# Aegis: Privacy-Preserving Market for Non-Fungible Tokens

Hisham S. Galal, Amr M. Youssef, *Senior Member, IEEE*

**Abstract**—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. Given the public state of Blockchain, NFTs owners face a privacy problem. More precisely, an observer can trivially learn the whole NFT collections owned by an address. For some categories of NFTs like arts and game collectibles, owners can sell them for a profit. However, popular marketplaces trade NFTs using public auctions and direct offers. Hence, an observer can learn about the new owner and the NFT purchase price. To tackle those problems, we propose Aegis, a protocol that allows users to add privacy to their NFTs ownership. In Aegis users can swap NFTs for payment amounts in fungible tokens while hiding the details (i.e., involved parties, the NFTs, and the payment amounts). One of the main properties of Aegis is its complete compatibility with existing NFT standards. We design Aegis by leveraging zkSNARK proof system and smart contracts. We build an open-source prototype and perform experiments to evaluate Aegis's performance.

**Index Terms**—Non-fungible tokens, Atomic Swap, zkSNARK, Blockchain.

## 1 Introduction

Non fungible tokens (NFTs) have gained significant interest and adoption. Sales volumes of NFTs surged to $10.7 billion in the third quarter of 2021 alone [1]. The popular and largest NFT marketplace, Opensea[1] hit $3.4 billion as a sales volume in August 2021 [1]. NFTs are tokens that represent ownership of unique digital items such as art [2], collectibles [3], essays [4], domains [5], and even tickets to access real-world events [6]. 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 [7] (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 standards and marketplaces design. For instance, ERC-721 specifications [7] require compatible smart contracts to expose the owner's address given the NFT identifier. 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. For example, 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 of some of the most expensive and premium NFTs in addition to significant 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.

Furthermore, the current design of NFTs marketplaces also lacks privacy. 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, an illegal act that provides risk-free profits to front-runners. Moreover, public auctions and swaps reveal sensitive information about the seller, buyer, NFT, and payment amount.

The contribution of this paper can be summarized as follows: We design Aegis as a privacy-preserving protocol that allows users to add privacy to their NFT ownership status. Aegis enables 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 way without revealing any information about the involved participants, NFTs, and payment amounts. Finally, we implemented a basic prototype to assess name's performance and released its source code [8].

The rest of this paper is organized as follows. Section 2 outlines the background before giving an overview of Aegis in Section 3. Section 4 presents a detailed construction of Aegis. Section 5 analyzes the Aegis's security and privacy. Section 6 evaluates Aegis's performance based on an open-source prototype. Section 7 summarizes related work on adding privacy to transactions on Ethereum. Finally, Section 4 concludes this paper.

## 2 Background and Preliminaries

**Ethereum Blockchain.** acts as a distributed virtual machine that supports quasi Turing-complete programs [9].


- *HS. Galal is with Faculty of Computers and Information, Assiut University, Egypt.*
  *E-mail: hisham_sg@aun.edu.eg*
- *AM. Youssef is with Concordia Institute for Information Systems Engineering, Concordia University, QC, Canada.*
  *E-mail: youssef@ciise.concordia.ca*


1. https://opensea.io

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 in gas units, and the transaction's sender pays the gas cost in Ether.

**NFT Smart Contracts.** NFT smart contracts adhere to the standard specifications outlined by ERC-721 [7]. Additionally, ERC-1155 [10] is a novel standard for creating fungibility-agnostic tokens. Both standards maintain mappings 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 [7] to the NFT smart contract to assign an address as the new owner. Alternatively, the owner can send `approve` [7] 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 smart contracts of marketplaces as operators to their NFTs, 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.

**Meta-Transactions.** 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 [11] 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.

## 3 Aegis OVERVIEW

We specify the system goals and threat model. Then, we provide the structure of Aegis and protocol participants. Finally, we briefly describe transactions in Aegis.

Aegis works in the Unspent Transaction Output (UTXO) model where users own multiple coins controlled by different keys in contrast to having a single state as in *Account* model. 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 solve this dilemma, users submit Zero-Knowledge Succinct Non-interactive ARgument of Knowledge (zkSNARK) [12] 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 incremental Merkle trees to accumulate commitments of new UTXOs. Additionally, to prevent double-spending, Aegis leverages the notion of *serial numbers* [13] to privately nullify consumed UTXOs without linking them to the new ones.

