AMR: Autonomous Coin Mixer with Privacy Preserving Reward Distribution

Duc V. Le
Purdue University

Arthur Gervais
Imperial College London

ABSTRACT

It is well known that users on open blockchains are tracked by an industry providing services to governments, law enforcement, secret services, and alike. While most blockchains do not protect their users' privacy and allow external observers to link transactions and addresses, a growing research interest attempts to design add-on privacy solutions to help users regain their privacy on non-private blockchains.

In this work, we propose to our knowledge the first censorship resilient mixer, which can reward its users in a privacy-preserving manner for participating in the system. Increasing the anonymity set size, and diversity of users, is, as we believe, an important endeavor to raise a mixer's contributed privacy in practice. 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 is the first mixer design that allows participating clients to earn financial interest on their deposited funds.

Our system AMR is autonomous as it does not rely on any external server or a third party. The evaluation of our AMR implementation shows that the system supports today on Ethereum anonymity set sizes beyond thousands of users, and a capacity of over $66,000$ deposits per day, at constant system costs. We provide a formal specification of our zk-SNARK-based AMR system, a privacy and security analysis, implementation, and evaluation with both the MiMC and Poseidon hash functions.

ACM Reference Format:
Duc V. Le and Arthur Gervais. 2021. AMR: Autonomous Coin Mixer with Privacy Preserving Reward Distribution. In *3rd ACM Conference on Advances in Financial Technologies (AFT '21), September 26–28, 2021, Arlington, VA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3479722.3480800

1 INTRODUCTION

More than a decade after the emergence of permissionless blockchains, such as Bitcoin, related work has thoroughly shown that the blockchain's pseudonymity is not offering its clients strong anonymity. Several works have therefore attempted to both, deanonymize clients, cluster addresses [10, 25] as well as to build privacy solutions to protect the clients' privacy [9, 28, 38, 43, 44, 46]. Those existing privacy solutions can be categorized into two classes: *(i)* a fundamental blockchain redesign to natively offer better privacy to clients, and *(ii)* add-on privacy solutions that aim to offer privacy for clients of existing, non-privacy-preserving blockchains.

This paper focuses on add-on privacy solutions that mix cryptocurrency coins within an anonymity set. One known problem of such mixers is that their provided privacy depends on the anonymity set size, i.e., on the protocol's number of active clients. Also, in those systems, to gain a certain degree of privacy, users often need to keep their digital assets locked in the system for a certain period before withdrawing. This locking period prevents users from performing any financial activities on those assets, i.e., there is an opportunity loss of investing the assets for a financial return.

Hence, this work's particular focus is to find new ways to incentivize clients to participate in the mixer. First, similar to popular "DeFi farming" protocols [2], our system, called AMR, chooses to reward mixer participants by granting governance tokens when a client deposits coins for at least time t within the mixer. Naturally, the reward payout must remain privacy-preserving, i.e., a reward payment must be unlinkable to a deposit from the same client of the mixer. Clients can utilize the collected tokens to govern AMR in a decentralized manner, without the need for an external server or centralized entity. Secondly, by leveraging existing popular lending platforms [1, 2, 6], AMR can allow clients to earn interest on their deposited funds. This approach makes AMR the first mixer design that generates financial interest on participants' funds. We hope that such a mixer attracts clients that are privacy-sensitive and interested in a token reward to maximize the anonymity set and client diversity within AMR.

We formalize the zk-SNARK-based AMR system, and implement the mixer in $1,013$ lines of Solidity code. A deposit costs $1.2m$ gas (31.95 USD), while a withdrawal costs $0.3m$ gas (9.12 USD), receiving a reward amounts to $1.5m$ gas (41.07 USD) [1] [2]. These numbers support a real-world deployment, that could support over $66,000$ deposits per day given Ethereum's transaction throughput (assuming no withdrawals). The resulting anonymity set sizes, which can easily exceed $1,000$ while operating at constant system costs, offer stronger privacy than, e.g. the ring signature-based privacy solution [35], whose costs scale linearly with the size of the anonymity set and are hence practically capped at anonymity set sizes of 24 ($8m$ gas for withdrawing).

Our contributions can be summarized as follows.

- We formalize and present a practical zk-SNARK based mixer AMR, which breaks the linkability between deposited and withdrawn coins of a client on a smart contract enabled blockchain, and we provide a formal security and privacy analysis of the proposed system.
- To decentralize AMR's governance and incentivise clients to join the system, we invent a privacy-preserving reward scheme for its clients. We believe that in practice, an incentive scheme would attract more and a wider variety of clients to such privacy solution, and hence contribute to a better anonymity for all involved clients.
- We leverage popular existing lending platforms [1, 2] to propose the first autonomous decentralized on-chain mixer that allows

[1] Estimated using Ethererum price of \$380.4 in 08/25/2020 14:39 UTC.

[2] Using the gas price of 70 Gwei. 1 GWei is 1×10^{-9} Ether.

AFT '21, September 26–28, 2021, Arlington, VA, USA
© 2021 Copyright is held by the owner/author(s).
ACM ISBN 978-1-4503-9082-8/21/09.
https://doi.org/10.1145/3479722.3480800

users to earn interest on their deposited fund. This approach further incentivises users to keep their funds in the system.

