate RTGS systems and require access to payment instructions of each local inter-bank. Accordingly, RTGS systems assume unconditional trust given to central banks, and they suffer from a single point of failure. Hence, we propose a decentralized netting protocol that ensures balance correctness while hiding the transferred amounts and recipients.

Finally, we switch gears to the booming Non-Fungible Tokens (NFTs) technology and tackle privacy issues with existing systems. NFTs are unique non-interchangeable digital assets verified and secured by blockchain technology. Current NFT standards lack privacy guarantees; hence any observer can trivially learn the whole NFT collection of an arbitrary user. Furthermore, popular marketplaces use public exchanges and auctions for trades which leak information about the trade parties and the payment amount for an NFT. We design Aegis as a protocol that adds privacy to NFTs ownership. More importantly, Aegis allows users to atomically swap NFTs for payment amounts while hiding the details of the transactions.

Acknowledgments

First and foremost, I would like to express my sincere gratitude to my supervisor, Dr. Amr Youssef. I am always indebted for his continuous support, encouragement, patience, guidance, and immense knowledge. I have learned a lot from him, and I'm very proud to have been his student.

I would like to sincerely thank the examining committee: Dr. Jermey Clark, Dr. Mohammed Mannan, and Dr. Wahab Hamou-Lhadj for their helpful insights, valuable discussion, and guidance throughout each milestone of the Ph.D. journey.

No words can express my gratitude to my beloved parents and siblings; you have always been loving, caring, and supporting in my entire life.

Lastly, but not least, there is a special person in my life, my beloved wife, to whom I owe a lot for her love and encouragement. Thank you for always being there for me and bringing our beautiful children: Larien and Adam. My dear little family, you mean everything to me.

Hisham Galal

Table of Contents

List of Figures

List of Tables

List of Acronyms

AES	Advanced Encryption Standard
AMM	Automated Market Maker
CCE	Consistent Commitment Encryption
CRS	Common Reference String
DL	Discrete Log
DDH	Decisional Diffie Hellman
ECDH	Elliptic Curve Diffie Hellman
ECDSA	Elliptic Curve Digital Signature Algorithm
ECIES	Elliptic Curve Integrated Encryption Scheme
EIP	Ethereum Improvement Proposal
EOA	Externally Owned Account
EPID	Enhanced Privacy ID
FIFO	First In First Out
FPSBA	First-Price Sealed-bid Auction
LSM	Liquidity Saving Mechanism
MCP	Market Clearing Price
MCV	Market Clearing Volume
MPC	Multi-party Computation
NIZK	Non-interactive Zero-Knowledge
OPE	Order Preserving Encryption

PPT	Probabilistic Polynomial Time
PRF	Pseudorandom Function
QE	Quote Enclave
RTGS	Real Time Gross Settlement
SHA	Secure Hash Algorithm
SGX	Software Guard eXtensions
TEE	Trusted Execution Environment
ZKP	Zero-Knowledge Proof
zkSNARK	Zero-Knowledge Succinct Non-interactive ARgument of Knowledge

Chapter 1

Introduction

1.1 Overview

Blockchain is an immutable append-only ledger storing transactions chronologically in a linked chain of blocks. With blockchain, mutually distrusting users can finally make transactions without relying on a trusted third party. Blockchain has the potential to reshape many industries and deploy complex decentralized protocols that were deemed hard in practice. For instance, there have been plenty of research proposals [20, 58, 64] to build cryptocurrency, yet without a notable adoption and success. On the other hand, using blockchain, Bitcoin [55] became the most popular and successfully deployed cryptocurrency. Additionally, distrusting users can run far more complex transactions beyond simple payments transfer. For instance, Ethereum is the pioneer blockchain platform that supports smart contracts with rich and expressive functionalities.

1.2 Motivation

Blockchain mainly relies on transparency as a feature to allow anyone to verify the integrity of transactions' records. However, this feature quickly became a tool for tracking users and compromising their privacy. Even with the use of *pseudonyms* as

users' identities, there are analysis techniques [8, 23, 48, 51] in practice that are robust enough to link and identify users' transactions.

In our research, we study how to enhance users' privacy on the blockchain. The easiest solution would be to modify the core design of blockchain technology. However, such an approach cannot reach a unanimous agreement in a decentralized environment. Hence, it can lead to hard forks. Furthermore, building a blockchain from scratch with better privacy may not prompt enough users to adopt it. Therefore, our research is mainly concerned with utilizing cryptographic protocols to enhance the privacy of decentralized applications on top of existing public blockchains, mainly on top of Ethereum.

1.3 Contributions

We study the privacy issue that is inherent in blockchain with public ledgers. The main focus is to design privacy-preserving protocols for decentralized applications running on Ethereum. Towards this objective, we utilize various cryptographic protocols, especially zero-knowledge proof systems. The contributions can be summarized as follows:

- We design three protocols [31–33] for publicly-verifiable sealed-bid auctions. The main focus is to hide the losing bids while publicly verifying the correctness of the winning bid. The protocols utilize zero-knowledge proof of range, zkSNARK proofs, and Intel SGX.

- We extend our work of sealed-bid auctions to design a privacy-preserving periodic auction protocol [35]. The motivation is to allow financial institutions to trade large orders without resorting to dark pools. The protocol utilizes Bulletproofs and a consistent commitment encryption scheme to prove the correctness of the market-clearing price while maintaining privacy.

- We address the privacy problem inherent in RTGS systems that settle inter-bank payments. We present a decentralized netting protocol [34] to resolve payment

instructions on a netting basis while hiding recipients and amounts. The protocol relies on zkSNARK proofs to ensure correctness in a privacy-preserving manner.

- We tackle the privacy issue inherent in standard NFT specifications. We present, Aegis a protocol that adds privacy to NFTs while being completely compatible with existing standards. Aegis allows users to swap NFTs for payments in a complete privacy-preserving manner.

- We implement open-source prototypes to assess the protocols' performance and feasibility.

Other research works conducted during the tenure of this Ph.D. have been published in [30, 72].

1.4 Thesis Outline

- Chapter 2 provides background on cryptographic primitive, blockchain, Ethereum, and smart contracts.

- Chapter 3 presents three protocols for publicly verifiable and privacy-preserving sealed-bid auctions.

- Chapter 4 introduces a periodic auction protocol as an alternative to dark pools favored by financial institutions.

- Chapter 5 provides a decentralized netting protocol for settling inter-bank payments without relying on central banks.

- Chapter 6 presents Aegis a protocol for adding privacy to NFTs and allowing users to trade them in a privacy-preserving manner.

- Chapter 7 provides the concluding remarks for the thesis and future work.

Chapter 2

Background

We provide an overview of cryptographic primitives, zero-knowledge proofs, Intel SGX, and Ethereum.

2.1 Cryptographic Primitives

2.1.1 Notation

We denote by 1^λ the security parameter in the unary representation. Let p and q be a large primes where $p = 2q + 1$, we define an elliptic curve group $\mathbb{G}$ of order q generated by g over a field $\mathbb{F}_p$. We assume the Decisional Diffie Hellman (DDH) problem, is intractable in $\mathbb{G}$. Let $x \xleftarrow{\$} \mathbb{Z}_q$ denote uniform sampling of an element x from a group $\mathbb{Z}_q$. We assume the adversary $\mathcal{A}$ runs in a probabilistic polynomial time (PPT). We use bold symbols to denote to vectors such as $\boldsymbol{a} = (a_1, ..., a_n)$ where n is the number of element Let $[N]$ denote to an interval of integers from 1 to N.

2.1.2 Digital Signatures

Definition 2.1 *(Digital Signature.) A digital signature scheme consists of a triple of efficient algorithms (`KeyGen, Sign, Verify`):*

- $(sk, pk) \leftarrow \texttt{KeyGen}(1^\lambda)$: *it is a PPT algorithm that takes a security parameter* 1^λ *as input; and it returns a signing key* sk *and a public key* pk*.*

- $\sigma \leftarrow \texttt{Sign}(sk, m)$: *it is a PPT algorithm that takes a signing key* sk *and a message* m *as input; and it returns a signature* σ*.*

- $0/1 \leftarrow \texttt{Verify}(pk, m, \sigma)$: *it is a deterministic algorithm that takes a public key* pk*, a message* m*, and a signature* σ *as input; and it returns 1 if* σ *is valid.*

2.1.3 ElGamal Encryption

Definition 2.2 *(ElGamal Encryption.) It consists of a triple of efficient algorithms (*`KeyGen, Enc, Dec`*):*

- $(x, y) \leftarrow \texttt{KeyGen}(\mathbb{G}, q, p, g)$*: it is a PPT algorithm that samples a private key* $x \xleftarrow{R} \mathbb{Z}_q$*, and generates a public key* $y = g^x$*.*

- $c \leftarrow \texttt{Enc}(m, r, y)$*: it is a PPT algorithm that encrypts a message* $m \in \mathbb{Z}_q$ *by the public key* y *and a randomness* $r \xleftarrow{\$} \mathbb{Z}_q$*; and it returns a ciphertext* $c = (c_1, c_2) = (g^r, g^m y^r)$*.*

- $g^m \leftarrow \texttt{Dec}(c, x)$*: it is a deterministic algorithm that decrypts the ciphertext* c *by the private key* x*, and outputs* $g^m \leftarrow c_2 \cdot c_1^{-x}$*. Recovering* m *from* g^m *requires discrete log brute-force which is affordable for small values (e.g., when* m *is a 32-bit value).*

2.1.4 Commitment Schemes

Definition 2.3 *(Commitment.) A non-interactive commitment scheme consists of a pair of PPT algorithms (*`Setup, Com`*):*

- $pp \leftarrow \texttt{Setup}(1^\lambda)$*: it is a PPT algorithm that generates public parameters* pp *for the commitment scheme given the security parameter* λ*.*

- $cm \leftarrow \mathtt{Com}_{pp}(x, r) : M_{pp} \times R_{pp} \rightarrow C_{pp}$: *it is a PPT algorithm that generates a commitment* $cm \in C_{pp}$ *for a message* $x \in M_{pp}$ *and a randomness* $r \xleftarrow{\$} R_{pp}$.

For ease of notation, we write $\mathtt{Com}$ instead of $\mathtt{Com}_{pp}$.

Definition 2.4 *(Homomorphic Commitments.) A homomorphic commitment scheme is a non-interactive commitment scheme where* M_{pp}, R_{pp} *and* C_{pp} *are all abelian groups and for* $x_1, x_2 \in M_{pp}, r_1, r_2 \in R_{pp}$, *we have*

$$\mathtt{Com}(x_1, r_1) \cdot \mathtt{Com}(x_2, r_2) = \mathtt{Com}(x_1 + x_2, r_1 + r_2)$$

Definition 2.5 *(Pedersen Commitment.) It is a homomorphic commitment scheme where* $M_{pp}, R_{pp} = \mathbb{Z}_q$ *and* $C_{pp} = \mathbb{G}$.

- $\mathtt{Setup}$: $g, h \xleftarrow{\$} \mathbb{G}$
- $\mathtt{Com}(x, r) : g^x h^r$

2.1.5 Pseudorandom Functions

Definition 2.6 *(Pseudorandom Function.) It is an efficient and deterministic function which return a pseudorandom output computationally indistinguishable from a truly random output.*

- $\mathsf{PRF} : \mathcal{K} \times \mathcal{X} \rightarrow \mathcal{Y}$: *it takes as input a seed from* $\mathcal{K}$, *a data input from* $\mathcal{X}$, *and generates a random output from* $\mathcal{Y}$.

2.1.6 Cryptographic Accumulator

A Merkle tree [52] is a cryptographic accumulator [6] where the tree root is the accumulator digest. It stores a set of elements in the labels of its leaves. An internal node's label is the hash of its children's labels. In Merkle trees, a user can generate proofs of membership that are logarithmic in the size of the tree.

2.2 Zero-Knowledge Proofs of Knowledge

A zero-knowledge proof (ZKP) of knowledge is a protocol in which a prover can convince a verifier that it knows a witness for an NP statement without revealing any information about why it holds. A prover can for example convince a verifier that a confidential transaction is valid without revealing the transferred amount. An *argument* is a proof which holds only if the prover is computationally bounded.

2.2.1 Zero-Knowledge Proof of Interval Membership

Using zero-knowledge proof of interval membership [12], a prover can convince a verifier that a value x with a Pedersen commitment X belongs to the interval $[0, B)$.

Definition 2.7 *Zero-knowledge proof of interval membership is a triple of efficient algorithms* ***(Setup, Prove, Verify)****:*

- $(pp) \leftarrow$ ***Setup***(1^λ)*: it takes as input a security parameter* 1^λ*; and returns public parameters pp.*
- $\pi \leftarrow$ ***Prove***(pp, X, x, r)*: it takes as input public parameters pp, a Pedersen commitment* X *as an NP statement, and the commitment opening* (x, r) *as a witness. It returns a proof* π *that asserts* $(x) \in [0, B)$*.*
- $0/1 \leftarrow$ ***Verify***(pp, X, π)*: it takes as input public parameters pp, a proof* π*, and an NP statement* X*. If* π *is a valid proof, it returns 1, otherwise 0.*

2.2.2 Bulletproofs

Bulletproofs [14] is a short Non-interactive Zero-knowledge (NIZK) proof of range. It allows a prover to convince a verifier that a value x with a Pedersen commitment X belongs to the interval $[0, 2^n - 1]$.

Definition 2.8 *Bulletproofs is a triple of efficient algorithms* ***(Setup, Prove, Verify)****:*

- $(pp) \leftarrow$ ***Setup***(1^λ)*: it takes as input a security parameter* 1^λ*; and returns public parameters pp.*
- $\pi \leftarrow$ ***Prove***(pp, X, x, r)*: it takes as input public parameters pp, a Pedersen commitment* X *as a statement, and the committed opening* (x, r) *as a witness. It returns a proof* π *that asserts* $(x) \in [0, 2^n - 1]$*.*
- $0/1 \leftarrow$ ***Verify***(pp, X, π)*: it takes as input public parameters pp, a proof* π*, and a statement* X*. If* π *is a valid proof, it returns 1, otherwise 0.*

2.2.3 zkSNARK

A zero-knowledge Succinct Non-interactive ARgument of Knowledge (zkSNARK) is essentially a non-interactive zero-knowledge (NIZK) proof system for arithmetic circuit satisfiability [36]. An arithmetic circuit satisfiability problem of a circuit $C : \mathbb{F}_p^n \times \mathbb{F}_p^h \rightarrow \mathbb{F}_p^l$ is captured by relation $R_C = \{(x, w) \in \mathbb{F}_p^n \times \mathbb{F}_p^h : C(x, w) = 0^l\}$ where x and w denote to an NP statement and a witness to the circuit C, respectively. The NP language for the circuit C is defined as $\mathcal{L}_C = \{x \in \mathbb{F}^n \mid \exists w \in \mathbb{F}^h \text{ s.t. } C(x, w) = 0^l\}$.

Definition 2.9 *zkSNARK is a triple of efficient algorithms* ***(Setup, Prove, Verify)****:*

- $(pk, vk) \leftarrow$ ***Setup***$(1^\lambda, C)$*: it takes as input a security parameter* 1^λ *and an arithmetic circuit* C*. It outputs a pair of public keys: proving key* pk *and verification key* vk*.*
- $\pi \leftarrow$ ***Prove***(pk, x, w)*: it takes as input a proving key* pk*, a statement* x *and a witness* w*. It outputs a proof* π *that asserts* $x \in \mathcal{L}_C$*.*
- $0/1 \leftarrow$ ***Verify***(vk, x, π)*: it takes as input a verification key* vk*, a proof* π*, and a statement* x*. It outputs 1 if* π *is a valid proof for asserting that* $x \in \mathcal{L}_C$

In addition to *correctness, soundness, and zero-knowledge* properties, a zkSNARK has the following properties [38]:

1. *Succinctness*: it requires that the proof π has $O_\lambda(1)$ size and the verifier runs in time $O_\lambda(\vec{x})$

2. *Simulation extractability*: it requires the prover to know a witness $\vec{w}$ for a statement $\vec{x}$ in order to generate a valid proof π. In other words, an adversary cannot come up with a new valid proof without knowing the witness.

2.3 Intel Software Guard Extensions

Intel SGX [2] is a TEE technology released by Intel in 2015. It provides an isolated secure environment referred to as *enclave* for code and data protection against violations of confidentiality and integrity. Note that an attacker can statically disassemble the enclave; hence, it must not contain any hard-coded secrets. However, once it is loaded and running, the processor enforces the confidentiality and integrity of the enclave state. Therefore, an observer will have an opaque view of the enclave's state.

2.3.1 Sealing

Intel SGX enclave can securely generate cryptographic keys at the run-time. However, once the enclave terminates execution, it loses its state. Therefore, Intel SGX provides the ability to cryptographically *seal* [2] secrets to untrusted storage in a secure way. It performs the encryption using a private *Seal* Key that is unique to that particular platform and enclave.

2.3.2 Remote attestation

Intel SGX provides the ability to cryptographically *attest* that a particular enclave is running on an authentic Intel SGX platform. The attestation process starts with an application requesting its enclave to generate a `report`. The enclave generates the `report` and authenticates it by a platform-specific hardware key. The `report` contains enclave-specific information, notably, the measurement *MRENCLAVE* and an auxiliary data `report_data` field. The main purpose of the `report_data` field is to bind a piece of data (e.g., a public key of a private key generated exclusively by the enclave) to the `report`.

Next, the application transmits the `report` to an architectural enclave known as *Quote Enclave* (QE) running on the same platform. After verifying the authenticity of the `report` using the same platform-specific hardware key, QE signs the `report` by the attestation key and returns a `quote` (i.e., a signed `report`) to the calling application, which in turn, transmits the `report` to the remote party. Note that, besides attesting to authentic Intel SGX platforms, a valid quote also implies the authenticity of `report_data`. For instance, this allows two enclaves to establish a secure channel after quotes verification by setting their ephemeral ECDH public keys in `report_data` in their corresponding quotes.

Currently, Intel SGX supports two models of remote attestations: EPID [42] and ECDSA [68]. In EPID attestation, the main focus is to preserve the attestor's privacy by utilizing an EPID group signature scheme. Hence, an entity with the group's public key can verify a quote without learning which group member (i.e., processor) has signed it. Intel exclusively maintains the group public key in its Attestation Service (IAS). Thereby, one must consult IAS to verify a quote, which replies with an attestation verification report that confirms or denies the quote's authenticity.

Conversely, the ECDSA attestation sacrifices the attestor's privacy since it attests to platforms within the same organization. More importantly, it does not require com-

munication with IAS. Therefore, ECDSA attestation is more convenient when attestors' identities are already known or when external communication with IAS is restricted.

2.4 Ethereum

Ethereum blockchain [77] acts as a distributed virtual machine that supports quasi Turing-complete programs. Developers can deploy smart contracts guaranteed by the blockchain consensus to run precisely according to their code. There are types of accounts: *externally-owned accounts* (EoA) controlled by users' private keys, and *contract accounts* owned by smart contracts . Only EoAs can send transactions that change the blockchain state. In particular, transactions can transfer Ethers and trigger the execution of smart contract code. The costs of executing smart contracts code are measured gas units, and the transaction's sender pays the gas cost in Ether.

In Ethereum, all transactions consume gas, and their senders must have enough Ether to pay for the gas cost. Furthermore, Ethereum uses *account-model* where an EoA trivially links all transactions performed by its user. One way to break that link is by having another EOA, not owned by the user, pay the gas cost. In particular, the user generates a meta-transaction [59] containing some parameters and sends it to a trustless relayer. Then, the relayer creates a transaction with those parameters and pays the gas cost. To compensate relayers, they often receive shares from protocols' fees which should cover their expenses plus an extra profit.

Chapter 3

Sealed-bid Auctions

3.1 Introduction

The unprecedented growing deployment of assets on Ethereum has created a vibrant market for assets exchange [22] which imposes a high demand for various trading tools such as verifiable and secure auctions. Auctions are platforms for vendors to advertise their assets where interested buyers deposit competitive bids based on their monetary valuation. The auction winner is commonly the highest bidder. Auctions promote many economic advantages for the efficient trade of goods and services. According to [45], there exist two types of sealed-bid auctions: (i) First-price sealed-bid auctions (FPSBA) where the winner pays the highest bid, and (ii) Vickrey auctions, where the winner pays the second-highest bid instead.

Arguably, the main objective behind concealing the losing bids in sealed-bid auctions is to prevent the use of bidders' valuations against them in future auctions. Therefore, bidders are motivated to submit bids without worrying about the misuse of their valuations. Nonetheless, an auctioneer colluding with malicious bidders can easily break this advantage. Consequently, the auctioneer has to be trusted to preserve bids' privacy and to correctly claim the auction winner. Therefore, various constructions of sealed-bid

auctions utilize cryptographic protocols to ensure the proper and secure implementation without harming the privacy of bids.

We design three protocols for sealed-bid auctions with a progressive improvement in performance. Bidders submit their bids to a smart contract. Then, the auctioneer determines the winning bid. Finally, the smart contract acts as a verifier that checks the validity of a zero-knowledge proof or a digital signature before accepting the winning bid.

3.2 Related Work

Kosba et al. [46] presented Hawk, a framework for creating Ethereum smart contract that does not store financial transactions in the clear on the blockchain. Hawk compiler utilizes different cryptographic primitives such as ZKP to generate privacy-preserving smart contracts. A Hawk program consists of public and private parts. The former contains the logic, while the latter includes the cryptographic primitive responsible for hiding the data. Up to our knowledge, the Hawk framework has never been released yet.

Blass and Kerschbaum [9] presented Strain, a protocol to implement sealed-bid auctions on top of blockchains that protects the bid privacy against fully-malicious parties. Strain is a two-party comparison mechanism executed between any pair of bidders in parallel. The comparison outcome is broadcast to all bidders. The main weakness of Strain is leaking the order of bids similar to Order Preserving Encryption (OPE) schemes.

Sánchez [66] proposed Raziel, a system that combines MPC and ZKP to guarantee smart contracts' privacy and verifiability. The associated proofs can effectively prove computation correctness, besides additional properties such as termination, security, pre-conditions, and post-conditions. However, the Raziel runs as an interactive protocol with multiple rounds that are not suitable for blockchain.

3.3 Protocol 1: Using ZKP of Interval Membership

To prove the correctness of the winning bid $v_w \in \mathbb{Z}_q$, the auctioneer needs to prove that v_w is larger than every $v_i \in \mathbb{Z}_q$ where $i \neq w$ and $i \in [N]$ for N bids. In particular, the auctioneer can utilize ZKP of interval membership to prove $v_w > v_i$ by proving:

- $v_w \in [0, \frac{q}{2}]$.
- $v_i \in [0, \frac{q}{2}]$.
- $\Delta v_{w,i} \in [0, \frac{q}{2}]$ where $\Delta v_{w,i} = (v_w - v_i) \, mod \, q$.

3.3.1 Auction Smart Contract

There are three sequential phases in sealed-bid auction: submitting sealed bids, revealing sealed bids to the auctioneer, and finally, verifying the correctness of the winning bid. Initially, the auctioneer deploys the auction smart contract on Ethereum while depositing a `collateral` value as shown in Fig. 3.1.

Initialize$(pk_a, t_{bid}, t_{rev}, t_{vrf})$

`Require` (`tx.value` = `collateral`)

`Store` $pk_a, t_{bid}, t_{rev}, t_{vrf}$

`auctioneer` $\leftarrow$ `msg.sender`

Figure 3.1: Pseudocode of Initialize function

- pk_a is the auctioneer's public key of an asymmetric encryption scheme.
- $t_{bid}, t_{rev},$ and t_{vrf} define the time periods for phases in sealed-bid auctions.