### 3.1 System Goals

We design Aegis such that it achieves the following goals:

*Unlinkability:* an adversary cannot link between NFTs deposit and swap transactions. Similarly, the adversary cannot link funds deposit to swap and withdrawal transactions.
*Balance:* an adversary cannot successfully 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 [7] without requiring any changes to the deployed contracts.

### 3.2 Threat Model

We assume the cryptographic primitives are secure. We further assume the adversary $A$ is computationally bounded and cannot tamper with the execution of the Aegis smart contracts. Additionally, $A$ has the capabilities of a miner (i.e., reorder transactions within a blockchain block, and inject its transactions before and after certain transactions). $A$ can always read all transactions issued to Aegis while propagating over the network. We assume users can always read from and write to the blockchain state. Moreover, users utilize *trustless* relayers to submit non-deposit transactions on their behalf.

### 3.3 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 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 implements an efficient incremental Merkle tree.
`Verifier:` it verifies zkSNARK proofs.

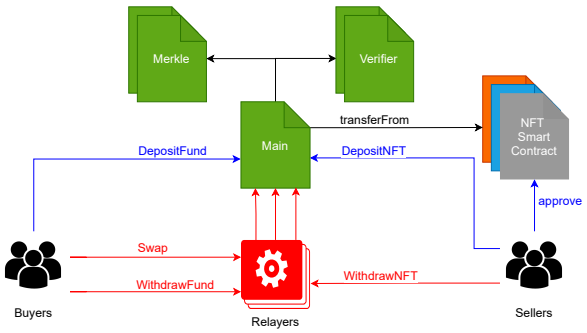


Figure 1. Green components are Aegis smart contracts. Users directly send transactions in blue, while relayers send transactions in red. Black arrows are calls between smart contracts

### 3.4 Aegis Transactions

Figure 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 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 [7] 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 for a payment amount $b$. She deploys a smart contract and initializes it with her and Bob's public signature verification keys. Additionally, the smart contract holds assets $a$ and $b$ as escrow. Then, Alic signs Bob's asset $b$, and Bob signs her asset $a$. Next, Bob sends both signatures to the smart contract. Finally, the smart contract settles the swap only if both signatures are valid for the counterparty'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* [14] over commitments instead of digital signatures over plaintext values.

## 4 Aegis Detailed Construction

### 4.1 Building Blocks

**Hash Functions.** Let $\mathsf{H}_2$ and $\mathsf{H}_3$ be collision-resistant hash functions that map two and three elements from $\mathsf{F}_p$ to an element in $\mathsf{F}_p$, respectively.

$$\mathsf{H}_2 : \mathsf{F}_p \quad \mathsf{F}_p \; ! \; \mathsf{F}_p$$
$$\mathsf{H}_3 : \mathsf{F}_p \quad \mathsf{F}_p \quad \mathsf{F}_p \; ! \; \mathsf{F}_p$$

**Pseudorandom Functions.** To utilize Aegis, a user samples a seed $s \xleftarrow{\$} \mathsf{F}_p$ and keeps it private. We construct $\mathsf{PRF}^{addr}$ and $\mathsf{PRF}^{sn}$ using $\mathsf{H}_3$ to generate spending addresses and serial numbers as follows:

$$\mathsf{PRF}^{addr}(s, \;) = \mathsf{H}_3(0, s, \;)$$
$$\mathsf{PRF}^{sn}(s, \;) = \mathsf{H}_3(1, s, \;)$$

**Commitment Scheme.** We instantiate a commitment scheme COMM using $\mathsf{H}_2$ to generate NFTs and funds commitments as follows:

$$\mathsf{COMM} : \mathsf{F}_p \quad \mathsf{F}_p \; ! \; \mathsf{F}_p$$
$$\mathsf{COMM}(v, addr) = \mathsf{H}_2(v, addr)$$

$v$ 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 $\mathsf{H}_2$ over the left and right children. More importantly, `Merkle` smart contract implements an efficient append-only Merkle tree as shown in Fig. 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. 2, the Merkle proof for the first empty leaf in the left tree is $\pi = (5, 00, 13)$. Using $\pi$ 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 $\pi$ 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 proof of membership for any commitment.