- We implement AMR and show that the system can be deployed and operated efficiently on a permissionless blockchain. A deposit into the system costs $1.2m$ (31.95 USD), a withdrawal costs $0.3m$ gas (9.12 USD) and collecting a reward costs $1.5m$ gas (41.07 USD) in transaction fees on the current Ethereum network. The anonymity set size of AMR could grow to up to 2^{d} [3], while operating at constant system costs once deployed (we applied a Merkle depth tree of $d = 30$ within this evaluation). Generating client-side zkSnark proofs costs 3.607 seconds respectively on commodity hardware.

Paper Organization. Section 2 outlines the necessary background before explaining an overview of AMR in Section 3 Section 4 formally outlines the algorithms of AMR, together with the desired security goals and threat model. Section 5 provides a detailed description of AMR. Section 6 discusses how AMR achieves the security goals. Section 7 presents an implementation and evaluation of AMR. Section 8 outlines different applications of AMR and discusses possible future works. Section 9 summarizes related work. Section 10 concludes this paper.

2 PRELIMINARIES

In this section, we define several building blocks for AMR.

2.1 Background on Smart Contract Blockchains and lending platforms

Ethereum Blockchain. The Ethereum blockchain acts as a distributed virtual machine that supports quasi Turing-complete programs. The capability of executing highly expressive languages in those blockchains enables developers to create *smart contract.* The blockchain also keeps track of the state of every account [49], namely *externally-owned accounts (EoA)* controlled by a private key, and *contract account* own by contract's code. Transactions from EoA determine the state transitions of the virtual machine. Transactions are either used to transfer Ether or to trigger the execution of smart contract code. The costs of executing functions are expressed in terms of *gas* unit. In Ethereum, the transaction's sender is the party that pays for the cost of executing all contract operations triggered by that transaction. For a more thorough background on blockchains, we refer the interested reader to [11, 15].

Lending platforms on Ethereum blockchain. Smart-contract-enabled blockchains like Ethereum give rise to many other decentralized financial (Defi) applications. Defi applications allow parties to participate in the financial market without relying on any trusted third party while retaining full custody of their funds. Defi applications appear in different forms, such as decentralized exchanges, lending platforms, or derivatives. At the time of writing, the Defi space accumulates over 10bn dollars of digital assets, and hundreds of millions of dollars of assets are traded daily in those Defi platforms.

For this work, we focus on existing lending protocols [1, 2]. At its core, lending protocols let *borrowers* acquire digital assets with a specified interest rate by placing upfront collaterals into the system. Later, to retrieve the collaterals, *borrowers* need to pay back the borrowed funds along with an additional interest amount. Similarly, users also act as *lenders* by depositing digital assets into the protocol, and the deposited amount will generate interest until users redeem those assets. Finally, the interest rates for borrowing and lending are determined by the state of the lending platforms. In this work, we are only interested in the depositing and redeeming functionalities of lending platforms.

Definition 1. *A lending protocol, Σ, reserves the following actions:*

- $\mathsf{amt}_{\Sigma} \leftarrow$ *Deposit*(amt) *takes as input of* amt *of coins, and outputs a corresponding amount of* amt_{Σ} *tokens.* amt_{Σ} *tokens are minted upon deposits and sent to the depositor, and the value of* amt_{Σ} *increases over time.*
- $\mathsf{amt}{+}R \leftarrow$ *Redeem*(amt_{Σ}) *takes as input* amt_{Σ} *tokens, and deposits* $\mathsf{amt}{+}R$ *to the function invoker. The interest amount R is determined by protocol Σ.*

This definition aims to capture a high-level overview of how the depositing and redeeming functionalities work in a lending platform. For a detailed constructions of each actions in these lending protocols, we refer interested readers to [1, 2, 6].

Governance Token and Yield Farming in Decentralized Finance (DeFi). Users of DeFi platforms are often awarded governance tokens for interacting or providing liquidity to DeFi platforms. These tokens can for instance be used for governance and value accrual/yield farming. Governance means that users can use their tokens to vote for changes in the contract during its lifetime. In term of value accrual, platforms [2, 3] allow users to lock their governance tokens in a pool to be eligible to obtain trading fees collected by the DeFi platform. This approach allows a fair distribution of protocol fees to users who take on the opportunity cost of holding the governance tokens. In this work, we adapt a similar technique of having a distribution pool to fairly distribute total accrued interest collected by the mixer to users.

2.2 Cryptographic Primitives

Notation. We denote by 1^{λ} the security parameter and by $\mathsf{negl}(\lambda)$ a negligible function in λ. We express by $(\mathsf{pk}, \mathsf{sk})$ a pair of public and secret keys. Moreover, we require that pk can always be efficiently derived from sk, and we denote $\textsc{extract}\mathrm{PK}(\mathsf{sk}) = \mathsf{pk}$ to be the deterministic function to derive pk from sk. $k||r$ denotes concatenation of two binary string k and r. We denote $\mathbb{Z}_{\geq a}$ to denote the set of integers that are greater or equal a, $\{a, a + 1, \dots\}$. We let PPT denote probabilistic polynomial time. We use $st[a, b, c \dots]$ to denote an instance of the statement where $a, b, c \dots$ have fixed and public values. We use a shaded area i, j, k to denote the private inputs in the statement $st[a, b, c; i, j, k]$.