Phase 1: Submitting Sealed Bids. This phase starts immediately after the deployment of the auction contract. Each bidder submits a bid commitment cm using Pedersen

SubmitBid(cm)

`Require` ($\texttt{now} < t_{bid} \wedge \texttt{tx.value} = \texttt{collateral}$)

`Bidders[msg.sender].Commit` $\leftarrow cm$

Figure 3.2: Pseudocode of Submit function

commitment scheme along with the collateral value to the function `SubmitBid` as shown in Fig 3.2.

Phase 2: Revealing Sealed Bids. Each bidder sends a ciphertext ct encrypting the committed bid and randomness (v_i, r_i) by the auctioneer's public key pk_a to the function `RevealBid` on the auction contract as shown in Fig. 3.3.

RevealBid(ct)

`Require` ($\texttt{now} < t_{rev}$)

`Bidders[msg.sender].Ciphertext` $\leftarrow ct$

Figure 3.3: Pseudocode of RevealBid function

The ciphertext is stored on the auction contract instead of being sent directly to the auctioneer in order to avoid the following attack scenario. Suppose a malicious auctioneer pretends that an arbitrary bidder Bob has not revealed the opening values of the associated commitment. In this case, Bob has no chance of denying this false claim. However, if the ciphertext is to be stored on the auction contract, then their mere existence successfully prevents this attack.

We have also taken into our account the possibility of the following attack as well. Suppose a malicious auctioneer intends to penalize an arbitrary bidder Bob by claiming that the decryption outcome of Bob's ciphertext ct_b does not successfully open Bob's commitment cm_b. We prevent this attack by requiring the auctioneer to verify the opening correctness of the commitments once they are submitted by the bidders. In the case of unsuccessful opening, the auctioneer declares on the auction contract that the ciphertext

associated with the Bob is invalid. Subsequently, Bob can deny this claim by revealing (v_b, r_b) to the auction contract. Subsequently, the auction contract encrypts the revealed values by the public key pk_a. If the outcome ciphertext is found to be equivalent to the previously submitted ciphertext ct_b, then the auction contract penalizes the auctioneer and terminates the auction after refunding the bidders. Otherwise, the bidder is penalized and the associated commitment is removed, such that only the valid commitments exist on the auction contract.

Phase 3: Verifying the Auction Winner. The auctioneer determines the winning bid v_w and proves that it is the highest bid. The auctioneer sends a `VerifyWinner` transaction as shown in Fig 3.4. The transaction parameters include the winner address $addr$, opening values of the bid commitment v_w, r_w, $\{\pi_{1,i}\}_{i=1}^{N-1}$ and $\{\pi_{2,i}\}_{i=1}^{N-1}$ proofs for proving $v_i \in [0, \frac{q}{2}]$ and $\Delta v_{w,i} \in [0, \frac{q}{2}]$, respectively.

VerifyWinner$(addr, v_w, r_w, \{\pi_{1,i}\}_{i=1}^{N-1}, \{\pi_{2,i}\}_{i=1}^{N-1})$

`Require` $(t_{rev} \leq \texttt{now} < t_{vrf} \wedge v_w < \frac{q}{2})$

`Require` $(\texttt{msg.sender} = \texttt{auctioneer})$

$cm_w \leftarrow \texttt{Com}(v_w, r_w)$

`Require` $(\texttt{Bidders[addr].Commit} = cm_w)$

$\{cm_i\}_{i=1}^{N} \leftarrow \texttt{Bidders.Commits}$

`For` $i \in [N] \wedge i \neq w$

  $\Delta cm_{w,i} \leftarrow cm_w cm_i^{-1}$

  `Require` $(1 = \texttt{Verify}(cm_i, \pi_{1,i}))$

  `Require` $(1 = \texttt{Verify}(\Delta cm_{w,i}, \pi_{2,i}))$

`winner`$\leftarrow addr$, `highestBid` $\leftarrow v_w$

Figure 3.4: Pseudocode of VerifyWinner function

As explained in 3.3, three interval membership proofs are required to prove that $v_w > v_i$. However, since the bid v_w of the winner is revealed, the smart contract can check

that it is less than the maximum value $\frac{q}{2}$. Therefore, the number of proofs is reduced to $2(N-1)$. In other words, the auctioneer has to prove the interval membership for all bids v_i other than the winning bid and the associated differences $\Delta v_{w,i}$.

3.4 Protocol 2: Using zkSNARK

The second protocol shares a lot of similarities with the first one, except it uses zkSNARK to prove the correctness of the winning bid. Initially, we design an arithmetic circuit `Auction` as shown in Fig. 3.5. The NP statement is of the form "v is the highest bid with a randomness r and a commitment in the list of commitments $\{cm_i\}_{i=1}^{N}$".

Auction(x, w)

Statement x:

- v the winning bid
- r the randomness for the winning bid commitment
- $\{cm_i\}_{i=1}^{N}$ list of N bid commitments

Witness w:

- j the index of the winning bid commitment
- $\{v_i\}_{i=1}^{N}$ committed bids
- $\{r_i\}_{i=1}^{N}$ randomness for the bid commitments

Assert the following:

$cm_j = \texttt{Com}(v, r)$
`For` $i \in [N] \wedge i \neq j$
    $cm_i = \texttt{Com}(v_i, r_i)$
    $v \geq v_i$

Figure 3.5: zkSNARK Auction circuit

The `Setup` algorithm of zkSNARK requires a trust assumption. However, this is against the whole premise of the blockchain as a decentralized platform that does not require a trusted party. Hence, we utilize MPC protocol [11] to generate the Common

Reference String (CRS) (i.e., proving and verification keys) such that at least a single honest participant is required to ensure the zkSNARK proofs security. Moreover, it is sufficient to generate CRS once as long as the problem statement does not change. In other words, we can initially generate the CRS for the sealed-bid auction. Then, we can utilize the resultant CRS in multiple auction instances.

While the auctioneer might be tempted to omit commitments to let a colluding bidder win the auction, doing so will result in a failed verification by the smart contract. In other words, the `Auction` circuit does not check whether the auctioneer supplies *all* commitments as part of the public inputs. On the other hand, the verification which is carried out by the smart contract does supply all commitments. As a consequence, there will be a difference between the commitments list used by the auctioneer to generate the proof, and the commitments list used by the smart contract to verify the proof. Therefore, the verification will fail, and the auctioneer will be penalized if he cannot supply a valid proof that uses the same commitments list as the smart contract.

3.4.1 Auction Smart Contract

Similar to the first protocol, this protocol consists of the same three sequential phases. The auctioneer deploys the smart contract and locks a `collateral` deposit as shown in Fig. 3.6.

Initialize$(pk_a, t_{bid}, t_{rev}, t_{vrf}, vk)$

`Require` (`tx.value` = `collateral`)

`Store` $pk_a, t_{bid}, t_{rev}, t_{vrf}, vk$

`auctioneer` $\leftarrow$ `msg.sender`

Figure 3.6: Pseudocode of Initialize function

- pk_a is the auctioneer's public key of an asymmetric encryption scheme.

- vk is the zkSNARK verification key generated by the MPC-based setup.
- $t_{bid}, t_{rev},$ and t_{vrf} define the time periods for phases in sealed-bid auctions.

Phase 1: Submitting Sealed Bids. Compared to the first protocol, in this phase, each bidder submits a bid commitment cm using `SHA256` hash function, and the collateral value to the function `SubmitBid` as shown in Fig 3.2.

Phase 2: Revealing Sealed Bids. Each bidder sends a ciphertext ct encrypting the committed bid and randomness (v_i, r_i) by the auctioneer's public key pk_a to the function `RevealBid` on the auction contract as shown in Fig. 3.3.

Phase 3: Verifying the Auction Winner. After decrypting the ciphertext for bids, the auctioneer can determine the winning bid, and hence the auctioneer generates a zkSNARK proof $\pi \leftarrow \texttt{Prove}(pk, x, w)$ where pk is the proving key, x and w are NP statement and witness satisfying the `Auction` arithmetic circuit, respectively. The auctioneer calls the function `VerifyWinner` by specifying a winner address $addr$, a tuple (v, r) from a statement x, and a zkSNARK proof π as shown in Fig. 3.7.

VerifyWinner$(addr, v, r, \pi)$

`Require` $(t_{rev} \leq \texttt{now} < t_{vrf})$

`Require` $(\texttt{msg.sender} = \texttt{auctioneer})$

`Require` $(\texttt{Bidders[addr].Commit} = \texttt{Com}(v, r))$

$\{cm_i\}_{i=1}^{N} \leftarrow \texttt{Bidders.Commits}$

$x \leftarrow (v, r, \{cm_i\}_{i=1}^{N})$

`Require` $(1 = \texttt{Verify}(vk, x, \pi))$

`winner`$\leftarrow addr$, `highestBid` $\leftarrow v$

Figure 3.7: Pseudocode of VerifyWinner function

3.5 Protocol 3: Using Intel SGX

Compared to previous protocols, this protocol, Trustee, does not utilize zero-knowledge proofs and consists of two phases only: submission of bids and verification of the auction winner. Trustee relies on Intel SGX to determine the auction winner in a complete privacy-preserving manner. It depends on basic cryptography primitives such as digital signatures and Elliptic Curve Integrated Encryption Scheme (ECIES) to encrypt bids. ECIES enables two parties to communicate authenticated confidential messages. As its name indicates, ECIES integrates the following functions:

1. $(sk, pk) \leftarrow \texttt{KeyGen}(1^\lambda)$: a key generation function that takes a security parameter 1^λ, and returns a private key sk and the corresponding public key pk.

2. $ss \leftarrow \texttt{KA}(sk_i, pk_j)$: a key agreement function to generate a shared secret ss based on the private key of party i and the public key of party j.

3. $(k_1, k_2) \leftarrow \texttt{KDF}(ss)$: a key derivation function to produce keys k_1 and k_2 from the shared secret ss.

4. $ct \leftarrow \texttt{Enc}(m, k_1)$: a symmetric encryption function to encrypt a message m using the symmetric key k_1.

5. $tag \leftarrow \texttt{MAC}(ct, k_2)$: a message authentication code function to generate a tag based on the key k_2 and the ciphertext ct.

3.5.1 System Overview

Trustee system consists of three components: a smart contract C which resides on top of Ethereum, a back-end Intel SGX enclave E and a relay R which both run off-chain on a server. We refer to the user who deploys C and controls R as the auctioneer. Furthermore, E is only accessible through R, and R interacts with C on behalf of the

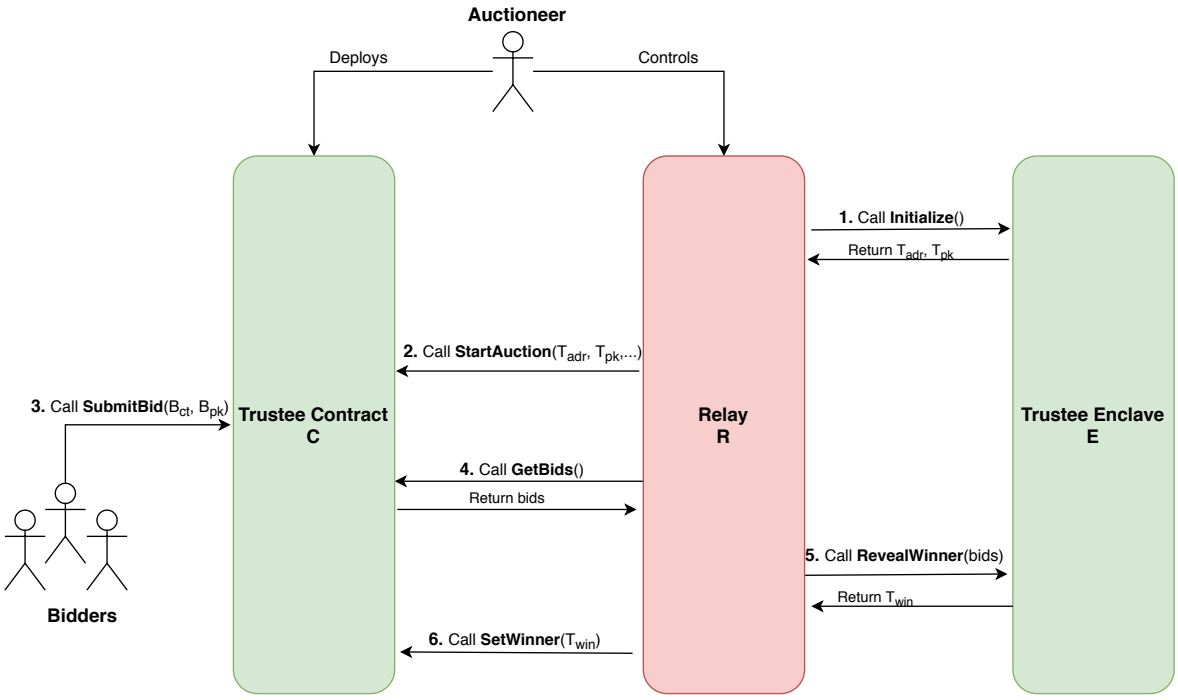

Figure 3.8: Interactions between Trustee's components and bidders

auctioneer and E. The general flow of interactions between Trustee's components, and bidders is depicted in Fig. 3.8.

Initially, the auctioneer deploys C on Ethereum and publishes its address so that interested sellers and buyers can learn about it. To start an auction, the auctioneer sends a request to R which loads E and calls the function `InitializeEnclave`. As a response, E generates an EOA account with the private key T_{sk} and the associated address T_{addr}, and an ECDH key-pair (T_{dh}, T_{pk}) where T_{dh} is the private key and T_{pk} is the associated public key. Then, it returns the values of T_{addr} and T_{pk} to R. Subsequently, the auctioneer instructs R to deploy the auction smart contract C and initializes it with parameters phases periods, in addition to T_{addr} and T_{pk}. Next, assume a bidder Bob is interested in the auction, then he utilizes ECIES protocol with T_{pk} as the public key of the recipient (i.e., Trustee's enclave E) to encrypt his bid. Subsequently, he submits his sealed bid ct along with his ECDH public key B_{pk} to C. Once the bidding interval is closed, R retrieves the sealed bids stored on the C, then it forwards them to E by calling the

function `RevealWinner`. As a result, E opens the sealed bids and determines the winner Then, it returns a transaction tx_{win} signed by the private key T_{sk} to R. Finally, R sends tx_{win} to C which is essentially a call to the smart contract function `VerifyWinner` that declares the auction winner and highest price.

3.5.2 Trustee Construction

Initializing the Enclave. The initialization process starts with the auctioneer requesting R to load E inside Intel SGX enclave and invoke the function `InitializeEnclave` which is implemented as shown in Algorithm 1.

Algorithm 1: `InitializeEnclave`

Input: 1^λ: Security parameter
Output:

- *sealed*: Sealed private keys
- T_{addr}: EOA address
- T_{pk}: ECDH public key

1 $(T_{pk}, T_{dh}) \leftarrow$ `GenerateECDHKeys`(1^λ)
2 $(T_{addr}, T_{sk}) \leftarrow$ `GenerateAccount`(1^λ)
3 $sealed \leftarrow$ `Seal`(T_{sk}, T_{dh})
4 **return** $sealed, T_{addr}, T_{pk}$

The `InitializeEnclave` function generates two key-pairs. More precisely, one key-pair (T_{pk}, T_{dh}) that enables bidders to seal their bids such that only E can open them, and the second one to authenticate the result (i.e., auction winner and second-highest price) generated by E. The former is an ECDH key-pair used as part of ECIES protocol between E and each bidder to securely transmit the sealed bids through C and R. The latter is an ECDSA key-pair used to sign the result. Verifying the signature on the result by C is a relatively expensive operation (i.e., roughly 120,000 gas for using *ecrecover*). Therefore, in Trustee, we utilize an intrinsic operation that happens on every transaction in Ethereum (i.e., transaction's signature verification) to indirectly verify the authenticity

of the result for us. Hence, E generates an ECDSA key-pair on curve `secp256k1` which essentially creates an EOA with the private key T_{sk} and the associated address T_{addr}. Then, whenever E determines the auction winner and the highest price, it generates a transaction tx_{win} signed by T_{sk}. Later, R relays tx_{win} to the Ethereum network, where the miners verify its signature. Finally, C checks that that the sender of tx_{win} is the T_{addr}. As a result, this approach yields a much cheaper transaction fee compared to the explicit signature verification by calling `ecrecover`.

Intel SGX enclaves are designed to be stateless. In other words, once an enclave is destroyed, its whole state is lost. However, in Trustee, we have to persist the generated keys as long as the current auction is running. Therefore, we utilize Intel SGX feature known as *Sealing* [2] to properly save the generated private keys. Sealing is the process of encrypting enclave secrets in order to persist them on a permanent storage such as a disk. This effectively allows us to retrieve the private keys (T_{sk}, T_{dh}) even if the enclave was brought down for any reason. The encryption is performed using a private *Seal Key* that is unique to the platform and enclave, and is not accessible by any other entity.

Initializing the Smart Contract. Upon the return from `InitializeEnclave`, R saves the values of *sealed* on a disk besides to having a backup. Furthermore, R deploys a smart contract C and initializes it with T_{addr} and T_{pk}, in addition to parameters t_{bid} and t_{vrf} which define the time period for bids submission and revealing auction winner. Furthermore, R locks a `collateral` deposit that gets slashed if it fails to relay T_{win} transactions in time as shown in Fig. 3.9.

Initialize$(T_{addr}, T_{pk}, t_{bid}, t_{vrf})$

`Require` (`tx.value` = `collateral`)

`Store` $T_{addr}, T_{pk}, t_{bid}, t_{vrf}$

`auctioneer` $\leftarrow$ `msg.sender`

Figure 3.9: Pseudocode of Initialize function

Phase 1: Submitting Sealed Bids. Once the C has been initialized, an interested bidder Bob can seal his bid x by running `EncryptBids` as shown in Algorithm 2. It starts with retrieving the public key T_{pk} from C. Then, it generates an ephemeral ECDH key-pair (B_{pk}, B_{sk}) on `curve25519` where B_{sk} is the private key and B_{pk} is the corresponding public key. Then, it computes the shared secret ss based on T_{pk} and B_{sk}. After that, it derives two symmetric keys k_1 and k_2 in order to perform an authenticated encryption on the bid value v.

Algorithm 2: `EncryptBids`

Input:

- 1^{λ}: Security parameter
- T_{pk}: Trustee's public key
- v: Bid value

Output:

- B_{pk}: Bidder's public key
- ct: Encrypted Bid
- tag: Authentication tag

1 $(B_{pk}, B_{dh}) \leftarrow \texttt{GenerateECDHKeys}(1^{\lambda})$
2 $ss \leftarrow \texttt{KA}(B_{dh}, T_{pk})$
3 $(k_1, k_2) \leftarrow \texttt{KDF}(ss)$
4 $ct \leftarrow \texttt{Enc}(v, k_1)$
5 $tag \leftarrow \texttt{MAC}(ct, k_2)$
6 **return** B_{pk}, ct, tag

Finally, it returns the sealed-bid ct, tag and the public key B_{pk}. Subsequently, Bob sends these values along with a `collateral` deposit to the function `SubmitBid` on C as shown in Fig. 3.10.

The function `SubmitBid` first asserts that the call is invoked before the end of the bidding interval t_{bid}. After that, it stores the ct and B_{pk} into the array $Bidders$.

Phase 2: Determining and Verifying the Auction Winner. Once the bidding interval is over, R retrieves the submitted array of sealed bids and their associate public

SubmitBid(ct, tag, B_{pk})

`Require` $(\texttt{now} < t_{bid} \wedge \texttt{tx.value} = \texttt{collateral})$

`Bidders[msg.sender]` $\leftarrow (ct, tag, B_{pk})$

Figure 3.10: Pseudocode of SubmitBid function

keys $(ct_i, tag_i, B_{pk,i})_{i=1}^{N}$ from C. Then, it passes them along with *sealed* (previously generated by the function `InitalizeEnclave`) to the function `RevealWinner` on E as shown in Algorithm 3.

Algorithm 3: `RevealWinner`

Input:

- *sealed*: Sealed private keys
- $(ct_i, tag_i, B_{pk,i})_{i=1}^{N}$: Encrypted bids and bidder's public keys

Output: tx_{win}: Signed transaction revealing auction winner and bid

```
1  max ← 0
2  index ← 0
3  (T_sk, T_dh, success) = Unseal(sealed)
4  for i ∈ [N] do
5      ss ← KA(T_dh, B_pk,i)
6      (k_1, k_2) ← KDF(ss)
7      Assert(tag_i = MAC(ct_i, k_2))
8      v_i ← Dec(ct_i, k_1)
9      if v_i > max then
10         max ← v_i
11         index ← i
12 h ← SHA256((ct_i, tag_i, B_pk,i)_{i=1}^{N})
13 tx_win ← GenerateTransaction(T_sk, h, index, max)
14 return tx_win
```

Initially, E unseals the private keys from *sealed*. Then, for every bidder i, it runs the decryption part of ECIES protocol based on the sealed-bid ct_i, the public key $B_{pk,i}$, and the private key T_{dh} to extract the bid value and find the winner. Once all sealed-bids are decrypted, the winner's index *index* and highest bid *max* are set accordingly . Subsequently, E binds the auction winner to the inputs it received by computing the

SHA256 hash value of the sealed bids list. Finally, E creates a transaction tx_{win} to the function VerifyWinner on C and signs it with the private key T_{sk} as shown in Fig. 3.11. Next, the auctioneer requests R to send the transaction tx_{win} to C. On its call, it asserts that: (i) tx_{win}'s origin is the address T_{addr}, the call happens before the end of reveal phase. Then, it checks if h is equal to the SHA256 hash value of the sealed bids and their associated public keys submitted by bidders. Accordingly, it decides whether to accept the submitted values or reject them.

VerifyWinner$(h, index, max)$

Require $(t_{bid} < \texttt{now} < t_{vrf})$

Require $(T_{addr} = \texttt{msg.sender})$

Require $(h = \texttt{SHA256(Bidders)})$

winner$\leftarrow$ Bidders$[index]$, highestBid $\leftarrow max$

Figure 3.11: Pseudocode of VerifyWinner function

3.6 Evaluation

We have implemented a prototype for each protocol, and we have also made it open-source and published it on Github repositories[1]. To determine the performance measurements, we have created a local Ethereum blockchain using the *Geth* client version 1.8.10. The genesis file that is responsible for the initialization of the local blockchain contains the following attribute to support the pre-compiled contracts EIP-196 and EIP-197: { *"byzantiumBlock": 0*}. We created a test case with the number of bidders $N = 10$. Table 3.1 lists the security properties of the proposed sealed-bid auction protocols. Furthermore, we compare the proposed sealed-bid auctions and report the performance measurements in Table 3.2.

[1] https://github.com/hsg88

Table 3.1: Comparison between sealed-bid auction protocols

	Protocol 1 [32]	Protocol 2 [31]	Protocol 3 [33]
Front-running resistance	Yes	Yes	Yes
Privacy	Partial	Partial	Full
Security Assumption	Discrete log	Knowledge of exponent	Hardware vendor
Dispute free	No	No	Yes
Fairness	Yes	Yes	Yes

Table 3.2: Gas cost evaluation for sealed-bid auction protocols with N bidders

	Protocol 1 [32]	Protocol 2 [31]	Protocol 3 [33]
`Initialize`	1346611	3131261	1173779
`SubmitBid`	130084	115583	123350
`RevealBid`	53231	44176	
`VerifyWinner`	$2002490(N-1)$	3395077	82847

Protocols 1 and 2 rely on zero-knowledge proofs to prove the correctness of the auction winner without revealing the losing bids to the public. However, the auctioneer needs to access the committed bids to generate valid zero-knowledge proofs. Hence, these protocols are partially privacy-preserving. In contrast, protocol 3 relies on a trusted execution environment to find the auction winner in a complete privacy-preserving manner.