**Coin Structure.** We use the term *coin* [13] 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. To generate a coin $\mathsf{c}$ for a value $v$, a

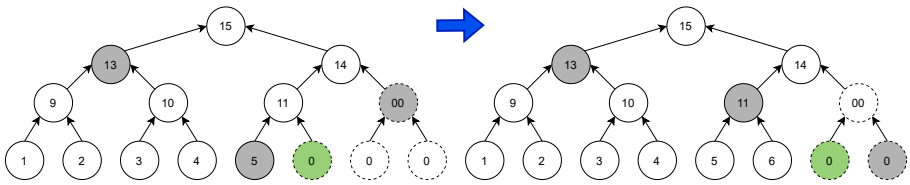


Figure 2. Illustration of accumulating a new element 6. The shaded circles are Merkle proof for the first empty leaf, which is depicted in green.

.

user with a private seed $s$ samples $\rho \xleftarrow{\$} \mathsf{F}_p$. Then, the user utilizes $s$ and $\rho$ to generate a spending address $addr$, a serial number $sn$ as follows:

$$\begin{aligned} addr &\quad \mathsf{PRF}^{addr}(s, \rho) \\ sn &\quad \mathsf{PRF}^{sn}(s, \rho) \\ cm &\quad \mathsf{COMM}(v, addr) \\ \mathtt{c} &\quad (\rho, v, addr, sn, cm) \end{aligned}$$

The user keeps $\rho$ private, which will be used as part of the witness to generate zkSNARK proofs for spending the coin `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.

### 4.2 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* [14] 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. 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.

**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). In addition, 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 $\mathsf{F}_p$. 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 providing 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.

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

Statement $\vec{x}$:
- $root$: the 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
- $\rho^{in}$: the randomness of input coin
- $\pi^{in}$: proof of membership for the commitment of input coin
- $addr^{out}$: spending address of output coin

Compute and assert:

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

**Setup and Deployment.** For Groth [12] zkSNARK construction, the circuit's signals must be fixed before running zkSNARK Setup (i.e., circuits cannot utilize a variable number of signals).

$$\begin{aligned} (pk_{own}, vk_{own}) &\quad \mathsf{zkSNARK.Setup}(1^{\lambda}, \mathtt{Ownership}) \\ (pk_{js}, vk_{js}) &\quad \mathsf{zkSNARK.Setup}(1^{\lambda}, \mathtt{JoinSplit}) \end{aligned}$$

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 fixed per circuit setup. Next, `Main` smart contract is deployed and initialized with the verifying keys and Merkle tree depth. 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.

### 4.3 Deposit Transactions

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

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

Statement $\vec{x}$:

- $root$: the 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 $i \in \{1, 2\}$

$cm_i^{out} = \mathsf{COMM}(v_i^{out}, addr_i^{out})$

$0 \le v_i^{out} \le v_{max}$

If $v_i^{in} \neq 0$

$addr_i^{in} \rightarrow \mathsf{PRF}^{addr}(s, \rho_i^{in})$

$cm_i^{in} \rightarrow \mathsf{COMM}(v_i^{in}, addr_i^{in})$

$\mathsf{Merkle.Verify}(cm_i^{in}, root, \pi_i^{in}) = 1$

$sn_i^{in} = \mathsf{PRF}^{sn}(s, \rho_i^{in})$

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

```
fundVerifier := Verifier(vk_js)
nftVerifier := Verifier(vk_own)
nftMerkle := Merkle(d)
fundMerkle := Merkle(d)
nftSerials := {}
fundSerials := {}
nftMap := mapping(v => (address, id))
```

the initial 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 deposits and other transactions (see Section 5).

**Depositing NFTs.** To deposit an NFT, a user sends an `approve` transaction to the NFT smart contract as shown in Fig. 1. Let $address$ denote the NFT smart contract's address on Ethereum, and $id$ denote to NFT token identifier. The user samples $\rho$ and generates an NFT coin $c = (\rho, v, addr, sn, cm)$ and sends the spending address $addr$ along with $address$ and $id$ as parameters to `DepositNFT` transaction.

`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

**DepositNFT**($address, id, addr$)

```
erc721 := ERC721(address)
require(tx.sender = erc721.OwnerOf(id))
erc721.transferFrom(tx.sender,
   Main.address, id)
v := H2(address, id)
nftMap[v] := (address, id)
cm := COMM(uid, addr)
nftMerkle.Accumulate(cm)
emit NewCommitment(NFT, cm)
```

**DepositFund**($addr$)

```
v := tx.value
cm := COMM(v, addr)
fundMerkle.Accumulate(cm)
emit NewCommitment(Fund, cm)
```

$cm$ so users can rebuild the Merkle tree off-chain.