Collision resistant hash function. a family H of hash functions is collision resistant, iff for all PPT $\mathcal{A}$ given $h \overset{\$}{\leftarrow} H$, the probability that $\mathcal{A}$ finds x, x', such that $h(x) = h(x')$ is negligible. we refer to the cryptographic hash function h as a fixed function $h : \{0, 1\}^* \rightarrow \{0, 1\}^{\lambda}$. For the formal definitions of cryptographic hash function family, we refer reader to [41].

[3] d is the depth of the Merkle tree

zk-SNARK. A zero-knowledge Succinct Non-interactive ARgument of Knowledge (zk-SNARK) can be considered as "succinct" NIZK for arithmetic circuit satisfiability. For a field $\mathbb{F}$, an arithmetic circuit C takes as inputs elements in $\mathbb{F}$ and outputs elements in $\mathbb{F}$. We use the similar definition from Sasson *et al.*'s Zerocash paper [46] to define arithmetic circuit satisfiability problem. An arithmetic circuit satisfiability problem of a circuit $C : \mathbb{F}^n \times \mathbb{F}^h \rightarrow \mathbb{F}^l$ is captured by relation $R_C = \{(st, \mathsf{wit}) \in \mathbb{F}^n \times \mathbb{F}^h : C(st, \mathsf{wit}) = 0^l\}$; the language is $\mathcal{L}_C = \{st \in \mathbb{F}^n \mid \exists\ \mathsf{wit} \in \mathbb{F}^l\ s.t\ C(st, \mathsf{wit}) = 0^l\}$.

Definition 2. *zk-SNARK for arithmetic circuit satisfiability is triple of efficient algorithms* ($\textsc{Setup}$, $\textsc{Prove}$, $\textsc{Verify}$)*:*

- $(\mathsf{ek}, \mathsf{vk}) \leftarrow \textsc{Setup}(1^\lambda, C)$ *takes as input the security parameter and the arithmetic circuit C, outputs a common reference string that contains the evaluation key* ek *later used by prover to generate proof, and the verification key* vk *later used by the verifier to verify the proof. The public parameters,* pp, *is given implicitly to both proving and verifying algorithms.*
- $\pi \leftarrow \textsc{Prove}(\mathsf{ek}, st, \mathsf{wit})$ *takes as input the evaluation key* ek *and* $(st, \mathsf{wit}) \in R_C$*, outputs a proof* π *for the statement* $st \in \mathcal{L}_C$
- $0/1 \leftarrow \textsc{Verify}(\mathsf{vk}, \pi, st)$ *takes as input the verification key, the proof* π*, the statement st, outputs* 1 *if* π *is valid proof for the statement* $st \in \mathcal{L}_C$.

In additional to *Correctness, Soundness,* and *Zero-knowledge* properties, a zk-SNARK requires two additional properties *Succinctness* and *Simulation extractability.* We defer the definitions of these properties to [27].

Commitment Scheme. A commitment scheme allows a client to commit to chosen values while keeping those values hidden from others during the committing round, and later during the revealing round, client can decide to reveal the committed value.

Definition 3. *A commitment scheme* com = $(P_{\text{com}}, V_{\text{com}})$ *consists of: A* committing *algorithm* $P_{\text{com}}(m, r)$ *takes as input a message m and randomness r, and outputs the commitment value c. A* Reveal *algorithm,* $V_{\text{com}}(c, m, r)$ *takes as input a message m, and the decommitment value r and a commitment c, and returns 1 iff* $c = P_{\text{com}}(m, r)$*. Otherwise, returns 0.*

We use commitment schemes that achieve two properties: *binding* means that given commitment c, it is difficult to find a different pair of message and randomness whose commitment is c, and *hiding* means that given commitment c, it is hard to learn anything about the committed message m.

Authenticated Data Structure (ADS). An authenticated data structure can be used to compute a short digest of a set $X = \{x_1, \ldots, x_n\}$, so that later one can prove certain properties of X with respect to the digest. In this work, we are only interested in a data structure for set membership:

Definition 4. *An authenticated data structure for set membership* $\Pi = (\textsc{Init}, \textsc{Prove}, \textsc{Verify}, \textsc{Update})$ *is a tuple of four efficient algorithms:*

- $y \leftarrow \textsc{Init}(1^\lambda, X)$ *the initialization algorithm takes as input the security parameter and the set* $X = \{x_1, \ldots, x_n\}$ *where* $x_i \in \{0, 1\}^*$*, output* $y \in \{0, 1\}^\lambda$.
- $\pi \leftarrow \textsc{Prove}(i, x, X)$ *takes as input an element* $x \in \{0, 1\}^*, 1 \le i \le n$*, and set X, outputs a proof that* $x = x_i \in X$.
- $0/1 \leftarrow \textsc{Verify}(i, x, y, \pi)$ *takes as input* $1 \le i \le n, x \in \{0, 1\}^*, y \in \{0, 1\}^\lambda$*, and proof* π*, output* 1 *iff* $x = x_i \in X$ *and* $y = \textsc{Init}(1^\lambda, X)$*. Otherwise, return 0.*
- $y' \leftarrow \textsc{Update}(i, x, X)$ *takes as input* $1 \le i \le n, x \in \{0, 1\}^*$ *and set X, output* $y' = \textsc{Init}(1^\lambda, X')$ *where* X' *is obtained by replace* $x_i \in X$ *with x.*