Protocol 1 relies on a standard security assumption (i.e., the intractability of discrete log problem). Protocol 2 requires a trusted setup ceremony where at least a single honest participant is needed to ensure the protocol's security. Protocol 2 additionally relies on a non-standard assumption referred to as *knowledge of exponent*. The security of protocol 3 assumes that the hardware vendor provides honest attestations besides correct hardware design and robustness against side-channel attacks.

From the gas cost perspective, the cost for verifying the auction winner in protocol 1 scales linearly with the number of bids, while protocols 2 and 3 incur a fixed cost. The deployment cost of protocol 2 is high due to the size of the zkSNARK verification key stored in the smart contract during initialization. A key advantage for protocol 3 compared to others is it consists of two phases to finalize an auction (i.e., bidders interact

with the smart contract only when they need to submit bids).

3.7 Summary

We designed three protocols for sealed-bid auctions with progressive improvement in performance and privacy. An auctioneer deploys a smart contract that acts as a trusted verifier. Then bidders submit their bids to the smart contract. Finally, the auctioneer determines the auction winner and convinces the smart contract of its correctness in a privacy-preserving manner. Furthermore, we implemented open-source prototypes for these protocols in order to assess their cost and security. In conclusion, protocols utilizing TEE achieve the highest privacy and lowest cost yet they require strong trust assumption in the hardware vendor. In contrast, protocols relying on zero-knowledge proofs provide partial privacy and high verification cost yet they rely on standard security assumptions.

Chapter 4

Periodic Auctions

4.1 Introduction

Investors use financial exchanges to trade equities and securities. Generally, an exchange is a continuous double-sided auction between buyers and sellers [74]. It records all outstanding limit orders in an order book. A *limit* order consists of a unit price, a quantity of an asset, and a direction to indicate whether it is a *bid* by a buyer or an *offer* by a seller. If the order book is transparent and accessible to the public, the exchange is known as a *Lit market*. On the contrary, a *dark pool* favored by financial institutions is an exchange that hides its order book from traders [74].

To understand the main benefit of dark pools, it is worth considering the problem institutional investors face in lit markets. Suppose that Bob is an institutional investor who uses a lit market to buy one million USD worth of an arbitrary asset. The sellers will notice Bob's bid. Hence, they anticipate the increased demand and react by moving their offers to higher prices to gain higher revenue. As a result, Bob will have a hard time filling his order. Thus, he either accepts the loss in buying the whole volume at higher prices or divides the quantity into smaller batches and buys at different prices. Although the latter approach may seem better, it still incurs high fees and commissions paid to the

exchange. Therefore, it is convenient for Bob to trade on a dark pool since the market impact will be minimal.

While dark pools provide a better trading platform for financial institutions, they have several issues. Most importantly, they hurt price discovery and put traders on other exchanges at a disadvantage. Furthermore, the lack of transparency could result in poor execution of trades or abuses such as front-running. Conflicts of interest are also a possibility since the operator could trade against pool clients. The U.S. Securities and Exchange Commission has found numerous violations and fined some dark pool operators [69–71]. Accordingly, several recent financial regulations, such as Europe's MiFID-II [54], call for limiting trades on dark pools. Interestingly, post enforcing MiFID-II, periodic auctions, which are considered regulation-compliant alternatives to dark pools, have witnessed a surge in the size of executed trading volumes.

In periodic auctions, the operator matches orders periodically, rather than continuously. Initially, traders submit orders privately to the operator. The submission phase ends at a random time. Next, the operator determines the market clearing price (MCP) and market clearing volume (MCV). Essentially, traders trust the operator to correctly calculate these values since they do not have access to the order book. To counter-balance this trust, regulators must audit the operator's work to reveal malicious behavior. However, the audit process is prohibitively expensive, and it might also be infrequent. One way to remove trust requirements and reduce costly audits is to utilize the blockchain technology.

4.2 Related Work

Thrope and Parkes [74] proposed a protocol for continuous double-sided auctions. Initially, each trader sends a price, a quantity, and a direction encrypted by the operator's public key of a homomorphic encryption scheme to a bulletin board. Then, the operator

decrypts the orders and tries to match them. Once a match is found, the operator executes the matched orders and publishes them in history. The main drawback of this protocol is its heavy computation burden on the operator since it requires ranking all orders and generating proofs of correctness after the execution of every matched order.

Jutla [43] presented a secure five-party computation protocol for periodic auctions. A small number of brokers and a regulatory authority run the protocol. The auction starts with traders sending limit orders to brokers. Next, brokers run the protocol to find MCP and settle matched orders. In each round of this protocol, the regulatory authority must audit extensive computation to ensure the correctness of MCP, which renders the protocol impractical.

Cartlidge *et al.* [18] utilized SCALE-MAMBA, a multi-party computation (MPC) framework, to emulate a trusted third party. The authors designed three constructions to assess the feasibility of using MPC in stock markets. They argue that it is not practical yet to run continuous double-sided auctions. On the other hand, the periodic auctions and volume matching constructions show promising results. Although this protocol provides strong secrecy, it requires a heavy pre-processing phase in addition to the inherent highly interactive communications between parties.

4.3 Preliminaries

4.3.1 Evaluator-Prover Model

The evaluator-prover (EP) model [53] provides a practical framework for secrecy preserving proofs of correctness. Involved parties secretly submit input values $(x_1, \ldots, x_n)$ to the EP entity. The EP privately computes a function $y = f(x_1, \ldots, x_n)$, outputs the value y, and generates a proof of the correctness. Parties accept the result on successful verification of the proof of correctness. The EP model is *secrecy preserving* if the proof does not reveal any information about the inputs beyond what is implied by the result.

Note that the EP model does not maintain *strong secrecy* [60], which mandates that the EP cannot disclose information about the inputs. However, the notion of *secrecy preserving* is still useful in the context of periodic auctions. More specifically, at the end of each round, information about the MCP and MCV are published, which gives more hints about the inputs. Hence, the main requirement here is to ensure that the operator cannot exploit this information to its advantage. In particular, the operator must not have access to the submitted orders until the end of the submission phase.

4.3.2 Consistent Commitment Encryption (CCE)

We design an honest-verifier zero-knowledge Σ protocol for consistent commitment encryption as shown in Fig. 4.1. It allows a prover $\mathcal{P}$ to convince a verifier $\mathcal{V}$ that an ElGamal ciphertext hides the committed value of a Pedersen commitment. To motivate the need for this protocol, suppose that Alice has sent Bob a commitment $X \leftarrow g^x h^r$ for a message x. Later, she reveals x to Bob by encrypting it using ElGamal encryption as $(c_1, c_2) \leftarrow (g^r, g^x y^r)$ where y is Bob's public key y. Alice wants to convince Carol that she has encrypted x in the ciphertext c using Bob's public key. More precisely, Alice wants to prove the following argument:

$$\{(c_1, c_2, X, y; x, r) : c_1 = g^r \wedge c_2 = g^x y^r \wedge X = g^x h^r\}$$

We utilize Fiat-Shamir heuristic to convert the protocol into NIZK argument by using a hash function to get the challenge $e \leftarrow H(c_1, c_2, X, y, a_1, a_2, A)$. We define the following two $\mathcal{PPT}$ algorithms for this protocol:

1. $\pi \leftarrow \texttt{Prove}(c_1, c_2, X, y, x, r)$. It generates a proof π to prove that the ciphertext (c_1, c_2) is an encryption of the opening value x for the commitment X.

2. $\{0, 1\} \leftarrow \texttt{Verify}(c_1, c_2, X, \pi)$. It returns 1 if it has successfully verified the proof π for a ciphertext (c_1, c_2) and a commitment X; otherwise, it returns 0.

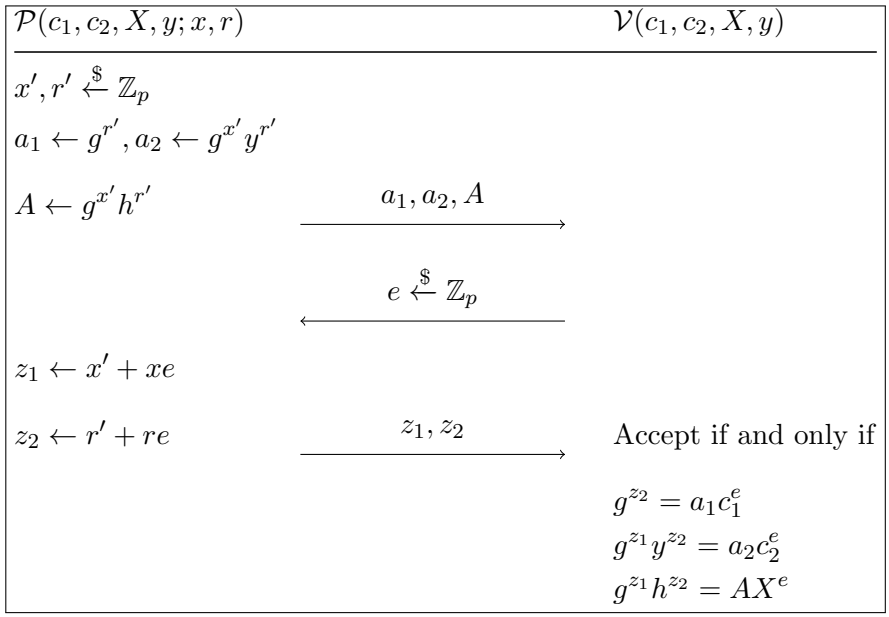

Figure 4.1: Protocol for consistent commitment encryption

4.3.3 Proving Correctness of Sort

We build a protocol to prove that the committed values for a vector of Pedersen commitments are in descending order without revealing any information beyond that fact, as shown in Fig. 4.2. More specifically, given a vector of commitments $\{X_i\}_{i=1}^m$ to a vector of elements $x_i \in [0, 2^n - 1]$, we say the elements are in descending order if the differences between successive elements are non-negative values. Furthermore, since Pedersen commitments are additively homomorphic, one can compute the commitment $\hat{X}_i$ to the differences between successive elements x_i, x_{i+1} given their commitments X_i, X_{i+1}. Now, we can utilize Bulletproofs to prove that the commitments $\hat{X}_i$ is a commitment to non-negative value (i.e., $\hat{x}_i \in [0, 2^n - 1]$). Furthermore, Bulletproofs allows efficient aggregation of separate proof. Note that we can also prove ascending order by simply reversing the elements in the vectors.

By default, this protocol inherits the completeness and zero-knowledge properties of Bulletproofs [14]. To ensure the soundness, we have a condition on the value of 2^n. Specifically, as the operation $x_i - x_{i+1}$ is carried out in $\mathbb{Z}_p$, the condition $2^n < \frac{p}{2}$ must hold to ensure that negative differences do not fall in the range $[0, 2^n - 1]$.

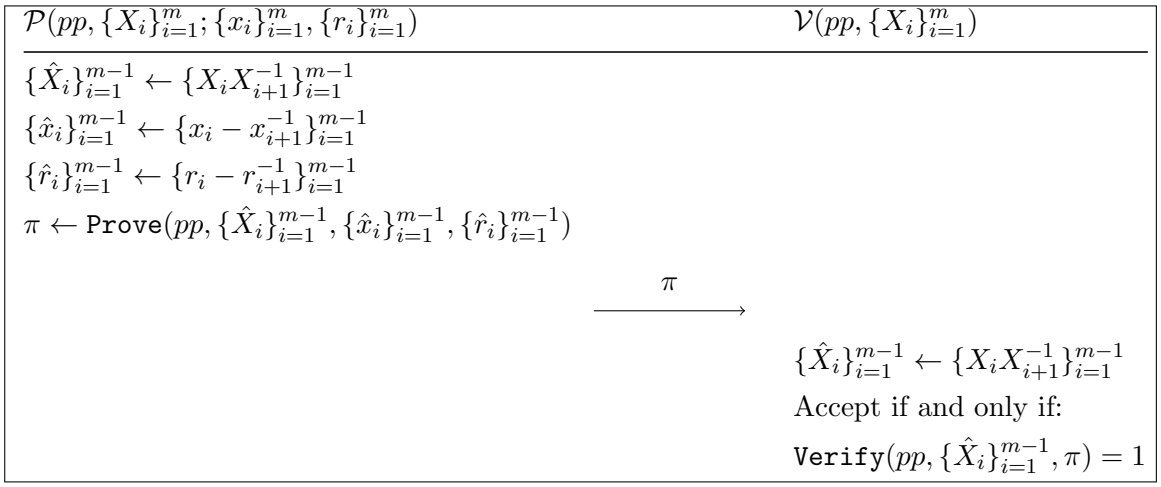

Figure 4.2: Protocol for proving correctness of sort

It is worth mentioning that the implementation of Fiat-Shamir heuristic can compromise the security. More precisely, these protocols are susceptible to *replay* attacks by the adversary when they are used with blockchain. For example, the adversary can replay an arbitrary trader's proof to the smart contract without knowing any witness, yet her proof will be successfully accepted. To prevent this attack, we include the *address* of the transaction sender as one of the inputs to the hash function that computes the verifier challenges. Consequently, the adversary's proof will be rejected because the verifier challenges computed by the smart contract will be different from those computed for the replayed proof.

4.4 Periodic Auction Protocol

4.4.1 System Model

In this protocol, there are three entities, namely, traders, an operator, and a smart contract. The operator and traders interact indirectly through the smart contract using their accounts on the blockchain.

1. Traders are the buyers and sellers who want to exchange their assets via the auction.

2. An operator is the EP entity that privately receives orders and evaluates the MCP and MCV, and proves their correctness to the smart contract.

3. A smart contract publicly verifies the zero-knowledge proofs submitted by traders and the operator, as well as serves as a secure bulletin-board.

4.4.2 High-Level Flow of the Protocol

The operator deploys the smart contract and initializes it by a set of public parameters. Each operation performed by the traders or the operator results in a piece of data and zero-knowledge proof, which will be submitted to the smart contract. The smart contract verifies the proof, and upon success, it stores the associated data. A single round of the periodic auction protocol consists of the following three phases:

1. Traders commit to their orders, and utilize Bulletproofs to generate an aggregate range proof.

2. Traders encrypt their orders by the operator's public key, and utilize CCE protocol to prove the consistency between ciphertext and commitments.

3. The operator does the following:

   (a) Access price and quantity values in orders.

   (b) Determine the MCP and MCV.

   (c) Generate proof of correctness for MCP and MCV.

4.4.3 Auction Smart Contract

The protocol starts by the operator Alice generating the public parameter pp by running the setup algorithm of Bulletproofs for a security parameter λ, a bit-width n, and number of commitments m. Then, she generates a key-pair x, y as the secret and

public keys for ElGamal encryption scheme, respectively. Additionally, she defines the time-window of each phase by the vector $t_{bid}, t_{rev}, t_{vrf}$.

$$\begin{aligned} pp &\leftarrow \texttt{BulletProofs.Setup}(1^\lambda, n, m) \\ x, y &\leftarrow \texttt{KeyGen}(\mathbb{G}, q, p, g) \end{aligned}$$

Next, she deploys the smart contract and initializes it with the parameters (pp, y, t) along with a collateral deposit. The auctioneer deploys the auction smart contract on Ethereum while depositing a `collateral` value as shown in Fig. 4.3.

Initialize$(pp, y, t_{bid}, t_{rev}, t_{vrf})$

`Require` (`tx.value` = `collateral`)

`Store` $pp, y, t_{sub}, t_{rev}, t_{vrf}$

`operator` $\leftarrow$ `msg.sender`

Figure 4.3: Pseudocode of Initialize function

Phase One: Submission of Orders. Traders submit their orders before the block-height t_{sub}. For example, a trader Bob wants to buy v units of the auctioned asset at a price u. He creates his order as follows:

$$\begin{aligned} \{r_1, r_2\} &\xleftarrow{\$} \mathbb{Z}_p^2 \\ U &\leftarrow \texttt{Com}(u, r_1) \\ V &\leftarrow \texttt{Com}(v, r_2) \\ \pi &\leftarrow \texttt{Bulletproofs.Prove}(pp, \{U, V\}, \{u, v\}, \{r_1, r_2\}) \end{aligned}$$

First, he creates Pedersen commitments U and V for the price and quantity, respectively. Subsequently, the trader generates an aggregate range proof π to assert that the price and quantity values are within the range, i.e. $u, v \in [0, 2^n - 1]$. It is worth mentioning that in the prototype, this phase uses a different Bulletproof setup where $m = 2$ since there

are two commitments only. Finally, he sends a transaction `SubmitOrder` that includes the parameters $(\texttt{dir}, U, V, \pi)$ where `dir` indicates whether the order is a bid or an offer as shown in Fig. 4.4.

SubmitOrder$(\texttt{dir}, U, V, \pi)$

`Require` $(\texttt{now} < t_{sub} \wedge \texttt{tx.value} = \texttt{collateral})$

`Require` $(1 = \texttt{Verify}(pp, \{U, V\}, \pi))$

```
if (dir = BID)
```

    Bids[msg.sender] $\leftarrow \{U, V\}$

```
else
```

    Offers[msg.sender] $\leftarrow \{U, V\}$

Figure 4.4: Pseudocode of SubmitOrder function

Upon receiving the transaction, the smart contract checks whether the current block-height is less than t_{bid}, and the transaction has the collateral deposit. Then, it verifies the aggregate range proof π for the commitments U and V. Finally, it stores the commitments in either the list of `Bids` or `Offers` based on the value of `dir`.

It is worth mentioning that front-running has a little impact in periodic auctions in contrast to continuous mainly because orders will be settled at a single MCP regardless of orders sequence. Still, this protocol provides protection against front-running for three main reasons. First, the commitments U and V are perfectly hiding. Second, the aggregate range proof π is zero-knowledge, hence, π does not reveal any information about the witness u and v beyond the fact that they are in range $[0, 2^n - 1]$. Third, there is an idle period between the first and second phases to consider the possibility of revealing orders on minor blockchain forks that will be discarded.

Phase Two: Revealing Orders. Traders utilize ElGamal encryption to reveal their orders to Alice before the block-height t_2. Therefore, Bob retrieves Alice's public key y

from the smart contract and encrypts the opening values (u, r_1) and (v, r_2) as follows:

$$c_u \leftarrow \texttt{Enc}_y(u, r_1), \quad \pi_u \leftarrow \texttt{CCE.Prove}(c_u, U, y, u, r_1)$$
$$c_v \leftarrow \texttt{Enc}_y(v, r_2), \quad \pi_v \leftarrow \texttt{CCE.Prove}(c_v, V, y, v, r_2)$$

Then, he utilizes CCE protocol to generate the proofs π_u and π_v to prove the consistency of ciphertext c_u and c_v for the commitments U and V, respectively. Subsequently, he sends a transaction `RevealOrder` which includes the parameters (c_u, c_v, π_u, π_v) as shown in Fig. 4.5.

RevealOrder(c_u, c_v, π_u, π_v)

`Require` $(t_{sub} < \texttt{now} < t_{rev})$

$\{U, V\} \leftarrow$ `GetOrder(msg.sender)`

`Require` $(1 = \texttt{Verify}(c_u, U, \pi_u) \wedge 1 = \texttt{Verify}(c_v, V, \pi_v))$

`Emit OrderRevealed(msg.sender,` c_u, c_v`)`

`Revealed[msg.sender]` $\leftarrow$ `true`

Figure 4.5: Pseudocode of RevealOrder function

Initially, the smart contract checks if the transaction is sent within the right time window between t_{sub} and t_{rev}. Then, it searches for the commitments (U, V) corresponding to transaction `msg.sender`. Subsequently, it verifies the proofs π_u and π_v. Alice can monitor the transactions submitted to the smart contract during this phase to recover the ciphertext c_u and c_v. In practice, Alice can efficiently retrieve the ciphertext by listening to *events* triggered on the smart contract.

4.4.4 Phase Three: Matching Orders

At the beginning of this phase, Alice instructs the smart contract to find unrevealed orders, remove them, and penalize their owners. Now, she has access to the price and

quantity values of revealed orders. She performs the following tasks to determine the MCP and MCV before block-height t_3:

1. Sort the bids descendingly and the offers ascendingly by price.
2. Compute the cumulative quantity in bids and offers.
3. Finds the MCP that clears the highest cumulative quantity, i.e. MCV.
4. Send the MCP and MCV along with proofs of correctness to the smart contract.

She can generate proof of correctness by creating an order with the MCP and MCV values. Then, she inserts that order in the sorted lists of bids and offers consisting of prices and cumulative quantities. Finally, she utilizes protocol 4.3.3 to prove the sort on price and cumulative quantity commitments. Note that cumulative quantity commitments can be easily computed since Pedersen commitments are additively homomorphic.

Let $\boldsymbol{B}$ and $\boldsymbol{S}$ denote the lists of bids and offers where the numbers of orders in each list are M and N, respectively. Each order in $\boldsymbol{B}$ and $\boldsymbol{S}$ is encoded as a tuple $(U, V, V_c, u, r_1, v, r_2, v_c, r_c)$ of price, quantity, and cumulative quantity commitments and their opening values. Note that, at the beginning, V_c, v_c, r_c are empty. Alice performs the first task as follows:

$$\texttt{Sort}(\boldsymbol{B}, \texttt{DESCENDING})$$

$$\texttt{Sort}(\boldsymbol{S}, \texttt{ASCENDING})$$

The `Sort` function sorts the elements in the input list according to the specified criteria on the price values. For example, the elements in $\boldsymbol{B}$ and $\boldsymbol{S}$ are relocated such that:

$$\forall i \in [1, M-1],\ B_i.u > B_{i+1}.u$$

$$\forall j \in [1, N-1],\ S_j.u < S_{j+1}.u$$

Next, for each order in $\boldsymbol{B}$ and $\boldsymbol{S}$, she computes the cumulative quantities as:

$$\begin{aligned}
\forall i \in [1, M],\ B_i.(V_c, v_c, r_c) &\leftarrow \boldsymbol{B}.(\prod_{k=1}^{i} V_k, \sum_{k=1}^{i} v_k, \sum_{k=1}^{i} r_{2,k}) \\
\forall j \in [1, N],\ S_j.(V_c, v_c, r_c) &\leftarrow \boldsymbol{S}.(\prod_{k=1}^{j} V_k, \sum_{k=1}^{j} v_k, \sum_{k=1}^{j} r_{2,k})
\end{aligned}$$

Subsequently, she finds the intersection range between prices in $\boldsymbol{B}$ and $\boldsymbol{S}$. Then, for this range, take the middle point as MCP denoted by p, and the lowest cumulative quantity as MCV denoted by l. She generates an order $\mathcal{M}$ with commitments to p and l as follows:

$$\begin{aligned}
P &\leftarrow \texttt{Com}(p, 0),\ L \leftarrow \texttt{Com}(l, 0) \\
\mathcal{M} &= (P, 0, L, p, 0, 0, 0, l, 0)
\end{aligned}$$

Note that, the blinding values in commitments of $\mathcal{M}$ are set to zero as we want the commitments to be binding only. Moreover, p and l will be posted on the smart contract eventually, we just need them in commitment form to be utilized in the Sort protocol. Finally, she inserts $\mathcal{M}$ in both $\boldsymbol{B}$ and $\boldsymbol{S}$ while preserving the ordering:

$$\boldsymbol{B}.\texttt{Insert}(\mathcal{M}),\ \boldsymbol{S}.\texttt{Insert}(\mathcal{M})$$

Now, Alice utilizes protocol 4.3.3 to prove the correctness of MCP p and MCV l as follows:

$$\begin{aligned}
\pi_1 &\leftarrow \texttt{Prove}(\boldsymbol{B}.(\boldsymbol{U}, \boldsymbol{u}, \boldsymbol{r_1}), \texttt{DESCEND}) \\
\pi_2 &\leftarrow \texttt{Prove}(\boldsymbol{B}.(\boldsymbol{V_c}, \boldsymbol{v_c}, \boldsymbol{r_c}), \texttt{ASCEND}) \\
\pi_3 &\leftarrow \texttt{Prove}(\boldsymbol{S}.(\boldsymbol{U}, \boldsymbol{u}, \boldsymbol{r_1}), \texttt{ASCEND}) \\
\pi_4 &\leftarrow \texttt{Prove}(\boldsymbol{S}.(\boldsymbol{V_c}, \boldsymbol{v_c}, \boldsymbol{r_c}), \texttt{ASCEND}) \\
\boldsymbol{\pi} &= (\pi_1, \pi_2, \pi_3, \pi_4)
\end{aligned}$$

In the smart contract, the indices of orders in `Bids` and `Offers` depend entirely on their arrival time. Hence, Alice creates two positioning vectors $\boldsymbol{\chi}$ and $\boldsymbol{\gamma}$ that will be used by the smart contract as proxies to access `Bids` and `Offers` in their sorted order, respectively.