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

### 4.4 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 $\pi$ that asserts knowledge of a valid witness $\vec{w}$ satisfying `Ownership` circuit for a statement $\vec{x}$. Then, the user sends $\pi$, $\vec{x}$, and the opening values $v$ and $addr^{out}$) of the output commitment $cm^{out}$ as parameters to `WithdrawNFT` transaction.

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 $\pi$ 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.

**WithdrawNFT**$(\vec{x}, \pi, v, addr^{out})$
Parse $\vec{x}$ as $(root, sn^{in}, cm^{out}, m)$
require($cm^{out}$ = COMM($v, addr^{out}$))
require(nftSerials.NotIn($sn^{in}$))
require(nftMerkle.ContainsRoot($root$))
require(nftVerifier.Verify($\vec{x}, \pi$))
nftSerials.Append($sn^{in}$)
$(address, id)$ := nftMap[$v$]
$erc721$ := ERC721($address$)
$recipient$ := address($m$)
$erc721$.transferFrom(Main.address, $recipient, id$)

**Withdrawing Funds.** To withdraw a fund, a user generates a zkSNARK proof $\pi$ that asserts knowledge of a valid witness $\vec{w}$ satisfying JoinSplit circuit for a statement $\vec{x}$. Then, the user sends $\pi$, $\vec{x}$, and the opening values $v_1^{out}$ and $addr_1^{out}$ of the output commitment $cm_1^{out}$ as parameters to WithdrawFund transaction.

**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($cm_1^{out}$ = COMM($v_1^{out}, addr_1^{out}$))
require(fundSerials.NotIn($sn_1^{in}, sn_2^{in}$))
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, $cm_2^{out}$)
$recipient$.address($m$)
$recipient$.transfer($v_1^{out}$)

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 $\pi$ 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.

### 4.5 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 specific 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 $c_{b;2}^{in}$. 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 $\pi_a^{in}$ 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 $\pi_a$ and $\pi_b$ for Ownership statement $\vec{x}_a$ and JoinSplit statement $\vec{x}_b$, respectively. Finally, Alice sends $\vec{x}_a$ and $\pi_a$ to Bob, who asserts that messages are valid 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 $\pi_b$ as parameters to Swap transaction. 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 storing the serial numbers, which nullify the input coins and accumulate 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.NotIn($sn_a^{in}$))
require(nftMerkle.ContainsRoot($root_a$))
require(nftVerifier.Verify($\vec{x}_a, \pi_a$))
require(fundSerials.NotIn($sn_{b;1}^{in}, sn_{b;2}^{in}$))
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', $cm_{b;1}^{out}, cm_{b;2}^{out}$)

### 4.6 Proving Ownership of an NFT

One of the main functionalities of public NFTs implemented by standard ERC721 is the verification of ownership. Typically by querying the owner's address associated with a given NFT identifier. With Aegis, users can prove the ownership of their NFTs in a privacy-preserving manner without leaking their addresses. Suppose that Bob wants to verify whether Alice is the owner of an NFT with an identifier $id$ and ERC721 $address$. First, Bob sends Alice a random challenge $ch$ to prevent her from replaying others' proofs successfully. With the knowledge of a valid witness $\vec{w}$, Alice generates a zkSNARK proof $\pi$ satisfying the `Ownership` circuit for a statement $\vec{x}$ where the message $m$ signal is set to the challenge $ch$. Subsequently, Alice sends $\pi$ and $\vec{x}$ to Bob. Finally, Bob *calls* `VerifyOwnership` on `Main`.

**VerifyOwnership**$(\vec{x}, \pi, id, address, ch)$
`Parse` $\vec{x}$ `as` $(root, sn^{in}, cm^{out}, m)$
`require`$(m = ch)$
$v :=$ `H`$_2(address, id)$
`require`$(cm^{out} =$ `COMM`$(v, 0))$
`require(nftSerials.NotIn(`$sn^{in}$`))`
`require(nftMerkle.ContainsRoot(`$root$`))`
`require(nftVerifier.Verify(`$\vec{x}, \pi$`))`

`Main` checks that the message $m$ is set to the challenge $ch$. Then, `Main` computes the image $v$ of the NFT $id$ and ERC721 $address$, and checks that $(v, addr = 0)$ are the opening values to the output commitment $cm^{out}$. Finally, `Main` 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 $\pi$ with respect to the statement $\vec{x}$ using `nftVerifier`. If the call returns successfully, then Bob has verified Alice's ownership.