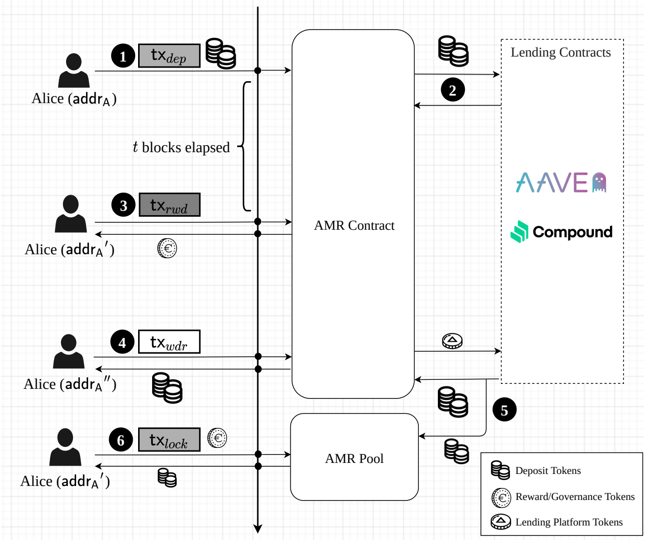

Figure 1: System Overview. In step ❶, clients deposit coins to the AMR **contract. Upon receiving a valid deposit,** AMR **deposit user's fund into lending platform ❷. In ❸, a client proves to the contract to own a deposit older than** t **blocks to obtain a reward. In ❹, a client with a previous deposit can withdraw the coin from the** AMR **contract using a different address. Upon receiving a valid withdrawing transaction,** AMR **redeems user's deposit from lending platform along with accrued interest in step ❺, and deposits into user's address** (addr$''_A$)**. In step ❻, users with** AMR **governance tokens can lock their tokens in an** AMR **pool at any given time to obtain their share of the total accrued interest.**

We require the ADS to be *correct* and *secure*. We defer the formal definitions of these properties to Boneh and Shoup's book [14]. Typical examples of authenticated data structures are Merkle tree [36] or RSA Accumulators [13, 32].

3 SYSTEM OVERVIEW

We proceed to define the system components, overview, goals and the threat model.

3.1 System Components

There are three components of this system: the client, the AMR smart contract, and onchain lending platforms. A **Client** interacts with the AMR smart contract through externally owned accounts. A client can either deposit coins, withdraw coins, or redeem a reward. The AMR **Contract** is the blockchain smart contract that holds deposits, and handles withdrawals and reward redemptions. The contract keeps track of different data structures and parameters to verify the correctness and the integrity of transactions sent to the contract. The AMR **Pool** is a smart contract that takes the accrued interest from a lending platform and proportionally distributes the reward among clients who lock their governance tokens to the pool. The **Lending Platforms** (cf. Section 2.1) are smart contracts that allow users to deposit digital assets and earn interest based on those assets.

3.2 System Overview

Figure 1 outlines the overview of interactions in AMR.

Deposits. In AMR, clients deposit a fixed amount of coins into the system. The client forms a depositing transaction to deposit coins, then sends this transaction through the P2P Network ❶. Once the transaction is validated, miners record the transaction in a blockchain block. Each deposit transaction decreases the balance of the clients' address by a fixed amount of coins. In step ❷, upon receiving a valid deposit from the user, AMR deposits users fund into lending platforms to obtain an equivalent amount of tokens for future withdrawals.

Reward Redemptions. AMR allows clients to earn governance tokens as rewards based on certain conditions. In Figure 1, the requirement for a client to redeem a reward is to keep the deposit inside the contract pool for t blocks. To obtain a reward, a client forms a redeeming transaction and forwards the transaction to the P2P Network ❸. The redeeming transaction includes a cryptographic proof certifying that the client has deposited coins at least t blocks in the past and that the coins remain in the AMR contract. Finally, miners validate the redeeming transaction using the current state of the AMR contract. Once the redeeming transaction gets validated, the transaction gets recorded to a blockchain block, and the network updates the state of the AMR contract.

Withdrawals. The client forms a withdrawing transaction to withdraw coins, then sends this transaction through the P2P Network ❹. The withdrawing transaction includes cryptographic proof certifying that the client has issued a depositing transaction in the past without revealing precisely which one the depositing transaction is. In step ❺, upon receiving valid withdrawing transactions from the client, the contract autonomously redeems the original deposit from lending platforms and the accrued interest. Finally, the contract deposits the redeemed amount into user's address and the accrued interest into a separate AMR pool.

Fair Interest Allocation. At any given time, clients can lock their governance tokens to the AMR pool ❻. AMR distributes the total accrued interest to addresses that lock their AMR governance tokens in AMR pool. This step is straightforward, but offers a fair allocation of interest to users who contribute more to AMR's privacy set.

4 AMR **SYSTEM**

In the following, we discuss various components of the AMR system and provide more details of how AMR operates. In the following algorithm descriptions, we use `tx.sender` to denote the address of the sender from which `tx` was sent.