Finally, Alice sends a transaction which contains the parameters $(p, l, \boldsymbol{\chi}, \boldsymbol{\gamma}, \boldsymbol{\pi})$ to the smart contract.

The smart contract checks that the transaction is sent by Alice between block-heights t_2 and t_3. Then, it checks whether $p, l \in [0, 2^n - 1]$. After that, it appends the order $\mathcal{M}$ in `Bids` and `Offers`. Finally, it verifies the proofs $\boldsymbol{\pi}$ before accepting and storing p and l.

Upon successful verification, the smart contract refunds the collateral deposits to Alice and owners of unsettled orders. On the other hand, the smart contract keeps the collateral deposits of owners of executed orders locked for the settlement of the assets exchange phase off-chain. Conversely, if the verification was not successful or the Alice failed to send the proofs $\boldsymbol{\pi}$ before block-height t_3, then the smart contract slashes her deposit and refunds the traders.

4.5 Performance Evaluation

4.5.1 Environment

We implemented a basic prototype on Ethereum. We used a local Ethereum full node on a Dell Inspiron 7577 laptop with the 6th generation Intel Core i5 CPU and 16-GB of RAM. The prototype consists of two key components: a smart contract and a client interface to create transactions and generate proofs. For compatibility purposes, we instantiate our protocol using the 128-bits security elliptic curve BN128 since it is the only supported elliptic curve by Ethereum. Furthermore, we set the bit-width for price and quantity values to $n = 16$ bit. The smart contract is written in Solidity, while the client interface is implemented in JavaScript.

Table 4.1: Performance of Sort and CCE protocols.

Performance	#	Sort protocol	CCE Protocol
Proof size	$\mathbb{G}$	$2(log_2(n) + log_2(m)) + 4$	3
	$\mathbb{Z}_p$	5	2
Verifier operations	`mul`	$11 + 2n + m$	8
	`add`	$7 + 2n + m$	5

4.5.2 Evaluation

We report the measurements of the two main building blocks that constitutes the periodic auction protocol, namely, CCE and Sort protocols n Table 4.1. The proof size is measured by the number of elements in $\mathbb{G}$ and $\mathbb{Z}_p$. For the verifier, we report the number of elliptic curve operations required to verify proofs.

In Ethereum, the point addition and point multiplication operations cost 150 and 6000 gas, respectively. Hence, we can measure the transaction gas cost of verifying the submitted proofs. In Fig.4.6, we report the performance measurements: proof size, prover time, and gas cost of proof verification by the smart contract with respect to the total number of orders for the transactions: `SubmitOrder`, `RevealOrder`, and `ClearMarket`. Obviously, the transaction `SubmitOrder` and `RevealOrder` have constant measurements as opposed to `ClearMarket` transaction which scale linearly with the number of orders.

The current block gas limit on Ethereum is roughly 10M gas. Hence, we can estimate the number of transactions that fit in a single block. More importantly, we can estimate the theoretical number of `SubmitOrder` and `RevealOrder` transactions that the smart contract can receive during the first and second phases for different phase lengths as shown in Fig 4.6. The `SubmitOrder` transaction incurs the cost of verifying Sort proof where $n = 16$ and $m = 2$. Similarly, the `RevealOrder` transaction incurs the cost of verifying CCE proof. Accordingly, the transaction cost of `SubmitOrder` and `RevealOrder` are roughly 276150 and 48750 gas, respectively. In practice, the gas costs for transactions are higher due to data access and control flow operations.

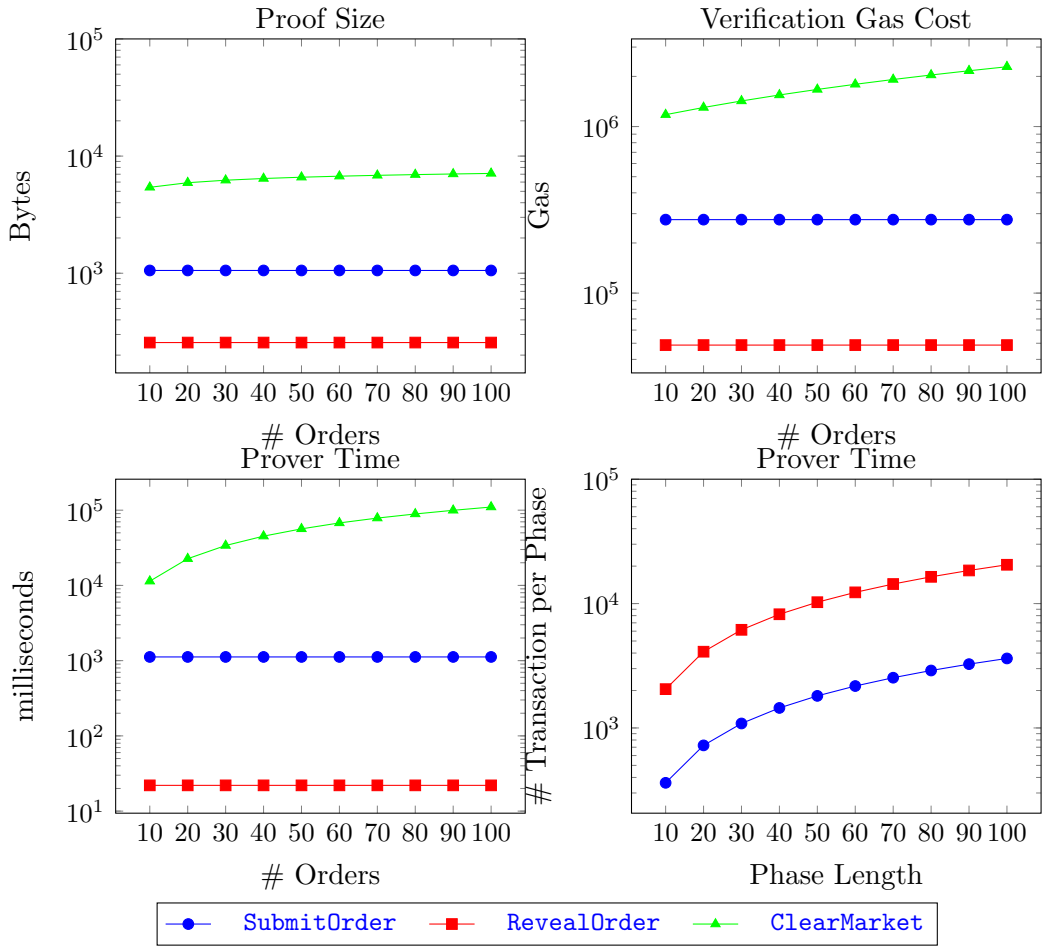

Figure 4.6: Performance measurements of the periodic auction protocol

Furthermore, we can estimate the highest number of orders that can be processed by a single `ClearMarket` transaction before exceeding the 10M gas block limit. Typically, the `ClearMarket` transaction requires verification of two Sort proofs for M bids and two Sort proofs for N offers. For convenience, assume that we have an equal number of bids and offers $M = N$, hence, the `ClearMarket` transaction incurs the verification cost of four Sort proofs of M commitments. Accordingly, the `ClearMarket` transaction can theoretically process up to ≈ 728 orders before exceeding the block gas limit. Certainly, in practice, this number is lower due to the gas cost associated with operations other than proof verification.

4.6 Summary

We presented publicly verifiable secrecy preserving periodic auction protocol. The protocol depends on two zero-knowledge proofs, namely, proof of consistent commitment encryption and proof of ordering. Furthermore, we implemented a prototype and evaluated its performance to assess its feasibility. Based on the result, we believe that the periodic auction protocol is a feasible and secure alternative to dark pools.

Chapter 5

Decentralized Netting Protocol

5.1 Introduction

Traditionally, banks use Real-Time Gross Settlement (RTGS) systems [44] to settle inter-bank payment instructions. The country's central bank is the sole operator of its RTGS system. It enforces each bank to have a local account with liquidity above a certain limit. To settle a payment instruction, the central bank debits the instruction's amount from the sender account while crediting the same amount to the recipient atomically. The settlement process is instantaneous if the sender has sufficient liquidity. Otherwise, the payment instruction is pushed to an outgoing queue associated with the sender's account. The outgoing queue is a priority queue where higher priority payment instructions are pushed ahead of lower ones. Moreover, payment instructions having the same priority level are settled according to a "First In First Out" (FIFO) policy. Once the sender's liquidity becomes sufficient, the highest priority pending outgoing payment instructions are settled automatically.

Gridlock refers to the state when pending payment instructions cannot be settled on a gross basis due to insufficient liquidity. Therefore, to resolve the gridlock state, RTGS systems utilize a Liquidity Saving Mechanism (LSM) [57] to settle payment instructions

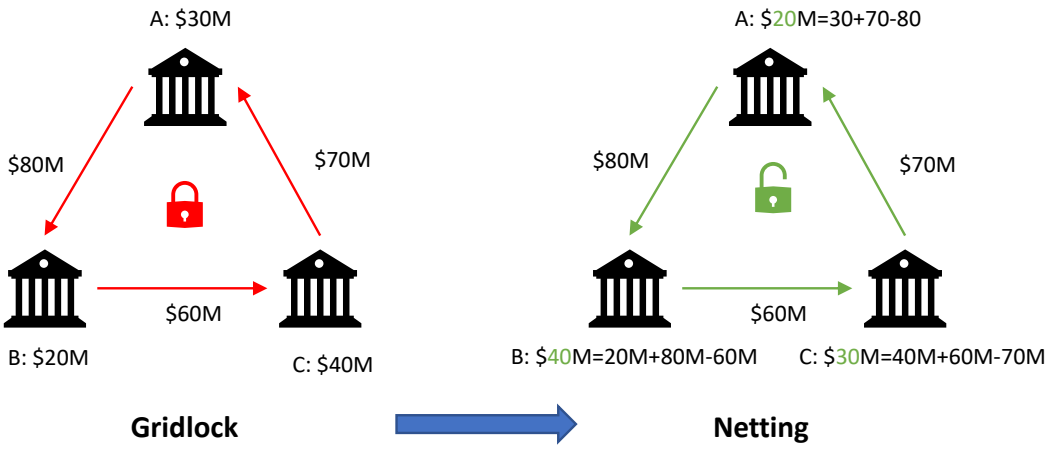

Figure 5.1: An example for resolving gridlock state in RTGS

on a netting basis. To illustrate the gridlock resolution, consider the scenario shown in Fig. 5.1 where the RTGS system is initially in a gridlock state because none of the banks has sufficient liquidity to settle its outgoing payment instruction. Although banks can borrow some funds from the central bank to resolve gridlock, the available credit may still be insufficient to settle payment instructions. Therefore, banks prefer to utilize LSM before resorting to credit. Central banks perform LSM since they have a global view of all pending payment instructions.

With the growing volume of inter-bank payments, contemporary RTGS systems face many security challenges. Most importantly, central banks have privileges on all payment instructions, and they require unconditional trust to maintain the ledgers in RTGS. Moreover, RTGS as centralized systems are vulnerable to the inherent single point of failure problem. To face these challenges, some RTGS operators are beginning to embrace the blockchain technology. Project Jasper [19] and Project Ubin [61] are some of such successful attempts to utilize blockchain. Blockchain alone is not a silver bullet to solve all of the above problems, however, one can build cryptographic protocols utilizing it to provide the required functionality in a trustless manner while preserving participants' privacy (e.g., see [31–33]). More specifically, the migration of traditional RTGS to the

blockchain requires an efficient decentralized netting method to resolve gridlock while delivering better privacy for participants.

5.2 Related Work

Project Jasper [19] and Project Ubin [61] are two successful deployed projects that investigate the advantages of migrating traditional RTGS to the blockchain. They managed to resolve the single point of failure issue while achieving an immediate gross settlement. However, they do not include LSM functionality, which is an important requirement in RTGS systems.

Wang *et al.* [76] introduced an end-to-end prototype based on Hyperledger Fabric enterprise blockchain platform [3]. The prototype supports gross settlement, gridlock resolution, and reconciliation for inter-bank payment business. Gridlocks are resolved through a timestamp-based algorithm, which shares enough information among participants without the risk of privacy violation. The prototype relies on a central party to check the correctness of the netting result. Furthermore, while it hides the amounts in payment instructions, it reveals the net amounts.

Cao *et al.* [17] proposed a decentralized netting protocol that guarantees netting correctness. The participants submit their local settlements to a smart contract. The protocol hides payment amounts using Pedersen commitments and utilizes extensive zero-knowledge range proofs. Furthermore, to obfuscate the links between senders and recipients, participants can send payment instructions with empty amounts. Obviously, these empty instructions are also associated with zero-knowledge proofs, which add extra overhead to the protocol.

5.3 The Netting Problem

We follow the notations defined in [17] to illustrate the netting problem. Let N denote the number of participants and P_i refer to the ith participant. Let $\boldsymbol{Q}$ denote the list of all outgoing payment queues, which is defined as:

$$\begin{aligned}
\boldsymbol{Q} &= [\boldsymbol{Q}_1, \ldots, \boldsymbol{Q}_N] \\
\boldsymbol{Q}_i &= [Q_{i,1}, \ldots, Q_{i,n_i}] \text{ where } n_i = |\boldsymbol{Q}_i| \\
Q_{i,k} &= (Rec_{i,k}, v_{i,k}) \text{ where } v_{i,k} > 0 \\
& \quad\; Rec_{i,k} \in \{P_j\}_{j=1,\, j\neq i}^{N}
\end{aligned}$$

A payment instruction $Q_{i,k}$ consists of two fields: a recipient denoted by $Rec_{i,k}$ and an outgoing amount denoted by $v_{i,k}$. Note that, if there are multiple payment instructions to the same recipient, then they are aggregated in a single payment instruction $Q_{i,k}$, where $v_{i,k}$ is the total amount. Furthermore, for each payment queue $\boldsymbol{Q}_i$, we define a settlement indicator $\boldsymbol{x}_i$ as:

$$\begin{aligned}
\boldsymbol{x} &= [\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N] \\
\boldsymbol{x}_i &= [x_{i,1}, \ldots, x_{i,n_i}], \text{ where } x_{i,k} = \begin{cases} 1 & \text{if payment } Q_{i,k} \text{ will be settled} \\ 0 & \text{otherwise} \end{cases}
\end{aligned}$$

Given $\boldsymbol{x}$, we define the following functions for a participant P_i:

$$\begin{aligned}
T_i(\boldsymbol{x}) &= \textstyle\sum_{k=1}^{n_i} x_{i,k} \\
S_i(\boldsymbol{x}) &= \textstyle\sum_{k=1}^{n_i} x_{i,k} v_{i,k} \\
R_i(\boldsymbol{x}) &= \textstyle\sum_{j=1}^{N} \sum_{k=1}^{n_i} x_{j,k} v_{i,k} \text{ where } Rec_{j,k} = P_i
\end{aligned}$$

where $T_i(\boldsymbol{x})$ denotes the number of payment instructions that will be settled, $S_i(\boldsymbol{x})$ denotes the total outgoing amount, and $R_i(\boldsymbol{x})$ denotes the total incoming amount. Let B_i and B_i' denote the ex-ante and ex-post balances of participant P_i (i.e., balance before and after

netting, respectively). The balance relationship is defined as:

$$B'_i = B_i - S_i(\boldsymbol{x}) + R_i(\boldsymbol{x}) \tag{5.1}$$

The netting problem is to find the solution that satisfies the following constraints:

1. The **liquidity** constraint which dictates that the ex-post balance of each participant after netting must be non-negative.

$$\forall\, i \in [N] \;\; B'_i \geq 0$$

2. The **sequence** constraint which requires payment instructions to be settled according to their priority order [4].

$$\forall\, i \in [N],\; k \in [n_i - 1] \;\; x_{i,k+1} \leq x_{i,k}$$

Let $h(\boldsymbol{x}_i)$ denote the index of lowest priority payment instruction in $\boldsymbol{Q}_i$ that will be settled, which is defined as:

$$h(\boldsymbol{x}_i) = \begin{cases} 0 & \text{if } \forall\, k \in [n_i] \;\; x_{i,k} = 0 \\ max(k) & \text{where } x_{i,k} = 1 \end{cases}$$

3. The **optimality** constraint which ensures settling the highest possible number of payment instructions by maximizing $\sum_{i=1}^{N} T_i(\boldsymbol{x})$.

5.3.1 Decentralized Netting Protocol

We describe the decentralized netting protocol introduced in [17] to resolve the gridlock over a variable number of rounds. The protocol assumes transparent communication between participants. In other words, the protocol does not provide any privacy,

and participants have access to all payment instructions $\boldsymbol{Q}$ and settlement indicators $\boldsymbol{x}$.

The protocol starts with the initial assumption that all payment instructions will be settled where $\forall\, i \in [N],\ k \in [n_i]\ \ \boldsymbol{x}_{i,k} = 1$. Certainly, this assumption may be invalid, (i.e., some but not all payment instructions will be actually settled). Accordingly, for each round t, participants run Algorithm 4 to update their settlement indicators $\boldsymbol{x}^t$. Eventually, the protocol stops when there are no further updates in the settlement indicators.

Algorithm 4: `UpdateIndicator`

Input:

- $\boldsymbol{x}_i^{t-1}$: Settlement indicator queue for round t and participant i
- $\boldsymbol{Q}$: Payment instructions

Output:

- $\boldsymbol{x}_i^t$: Settlement indicator for next round
- B_i': Ex-post balance

$\boldsymbol{x}_i^t \leftarrow \boldsymbol{x}_i^{t-1}$
$k \leftarrow h(\boldsymbol{x}_i^t)$
while $k \geq 0$ **do**
    $B_i' \leftarrow B_i - S_i(\boldsymbol{x}^{t-1}) + R_i(\boldsymbol{x}^{t-1})$
    if $B_i' < 0$ **then**
        $x_{i,k}^t \leftarrow 0$
        $k \leftarrow k - 1$
    else
        `break`
return $(\boldsymbol{x}_i^t, B_i')$

The main objective of Algorithm 4 is to iteratively unsettle lowest priority payment instructions until the three constraints are satisfied. It checks for non-negative ex-post balance, thereby enforcing the liquidity constraint. Furthermore, it enforces the sequence constraint by iterating over payment instructions from lowest to highest priority. Finally, it stops when the optimality constraint is satisfied once the highest possible number of payment instructions that meet the liquidity and sequence constraints.

Eventually, the protocol halts when there is no further change in settlement indi-

cators in the final round (i.e., $\boldsymbol{x}^t = \boldsymbol{x}^{t-1}$). The end result is either learning the netting solution or reaching a deadlock where the settlement indicator of each participant is $\boldsymbol{x}_i^t = [0]_1^{n_i}$ (i.e., no participant can settle any payment instruction on a netting basis). In the former, participants can update their ex-post balance B_i' of participants. Conversely, in the latter, participants have to deposit more liquidity to resolve the deadlock and reset the protocol.

5.4 Privacy Preserving Netting Protocol Design

To protect the privacy of participants, we modify the decentralized netting protocol to (i) conceal payment amounts and (ii) hide links between senders and recipients. It is worth mentioning that participants in our context refer to banks not individuals. In particular, inter-bank payments are aggregated individuals transactions. Hence, our protocol is designed to preserve the privacy of involved banks in inter-bank payments transactions. Our approach to fulfilling the first objective is to utilize ElGamal encryption. We did not use Pedersen commitments as followed in [17], mainly because the sender would have to open a communication link with the recipient to send the opening values, which can be leaked and compromise the recipient's privacy. When a participant sends a payment instruction to the smart contract, it sends an encrypted payment amount without including the recipient's identity. Subsequently, other participants will individually try to decrypt it locally using their own private keys. Certainly, only the intended recipient will successfully decrypt it. Consequently, no link is publicly established on the smart contract between senders and recipients, which fulfills our second objective.

Later on, after resolving the gridlock, each participant will send a zero-knowledge proof to the smart contract to prove the correctness of its ex-post balance based on ex-ante balance, outgoing amounts, and incoming amounts. However, since we removed recipients' identities from payment instructions, the smart contract cannot determine the

incoming payment instruction for any participant. Nonetheless, we must ensure that participants do not use arbitrary ciphertext as their incoming amounts. In other words, each participant must prove that its incoming payment instructions are part of the list containing all payment instructions sent to the smart contract. To solve this challenge, the smart contract utilizes a Merkle tree to accumulate all payment instructions. Given the public state of the smart contract, participants can clearly see the paths in that tree. As a result, each participant can prove that it knows the Merkle tree paths for its incoming payment instructions without revealing them, which in turn effectively enforces the recipient's privacy (more detail in Section 5.4.6).

5.4.1 Overview of the Protocol

1. The protocol starts with a Setup phase where an MPC protocol is used to generate the CRS for the zkSNARK proof system, and a smart contract is deployed on the blockchain.

2. Participants initialize their encrypted ex-ante balance on the smart contract.

3. Each participant submits its payment instructions to the smart contract and utilize zkSNARK to prove their correctness.

4. Participants scan the smart contract to learn their incoming amounts based on their success in decrypting the submitted payment instructions.

5. Participants begin to resolve the gridlock over a number of variable rounds as follows:

   (a) In the first round, they assume all payment instructions will be settled and set their settlement indicator $\boldsymbol{x}_i^{t=1} = [1]^{n_i}$.

   (b) In subsequent round t, each participant P_i locally runs Algorithm 4 to update its settlement indicator from $\boldsymbol{x}_i^{t-1}$ to $\boldsymbol{x}_i^t$, and send $\boldsymbol{x}_i^t$ to the smart contract.

(c) If there are no changes (i.e., $\boldsymbol{x}^t = \boldsymbol{x}^{t-1}$), the smart contract builds a Merkle tree over payment instructions that will be settled; otherwise, participants repeat the above step again for the new round $t \leftarrow t + 1$.

6. Participants send transactions to the smart contract containing their encrypted ex-post balance and a zkSNARK proof to prove its correctness.

7. Upon successful verification, the smart contract accepts the ex-post balance.

5.4.2 Setup

To promote a modular design, the protocol utilizes several smart contracts that are deployed on the blockchain.

1. `Registry`: it contains ElGamal public encryption keys of all participants.