Note that calling `VerifyOwnership` does not change any state, and hence it does not incur any gas cost. More importantly, Bob can maliciously abuse the code logic similarities between `WithdrawNFT` and `VerifyOwnership` to steal her NFT. In this attack, Bob sets $ch$ to his address; then, he follows the same procedure with Alice to verify her ownership. Finally, he sends $(\vec{x}, \pi, v, 0)$ to `WithdrawNFT` where $m$, which is set to $ch$, is used as the recipient address. To prevent this attack, Alice must check that $ch$ is not a valid address (i.e., an address is 160-bits).

## 5 Security Analysis

We informally discussed how Aegis achieves the security goals mentioned in Section 3.1. Aegis has separate anonymity sets for NFTs and funds, and their settings are similar regarding data structures and protocols. Hence for the sake of a clearer exposition of the security analysis, we will discuss the security properties from the viewpoint of NFTs. In particular, an adversary may try to guess the pairwise link between deposit and swap transactions. Additionally, we analyze whether an adversary can successfully withdraw an NFT that belongs to an honest user.

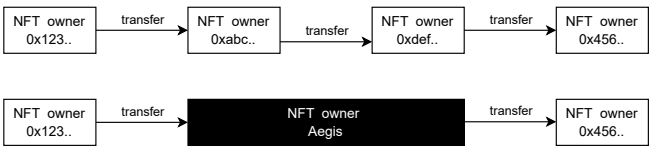


Figure 3. Transactions graph with and without Aegis

### 5.1 Unlinkability

This property captures the requirement that an adversary cannot link deposited NFTs to swap transactions. Note that, it is trivial to link `DepositNFT` to `WithdrawNFT` since the NFT identifier revealed in the latter will *uniquely* determine the former transaction. However, the adversary will not be able to tell how the NFT changed hands in between via swap transactions, as shown in Fig. 3.

We describe a game UNLINK between a challenger $C$ and an adversary $A$ where $A$ wins if $C$ outputs 1 at the end. Initially, $C$ submits two `DepositNFT` transactions $tx_0$ and $tx_1$ to Aegis for two unique NFTs $id_0$ and $id_1$. Then, $C$ samples $b \xleftarrow{2} \{0, 1\}$ at random, and submits a `Swap` transaction $tx_2$ to swap NFT $id_b$. Next, $A$ gets access to Aegis's state where $A$ can find the public parameters of $tx_0$ and $tx_1$. Finally, $A$ guesses $b'$ and sends it to $C$, who outputs 1 if $b = b'$, else outputs 0. Let $\mathbf{tx} = tx_0, tx_1, tx_2$. The adversarial advantage is defined as:

$$\mathsf{Adv}^{\mathsf{UNLINK}}_{A}(\lambda) \quad \Pr\left[b' = b \;\middle|\; \begin{array}{l} pp \leftarrow \mathtt{Setup}(\lambda) \\ (b, \mathbf{tx}) \leftarrow C(pp) \\ b' \leftarrow A(pp, \mathbf{tx}) \end{array}\right] \quad \frac{1}{2}$$

*Definition 1.* (Unlinkability) Aegis maintains the unlinkability property if the adversarial advantage to win UNLINK is negligible.

$$\mathsf{Adv}^{\mathsf{UNLINK}}_{A}(\lambda) < negl(\lambda)$$

*Claim 1.* Aegis satisfies the unlinkability property.

*Proof 1.* Typically, if $A$ wins the game with a non-negligible probability, then $A$ must have found a way to break the pre-image resistance property of $\mathsf{H}_2$ and $\mathsf{H}_3$ used in the computation of $tx_b.cm = \mathsf{H}_2(id, \mathsf{H}_3(0, s, \rho))$ and $tx_2.sn = \mathsf{H}_3(1, s, \rho)$ such that he could determine the same pre-image to compute them: seed $s$ and randomness $\rho$. However, this contradicts the main assumptions of secure cryptographic hash functions.