Condition for reward in AMR**.** The longer time the clients wait before withdrawing/redeeming, the more deposit transactions are issued to the AMR contract. Thus, as the number of deposit transaction (i.e. the anonymity set) increases, the harder it is to link a withdrawing/redeeming transaction with the original deposit transaction. In AMR, we incentivise clients by providing rewards to clients who can prove that the deposit funds are not withdrawn before a certain time, measured in a number of blocks. The provided reward can for instance represent a governance token for a client to participate in the decentralized governance of AMR parameters.

4.1 AMR **Contract Setup**

The setup phase generates public parameters and data structures for the AMR contract and clients. In particular, all cryptographic parameters are generated for the contract. The contract is also initialized with different data structures to prevent clients from double-withdrawal and double-redemption. The deposit and reward amounts, amt and amt_{rwd}, are specified as a fixed deposit amount of coins and a fixed reward amount of governance tokens. The condition for redeeming rewards, t_{con}, is also declared. A lending platform, Σ, (cf. Section 2.1) is determined during this setup phase. A pool, Γ_{AMR}, is deployed, and this pool periodically distributes the accrued interest to addresses that lock governance tokens.

We denote pp^h to be the state of the contract at block h. The state contains all data structures initialized during the setup phase. Moreover, this state is given implicitly to all clients' and contract's algorithms. Finally, the AMR contract is deployed during this phase.

4.2 AMR **Client Algorithms**

In our system, clients have access to the following algorithms to interact with the AMR smart contract. Also, all transactions are implicitly signed by the client using the private key of the Ethereum account that creates the transaction.

- $(\mathsf{wit}, \mathsf{tx}_{dep}) \leftarrow \textsc{CreateDepositTx}(\mathsf{sk}, \mathsf{amt})$ takes as input the private key sk and the amount, amt, coins specified in the setup phase, outputs a deposit transaction tx_{dep} and the secret note wit which is used as witness for creating future withdraw and reward transactions.
- $(\mathsf{wit}', \mathsf{tx}_{rwd}) \leftarrow \textsc{CreateRedeemTx}(\mathsf{sk}', \mathsf{wit})$ takes as input a private key sk′ and the secret note wit, outputs a reward-redeeming transaction tx_{rwd} along with a new secret note, wit′.
- $\mathsf{tx}_{wdr} \leftarrow \textsc{CreateWithdrawTx}(\mathsf{sk}'', \mathsf{wit})$ takes as input a private key sk″ and the secret note wit, outputs a withdrawing transaction tx_{wdr}.
- $\mathtt{tx}_{lock} \leftarrow \textsc{CreateLockTransaction}(\mathsf{sk}, \gamma_{rwd}, t_{lock})$ takes as input an amount, γ_{rwd}, of governance tokens and an unlock value, t_{lock}, specifying how long, γ_{rwd}, will remain locked in the system, outputs a locking transaction, $\mathtt{tx}_{lock}$.

4.3 AMR **Contract Algorithms**

The AMR contract should accept the deposit of funds, handle withdrawals, and reward redemptions. Summarizing, the AMR contract should provide the following functionalities.

- $0/1 \leftarrow \textsc{AcceptDeposit}(\mathsf{tx}_{dep})$ takes as input the deposit transaction tx_{dep}. The AMR contract deposits amt into the lending platform Σ to obtain amt_{Σ}. Finally, the algorithm outputs 1 to denote a successful deposit, otherwise 0.

- $0/1 \leftarrow$ IssueWithdraw(tx_{wdr}) takes as input the withdraw transaction tx_{wdr}. The AMR contract uses amt_{Σ} to redeem $\mathsf{amt} + R$ from Σ. The algorithm outputs 1 to denote a successful withdraw and deposits $\mathsf{amt} + R$ into tx_{wdr}.sender. Otherwise, outputs 0.
- $0/1 \leftarrow$ IssueReward(tx_{rwd}) takes as input the reward transaction tx_{rwd} and the condition t_{con} specified during the setup algorithm, outputs 1 if tx_{rwd} satisfies the t_{con} for reward and deposit amt_{rwd} governance tokens as reward to tx_{rwd}.sender. Otherwise, output 0.

4.4 System Goals

In the following, we outline our system goals.

Correctness. Generally, AMR needs to ensure that clients should not be able to steal coins from the AMR contract or from other clients. Moreover, we design AMR such that clients can redeem a reward after they have deposited their coins into the AMR contract for a fixed period, as a reward system will incentivise clients to deposit more into the system while contributing to the size of the anonymity set. AMR needs to provide the following guarantees: *(i)* It is infeasible for clients to issue n withdrawal transactions without issuing at least n deposit transactions into the AMR contract beforehand. *(ii)* It is infeasible for a client to issue a redeeming transaction without having any coins locked in the AMR contract. *(iii)* A valid redeeming transaction indicates that a client has at least one deposit locked in the AMR contract for a specified duration.

Privacy. AMR needs to ensure the privacy to clients of the system. In particular, considering an adversary that has access to the history of all depositing, withdrawing, and redeeming transactions sent to AMR contract, the system needs to ensure *(i)* the unlinkability between deposit and withdrawing transactions *(ii)* the unlinkability between deposit and redeeming transactions *(iii)* the unlinkability between withdrawing and redeeming transactions.