2. `Serials`: The sole purpose of this smart contract is to prevent double-spending of incoming payment instructions by tracking the serial numbers (also known as nullifiers [67]) of the settled ones (more detail in Section 5.4.6).

3. $\texttt{MerkleTree}_{pk}$: it accumulates the public keys of all participants.

4. $\texttt{MerkleTree}_{s}$: it accumulates all payment instructions that will be settled after gridlock resolution.

5. `Verifier`: it verifies zkSNARK proofs submitted by participants, and it contains the verification key vk generated by zkSNARK `Setup` algorithm.

6. `Main`: this is the main smart contract which handles the deposit of funds, controls the transitions between phases and rounds, and utilizes the above smart contracts.

5.4.3 Initializing Ex-ante Balance

To join the protocol, a participant P_i deposits its ex-ante balance B_i in `Main` as shown in Fig. 5.2. Initially, `Registry` checks whether the transaction sender is authorized based on its address and returns its public key on success. If this is the first deposit by the sender, then the sender's ex-ante balance is set to the encryption of the `msg.value` by the sender's public key and randomness $r = 0$. Otherwise, the deposit is to inject more liquidity into the sender's balance by utilizing the additively homomorphic property of ElGamal encryption.

Deposit()

```
Require(Registry.ContainsAddress(msg.sender))
y ← Registry.GetPublicKey(msg.sender)
v ← msg.value
c ← Enc(v, 0, y)
If Balance[msg.sender] is null
    Balance[msg.sender] ← c
Else
    Balance[msg.sender] ← Balance[msg.sender] + c
```

Figure 5.2: Pseudocode for Deposit function

Generally, a blockchain with a public state such as Ethereum cannot hide the deposit value of ex-ante balance. However, after resolving gridlock, participants will send their ex-post balances in encrypted form to `Main`.

5.4.4 Submitting Payment Instructions

To submit payment instructions while hiding amounts and preserving recipient's privacy, a participant P_i sends a transaction to `Main` containing its outgoing payment

$\underline{\texttt{Submit}(X, W)}$

Statement X:

- $\boldsymbol{Q}_i$: Encrypted payment instructions from a participant i
- $root_{pk}$: Root of $\texttt{MerkleTree}_{pk}$

Witness W:

- $\{v_{i,k}\}_{k=1}^{n_i}$: Payment amounts
- $\{y_{i,k}\}_{k=1}^{n_i}$: Recipients public keys
- $\{\pi_k\}_{k=1}^{n_i}$: Merkle proofs of membership

Assert the following:

$\texttt{For}\ \ k \in [n_i]$

$v_{i,k} \in [0, 2^l - 1]$

$c_{i,k} = \texttt{Enc}(v_{i,k}, y_{i,k})$

$1 = \texttt{Merkle.Verify}(y_{i,k}, root_{pk}, \pi_k)$

Figure 5.3: zkSNARK Submit circuit

queue $\boldsymbol{Q}_i$ without indicating recipients' identities (i.e., their public keys).

$$\begin{aligned} \boldsymbol{Q}_i &\leftarrow [c_{i,1}, \ldots, c_{i,n_i}] \\ c_{i,k} &\leftarrow \texttt{Enc}(v_{i,j}, y_{i,k}) \\ y_{i,k} &\in [y_j]_{j=1,\, j\neq i}^{N} \end{aligned}$$

A participant P_i utilizes zkSNARK to prove that (i) the amounts encrypted in $\boldsymbol{Q}_i$ fall in range $[0, 2^l - 1]$ for a system parameter l, and (ii) the ciphertext is generated using public keys accumulated in $\texttt{MerkleTree}_{pk}$ with $root_{pk}$ as its root. Technically speaking, P_i utilize zkSNARK to generate a proof for the `Submit` circuit as shown in Fig. 5.3.

The participant P_i sends a transaction `Submit` to `Main` containing the statement X and zkSNARK proof π as shown in Fig. 5.4. `Main` checks that the sender is one of the authorized participants. Then, if it successfully verifies the proof π_i, it stores $\boldsymbol{Q}_i$ in

the list `Out` and initializes the settlement indicator by setting 1 in all indices (see Section 5.3.1) which gives the assumption that all payment amounts will be settled. Otherwise, it reverts the transaction.

Submit$(\boldsymbol{Q}_i, \pi_i)$

$(\boldsymbol{Q}_i, root_{pk}) \leftarrow$ `Parse`(X)

`Require(Registry.ContainsAddress(msg.sender))`

`Require(MerkleTree`$_{pk}$`.Roots.Contains(`$root_{pk}$`)`

`Require(Verifier.Verify(`X, π`))`

`Out[msg.sender]` $\leftarrow \boldsymbol{Q}_i$

$n_i \leftarrow |\boldsymbol{Q}_i|$

`Indicator[msg.sender]`$\leftarrow [1]^{n_i}$

Figure 5.4: Pseudocode for Submit function

Once the transaction is accepted by `Main`, each participant P_j, where $j \neq i \ \wedge \ j \in [N]$ scans payment instructions in `Main.Out`$[P_i$`.Address`$]$ and try to decrypt them locally using its private key. Accordingly, P_j will learn if it is the intended recipient based on successful decryption.

5.4.5 Updating Settlement Indicators

In this phase, participants utilize decentralized netting protocol to resolve gridlock over a variable number of rounds before converging to the netting solution. `Main` implements the transitions of rounds as shown in Fig. 5.5. An important caveat here is that smart contracts do not execute code unless they are instructed to do so via transactions. Hence, we require one of the participants P' to send transactions to `Main` to transit between rounds and states. The gridlock resolution proceeds as follow:

1. Participant P' submits a `Start` transaction to initiate the first round of decentralized

netting protocol.

2. Each participant P_i runs Algorithm 4 to update its settlement indicator $\boldsymbol{x_i}$ and submits a `Receive` transaction within its predefined time-window.

3. Participant P' submits a `Final` transaction to determine the next state of this phase. More precisely, if there are changes in settlement indicators, then participants proceed to the next round and repeat Step 2. Otherwise, the protocol has either reached a deadlock state or it has successfully resolved the gridlock. In the former case, participants have to inject more liquidity by submitting `Deposit` transactions, then reset the current round in Step 2. Conversely, in the latter case, `Main` initializes $\texttt{MerkleTree}_s$ to accumulate all payment instructions that will be settled according to the current indicators, and sets the flag `Resolved` to transit to next phase.

5.4.6 Updating Ex-Post Balance

After resolving the gridlock, each participant P_i utilizes the `UpdateBalance` zk-SNARK circuit shown in Fig. 5.6 to generate a proof π about the correctness of its encrypted ex-post balance B'. Then, P_i submits an `Update` transaction containing the parameters π, B' and $\boldsymbol{sn}$ as shown in Fig. 5.7.

The `UpdateBalance` circuit asserts the following:

- The participant P_i knows the private key x corresponding to the public key y.
- $\boldsymbol{r}$ contains the incoming payment amounts decrypted from $\boldsymbol{R}$ by its private key x.
- The serial numbers in $\boldsymbol{sn}$ correspond to the hash of incoming payment instructions with the private encryption key x. Accordingly, valid incoming payment instructions must have unique serial numbers to prevent double-spending.
- P_i knows valid membership proofs $\boldsymbol{\psi}$ to prove that the payment instructions in $\boldsymbol{R}$ are accumulated in $\texttt{MerkleTree}_s$.

Start ()

```
Require(start = false)
start ← true
t ← 1
updated ← false
```

Receive($\boldsymbol{x}_i$)

```
Require(start = true)
Require(ValidReceiveWindow(Block.Number))
Require(Registry.ContainsAddress(msg.sender))
If Indicator[msg.sender] ≠ x_i
   Indicator[msg.sender] ← x_i
   updated ← true
```

Final()

```
Require(ValidFinalWindow(Block.Number))
If updated = true
   t ← t + 1
   updated ← false
Else
   If Indicator.AllZeros() = true
      DEADLOCK ← true
   Else
      Out_s ← GetSettledOnly(Out,Indicator)
      MerkleTree_s ← MerkleTree(Out_s)
      RESOLVED ← true
```

Figure 5.5: Pseudocode for updating indicators

$\underline{\texttt{UpdateBalance}(X, W)}$

Statement X:

- $root_s$: Root of $\texttt{MerkleTree}_s$.
- y: Public key of P_i.
- $\boldsymbol{S}$: List of settled payment instructions according to indicator $\boldsymbol{x}_i$.
- U: Encryption of highest priority unsettled payment instruction.
- B, B': Encrypted ex-ante and ex-post balances.
- $\boldsymbol{sn}$: List containing serial numbers of incoming payment instructions.

Witness W:

- x: Private key corresponding to the public key y.
- $\boldsymbol{R}, \boldsymbol{r}$: List of encrypted and decrypted payments instructions.
- $\boldsymbol{\psi}$: List of Merkle proofs of membership for $\boldsymbol{R}$ in $\texttt{MerkleTree}_s$.
- $\boldsymbol{s}$: List of outgoing payment amounts encrypted in $\boldsymbol{S}$.
- $\boldsymbol{y}_s$: List of public keys for recipients of payment instructions in $\boldsymbol{S}$.
- y_u, u: recipient's public key and decrypted highest priority unsettled payment amount.
- b, b': Ex-ante and ex-post balances encrypted in B

Assert the following:

$$
\begin{aligned}
&y = g^x \\
&\texttt{For } k \in [n_i] \\
&\quad r_k = \texttt{Dec}(R_k, x) \\
&\quad sn_k = H(x || R_k) \\
&\quad \texttt{Merkle.Verify}(root_s, R_k, \psi_k) = 1 \\
&\texttt{For } s_j \in \boldsymbol{s}, y_j \in \boldsymbol{y}_s \\
&\quad S_j = \texttt{Enc}(s_j, y_j) \\
&U = \texttt{Enc}(u, y_u) \ \wedge \ B = \texttt{Enc}(b, y) \ \wedge \ B' = \texttt{Enc}(b', y) \\
&b' = b - \sum \boldsymbol{s} + \sum \boldsymbol{r} \geq 0 \\
&b' - u < 0
\end{aligned}
$$

Figure 5.6: zkSNARK UpdateBalance circuit

- The outgoing payment amounts in $\boldsymbol{s}$ are encrypted to ciphertext in $\boldsymbol{S}$ by the public keys in $\boldsymbol{y}_s$.
- U, B, and B' are the ciphertext corresponding to u, b, and B', respectively.
- The liquidity constraint is satisfied.
- The optimality constrains is satisfied by showing that the highest priority unsettled payment will result in negative ex-post balance.

Update$(\pi, B', \boldsymbol{sn})$

```
Require(Registry.ContainsAddress(msg.sender)
Require(RESOLVED = true)
Require(Unique(sn))
Require(Serials.ContainsAny(sn) = false)
y ← Registry.GetPublicKey(msg.sender)
root_s ← MerkleTree_s.GetRoot()
B ← GetBalance(msg.sender)
S ← GetSettledPayments(msg.sender)
U ← GetHighestUnsettled(msg.sender)
X ← (B', B, y, root_s, S, U, sn)
Require(Verifier.verify(X, π))
Serials.append(sn)
Balance[msg.sender] ← B'
```

Figure 5.7: Pseudocode for Update function

5.5 Performance Evaluation

We evaluate the protocol's performance on Ethereum. The results show that the protocol is feasible and practical to deploy. Recently, Ethereum has gone through multiple planned hard-forks to upgrade its virtual machine (EVM) with new op-codes and features that will help Ethereum transit to Proof-of-Stake (PoS). One of these features is the support for running EC operations inside smart contracts. More specifically, the fork *Byzantium* [21] introduced two pre-compiled contracts: EIP-196 to perform point addition and multiplication operations, and EIP-197 for pairing checks on the curve *BN128*. Accordingly, smart contracts can efficiently verify many cryptographic proofs including zkSNARK [5]. However, the gas cost incurred by EC operations were relatively high which limited the feasibility of some cryptographic protocols on Ethereum. Therefore, the fork code-named *Istanbul* [25] made adjustments to the cost of EC operations. Strictly speaking, the gas cost of point addition, multiplication, and k pairing check got reduced from $500; 40000; 100000 + k \times 80000$ down to $150; 6000; 45000 + k \times 34000$, respectively.

5.5.1 Evaluations

We utilize the zkSNARK construction proposed in [38]. The major advantage of this construction is the small proof size 128-bytes and efficient verifier which requires three pairing checks. Using the gas adjustments brought by *Istanbul* fork in Ethereum, pairing checks cost 45000 gas in addition to 34000 gas for each check. Thus, the verification cost of zkSNARK proof is $147000 = 45000 + 3 \times 34000$ gas. Moreover, the verifier performs EC operations on the public input before verifying the proof. Specifically, for each public input encoded as $\mathbb{F}_p$ element, the verifier invokes two operations: point multiplication and point addition. Consequently, for a proof π with public input of size m elements in $\mathbb{F}_p$, the verification gas cost is $6150m + 147000$. The encoding of an ElGamal ciphertext, a public key, and a hash value are four, two, and one $\mathbb{F}_p$ elements, respectively.

To evaluate our protocol, we estimate the gas cost associated with the elliptic curve operations performed during zkSNARK proofs verification. Generally, participants send zkSNARK proofs in `Submit` and `Update` transactions. In Fig. 5.8, we report the gas cost for these transactions with respect the numbers of outgoing and incoming payment instructions n_i and m_i, respectively.

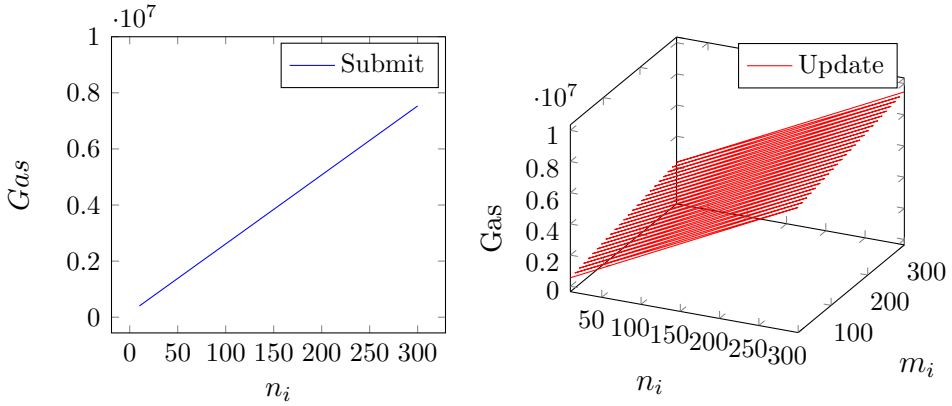

Figure 5.8: Gas cost for Submit and Update transactions

In the `Submit` transaction, the public input consists of a list of n_i ciphertext for payment instructions in $\boldsymbol{Q}_i$, and a hash value for the root of $\texttt{MerkleTree}_{pk}$. Accordingly, we can estimate the gas cost for proof verification as $6150(4n_i+1)+147000$ gas. Similarly, in the `Update` transaction, the public input consists of seven parameters: three ciphertext (B', B, U), a list of n_i ciphertext for payment instructions in $\boldsymbol{S}$, a public key y, a hash value for the root of $\texttt{MerkleTree}_s$, and a list of m_i serial numbers. Consequently, we can estimate the gas cost as $6150(15 + 4n_i + m_i) + 147000$.

Table 5.1 shows the size and cost in USD for `Submit` and `Update` transactions in the setting where $n_i = m_i = 10$. At the time of writing in August 2020, one Ether coin costs ≈\$400 USD, and the average gas price is $\approx 200\ Gwei = 200 \times 10^{-9}$ Ether. The gas cost of a transaction is computed as the incurred gas times gas price in Ether times Ether price in USD.

Based on the evaluation results, we believe the protocol is practical to deploy on Ethereum. For example, with the current block gas limit ≈ 10M gas on Ethereum,

Table 5.1: Size and cost in USD for `Submit` and `Update` transactions

Transaction	Size (Bytes)	Gas	Cost (USD)
Submit	1408	399150	$32
Update	2080	546750	$43.75

we find that the proof verification for `Submit` and `Update` transactions cost 3.9% and 5.4% of the block limit when the number of outgoing and incoming payment instructions $n_i = m_i = 10$, respectively. Furthermore, the protocol will perform much better on permissioned blockchain preferred by banks such as Corda [40] or Hyperledger [3] since there are neither lengthy block mining time nor block gas limit.

5.5.2 Limitations of Decentralized Netting Protocol

In this section, we discuss some limitations that apply to the decentralized netting protocol in [17], and naturally extend to our implementation. First, the protocol in [17] requires all participants to be online at the time of resolving gridlock. However, we argue that participants do not resolve gridlocks immediately once they occur, rather they keep aggregating individuals' transactions during the day. In fact, given the slow settlement of inter-bank payments in practice which takes roughly couple of days, banks can setup a specific time at the end of a business day to resolve gridlock. Therefore, we can safely assume that the online requirement is practically relaxed and does not cause in severe liveness issues. Furthermore, to improve the speed of gridlock resolution, participants can run the decentralized netting protocol off-chain to resolve gridlock without being controlled by the `Main` smart contract. Then, participants can submit correctness proofs and utilize `Main` for settlement only.

5.6 Summary

Financial institutions are embracing blockchain technology to address challenges in traditional RTGS. We designed a privacy-preserving decentralized protocol that resolves gridlock in RTGS. Furthermore, we enhanced the participants' privacy by hiding the links between senders and recipients while providing confidentiality to payment amounts. To assess the feasibility and performance of our protocol, we estimate the proof verification gas cost when deployed on Ethereum.

Chapter 6

Privacy Preserving Market for Non-Fungible Tokens

6.1 Introduction

Non-fungible tokens (NFTs) are unique non-interchangeable digital assets verified and stored using blockchain technology. Quite recently, there has been a surging interest and adoption of NFTs, with sales exceeding $10 billion in the third quarter of 2021. The popular and largest NFT marketplace, Opensea[1] hit $3.4 billion as a sales volume in August 2021 [63]. NFTs are tokens that represent ownership of unique digital items such as art [28], collectibles [24], essays [78], domains [26], and even tickets to access real-world events [75]. Although anyone can trivially copy digital assets, an NFT can have one owner only at a time, and the blockchain secures the ownership status. In particular, the standard NFT smart contract [27] (ERC-721) contains a mapping that associates each NFT identifier with its corresponding owner's address. The smart contract code guarantees that only the owner or approved operators can assign a new owner.

The design of Aegis is primarily motivated by the current limitations of NFT stan-

[1] https://opensea.io

dards and marketplaces design. For instance, ERC-721 specifications [27] require compatible smart contracts to expose the owner's address given the NFT identifier. Typically, privacy-advocate users cannot tolerate this limitation as none would like their entire NFT collections to be accessible to the public. The lack of privacy could also introduce the owners to life-threatening situations. Suppose Bob has an address x that is the owner of an NFT ticket for a real-world event. After scanning x using online services such as *Etherscan*, it turns out that x is also the owner to some of the most expensive and premium NFTs in addition to large fungible assets. Effectively, this information could attract bandits in an attempt to find Bob in the event's small proximity and find ways to extort his private key.

NFT smart contracts adhere to the standard specifications outlined by ERC-721 [27]. In particular, an NFT smart contract maintains a mapping from NFT identifiers (IDs) to their owners' addresses. This mapping is publicly accessible; hence, an observer can trivially determine the NFTs collection owned by an arbitrary user. Moreover, the observer can track how the ownership status of an arbitrary NFT changes over time.

To transfer the ownership of an NFT, the owner sends a `transferFrom` transaction [27] to the NFT smart contract to assign an address as the new owner. Alternatively, the owner can send `approve` [27] transaction to set an *operator*. In practice, the *operator* is a smart contract that changes ownership status based on a specific trigger. For example, users set marketplace smart contracts as operators to their NFTs; thereby, allowing the marketplace to transfer ownership from a seller (i.e., current owner) to a buyer (i.e., new owner) once the trade is complete.

Furthermore, the current design of NFTs marketplaces lacks privacy as well. For instance, the popular NFTs marketplace Opensea allows users to trade NFTs only via public auctions and swaps. Unfortunately, in public auctions, an observer can trivially learn the submitted bids even before they get mined by simply inspecting the *mempool*. Hence, these auctions are susceptible to front-running which is an illegal act that provides

risk-free profits to the front-runners. Moreover, public auctions and swaps reveal sensitive information about the seller, buyer, NFT, and payment amount.

To tackle those problems, we propose Aegis[2], a protocol with the following contributions:

- We design Aegis as a privacy-preserving protocol that allows users to add privacy to their NFT ownership status.
- Similarly, Aegis allows users to maintain private balances of funds in a non-custodial manner.
- More importantly, Aegis allows users to atomically swap their NFTs for payment amounts in a complete privacy-preserving manner without revealing any information about the involved participants, NFTs, and payment amounts.
- We implemented a basic prototype to assess Aegis's performance, and we released its source code on Github[3]. Furthermore, we improved the zkSNARK verification gas cost to be constant without impacting the anonymity set (which can grow up to 2^d, in evaluation, we set $d = 20$, where d is a Merkle tree depth).

6.2 Aegis Overview

Aegis works in UTXO model similar to ZeroCash [67] rather than in the *account* model since it allows for an always increasing anonymity set (see Section 6.4). Hence, transactions in Aegis privately consume old UTXO(s) and generate new ones. Since Ethereum has a public state, then Aegis smart contract needs to conceal users' state in the form of *commitments* to NFT IDs and funds. However, as the smart contract cannot access the committed values, it cannot determine if they are updated correctly. To

[2]Aegis is a shield carried by Zeus and Athena. It is a symbol of protection.
[3]https://github.com/hsg88/Aegis

solve this dilemma, users submit zkSNARK proofs that assert the correctness of state updates without revealing any further information. Upon successful verification, the smart contract accepts the updated state. Furthermore, the smart contract utilizes efficient incremental Merkle trees to accumulate commitments; thereby, reducing the storage requirements and gas cost. Additionally, to prevent double-spending, Aegis leverages the notion of *serial numbers* [67] to privately nullify consumed UTXOs.

6.2.1 Protocol Participants

In Aegis, there are four participants: sellers, buyers, trustless relayers, and a smart contract. Sellers privately own NFTs in Aegis, and they can swap for payment amounts or withdraw them by transferring the ownership from Aegis. Similarly, buyers have private funds in Aegis, which they can swap for NFTs or withdraw. Relayers receive meta-transactions from sellers and buyers and submit transactions to Ethereum while paying the gas cost. They are incentivized by receiving rewards from Aegis based on their contribution. We assume sellers and buyers are implicitly using relayers. For example, *a user sends a transaction to* Aegis implies that the user sends a meta-transaction to a relayer, and the relayer sends a transaction to Aegis.

Aegis has a `Main` smart contract in addition to two internal smart contracts:

- `Main:` it receives transactions from users, and it is the *non-custodial* owner of all deposited NFTs and funds.
- `Merkle:` it is an internal smart contract that implements an efficient incremental Merkle tree.
- `Verifier:` it is an internal smart contract that contains verifying keys and functions for verifying zkSNARK proofs.