### 5.2 Balance

This property requires that an adversary cannot withdraw an NFT that an honest user owns. We analyze this property by describing a game BAL between a challenger $C$ and an adversary $A$ where $A$ wins if $C$ outputs 1 at the end. Initially, $C$ submits a `DepositNFT` transaction $tx_d$ to deposit an NFT with an identifier $id$. Next, $A$ sends a `WithdrawNFT` transaction $tx_w$ which reveals the identifier $id'$ of the withdrawn NFT. Finally, $C$ outputs 1 if $id' = id$, else outputs 0. The adversary advantage is defined as:

$$\mathsf{Adv}^{\mathsf{BAL}}_{A}(\lambda) \quad \Pr\left[id' = id \;\middle|\; \begin{array}{l} pp \leftarrow \mathtt{Setup}(\lambda) \\ (tx_d) \leftarrow C(pp) \\ (tx_w) \leftarrow A(pp, tx_d) \end{array}\right]$$

*Definition 2.* (Balance) Aegis maintains the balance property if the adversarial advantage to win BAL is negligible.

$$\mathsf{Adv}_{A}^{\mathsf{BAL}}(\lambda) < negl(\lambda)$$

*Claim 2.* Aegis satisfies the balance property.

*Proof 2.* There are two ways the adversary $A$ can successfully withdraw an NFT that belongs to an honest user. Firstly, $A$ controls more than 51% of the blockchain mining/validation nodes, then $A$ can tamper with the execution of `Main` to bypass the zkSNARK verification check. Finally, $A$ breaks the soundness property of zkSNARK construction and generates a bogus proof accepted by `Main`. In the former, the Blockchain is no longer secure, and the assets hold no actual value. In the latter, the ability to break the soundness contradicts the main assumption of secure zkSNARK construction.

**Atomic Swap.** The security of swap transactions relies on name's guarantees of securing the balance property. In other words, the adversary $A$ 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 reverts to a previous state. More importantly, users follow the off-chain interaction protocol 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 revert due to a 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 value. Hence, we develop Aegis such that it can interact with any NFT smart contract as long as it supports the standard interface ERC-721 [7]. 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 that 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. It is acceptable in such circumstances to sacrifice privacy rather than deny 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 know 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, users can operate full nodes or utilize network-level anonymity services such as Tor or VPNs to have better privacy before using MetaMask.

Table 1
Comparison between hash functions measurements

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

## 6 Evaluation

We implemented a prototype of Aegis [8] to assess its performance and feasibility. More importantly, we measure the key performance metrics for proof generation and verification. These measurements depend on the number of constraints in each circuit and the depth of Merkle trees.

### 6.1 Cryptographic Primitives

In Aegis, we use Groth [12] zkSNARK protocol due to its high efficiency in terms of proof size and verification cost compared to other state-of-art zkSNARK protocols [15]–[18]. More specifically, Groth [12] protocol generates the smallest proof (i.e., two elements in $\mathsf{G}_1$ and one element in $\mathsf{G}_2$, where $\mathsf{G}_1$ and $\mathsf{G}_2$ are asymmetric bilinear groups). The verifier checks three pairing operations before accepting or rejecting 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 [19] and Poseidon [20] which are arithmetic circuit friendly hash function. Both hash functions yield a much lower number of constraints when compared to other standard hash functions such as SHA-256 and Keccak [21]. However, they consume more gas when executed in Ethereum smart contract. Table 1 shows a comparison between Poseidon, MiMC, and SHA-256 hash functions. We choose Poseidon for computing commitments and Merkle proof verification due to its low number of constraints and lower gas cost than MiMC.

**Cryptographic Libraries.** We utilize *Circom* (v2.0) library [22] to compile the arithmetic circuits `JoinSplit` and `Ownership`. For the proof system, we utilize *snarkjs* (v0.4.10) library [23] to (i) run an MPC-based setup ceremony for generating the proving and verifying keys and (ii) generate the zkSNARK proofs. We leverage the precompiled Ethereum contracts: EIP-196 [24] and EIP-197 [25] to perform point addition and multiplication and pairing operations on the elliptic curve *bn256* where the size of elements in $\mathsf{F}_p$ and curve points in $\mathsf{G}_1$ and $\mathsf{G}_2$ are 32, 64, 128 bytes, respectively.

**Hardware, Operating System, and Environment.** We run our experiments on commodity hardware, which runs *Ubuntu* (v21.04) on a laptop equipped with an Intel i7-10700K CPU with clock frequency up to 5.1 GHz and eight 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 and a block gas limit set to a maximum of 30 million gas.