Availability. Similar to the availability definition proposed by Meiklejohn and Mercer's Möbius system [35], AMR should ensure that *(i)* no one can prevent clients from using the mixer, and *(ii)* once the coins are deposited to the contract, no one can prevent clients from withdrawing their coins.

Frontrunning Resilience. Some transactions (i.e. deposit transactions) in AMR alter the state of the AMR contract, while other transactions (i.e. withdrawing/redeeming transactions) have to rely on the state of the contract to form the cryptographic proofs. Thus, if there are multiple concurrent deposit transactions issuing to the contract, some transactions will get invalidated by those transactions that modify the state of the contract. For example, in AMR, to withdraw or redeem a reward, a client Alice has to issue a withdrawal and a redemption transactions that contains cryptographic proofs proving that Alice deposited a coin in the past. Alice generates those cryptographic proofs w.r.t all current deposit transactions issued to the AMR contract. However, if another client Bob tries to deposit coins into the AMR contract, and Bob's transaction gets mined before Alice withdrawing/redeeming transactions, the proofs included in Alice transactions are no longer valid (because the state used for her proofs is outdated). This is a *front-running* problem [19, 23]. Therefore, to ensure the usability of the system, the AMR contract should be resilient against *front-running* by both clients and miners.

4.5 Threat Models

We assume that the cryptographic primitives (cf. Section 2) are secure. We further assume that adversaries are computationally bounded and can only corrupt at most 1/3 of the consensus participants of the blockchain. Thus, we assume that an adversary cannot tamper with the execution of the AMR smart contract. We assume that clients can always read the blockchain state and write to the blockchain. Note that blockchain congestion might temporarily affect the *availability* property of AMR, but does not impact the *correctness* and *privacy* properties. We assume that the adversary has the capabilities of a miner, i.e. can reorder transactions within a blockchain block, inject its own transactions before and after certain transactions. Also, we assume that the adversary can always read all transactions issued to the AMR contract, while the transactions are propagating on the P2P network, and afterwards when they are written to the blockchain. For a withdrawal and a redeem transaction, we assume that the client pays transaction fees either through a non-adversarial relayer (cf. Section 8), or the client possesses a blockchain address with funds that are not linkable to his deposit transaction. Finally, we assume that the underlying lending platforms used by AMR are secure.

5 DETAILED ZKSNARK-BASED SYSTEM CONSTRUCTION

We now present a zk-SNARK-based construction of AMR.

5.1 Building Blocks

Hash Functions. $H_p : \{0,1\}^* \rightarrow \mathbb{F}$ is a preimage-resistant and collision-resistant hash function that maps binary string to an element in $\mathbb{F}$, $H_{2p} : \mathbb{F} \times \mathbb{F} \rightarrow \mathbb{F}$ be a collision-resistant hash function that maps two elements in $\mathbb{F}$ into an element in $\mathbb{F}$.

Deposit Commitments. A secure commitment scheme ($\mathsf{P}_{\mathsf{com}}$, $\mathsf{V}_{\mathsf{com}}$) can be constructed using a secure hash function, $H_p : \{0,1\}^* \rightarrow \mathbb{F}$, as follows: (1) $\mathsf{P}_{\mathsf{com}}(m, r)$ returns $c = H_p(m||r)$, (2) $\mathsf{V}_{\mathsf{com}}(c, m, r)$ verifies if $c \stackrel{?}{=} H_p(m||r)$.

In AMR, to deposit, a client samples randomnesses, k_{dep}, r and computes the commitment: $\mathsf{cm} = H_p(k_{dep}||r)$ as a part of a deposit transaction.

Merkle Tree over Deposit Commitments, T_{dep}. The AMR contract maintains a Merkle tree, T_{dep}, over all commitments. a Merkle tree is an instance of an authenticated data structures for testing set membership [14] (cf. Section 2). The Merkle tree in the AMR contract is a complete binary tree and initialized with zero values at its leaves. As deposit transactions arrive, the AMR contract keeps track of the number of deposit transactions and updates the trees through the AcceptDeposit algorithm. A Merkle tree can be constructed using a collision-resistant hash function, H_{2p}.