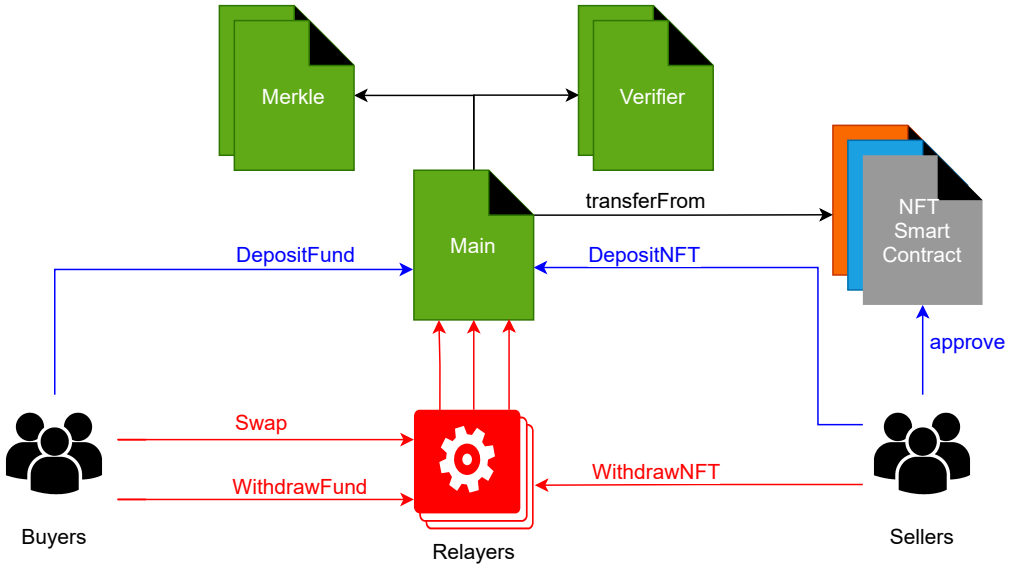

Figure 6.1: Interactions in Aegis. Users directly send transactions in blue, while relayers send transactions in red. Black arrows are calls between smart contracts.

6.2.2 Aegis Transactions

Figure 6.1 shows a high-level overview of transactions in Aegis. Users can directly deposit NFTs and funds to `Main`. A seller and a buyer communicate off-chain to agree on the swap detail, such as the NFT ID and payment amount. Then, the buyer can send `Swap` transaction to settle the exchange. Furthermore, users can withdraw their assets from `Main` to public recipient addresses. More importantly, swap and withdrawal transactions are sent via relayers so that an observer cannot link them to deposit transactions using the gas payer's address.

NFT Transactions.. Aegis does not *mint* NFTs; therefore, sellers have to transfer the ownership of their NFTs in a non-custodial manner to `Main`. Accordingly, sellers initially send `approve` transactions [27] to their NFT smart contracts to assign `Main`'s address as an *operator*. Then, sellers can add privacy to their NFTs ownership status by sending `DepositNFT` transactions to `Main`. Finally, `Main` (i) transfers ownership from the user's address to its address, (ii) generates an NFT commitment, (iii) and accumulates it in the NFTs Merkle tree. Later, a seller can send `WithdrawNFT` transaction containing

a zkSNARK proof of NFT ownership. Upon successful verification, `Main` transfers the ownership to the recipient's address which should be different from the deposit address.

Fund Transactions. Aegis allows buyers to build private funds that they can swap for NFTs in a privacy-preserving manner. Buyers send `DepositFund` transactions that include amounts in Ethers to `Main`. Then, `Main` generates a fund commitment for the deposited amount and accumulates the commitment in the funds Merkle tree. Later, a buyer can send `WithdrawFund` transaction containing a zkSNARK proof of funds. Upon successful verification, `Main` transfers the requested amount to the recipient's address.

Swap Transaction. Aegis allows users to trade NFTs without revealing IDs, payment amounts, and identities. The NFT trade is an atomic swap that either completes successfully or reverts without causing any loss. To understand how atomic swap works in Aegis, consider the following basic protocol that uses digital signatures. Alice wants to transfer an NFT a to Bob in exchange for a payment amount b. Then, Alice signs Bob's asset b, and Bob signs Alice's asset a. Then, Bob sends both signatures to a smart contract which acts as a trusted party holding a and b in its escrow. The smart contract settles the swap only if both signatures are valid for the counter-party's asset. This simple protocol correctly performs atomic swap; however, it lacks privacy since assets and owners are public. Aegis fixes this issue by utilizing zkSNARK proofs as *signatures of knowledge* [39] over commitments instead of digital signatures over plaintext values.

6.2.3 Threat Model

We assume the cryptographic primitives are secure. We further assume the adversary $\mathcal{A}$ is computationally bounded and cannot tamper with the execution of the Aegis smart contracts. Additionally, $\mathcal{A}$ has the capabilities of a miner (i.e., reorder transactions within a blockchain block, and inject its transactions before and after certain transactions). $\mathcal{A}$ can always read all transactions issued to Aegis while propagating over the network. We assume that users can always read from and write to the blockchain state. Moreover,

users utilize *trustless* relayers to submit non-deposit transactions on their behalf.

6.2.4 System Goals

We design Aegis such that it achieves the following goals:

- *Privacy:* an adversary should not link a deposit transaction to any swap or withdrawal transaction.
- *Balance:* an adversary cannot successfully swap or withdraw assets belonging to honest users.
- *Atomic Swap:* honest sellers and buyers can successfully swap their assets atomically, or the swap reverts entirely without causing any loss.
- *Availability:* users should always be able to submit transactions without any risk of censorship.
- *Compatibility:* Aegis should be compatible with existing NFT smart contracts standard [27] without requiring any changes to the deployed contracts.

6.3 Aegis Detailed Construction

6.3.1 Building Blocks

Hash Functions. Let H_2 and H_3 be collision-resistant hash functions that map two and three elements from $\mathbb{F}_p$ to an element in $\mathbb{F}_p$, respectively.

$$\mathsf{H}_2 : \mathbb{F}_p \times \mathbb{F}_p \to \mathbb{F}_p$$

$$\mathsf{H}_3 : \mathbb{F}_p \times \mathbb{F}_p \times \mathbb{F}_p \to \mathbb{F}_p$$

Pseudorandom Functions. To utilize Aegis, a user samples a seed $s \xleftarrow{\$} \mathbb{F}_p$ and keeps it private. We construct PRF^{addr} and PRF^{sn} using H_3 to generate spending addresses and

serial numbers as follows:

$$\mathsf{PRF}^{addr}(s, \cdot) = \mathsf{H}_3(0, s, \cdot)$$

$$\mathsf{PRF}^{sn}(s, \cdot) = \mathsf{H}_3(1, s, \cdot)$$

Commitment Scheme. We instantiate a commitment scheme `Com` using H_2 to generate NFTs and funds commitments as follows:

$$\mathtt{Com} : \mathbb{F}_p \times \mathbb{F}_p \rightarrow \mathbb{F}_p$$

$$\mathtt{Com}(v, addr) = \mathsf{H}_2(v, addr)$$

where v denotes an NFT ID or a fund amount, and $addr$ is a spending address generated randomly for each commitment (i.e., commitment randomness).

Merkle Trees. We instantiate Merkle trees as incremental binary trees of depth d using H_2 over the left and right children. More importantly, `Merkle` smart contract implements an efficient append-only Merkle tree as shown in Fig. 6.2. In particular, `Merkle` stores the last l roots and d elements comprising the Merkle proof for the first empty leaf. For example, in Fig. 6.2, the Merkle proof for the first empty leaf in the left tree is $\pi = (5, 00, 13)$. Using π and the new element 6, `Merkle` can compute the new root 15 in the right tree. Finally, `Merkle` sets $\pi = (0, 11, 13)$ as the Merkle proof for the next empty leaf.

It is worth mentioning that using the Merkle proof π stored on `Merkle` is insufficient for users to generate Merkle proofs for their commitments. Therefore, `Main` emits `NewCommitment` *event* for every new commitment. Consequently, by scanning Ethereum for these events, users can collect every accumulated commitment for building the entire Merkle tree off-chain. Hence, users can successfully generate proofs of membership for any commitment.

Coin Structure. We use the term *coin* [67] to refer to a data object that represents NFTs and funds in Aegis. The coins for NFTs and funds have the exact structure, yet their commitments are accumulated in two separate instances of `Merkle` smart contract.

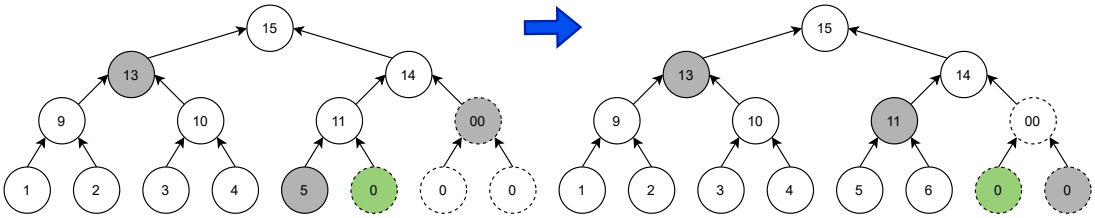

Figure 6.2: Illustration of accumulating a new element 6. The shaded circles are Merkle proof π for the first empty leaf which is depicted in green.

.

To generate a coin $\boldsymbol{c}$ for a value v, a user with a private seed s samples $\rho \xleftarrow{\$} \mathbb{F}_p$. Then, the user utilizes s and ρ to generate a spending address $addr$, a serial number sn as follows:

$$addr \leftarrow \mathsf{PRF}^{addr}(s, \rho)$$

$$sn \leftarrow \mathsf{PRF}^{sn}(s, \rho)$$

$$cm \leftarrow \mathtt{Com}(v, addr)$$

$$\boldsymbol{c} \leftarrow (\rho, v, addr, sn, cm)$$

The user keeps ρ private which will be used as part of the witness to generate zkSNARK proofs for spending the coin $\boldsymbol{c}$. For transactions in `Main`, the user sends cm for new coins and sn for spent coins in swap and withdrawal transactions. Additionally, in deposit and withdrawal transactions, `Main` must validate the transferred in/out value v; therefore, the user sends the committed values $(v, addr)$ for new and withdrawn coins, respectively.

6.3.2 Aegis Setup

Aegis protocol relies mainly on zkSNARK proofs that assert the satisfiability of constraints in circuits. We design `Ownership` and `JoinSplit` circuits for checking the validity of NFTs and funds coins, respectively. More importantly, both circuits include a *message* signal for defining an extra parameter to facilitate swaps and withdrawals (e.g., the recipient's Ethereum address and counterparty's coin commitment). In particular, this signal allows users to *signatures of knowledge* [39] on a message m given knowledge

of a valid witness.

Ownership. It allows users to prove the correctness of *transferring* NFT ownership from an input coin to an output one as shown in Fig. 6.3. It checks (i) knowledge of the sender's seed and randomness for the input commitment, (ii) validity of Merkle proof of membership, (iii) correctness of the serial number, (iv) and correctness of the output coin's commitment on the same NFT.

Ownership$(\vec{x}, \vec{w})$

Statement $\vec{x}$:

- $root$: root of NFTs Merkle tree
- sn^{in}: serial number of input coin
- cm^{out}: commitment of output coin
- m: message

Witness $\vec{w}$:

- s: sender's seed
- v: NFT ID
- ρ^{in}: randomness of input coin
- π^{in}: proof of membership for commitment of input coin
- $addr^{out}$: spending address of output coin

Compute and assert:

$$addr^{in} \leftarrow \mathsf{PRF}^{addr}(s, \rho^{in})$$
$$cm^{in} \leftarrow \mathtt{Com}(v, addr^{in})$$
$$\mathsf{Merkle.Verify}(cm^{in}, root, \pi^{in}) = 1$$
$$sn^{in} = \mathsf{PRF}^{sn}(s, \rho^{in})$$
$$cm^{out} = \mathtt{Com}(v, addr^{out})$$

Figure 6.3: zkSNARK Ownership circuit

JoinSplit. It allows users to prove the correctness of *joining* funds from up to two input coins and *splitting* that amount into two output coins (e.g., a recipient coin and a change coin for the sender) as shown in Fig. 6.4. It checks (i) equality of the input and output

balances, (ii) knowledge of the sender's seed and randomnesses for the input commitments, (iii) correctness of serial numbers, (iv) validity of Merkle proofs of membership, (v) correctness of the output commitments, (vi) and the output values lie in the range of $[0, v_{max}]$ to avoid arithmetic overflow and underflow in $\mathbb{F}_p$. More importantly, it skips constraints check for a *dummy* input commitment with a zero value. Therefore, a user with one input coin can still utilize the circuit by supplying a dummy coin as the second input. Additionally, a user can join the entire input funds into one output coin by using a dummy coin for the other output.

Setup and Deployment. For Groth [38] zkSNARK construction, the circuit's signals must be fixed before running zkSNARK Setup (i.e., circuits cannot utilize a variable number of signals).

$$(pk_{own}, vk_{own}) \leftarrow \mathsf{zkSNARK.Setup}(1^{\lambda}, \texttt{Ownership})$$

$$(pk_{js}, vk_{js}) \leftarrow \mathsf{zkSNARK.Setup}(1^{\lambda}, \texttt{JoinSplit})$$

Both `JoinSplit` and `Ownership` circuits verify Merkle proofs of membership which rely on the Merkle tree depth d. Therefore, d is one of the public parameters that is fixed per circuit setup. Next, `Main` smart contract is deployed and initialized with the verifying keys and Merkle tree depth as shown in Fig. 6.5. In turn, `Main` initializes `Verifier` and `Merkle` smart contracts for NFTs and funds. Additionally, it maintains two lists for storing and tracking revealed serial numbers from spent NFT and funds coins. Finally, it initializes a mapping for translating an NFT globally unique identifier into an NFT smart contract address and token ID.

6.3.3 Deposit Transactions

To trade on Aegis, a user initially deposit an NFT or funds. These deposits are non-custodial in the sense that a user can withdraw them at any time. Furthermore, due to the inherent public state of the blockchain, a deposit transaction reveals the initial

JoinSplit$(\vec{x}, \vec{w})$

Statement $\vec{x}$:

- $root$: root of funds Merkle tree
- $\{sn_i^{in}\}$: serial numbers of input coins
- $\{cm_i^{out}\}$: commitments of output coins
- m: message

Witness $\vec{w}$:

- s: sender's seed
- $\{\rho_i^{in}\}$: randomnesses of input coins
- $\{v_i^{in}\}$: values of input coins
- $\{\pi_i^{in}\}$: proofs of membership for commitments of input coins
- $\{addr_i^{out}\}$: spending addresses of output coins

Compute and assert:

$\sum_{i=1}^{2} v_i^{in} = \sum_{i=1}^{2} v_i^{out}$
`For` $i \in \{1, 2\}$
  $cm_i^{out} = \mathtt{Com}(v_i^{out}, addr_i^{out})$
  $0 \leq v_i^{out} \leq v_{max}$
  `If` $v_i^{in} \neq 0$
    $addr_i^{in} \leftarrow \mathsf{PRF}^{addr}(s, \rho_i^{in})$
    $cm_i^{in} \leftarrow \mathtt{Com}(v_i^{in}, addr_i^{in})$
    $\mathsf{Merkle.Verify}(cm_i^{in}, root, \pi_i^{in}) = 1$
    $sn_i^{in} = \mathsf{PRF}^{sn}(s, \rho_i^{in})$

Figure 6.4: zkSNARK JoinSplit circuit

NFT ownership, funds, and the sender's identity. However, we argue that these initial facts can change behind the scenes due to the *unlinkability* between deposit and other transactions (see Section 6.4).

Depositing NFTs. To deposit an NFT, a user sends an `approve` transaction to the NFT smart contract as shown in Fig. 6.1. Let *address* denote to the NFT smart contract's address on Ethereum, and *id* denote to NFT token identifier. The user samples ρ and

Initialize$(\lambda, vk_{js}, vk_{own}, d)$

`fundVerifier` $\leftarrow$ `Verifier`(vk_{js})

`nftVerifier` $\leftarrow$ `Verifier`(vk_{own})

`nftMerkle` $\leftarrow$ `Merkle`(d)

`fundMerkle` $\leftarrow$ `Merkle`(d)

`nftSerials` $\leftarrow \{\}$

`fundSerials` $\leftarrow \{\}$

`nftMap` $\leftarrow$ `mapping`$(v \rightarrow (address, id))$

Figure 6.5: Pseudocode for Initialize function

generates an NFT coin $\boldsymbol{c} = (\rho, v, addr, sn, cm)$ and sends the spending address $addr$ along with $address$ and id as parameters to `DepositNFT` transaction as shown in Fig. 6.6.

DepositNFT$(address, id, addr)$

`erc721` $\leftarrow$ `ERC721(`$address$`)`

`Require(tx.sender` $=$ `erc721.OwnerOf(`id`))`

`erc721.transferFrom(tx.sender, Main.address,` id`)`

$v \leftarrow \mathsf{H}_2(address, id)$

`nftMap`$[v] \leftarrow (address, id)$

$cm \leftarrow$ `Com`$(v, addr)$

`nftMerkle.Accumulate(`cm`)`

`emit NewCommitment('NFT', cm)`

Figure 6.6: Pseudocode for DepositNFT function

`Main` checks whether the transaction sender `tx.sender` is the owner of the NFT with an identifier id. Upon success, it transfers the ownership to its address, note that this call will fail if the user has not *approved* `Main.address` previously. Next, it generates a

global unique identifier v based on the NFT contract address $address$ and token's identifier id, and stores the association between them in the mapping `nftMap` for easier lookup in `WithdrawTransaction`. Subsequently, it generates a commitment cm, and accumulates it in the NFT Merkle tree. Finally, it emits an event containing cm so that users can rebuild the Merkle tree off-chain.

Depositing Funds. To deposit funds, a user samples ρ and generates a fund coin $\boldsymbol{c} = (\rho, v, addr, sn, cm)$. Then, the user sends $addr$ as a parameter to `DepositFund` transaction with a value of v as shown in Fig. 6.7. `Main` generates a commitment to the amount `tx.value`. Subsequently, it accumulates cm in the funds Merkle tree, and it emits the `NewCommitment` event containing cm.

DepositFund($addr$)

$v \leftarrow$ `tx.value`

$cm \leftarrow$ `Com`$(v, addr)$

`fundMerkle.Accumulate(`cm`)`

```
emit NewCommitment('Fund', cm)
```

Figure 6.7: Pseudocode for DepsoitFund function

6.3.4 Withdrawal Transactions

Users can withdraw their NFTs and funds from `Main` public Ethereum addresses of recipients given valid zkSNARK proofs to `Ownership` and `JoinSplit` statements, respectively. One of the witness parameters in both circuits is proof of membership for the commitments of input coins. To generate these proofs, users rebuild Merkle trees off-chain by scanning `NewCommitment` events from `Main`. In practice, relayers can maintain synchronized off-chain Merkle trees and expose them as a service to users. Furthermore, users utilize the message m signal to specify the recipient's Ethereum address. Additionally,

users open one of the output commitments so that `Main` can determine which NFT and how much funds to transfer out.

Withdrawing NFTs. To withdraw an NFT, a user generates a zkSNARK proof π that asserts knowledge of a valid witness $\vec{w}$ satisfying `Ownership` circuit for a statement $\vec{x}$. Then, the user sends π, $\vec{x}$, and the opening values v and $addr^{out}$) of the output commitment cm^{out} as parameters to `WithdrawNFT` transaction as shown in Fig. 6.8.

WithdrawNFT$(\vec{x}, \pi, v, addr^{out})$

`Parse` $\vec{x}$ `as` $(root, sn^{in}, cm^{out}, m)$

`Require`$(cm^{out} =$ `Com`$(v, addr^{out}))$

`Require(nftSerials.Contains(`sn^{in}`) = false)`

`Require(nftMerkle.ContainsRoot(`$root$`))`

`Require(nftVerifier.Verify(`$\vec{x}, \pi$`))`

`nftSerials.Append(`sn^{in}`)`

$(address, id) \leftarrow$ `nftMap`$[v]$

$erc721 \leftarrow$ `ERC721`$(address)$

$recipient \leftarrow$ `address`(m)

$erc721$`.transferFrom(Main.address,` $recipient, id$`)`

Figure 6.8: Pseudocode for WithdrawNFT function

Initially, `Main` checks the output commitment is computed based on v and $addr^{out}$. Then, it checks that the serial number sn^{in} has not been seen before, and $root$ is one of the recent l roots in the NFT Merkle tree. Subsequently, it verifies the proof π with respect to the statement $\vec{x}$ using `nftVerifier`. Upon success, it appends sn^{in} to the list of NFT serial numbers. Next, it retrieves the NFT address and identifier corresponding to the value v using `nftMap`. Finally, it sets m as the *recipient* address that receives the NFT ownership.

Withdrawing Funds. To withdraw fund, a user generates a zkSNARK proof π that as-

serts knowledge of a valid witness $\vec{w}$ satisfying `JoinSplit` circuit for a statement $\vec{x}$. Then, the user sends π, $\vec{x}$, and the opening values v_1^{out} and $addr_1^{out}$ of the output commitment cm_1^{out} as parameters to `WithdrawFund` transaction as shown in Fig. 6.9.

WithdrawFund$(\vec{x}, \pi, v_1^{out}, addr_1^{out})$

`Parse` $\vec{x}$ `as` $(root, sn_1^{in}, sn_2^{in}, cm_1^{out}, cm_2^{out}, m)$

`Require`$(cm_1^{out} =$ `Com`$(v_1^{out}, addr_1^{out}))$

`Require(fundSerials.Contains(`sn_1^{in}, sn_2^{in}`) = false)`

`Require(fundMerkle.ContainsRoot(`$root$`))`

`Require(fundVerifier.Verify`$(\vec{x}, \pi)$`)`

`fundSerials.Append`(sn_1^{in}, sn_2^{in})

`fundMerkle.Accumulate(`cm_2^{out}`)`

`emit NewCommitment('Fund',` cm_2^{out}`)`

$recipient \leftarrow$ `address`(m)

$recipient$`.transfer(`v_1^{out}`)`

Figure 6.9: Pseudocode for WithdrawFund function

The logic for `WithdrawFund` has some similarities to `WithdrawNFT`. Initially, `Main` checks the output commitment cm_1^{out} is correctly computed based on v_1^{out} and $addr_1^{out}$. Then, it checks the serial numbers sn_1^{in} and sn_2^{in} have not been seen before in `fundSerials`. Additionally, it asserts that $root$ is one of the recent l roots in the Merkle tree of fund coins. Subsequently, it verifies the proof π for the statement $\vec{x}$ using `fundVerifier`. Upon success, it appends the input serial numbers to `fundSerials`. Next, it accumulates the unspent output commitment cm_2^{out} in the funds Merkle tree, and emit the `NewCommitment` event. Finally, it sets m as the recipient address that receives an amount v from `Main`.

6.3.5 Atomic Swap

Aegis allows two users to swap an NFT for a payment amount in an atomic transaction. The atomic swap relies mainly on designing *contingent* transfer of coins. Initially, each user generates a zkSNARK proof for a statement transferring its coin to the counterparty. More importantly, `Main` accepts proofs if and only if (i) they are valid, (ii) and the statement's message is *equal* to the output commitment of the counterparty's statement. Informally speaking, each statement is interpreted as "I'm transferring my coin to the counterparty if and only if the counterparty transfers a coin with a certain commitment". The atomic swap process consists of an off-chain interaction protocol and an on-chain settlement by `Main`.

Off-chain Interaction Protocol. Suppose Alice owns an NFT coin c_a^{in} with a value v_a^{in} and she wants to swap it for a payment amount $v_{b,1}^{out}$ with Bob who owns fund coins $c_{b,1}^{in}$ and $c_{b,2}^{in}$. They run the protocol shown in Fig. 6.10. Initially, Alice generates a spending address $addr_a^{out}$ that Bob uses to generate an output fund coin's commitment $cm_{b,1}^{out}$ with the value $v_{b,1}^{out}$ for her. Similarly, Bob generates two spending addresses: $addr_{b,1}^{out}$ for receiving an output NFT coin's commitment cm_a^{out} with the value v_a^{in} from Alice, and $addr_{b,2}^{out}$ for receiving an output fund coin's commitment $cm_{b,2}^{out}$ with the change value $v_{b,2}^{out}$ from himself.