### 6.2 Performance Measurement

We conduct 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.

**Merkle Tree Depth.** Essentially, for Groth [12] protocol, the circuit's wires are fixed before running zkSNARK `Setup` to generate the proving and verifying keys. Both `JoinSplit` and `Ownership` circuits verify Merkle proofs of membership which rely on the Merkle tree depth $d$. More importantly, in the prototype implementation, Aegis creates an additional Merkle tree whenever the current one becomes full. Therefore, the maximum number of accumulated commitments is no longer limited by $d$. Consequently, we can further reduce the proof generation and verification cost by using smaller $d$. Figure 4 shows the gas cost for transactions with Merkle tree depth $d \in [8, 16]$.

**Circuit Measurements.** The performance measurement for the circuits scales linearly with $d$. In Table 2, we report the equations to compute the number of constraints and $\mathsf{F}_p$ elements for statements and witnesses. Note that only the statement size is constant since Merkle proofs of membership are part of the witness only. Furthermore, one may notice that the measurements for `JoinSplit` are two times those for `Ownership` mainly because both circuits are implemented similarly. Yet, the number of inputs and outputs in `JoinSplit` is twice those in `Ownership`.

Table 2
Performance metric for `JoinSplit` and `Ownership` circuits

| | JoinSplit | Ownership |
|---|---|---|
| Constraints | $1876 + 484d$ | $938 + 242d$ |
| Statement size | 6 | 4 |
| Witness size | $10 + 2d$ | $5 + d$ |

**Smart Contracts Measurement.** We deploy the smart contracts on a local in-memory Ethereum blockchain to measure the gas cost and assess the feasibility of Aegis transactions with variable $d$. The cost of `WithdrawNFT` is constant regardless of $d$ since there is no accumulation of the output commitment. On the other hand, the remaining transactions incur a cost that scales linearly with $d$. Furthermore, `swap` incurs the highest cost as it involves two zkSNARKs verification and two Merkle tree updates.

## 7 Related Work

To the best of our knowledge, Aegis is the first academic work that presents privacy-preserving ownership of NFTs and funds in addition to atomic swap. Other protocols in literature [26]–[31] tackle funds privacy only.

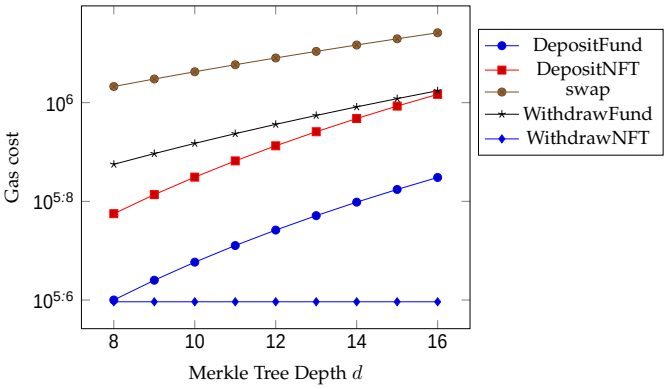


Figure 4. Transactions gas cost with respect to Merkle tree depth $d$

Zether [26] is a protocol for making private payments in Ethereum. It utilizes Elgamal encryption to hide users' balance in addition to Bulletproofs [32] 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 of gas. In Aegis, users' identities are hidden using SoK and relayers. Moreover, the gas cost for transactions in Aegis is very cheap compared to Zether.

Zeth [27] 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. Furthermore, Zeth utilizes a complicated `JoinSplit` circuit, which also involves verification of ciphertext. Additionally, it uses SHA-256 as a hash function for generating commitments and building Merkle trees which is heavy in terms of circuit constraints. Aegis solves the relayers trust problem using SoK, besides using Poseidon hash function for computing commitments and verifying Merkle proofs.

Möbius [28] is a mixer protocol on top of Ethereum. It utilizes linkable ring signature and stealth address primitives [33] to hide the address of the true sender and the recipient. However, the anonymity set is limited to the ring size, and the gas cost of the withdrawing transaction increases linearly with the size of the ring. Thus, in terms of privacy, Aegis balance pool offers a bigger anonymity set that scales exponentially with Merkle tree depth while incurring a fixed verification cost.

AMR [34] 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 already inherent in Aegis, yet without an added incentive.