We denote path_i the Merkle proof of cm_i. We denote the Merkle tree root at block h to be root^{h}_{dep}. We let root_{dep}.blockheight to be the height of the blockchain block when root_{dep} gets updated.

```
ContractSetUp($1^\lambda$)
Sample $H_p : \{0,1\}^* \rightarrow \mathbb{F}$ and $H_{2p} : \mathbb{F} \times \mathbb{F} \rightarrow \mathbb{F}$
Choose $\mathsf{amt} \in \mathbb{Z}_{>0}$ to be a fixed deposit amount
Choose $\mathsf{amt}_{rwd} \in \mathbb{Z}_{>0}$ to be a fixed reward amount
Choose $t_{con} \in \mathbb{Z}_{>0}$ to be condition for getting reward
Choose $d \in \mathbb{Z}_{>0}$, Let $X = \{x_1, \ldots, x_{2^d}\}$
        where $x_i = 0^\lambda$ for all $x_i \in X$
Choose $\Sigma$ to be the lending platform
Deploy $\Gamma_{\mathsf{AMR}}$ to be the interest distribution pool
Initialize an empty tree $\mathsf{root}_{dep} = T.\textsc{Init}(1^\lambda, X)$,
Choose $k \in \mathbb{Z}_{>0}$, set $\mathsf{RootList}_{wdr,k}[i] = \mathsf{root}_{dep}$,
    for $1 \leq i \leq k$
Set $\mathsf{root}^{\mathsf{curr}}_{rwd} = \mathsf{root}^{\mathsf{next}}_{rwd} = \mathsf{root}_{dep}$, $\mathsf{index} = 1$
Construct $C_{wdr}$ for statement described in Equation (1).
Let $\Pi$ be the zk-SNARK instance.
    – Run $(\mathsf{ek}_{dep}, \mathsf{vk}_{dep}) \leftarrow \Pi.\textsc{Setup}(1^\lambda, C_{wdr})$
Initialize: $\mathsf{DepositList} = \{\}$, $\mathsf{NullifierList} = \{\}$,
Deploy smart contract AMR with parameters :
pp = $(\mathbb{F}, H_p, H_{2p}, \mathsf{amt}, \mathsf{amt}_{rwd}, t_{con}, \Sigma, \Gamma_{\mathsf{AMR}}$
        $T, \mathsf{index}, \mathsf{RootList}_{wdr,k}, \mathsf{root}^{\mathsf{curr}}_{rwd}, \mathsf{root}^{\mathsf{next}}_{rwd},$
        $(\mathsf{ek}_{dep}, \mathsf{vk}_{dep}), \mathsf{DepositList}, \mathsf{NullifierList})$
```

Figure 2: AMR **Setup. The public parameters,** pp**, contains all information needed to interact with the** AMR **contract, and** pp **can be queried by any client.**

Withdrawal Proof. To withdraw coins from AMR, a client needs to prove three conditions: *(i)* the client knows the committed values of some existing commitments used to compute the tree root via zkSnark proof, *(ii)* the client did not withdaw in the past by passing a *fresh* nullifier value, *(iii)* the client knows the secret key used to issue the withdrawing transaction.

The last condition prevents network adversaries from stealing a valid proof by binding the public/private key to the zksnark proof. In particular, for a Merkle tree T with a root, root_{dep}, a client needs issue a proof proving the following statement:

$$st_{wdr} : \{\mathsf{pk}, \mathsf{sn}, \mathsf{root}_{dep}; \mathsf{sk}, k_{dep}, r, \mathsf{path}_i : \\ \mathsf{pk} = \textsc{extract}\mathrm{PK}(\mathsf{sk}) \wedge \mathsf{sn} = H_p(k_{dep}) \wedge \\ \mathsf{cm} = H_p(k_{dep}||r) \wedge T.\textsc{Verify}(i, \mathsf{cm}, \mathsf{root}_{dep}, \mathsf{path}_i))\} \quad (1)$$

Where pk, sn, root_{dep} are public values and

sk, k_{dep}, r, path_i are private values.

The nullifier value is used to ensure correctness by preventing clients from double-withdrawal.

Reward Proof. Intuitively, to prove that funds remained in the system for a certain time period, users can simply prove to the contract that they know some commitment their, cm, that is a member of an older Merkle root. To achieve such condition, the AMR contract always maintains an t_{con}-blocks-old Merkle root that serves as an anchor for clients to issue the reward proof. Similar to withdrawing, to redeem, clients need to nullify the old commitment, cm, by issuing a nullifier value, sn, and submit a new commitment, cm' to be eligible for future redeems and withdrawals. This requirement allows AMR to maintain system correctness and hide the link between reward-redeeming and withdrawing transactions.

In summary, to redeem coins from AMR, a client needs to prove that: *(i)* the client knows the committed value of some existing commitments used to compute the current reward Merkle tree root via a zkSnark proof, *(ii)* the client did not withdraw in the past by passing a *fresh* nullifier value sn, and *(iii)* Finally, the client needs to refresh its original deposit by submitting a new commitment to be eligible for future reward redemptions and withdrawals.

5.2 Contract Setup

Let $\mathbb{F}$ be the finite field used in AMR, during the AMR contract setup phase, the setup algorithm samples secure hash functions $H_p : \{0,1\}^* \rightarrow \mathbb{F}, H_{2p} : \mathbb{F} \times \mathbb{F} \rightarrow \mathbb{F}$ from secure collision-resistant hash families. The AMR contract is initialized with several parameters: amt for the fixed amount of coins to be mixed, amt_{rwd} indicating the fixed amount of coins to be rewarded, and t_{con} specifying the minimum number of blocks that clients need to wait before redeeming rewards.

Setting up Merkle Trees. Let T be the Merkle tree of depth d, the setup algorithm described in section 4.1 initializes T with zero leaves and initializes $\mathsf{index} = 1$ to keep track of latest deposits. Also, the algorithm initializes two lists: $\mathsf{RootList}_{wdr,k}$ to be the list of k most recent roots of T. Finally, the contract keeps track of the current reward root, $\mathsf{root}^{\mathsf{curr}}_{rwd}$ that is used by clients to form reward proofs. and the next reward root $\mathsf{root}^{\mathsf{next}}_{rwd}$. Recall that t_{con} to be the minimum number of blocks that clients need to wait before redeeming a reward, we require: $\mathsf{root}^{\mathsf{next}}_{rwd}.\mathsf{blockheight} - \mathsf{root}^{\mathsf{curr}}_{rwd}.\mathsf{blockheight} \geq t_{con}$. This approach helps the AMR contract maintain t_{con}-blocks-old reward root without storing all other roots.