More importantly, Alice and Bob set their messages m_a and m_b to the output coin's commitment $cm_{b,1}^{out}$ and cm_a^{out} expected from the counterparty, respectively. Next, Alice and Bob query off-chain Merkle trees MerkleNFT and MerkleFund to generate proofs of membership π_a^{in} and $(\pi_{b,1}^{in}, \pi_{b,2}^{in})$ for their input coins' commitments cm_a^{in} and $(cm_{b,1}^{in}, cm_{b,2}^{in})$, respectively. Afterwards, Alice and Bob generate zkSNARK proofs π_a and π_b for `Ownership` statement $\vec{x}_a$ and `JoinSplit` statement $\vec{x}_b$, respectively. Finally, Alice sends $\vec{x}_a$ and π_a to Bob who asserts that messages are valid with respect to the counterparty's output commitment.

On-chain Settlement. To settle the atomic swap, Bob sends $\vec{x}_a, \vec{x}_b, \pi_a$ and π_b as param-

$\texttt{Alice}(s_a, c_a^{in}, v_{b,1}^{out})$		$\texttt{Bob}(s_b, c_{b,1}^{in}, c_{b,2}^{in}, v_a^{in}, v_{b,1}^{out})$
$c_a^{in} = (\rho_a^{in}, sn_a^{in}, cm_a^{in}, v_a^{in})$		$c_{b,1}^{in} = (\rho_{b,1}^{in}, sn_{b,1}^{in}, cm_{b,1}^{in}, v_{b,1}^{in})$
		$c_{b,2}^{in} = (\rho_{b,2}^{in}, sn_{b,2}^{in}, cm_{b,2}^{in}, v_{b,2}^{in})$
$\rho_a \xleftarrow{\$} \mathbb{F}_p$		$\rho_{b,1} \xleftarrow{\$} \mathbb{F}_p, \rho_{b,2} \xleftarrow{\$} \mathbb{F}_p$
$addr_a^{out} \leftarrow \mathsf{PRF}^{addr}(s_a, \rho_a)$		$addr_{b,1}^{out} \leftarrow \mathsf{PRF}^{addr}(s_b, \rho_{b,1})$
		$addr_{b,2}^{out} \leftarrow \mathsf{PRF}^{addr}(s_b, \rho_{b,2})$
		$v_{b,2}^{out} \leftarrow v_{b,1}^{in} + v_{b,2}^{in} - v_{b,1}^{out}$
	$\xrightarrow{addr_a^{out}}$	
	$\xleftarrow{addr_{b,1}^{out}}$	
$cm_a^{out} \leftarrow \mathsf{Com}(v_a^{in}, addr_{b,1}^{out})$		$cm_{b,1}^{out} \leftarrow \mathsf{Com}(v_{b,1}^{out}, addr_a^{out})$
$m_a \leftarrow \mathsf{Com}(v_{b,1}^{out}, addr_a^{out})$		$cm_{b,2}^{out} \leftarrow \mathsf{Com}(v_{b,2}^{out}, addr_{b,2}^{out})$
$sn_a^{in} \leftarrow c_a^{in}.sn$		$m_b \leftarrow \mathsf{Com}(v_a^{in}, addr_{b,1}^{out})$
$root_a \leftarrow \mathsf{MerkleNFT}.root$		$root_b \leftarrow \mathsf{MerkleFund}.root$
$\pi_a^{in} \leftarrow \mathsf{MerkleNFT.Prove}(cm_a^{in})$		$\pi_{b,1}^{in} \leftarrow \mathsf{MerkleFund.Prove}(cm_{b,1}^{in})$
		$\pi_{b,2}^{in} \leftarrow \mathsf{MerkleFund.Prove}(cm_{b,2}^{in})$
$\vec{x}_a \leftarrow (root_a, sn_a^{in}, cm_a^{out}, m_a)$		$\vec{x}_b \leftarrow (root_b, sn_{b,1}^{in}, sn_{b,2}^{in}, cm_{b,1}^{out}, cm_{b,2}^{out}, m_b)$
$\vec{w}_a \leftarrow (s_a, c_a^{in}.\rho, v_a, \pi_a^{in}, addr_{b,1}^{out})$		$\vec{w}_b \leftarrow (s_b, \rho_{b,1}^{in}, \rho_{b,2}^{in}, v_{b,1}^{in}, \pi_{b,1}^{in}, \pi_{b,2}^{in}, addr_{b,1}^{out}, addr_{b,2}^{out})$
$\Psi_{own} \leftarrow \mathsf{Ownership.Prove}$		$\Psi_{js} \leftarrow \mathsf{JoinSplit.Prove}$
$\pi_a \leftarrow \Psi_{own}(pk_{own}, \vec{x}_a, \vec{w}_a)$		$\pi_b \leftarrow \Psi_{js}(pk_{js}, \vec{x}_b, \vec{w}_b)$
	$\xrightarrow{\pi_a, \vec{x}_a}$	
		$\mathsf{Assert}(\vec{x}_a.m_a = \vec{x}_b.cm_{b,1}^{out})$
		$\mathsf{Assert}(\vec{x}_b.m_b = \vec{x}_a.cm_a^{out})$

Figure 6.10: Off-chain protocol for swapping an NFT for a payment amount

eters to `Swap` transaction as shown in Fig. 6.11. Initially, `Main` checks that the message of each statement is equal to the output commitment of the other statement, which ensures that both parties have a mutual agreement on the swapped coins. Then, for each statement, `Main` checks (i) the freshness of serial numbers, the validity of Merkle root, (iii) and the validity of zkSNARK proof. Upon success, `Main` settles the atomic swap by

(i) storing the serial numbers which nullify the input coins, (ii) and accumulating the new output commitments in the corresponding Merkle trees; thereby, enforcing the transfer of coins. Finally, `Main` emits `NewCommitment` events for each accumulated commitment.

Swap$(\vec{x}_a, \vec{x}_b, \pi_a, \pi_b)$

`Parse` $\vec{x}_a$ `as` $(root_a, sn_a^{in}, cm_a^{out}, m_a)$

`Parse` $\vec{x}_b$ `as` $(root_b, sn_{b,1}^{in}, sn_{b,2}^{in}, cm_{b,1}^{out}, cm_{b,2}^{out}, m_b)$

`Require(`$m_a = cm_{b,1}^{out}$`)`

`Require(`$m_b = cm_a^{out}$`)`

`Require(nftSerials.Contains(`sn_a^{in}`)=false)`

`Require(nftMerkle.ContainsRoot(`$root_a$`))`

`Require(nftVerifier.Verify(`$\vec{x}_a, \pi_a$`))`

`Require(fundSerials.Contains(`$sn_{b,1}^{in}, sn_{b,2}^{in}$`)=false)`

`Require(fundMerkle.ContainsRoot(`$root_b$`))`

`Require(fundVerifier.Verify(`$\vec{x}_b, \pi_b$`))`

`nftSerials.Append(`sn_a^{in}`)`

`nftMerkle.Accumulate(`cm_a^{out}`)`

`fundSerials.Append(`$sn_{b,1}^{in}, sn_{b,2}^{in}$`)`

`fundMerkle.Accumulate(`$cm_{b,1}^{out}, cm_{b,2}^{out}$`)`

`emit NewCommitment(NFT,` cm_a^{out}`)`

`emit NewCommitment(Fund,` $cm_{b,1}^{out}, cm_{b,2}^{out}$`)`

Figure 6.11: Pseudocode for Swap function

6.4 Security Analysis

We informally discuss how Aegis achieves the security goals mentioned in Section 6.2.4. In particular, an adversary may try to guess the pair-wise link between deposit

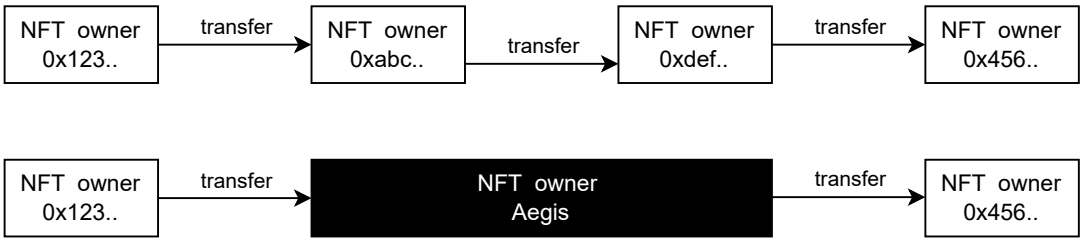

Figure 6.12: Transactions graph with and without Aegis

and withdrawal transactions. Next, we analyze whether an adversary can successfully swap or withdraw an NFT or fund that belongs to an honest user.

6.4.1 Privacy Analysis

Aegis allows users to deposit NFTs and funds and use them within the protocol without leaking any information. For example, users do not have to withdraw NFTs from Aegis since they can prove their ownership and trade them which are the two main features for NFTs. However, if the user decides to withdraw, then Aegis obfuscates the transactions links between deposit and withdraw transactions. In particular, an observer cannot determine whether a certain NFT has changed hands within the system and how many previous owners for that NFT. More precisely, Aegis acts as a black box that hides the transactions graph between deposits and withdrawals as shown in Fig. 6.12

The probability of an adversary correctly linking a deposit and withdrawal transactions largely depends on the anonymity set (i.e., number of deposit transactions). It is worth mentioning that Aegis maintains separate anonymity sets for NFTs and funds. Let $* \in \{nft, fund\}$, we define $\texttt{CM}_h^*$ and $\texttt{SN}_h^*$ to be the set of $*$ accumulated commitments and the set of revealed serial numbers in Aegis by the block height h, respectively. More importantly, those sets are the result of transactions made by *honest* users not colluding with an adversary. Furthermore, those sets are publicly accessible on Aegis to anyone. For any block height h, we assume that $|\texttt{CM}_h^*| - |\texttt{SN}_h^*| > 0$, which implies that at least one NFT and one fund coin still have not been withdrawn from Aegis.

We define a link property for a commitment $cm \leftarrow \mathtt{Com}(\cdot, addr)$ and a serial number $sn \leftarrow \mathsf{PRF}^{sn}(s, \rho)$. We say they are linked $cm \xleftarrow{link} sn$ if the spending address $addr \leftarrow \mathsf{PRF}^{addr}(s, \rho)$ is computed by the same seed s and randomness ρ. The linking advantage is the probability that an adversary can output the correct commitment in a deposit transaction linked to a serial number in a swap or withdrawal transaction.

Definition 6.1 *(Linking Advantage) Let tx^*_{h+1} be a valid withdrawal transaction submitted by an honest user at block height $h+1$. Let sn^*_{h+1} be the serial number revealed in tx^*_{h+1}, then the adversarial linking advantage is:*

$$\mathit{Adv}^{link,*}_{h+1} = \mathit{Pr}[\mathcal{A}(tx^*_{h+1}) \rightarrow cm \in \mathit{CM}^*_h \ s.\ t.\ cm \xleftarrow{link} sn^*_{h+1}]$$

Claim 1 *Assuming the underlying cryptographic primitives are secure, the adversarial linking advantage is* $(1/|\mathit{CM}^*_h| + \mathit{negl}(\lambda))$

Proof: In Aegis, the origin of a withdrawal transaction at block height $h+1$ can be any of the deposit transactions made by honest users up to h blocks ago. In particular, for a withdrawal transaction at block height $h+1$ revealing a serial number sn^*_{h+1}, there exists a deposit transaction with a commitment $cm \in \mathtt{CM}^*_h$ such that $cm \xleftarrow{link} sn^*_{h+1}$. Therefore, the probability that the adversary $\mathcal{A}$ successfully links a deposit to a withdrawal transaction is $(1/|\mathtt{CM}^*_h| + \mathsf{negl}(\lambda))$. ■

6.4.2 Balance Analysis

Similar to ZeroCash [67], Aegis satisfies the balance security property. More precisely, an adversary cannot successfully issue a valid withdrawal transaction without depositing any coins into Aegis. Let $\mathtt{BAL}^{fund}_h$ be the total *unspent* balance of funds deposited by honest users at block height h. Similarly, $\mathtt{BAL}^{nft}_h$ denotes the set of NFTs owned by `Main` due to non-custodial transfer of ownership by honest users. Formally speaking, let $\boldsymbol{c}^*_i$ denote to each coin generated in Aegis, we define the following functions:

$$\texttt{Sum}(\boldsymbol{c}_i^*) \leftarrow \begin{cases} \bigcup\{\boldsymbol{c}_i^*.v\} & \text{where } * = nft \\ \sum \boldsymbol{c}_i^*.v & \text{where } * = fund \end{cases}$$

$$\texttt{Less}(x^*, y^*) \leftarrow \begin{cases} x^* \subset y^* & \text{where } * = nft \\ x^* < y^* & \text{where } * = fund \end{cases}$$

$$\texttt{BAL}_h^* \leftarrow \texttt{Sum}(\boldsymbol{c}_i^*) \text{ s.t. } \boldsymbol{c}_i^*.cm \in \texttt{CM}_h^* \wedge \boldsymbol{c}_i^*.sn \notin \texttt{SN}_h^*$$

Definition 6.2 *(Balance Advantage) Let $\boldsymbol{CM}_h^*$ and $\boldsymbol{SN}_h^*$ be public data of coins' commitments and serial numbers available at block height h. Then, the adversarial balance advantage to generate a valid withdrawal transaction tx_{h+1}^* is:*

$$\boldsymbol{Adv}_{h+1}^{bal,*} = \boldsymbol{Pr}[\mathcal{A}(\boldsymbol{CM}_h^*, \boldsymbol{SN}_h^*) \rightarrow tx_{h+1}^* \ s.t. \ \boldsymbol{Less}(\boldsymbol{BAL}_{h+1}^*, \boldsymbol{BAL}_h^*)]$$

Claim 2 Aegis *satisfies the balance security. Assuming the adversary $\mathcal{A}$ cannot tamper with the execution of* Aegis *smart contracts, then the adversarial balance advantage to successfully withdraw a coin that belongs to an honest user is $\boldsymbol{negl}(\lambda)$*

Proof: There are three scenarios where the adversary $\mathcal{A}$ can successfully withdraw a coin that belongs to an honest user. Firstly, $\mathcal{A}$ controls more than 51% of the blockchain mining/validation nodes, then $\mathcal{A}$ can tamper with the execution of `Main` to bypass the zkSNARK verification check. Secondly, $\mathcal{A}$ breaks the collision-resistance property of H_2 such that $\mathcal{A}$ can generate zkSNARK proof for an existing coin commitment with a different seed and randomness, and thus a different serial number. Finally, $\mathcal{A}$ breaks the soundness property of zkSNARK construction and generates a bogus proof that is accepted by `Main`, thereby withdrawing coins that belong to honest users. In the first scenario, the blockchain is no longer secure, and the assets hold no actual value on it. In the second and third

scenarios, assuming the cryptographic primitives are secure then the probability that $\mathcal{A}$ can break them is $\mathsf{negl}(\lambda)$ ■

6.4.3 Analysis of Other Goals

Atomic Swap. The security of swap transactions relies on Aegis's guarantees of securing the balance property. In other words, the adversary $\mathcal{A}$ cannot swap coins that belong to honest users without breaking the balance property. Furthermore, `Main` executes swap transactions atomically such that either the swap completes, or it reverts to a prior state. More importantly, users follow the off-chain interaction protocol shown in Fig. 6.10 without any trust assumptions. If someone aborts the protocol, the counterparty does not lose assets. For example, assume Alice executed the protocol with Bob who disappeared at the end. Then, Alice can run the protocol with Charlie, who completes the protocol and submits the `Swap` transaction. If Bob tries to resume the protocol and submit a `Swap` transaction, then `Main` will reject and revert his transaction due to duplicate serial number in Alice's statement.

Compatibility. A key design goal of Aegis is to be compatible with existing NFT smart contracts. The motivation is to develop a practical system without modifying current NFT smart contracts that might hold millions of dollars in their values. Hence, we develop Aegis such that it can interact with any NFT smart contract as long as it supports the standard interface ERC-721 [27]. More precisely, in `DepositNFT` and `WithdrawNFT` transactions, Aegis calls ERC-721 `ownerOf` and `transferFrom` functions to manage the ownership in a non-custodial way.

Availability. Aegis operates entirely as smart contracts running on layer-1. In other words, it does not rely on layer-2 services, which might censor users' transactions. Therefore, Aegis has the *availability* guarantees of the underlying blockchain. Users can always read from or write to Aegis smart contracts. Recall that we mentioned non-deposit transactions are sent via trustless relayers. There could be a chance where the entire relayers

network is colluding to censor an arbitrary transaction. In this unlikely case, the transaction sender can utilize its wallet to submit the transaction, which links the origin deposit transaction. We argue that it is acceptable in such circumstances to sacrifice privacy rather than denying users the ability to withdraw their assets.

Blockchain Client Services. In Ethereum, a popular user wallet *MetaMask* outsources all its transactions to centralized services *Infura*. Those centralized services are aware of the users' blockchain address(es), IP address, and the transactions sent to Aegis. Therefore, they can weaken users' privacy, as they may link different transactions from the same wallet and IP address. Consequently, to have better privacy guarantees, a user can operate a full-node or utilize a network-level anonymity service such as Tor or VPNs before using MetaMask.

6.5 Evaluation

6.5.1 Cryptographic Primitives

In Aegis, we use Groth [38] zkSNARK protocol due to its high efficiency in terms of proof size and verification cost compared to other state-of-art zkSNARK protocols [10, 15, 29, 47]. More specifically, Groth [38] protocol generates the smallest proof (i.e., two elements in $\mathbb{G}_1$ and one element in $\mathbb{G}_2$, where $\mathbb{G}_1$ and $\mathbb{G}_2$ are asymmetric bilinear groups). The verifier checks three pairing operations before deciding whether to accept or reject the proof. More importantly, there is a pre-processing phase for the verifier, where it performs ECADD and ECMUL for each public input, before verifying the proof.

For cryptographic hash functions, we evaluate MiMC [1] and Poseidon [37] which are arithmetic circuit friendly hash function. Both hash functions yield much lower number of constraints when compared to other standard hash functions such as SHA-256 and Keccak [7]. However, they consume more gas when executed in Ethereum smart contract. Table 6.1 shows a comparison between Poseidon, MiMC, and SHA-256 hash functions.

Table 6.1: Comparing hash functions in terms of constraints number and gas costs

	Poseidon	MiMC	SHA-256
Constraints	240	2640	59793
Gas cost	49858	59840	23179

We opt to choose Poseidon for computing commitments and Merkle proof verification due to its low number of constraints in the arithmetic circuit while also having lower gas cost than MiMC.

Cryptographic Libraries. We utilize *Circom* (v2.0) [41] to compile the arithmetic circuits `JoinSplit` and `Ownership`. For the proof system, we utilize *snarkjs* (v0.4.10) library [73] to (i) run an MPC-based setup ceremony for generating the proving and verifying keys, and (ii) generate the zkSNARK proofs. We leverage the pre-compiled Ethereum contracts: EIP-196 [62] and EIP-197 [16] to perform point addition and multiplication, and pairing operations on the elliptic curve *bn256* where the size of elements in $\mathbb{F}_p$ and curve points in $\mathbb{G}_1$ and $\mathbb{G}_2$ are 32, 64, 128 bytes, respectively.

Hardware, Operating System, and Environment. We run our experiments on a commodity hardware, which run *Ubuntu* (v21.04) on a laptop equipped with an Intel i7-10700K CPU with clock frequency up-to 5.1 GHz and 8 cores, and 32GB RAM. Additionally, we install *Rust* (v1.57) and *Nodejs* (v16.3) libraries, which are required to compile and run Aegis prototype. We develop the smart contracts in *Solidty* (v0.8.0). We utilize *Hardhat* framework to deploy and test the smart contracts in a local in-memory Ethereum blockchain featuring all improvements added up to the London hard fork and a block gas limit set to a maximum of 30 million gas.

6.5.2 Performance Measurement

We carry out several experiments to assess the performance and feasibility of Aegis prototype. More precisely, we measure the performance of `JoinSplit` and `Ownership` circuits in terms of (i) number of circuit constraints, (ii) time to complete the MPC

ceremony for generating the CRS (i.e., proving and verifying keys), (iii) time to generate the proofs. and (iv) the size of generated CRS. Furthermore, we measure the gas cost for deploying the smart contracts and running Aegis transactions. Finally, we optimize the circuits to yield the most efficient proof verification cost at the expense of increasing the prover's time and computation. The motivation behind this is to minimize on-chain execution while delegating computation intensive tasks to off-chain.

Merkle Tree Depth. Essentially, for Groth [38] protocol, the circuit's wires must be fixed before running zkSNARK `Setup` to generate the proving and verifying keys (i.e., circuits cannot utilize a variable number of signals). Both `JoinSplit` and `Ownership` circuits verify Merkle proofs of membership which rely on the Merkle tree depth. Furthermore, Aegis smart contract utilizes a Merkle tree to accumulate commitments (UTXOs) generated by deposit, withdraw, and swap transactions. Hence, the Merkle tree depth affects the maximum number of commitments in Aegis which is computed as 2^{depth}. In other words, the Merkle tree depth is a critical system parameter which affects the maximum number of transactions and prover performance. We perform experiments with Merkle tree $depth = 10$ and $depth = 20$. For instance, with a $depth = 20$ and separate Merkle trees for NFTs and funds, Aegis can support up to 2^{20} (`DepositNFT, WithdrawNFT, DepositFund`) and 2^{19} (`WithdrawFund, Swap`) transactions. The former transactions accumulate one commitment each, while the latter transactions accumulate two commitments.

Circuit Measurements. We run an MPC ceremony between three parties to generate the proving and verifying keys for `JoinSplit` and `Ownership` circuits. In Table 6.2, we report various performance metrics for each circuit. In particular, we report the average time for a running zkSNARK `Setup` using MPC ceremony excluding the time of network transfer and coordination. Additionally, we report the number of circuit constraints, the size of generated CRS, proof generation time, and the number of $\mathbb{F}_p$ elements of the proof statement and witness.

Table 6.2: Performance measurements for `JoinSplit` and `Ownership` circuits

	Depth = 10		Depth = 20	
	JoinSplit	Ownership	JoinSplit	Ownership
Constraints	6716	3358	11556	5778
CRS size (bytes)	3719	1860	6556	3279
Proving time (ms)	637	439	1005	653
Statement size	6	4	6	4
Witness size	30	15	50	19

The size of the proving key scales with the witness size (i.e., number of private inputs) and constraints. While, the verifying key scales with the statement size. Since the Merkle tree depth does not affect the statement size in `JoinSplit` and `Ownership`, then the verifying key will have the same size in each experiment. In particular, the verifying keys size are 832 and 576 bytes for `JoinSplit` and `Ownership`, respectively.

Smart Contracts Measurement. We deploy the smart contracts with on local in-memory Ethereum blockchain to measure the gas cost and assess the feasibility of Aegis transactions. In particular, we perform experiments with Merkle tree $depth = 10$ and $depth = 20$. Currently, the average mining time for Ethereum block on the mainnet is 12 seconds, and the gas prices are relatively high (e.g., WithdrawFund transaction costs roughly 1k USD). One way to mitigate these problems in the meantime is to utilize (zk/optimistic) rollup solutions such as *zkSync* and *Arbitrum*. In Table 6.3, we report the gas cost for every transaction and its USD cost when deployed on Arbitrum where $1 gwei = \$3.5 \times 10^{-6}$ as of February 9^{th}, 2022.

Table 6.3: Measurement of gas units and fees of transactions in Aegis

	Sender	Depth = 10		Depth = 20	
		Gas	Fees	Gas	Fees
Deployment		5347036	$18.71	5867782	$20.53
DepositFund	User	462223	$1.62	841899	$2.94
DepositNFT	User	693552	$2.42	1244229	$4.35
WithdrawFund	Relayer	813994	$2.85	1193669	$4.18
WithdrawNFT	Relayer	394739	$1.38	394739	$1.38
Swap	Relayer	1141390	$3.99	1521118	$5.32

The deployment cost includes deploying Aegis smart contract, Poseidon hash library, initializing empty Merkle trees, and storing the verifying keys. Hence, the cost scales with the Merkle tree depth and size of verifying keys. `WithdrawFund, WithdrawNFT` and `Swap` transactions involve verification of zkSNARK proofs (i.e., SoKs). We can estimate the minimum gas cost for verifying zkSNARK proof which involves three pairing checks, and EC addition and EC multiplication for each element in the statement. based on the statement size. More precisely, according to EIP-1088 [25], we can estimate the theoretical gas cost as:

$$\begin{aligned} \mathsf{cost} &= 3 \times \texttt{EC-Pairing} + h \times (\texttt{EC-Add} + \texttt{EC-Mul}) \\ \mathsf{cost} &= 3 \times 34000 + 45000 + h \times (150 + 6000) \end{aligned}$$

where h denotes to the statement size, and the gas cost of EC point addition, multiplication, and three pairing operations is 150, 6000, and $3 \times 34000 + 45000$, respectively [25]. In `JoinSplit`, the statement size $h = 6$, hence the theoretical verification cost is 183900. Similarly, in `Ownership`, the number of public inputs $h = 4$, hence the theoretical verification cost is 171600.

Optimization. In order to reduce the gas cost for proof verification, we have to reduce the statement size h. Therefore, we change the circuits implementation such that the statement signals become part of the witness. Obviously, this will increase the proving key size and decrease the verifying key size. More importantly, the hash for those signals is used as the new statement. Thereby, the gas cost is now constant and minimum since the new statement has one element (i.e., $h = 1$). More precisely, the cost is computed as:

$$\mathsf{cost} = \ 3 \times 34000 + 45000 + 1 \times (150 + 6000) = 153150$$

Furthermore, to save more gas, the hash function that binds the statement signals into the witness signals must have low gas cost. Hence, we opt to use SHA256 for this part only

while still using Poseidon for computing commitments and verifying Merkle proofs. In other words, the optimized circuits utilizes both hash functions such that it yields smaller gas cost. Note that while using Poseidon for binding the statement signals will not bloat the circuit constraints, it will definitely eat the savings in gas cost and thus increase the verification cost. Therefore, SHA256 sounds a better alternative since it incurs the smallest gas cost at the expense of significantly increases the number of constraints. Furthermore, that increase does not scale with the Merkle tree depth, and it is a constant overhead. Consequently, we reduce the verification cost on Ethereum while slightly increasing the proving cost off-chain. In Table 6.4, we show the circuit and transactions measurements for the verification-optimized circuits.

Table 6.4: Performance measurement for verification-optimized `JoinSplit` and `Ownership` circuits

	Depth = 10		Depth = 20	
	JoinSplit	Ownership	JoinSplit	Ownership
Constraints	128189	93991	133029	96411
CRS size (bytes)	81317	61982	92019	63138
Proving time (ms)	4094	3515	5841	3672
Statement size ($\mathbb{F}_p$)	1	1	1	1
Witness size ($\mathbb{F}_p$)	36	19	56	29

Similarly, Table 6.5 lists the reduced gas cost for `WithdrawFund, WithdrawNFT` and `Swap` transactions which involve zkSNARK proof verification.

Table 6.5: Measurement of optimized gas units and fees of transactions in Aegis

	Depth = 10		Depth = 20	
	Gas	Fees	Gas	Fees
WithdrawFund	756501	$2.64	1136215	$3.97
WithdrawNFT	360716	$1.26	360740	$1.26
Swap	1049796	$3.67	1429487	$5.00

6.6 Related Work

Up to the best of our knowledge, Aegis is the first academic work which presents privacy preserving atomic swap for NFTs. Aegis has two key components that provide privacy for NFTs and balances. Although, there is no academic work tackling NFTs privacy, there are existing protocols [13,50,65] for adding privacy to users' balance transfer transactions.

Zether [13] is a protocol for making private payments in Ethereum. It utilizes Elgamal encryption to hide users' balance in addition to Bulletproofs [14] to prove the correctness of transferred amounts. The key disadvantage of Zether is that it only hides the transferred amounts leaving the sender and recipient identities public. Furthermore, a single Zether transaction costs roughly 7.8m gas. In Aegis, the identities of users are hidden using SoK and relayers. Moreover, the gas cost for transactions in Aegis is very cheap compared to Zether.

Zeth [65] is a protocol that implements ZeroCash on top of Ethereum. In the Zeth, the identities of sender and recipients are not fully hidden since an observer can track identities by checking the gas payer. A simple solution using relayers alone is insufficient as relayers can hijack users' withdraw proofs and steal the withdrawn amounts to their addresses. Furthermore, Zeth utilize a complicated `JoinSplit` circuit which also involves verification of ciphertext, in addition to using SHA-256 as a hash function for generating commitments and building Merkle trees. Aegis solves the relayers trust problem as well as using Poseidon hash function for computing commitments and verifying Merkle proofs, while using SHA256 to bind the statement signals in the optimized circuits which adds a constant overhead that does not scale with Merkle tree depth.

Möbius [50] is a mixer protocol on top of Ethereum. It utilizes linkable ring signature and stealth address primitives [56] to hide the address of the true sender and the recipient. However, in Möbius, the size of the anonymity set is limited to the size of the ring, and the gas cost of the withdrawing transaction increases linearly with the size of

the ring. Thus, in term of privacy, Aegis balance pool offers a bigger anonymity set that scales exponentially with Merkle tree depth, while incurring a constant verification cost.

AMR [49] is a censorship resilient mixer, which incentives users in a privacy-preserving manner for participating in the system. The paid-out rewards can take the form of governance tokens to decentralize the voting on system parameters, similar to how popular "Decentralized Finance (Defi) farming" protocols operate. Moreover, by leveraging existing Defi lending platforms, AMR allows participating clients to earn financial interest on their deposited funds. While AMR and Aegis share the objective of adding privacy to users' transactions, they have different goals and properties. Furthermore, Aegis provides a complete system for trading NFTs in a privacy preserving manner, while AMR provides a mixing service which is inherent in Aegis; however, without an added incentive.

6.7 Summary

We present Aegis as a privacy-preserving protocol for trading NFTs. Aegis ensures that an observer cannot determine the seller, buyer, NFT, and the payment amount. Furthermore, the swap is guaranteed to be fair such that either both parties get the counter-party's asset or none does. We develop a prototype to evaluate the performance of Aegis in terms of off-chain and on-chain performance. Based on the results, we believe that Aegis is practical to deploy and compatible with existing NFT standard interface. For future work, we will investigate improving Aegis such that it also handles sealed-bid auction for NFTs in order to prevent front-running while preserving users' privacy. Additionally, we will investigate adding support for fractional and composite NFTs.

Chapter 7

Conclusion and Future Work

7.1 Summary

In this section, we give a brief summary of the contribution accomplished in this thesis. We have studied the inherent privacy issue in blockchain with a public ledger. Particularly, we have focused on the lack of privacy in protocols utilizing Ethereum smarts contract. To address those problems, we have proposed privacy-preserving protocols using cryptographic primitives, especially, zero-knowledge proofs. In the following, we report the conclusion of the thesis.

In Chapter 3, we studied the privacy issue with digital assets trading on Ethereum using auctions. Open cry auctions on Ethereum are susceptible to front-running and manipulation by a malicious auctioneer. Hence, we proposed three sealed-bid auction protocols that are resilient to front-running and are publicly verifiable. The first protocol was our initial attempt, and it provided partial privacy to bidders since the auctioneer could access the plaintext bids. Furthermore, the verification cost was high and scaled linearly with the number of bidders. In the second protocol, we utilized zkSNARK to significantly reduce the verification cost; however, it had the same weakness of providing partial privacy. Finally, in the third protocol, we managed to achieve the cheapest veri-

fication cost and full privacy by utilizing a trusted execution environment. However, the protocol requires stronger security assumptions in the hardware vendor.

In Chapter 4, we analyzed the privacy issue with decentralized exchanges, which posed a major disadvantage for financial institutions. We designed publicly verifiable secrecy preserving periodic auction protocol. The building blocks for the protocol are two zero-knowledge proofs, namely, proof of consistent commitment encryption and proof of ordering. The protocol scales logarithmically with the number of orders and provides partial privacy.

In Chapter 5, we investigated the privacy issue in inter-bank payments and the dependency on central banks to settle payments on a gross basis. We designed a privacy-preserving decentralized protocol that resolves payments gridlock on a net basis. Furthermore, we enhanced participants' privacy by hiding the links between senders and recipients while providing confidentiality to payment amounts. The protocol utilizes zkSNARK, which yields a constant verification cost and proof size. Furthermore, the protocol maintains full privacy as each party has access to their private data only.

In Chapter 6, we studied the lack of privacy in the standard specifications of NFTs. We presented Aegis as a privacy-preserving protocol for trading NFTs. Aegis ensures that an observer cannot determine the seller, buyer, NFT, and the payment amount. Furthermore, the swap is guaranteed to be fair such that either both parties get the counter-party's asset or none does. Aegis utilizes zkSNARK to achieve constant verification cost and proof size.

7.2 Future Work

In this section, we discuss some topics that could be of interest for future research.

- There is a potential for using secure multi-party computation to build full privacy-preserving protocols, yet their performance could be much lower than those using

zero-knowledge proofs. Furthermore, the number of rounds and interactivity in MPCs could be a challenging issue. Hence, there are open problems in this direction for future research.

- For decentralized exchanges, we focused on the order-book type of exchanges which requires order matching. Fortunately, there is a better design that utilizes *Automated Market Maker* (AMM) to swap assets using *constant product function*. Current AMM design lacks privacy; therefore, improving its privacy is an interesting research problem to investigate.

- NFTs are gaining a lot of attention, and definitely, privacy is a crucial feature to maintain to gain mass adoption. We developed Aegis as a privacy solution for trading NFTs using atomic swap. Recent constructions such as *fractionalized* NFTs are far challenging and need further analysis and research to improve their privacy.

Bibliography

[1] M. Albrecht, L. Grassi, C. Rechberger, A. Roy, and T. Tiessen. MiMC: Efficient encryption and cryptographic hashing with minimal multiplicative complexity. In *Advances in Cryptology – ASIACRYPT 2016*, pages 191–219. Springer, 2016.

[2] I. Anati, S. Gueron, S. Johnson, and V. Scarlata. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, volume 13, page 7. ACM, 2013.

[3] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, and J. Yellick. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–15. ACM, 2018.

[4] M. L. Bech and K. Soramäki. Gridlock resolution in interbank payment systems. *Bank of Finland Research Discussion Paper*, 2001.

[5] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Succinct non-interactive zero knowledge for a von Neumann architecture. In *USENIX Security Symposium*, pages 781–796. USENIX, 2014.

[6] J. Benaloh and M. De Mare. One-way accumulators: A decentralized alternative to digital signatures. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 274–285. Springer, 1993.

[7] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Keccak. In *Advances in Cryptology – EUROCRYPT 2013*, pages 313–314. Springer, 2013.

[8] A. Biryukov, D. Khovratovich, and I. Pustogarov. Deanonymisation of clients in Bitcoin P2P network. In *Proceedings of the 2014 SIGSAC Conference on Computer and Communications Security*, pages 15–29. ACM, 2014.

[9] E.-O. Blass and F. Kerschbaum. Strain: A secure auction for blockchains. In *ESORICS*, volume 11098, pages 87–110. Springer, 2018.

[10] D. Boneh, J. Drake, B. Fisch, and A. Gabizon. Halo infinite: Recursive zk-snarks from any additive polynomial commitment scheme. *IACR Cryptol. ePrint Arch.*, 2020:1536, 2020.

[11] S. Bowe, A. Gabizon, and M. D. Green. A multi-party protocol for constructing the public parameters of the pinocchio zk-SNARK. In *International Conference on Financial Cryptography and Data Security*, volume 10958, pages 64–77. Springer, 2018.

[12] E. F. Brickell, D. Chaum, I. B. Damgård, and J. van de Graaf. Gradual and verifiable release of a secret. In *Conference on the Theory and Application of Cryptographic Techniques*, volume 293, pages 156–166. Springer, 1987.

[13] B. Bünz, S. Agrawal, M. Zamani, and D. Boneh. Zether: Towards privacy in a smart contract world. In *Financial Cryptography and Data Security*, volume 12059, pages 423–443. Springer, 2020.

[14] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy*, pages 315–334. IEEE, 2018.

[15] B. Bünz, B. Fisch, and A. Szepieniec. Transparent snarks from dark compilers. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 677–706. Springer, 2020.

[16] V. Buterin and C. Reitwiessner. Precompiled contracts for optimal ate pairing check on the elliptic curve alt_bn128. https://eips.ethereum.org/EIPS/eip-197. Accessed: 20-12-2021.

[17] S. Cao, Y. Yuan, A. De Caro, K. Nandakumar, K. Elkhiyaoui, and Y. Hu. Decentralized privacy-preserving netting protocol on blockchain for payment systems. In *International Conference on Financial Cryptography and Data Security*. Springer, 2020.

[18] J. Cartlidge, N. P. Smart, and Y. Talibi Alaoui. MPC joins the dark side. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, pages 148–159. ACM, 2019.

[19] J. Chapman, R. Garratt, S. Hendry, A. McCormack, and W. McMahon. Project Jasper: Are distributed wholesale payment systems feasible yet. *Financial System*, 59, 2017.

[20] D. Chaum. Blind signatures for untraceable payments. In *Advances in cryptology*, pages 199–203. Springer, 1983.

[21] C. Chinchilla. Byzantium Hard Fork. https://github.com/ethereum/wiki/wiki/Byzantium-Hard-Fork-changes. Accessed: 20-12-2021.

[22] Top 100 cryptocurrencies by market capitalization. https://coinmarketcap.com. Accessed: 20-12-2021.

[23] M. Conti, E. S. Kumar, C. Lal, and S. Ruj. A survey on security and privacy issues of Bitcoin. *IEEE Communications Surveys and Tutorials*, 20:3416–3452, 2018.

[24] Decentraland. https://market.decentraland.org. Accessed: 20-12-2021.

[25] Reduce alt_bn128 precompile gas costs. https://eips.ethereum.org/EIPS/eip-1108.

[26] Ethereum Name Service. https://app.ens.domains/. Accessed: 20-12-2021.

[27] W. Entriken, D. Shirley, J. Evans, and N. Sachs. EIP-721: Non-fungible token standard, 2018. URL https://eips.ethereum.org/EIPS/eip-721.

[28] Foundation. https://foundation.app/artworks. Accessed: 20-12-2021.

[29] A. Gabizon, Z. J. Williamson, and O. Ciobotaru. Plonk: Permutations over Lagrange-bases for oecumenical noninteractive arguments of knowledge. *IACR Cryptol. ePrint Arch.*, 2019:953, 2019.

[30] H. S. Galal, M. ElSheikh, and A. M. Youssef. An efficient micropayment channel on Ethereum. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, volume 11737, pages 211–218. Springer, 2019.

[31] H. S. Galal and A. M. Youssef. Succinctly verifiable sealed-bid auction smart contract. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, volume 11025, pages 3–19. Springer, 2018.

[32] H. S. Galal and A. M. Youssef. Verifiable sealed-bid auction on the Ethereum blockchain. In *Financial Cryptography and Data Security, FC 2018 International Workshops*, volume 10958, pages 265–278. Springer, 2018.

[33] H. S. Galal and A. M. Youssef. Trustee: Full privacy preserving vickrey auction on top of Ethereum. In *Financial Cryptography and Data Security, FC 2019 International Workshops*, volume 11599. Springer, 2019.

[34] H. S. Galal and A. M. Youssef. Privacy preserving netting protocol for inter-bank payments. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, pages 319–334. Springer, 2020.

[35] H. S. Galal and A. M. Youssef. Publicly verifiable and secrecy preserving periodic auctions. In *Financial Cryptography and Data Security, FC 2021 International Workshops*, volume 12676, pages 348–363. Springer, 2021.

[36] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *Advances in Cryptology – EUROCRYPT*, pages 626–645. Springer, 2013.

[37] L. Grassi, D. Khovratovich, C. Rechberger, A. Roy, and M. Schofnegger. Poseidon: A new hash function for zero-knowledge proof systems. In *30th USENIX Security Symposium*, pages 519–535. USENIX, 2021.

[38] J. Groth. On the size of pairing-based non-interactive arguments. In *Advances in Cryptology – EUROCRYPT 2016*, pages 305–326. Springer, 2016.

[39] J. Groth and M. Maller. Snarky signatures: Minimal signatures of knowledge from simulation-extractable snarks. In *Advances in Cryptology – CRYPTO 2017*, pages 581–612. Springer, 2017.

[40] M. Hearn. Corda: A distributed ledger. *Corda Technical White Paper*, 2016.

[41] Iden3. Circom: Circuit compiler. https://github.com/iden3/circom, 2021. Accessed: 20-12-2021.

[42] S. Johnson, V. Scarlata, C. Rozas, E. Brickell, and F. Mckeen. Intel® software guard extensions: EPID provisioning and attestation services. *White Paper*, 1:119, 2016.

[43] C. S. Jutla. Upending stock market structure using secure multi-party computation. *IACR Cryptology ePrint Archive*, 2015:550, 2015.

[44] C. M. Kahn and W. Roberds. Real-time gross settlement and the costs of immediacy. *Journal of Monetary Economics*, 47(2):299–319, 2001.

[45] S. Klein. *Introduction to electronic auctions*, volume 7. Taylor & Francis, 1997. 3–6 pp.

[46] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *IEEE Symposium on Security and Privacy*, pages 839–858. IEEE, 2016.

[47] A. Kosba, D. Papadopoulos, C. Papamanthou, and D. Song. MIRAGE: Succinct arguments for randomized algorithms with applications to universal zkSNARKs. In *29th USENIX Security Symposium*, pages 2129–2146. USENIX, 2020.

[48] P. Koshy, D. Koshy, and P. McDaniel. An analysis of anonymity in Bitcoin using P2P network traffic. In *Financial Cryptography and Data Security*, pages 469–485. Springer, 2014.

[49] D. V. Le and A. Gervais. Amr: Autonomous coin mixer with privacy preserving reward distribution. In *Proceedings of the 3rd ACM Conference on Advances in Financial Technologies*, pages 142–155. ACM, 2021.

[50] S. Meiklejohn and R. Mercer. Möbius: Trustless tumbling for transaction privacy. *Proceedings on Privacy Enhancing Technologies*, pages 105–121, 2018.

[51] S. Meiklejohn, M. Pomarole, G. Jordan, K. Levchenko, D. McCoy, G. M. Voelker, and S. Savage. A fistful of bitcoins: Characterizing payments among men with no

names. In *Proceedings of the 2013 Conference on Internet Measurement Conference*, page 127–140. ACM, 2013.

[52] R. C. Merkle. A digital signature based on a conventional encryption function. In *Conference on the theory and application of cryptographic techniques*, pages 369–378. Springer, 1987.

[53] S. Micali and M. O. Rabin. Cryptography miracles, secure auctions, matching problem verification. *Communications of the ACM*, 57:85–93, 2014.

[54] Markets in Financial Instruments Directive II . https://www.esma.europa.eu/policy-rules/mifid-ii-and-mifir, 2018. Accessed: 20-12-2021.

[55] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[56] S. Noether. Ring signature confidential transactions for monero. *IACR Cryptol. ePrint Arch.*, 2015:1098, 2015.

[57] B. Norman. Liquidity saving in real-time gross settlement systems. *Journal of Payments Strategy & Systems*, 4:261–276, 2010.

[58] D. Omahony, M. Peirce, and H. Tewari. *Electronic payment systems*. Artech House Norwood, 1997.

[59] OpenZeppelin. Sending gasless transactions. https://docs.openzeppelin.com/learn/sending-gasless-transactions. Accessed: 20-12-2021.

[60] D. C. Parkes, C. Thorpe, and W. Li. Achieving trust without disclosure: Dark pools and a role for secrecy-preserving verification. In *Proceedings of the Third Conference on Auctions, Market Mechanisms and Their Applications*. ACM, 2015.

[61] Project Ubin: central bank digital money using distributed ledger technology. https://www.mas.gov.sg/schemes-and-initiatives/Project-Ubin. Accessed: 20-12-2021.

[62] C. Reitwiessner. Precompiled contracts for addition and scalar multiplication on the elliptic curve alt_bn128. https://eips.ethereum.org/EIPS/eip-196. Accessed: 20-12-2021.

[63] Reuters. NFT sales surge to $10.7 billion in q3. https://money.usnews.com/investing/news/articles/2021-10-04/nft-sales-surge-to-107-billion, 2021. Accessed: 20-12-2021.

[64] R. L. Rivest and A. Shamir. PayWord and MicroMint: two simple micropayment schemes. In *International workshop on security protocols*, pages 69–87. Springer, 1996.

[65] A. Rondelet and M. Zajac. Zeth: On integrating zerocash on Ethereum. *arXiv preprint arXiv:1904.00905*, 2019.

[66] D. C. Sánchez. Raziel: Private and verifiable smart contracts on blockchains. *arXiv preprint arXiv:1807.09484*, 2018.

[67] E. B. Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. Zerocash: Decentralized anonymous payments from Bitcoin. In *IEEE Symposium on Security and Privacy*, pages 459–474. IEEE, 2014.

[68] V. Scarlata, S. Johnson, J. Beaney, and P. Zmijewski. Supporting third party attestation for intel® SGX with intel® data center attestation primitives. *Intel White paper*, 2018.

[69] SEC charges Citigroup for dark pool misrepresentations. https://www.sec.gov/news/press-release/2018-193, 2018. Accessed: 20-12-2021.

[70] SEC charges ITG with misleading dark pool subscribers. https://www.sec.gov/news/press-release/2018-256, 2018. Accessed: 20-12-2021.

[71] Barclays, credit suisse charged with dark pool violations. https://www.sec.gov/news/pressrelease/2016-16.html, 2016. Accessed: 20-12-2021.

[72] M. Seifelnasr, H. S. Galal, and A. M. Youssef. Scalable open-vote network on Ethereum. In *Financial Cryptography and Data Security, FC 2020 International Workshops*, volume 12063, pages 436–450. Springer, 2020.

[73] SnarkJS: JavaScript implementation of zkSNARKs. https://github.com/iden3/snarkjs. Accessed: 20-12-2021.

[74] C. Thorpe and D. C. Parkes. Cryptographic securities exchanges. In *Financial Cryptography and Data Security*, pages 163–178. Springer, 2007.

[75] Yellowheart. https://yh.io. Accessed: 20-12-2021.

[76] X. Wang, X. Xu, L. Feagan, S. Huang, L. Jiao, and W. Zhao. Inter-bank payment system on enterprise blockchain platform. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 614–621. IEEE, 2018.

[77] G. Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 151, 2014.

[78] Zora. https://zora.co/collections/zora/145. Accessed: 20-12-2021.