Setting up zk-SNARK parameters. Let Π be the zk-SNARK instance used in AMR, the setup algorithm Section 4.1 constructs circuit C_{wdr} capturing the relation described in Equation (1). Then, the setup algorithm runs $\Pi.\textsc{Setup}$ on the circuit to obtain two keys, $(\mathsf{ek}_{dep}, \mathsf{vk}_{dep})$.

Setting up commitments and nullifier lists. The AMR contract is initialized with two empty lists: a list, DepositList, that contains all cm included in depositing and reward-redeeming transactions, a list, NullifierList, that contains all unique identifiers (i.e. sn) appeared in withdrawing and reward-redeeming transactions. Figure 2 formally describes this setup algorithm.

5.3 Client Algorithms

These following algorithms specify how clients interact with the AMR smart contract.

Depositing. CreateDepositTx allows a client to deposit coins into the contract and outputs secret notes, wit, that can later be used to withdraw coins or obtain a reward.

Redeeming Reward. CreateRedeemTx allows clients with the secret note, wit, and the secret key sk' to issue a proof, π_{rwd}, to prove to the AMR contract that that client has not withdrawn their deposited coins after certain number of block counts. In order to form such NIZK proof, the client obtains the current state of the

Deposit Interactions

Client(sk, amt) $\xrightarrow{\mathsf{tx}_{dep}}$ AMR **Contract**

```
CreateDepositTx(sk, amt) :
    Sample $(k_{dep}, r) \xleftarrow{\$} \{0,1\}^{\lambda}$
    Compute $\mathsf{cm} = H_p(k_{dep} || r)$
    return $\mathsf{tx}_{dep}$ = (amt, cm) and wit = $(k_{dep}, r)$
```

```
AcceptDeposit($\mathsf{tx}_{dep}$)
    Parse $\mathsf{tx}_{dep}$ = (amt′, cm)
    Require amt=amt′ and index $< 2^{d}$
    /* Invest to the lending platform */
    Execute $\Sigma$.Deposit(amt) to obtain $\mathsf{amt}_{\Sigma}$
    Append cm to DepositList
    Increment index = index + 1
    Compute
        - $\mathsf{root}_{new} = T_{dep}$.Update(index, cm, DepositList)
    Append $\mathsf{root}_{dep}$ to $\mathsf{RootList}_{wdr,k}$
    /* Update reward roots*/
    If Block.Height − $\mathsf{root}^{\mathsf{next}}_{rwd}$.blockheight $\geq t_{con}$ :
        - Set $\mathsf{root}^{\mathsf{curr}}_{rwd} = \mathsf{root}^{\mathsf{next}}_{rwd}$
        - Set $\mathsf{root}^{\mathsf{next}}_{rwd} = \mathsf{root}_{new}$
    return 1
```

Figure 3: AMR's deposit interactions between the client (CreateDepositTx algorithm) and AMR contract (AcceptDeposit algorithm). Transaction tx_{dep} is signed by sk. Block.Height denotes the block height of the block containing tx_{dep}

contract to compute private inputs (i.e. Merkle path) for the zk-SNARK proof generation. Also, the proof generation requires the client to use an older root maintained by the contract as part of the computation. This approach allows a client to prove to the contract that the client's transaction was deposited before the root was computed. Along with the NIZK proof, a client will include the nullifier value as part of the transaction to prevent double-redemption from the AMR contract. Finally, the client includes a new commitment value, cm′, in the reward-redeeming transaction to be eligible for future withdrawal and reward-redemption.

Withdrawing. CreateWithdrawTx allows a client with the secret note, wit, and secret key sk′ to issue proofs, π_{wdr}, to withdraw amt to the public key pk′. In this step, AMR requires the client to issue a proof to verify that the client has deposited coins in the past along with a nullifier value sn to prove that those coins have not been withdrawn before and to prevent clients from withdrawing coins without having any coins deposited to the AMR contract.

5.4 Contract Algorithms

In this part, we formally define the contract algorithms.

Accepting Deposit. Upon receiving a deposit transaction from an externally owned account, the contract verifies the amount amt, updates the tree structure, recomputes the Merkle roots for the Merkle tree, and updates the DepositList list. Next, the AMR contract deposits amt into the lending platform Σ to retrieve amt_{Σ}. Finally, depending on the number of blocks mined, the contract always maintains a t_{con}-blocks-old root, so that clients use it to redeem rewards. Figure 3 formally describes this procedure.

Issuing Reward. Upon receiving reward transactions, the contract verifies that the proof, π_{rwd}, is valid with the $\mathsf{root}^{\mathsf{curr}}_{rwd}$, and the nullifier sn_{rwd} is not in the NullifierList. Then the contract updates the NullifierList to prevent future double-redemption and double-withdrawal. Here, we note that $\mathsf{root}^{\mathsf{curr}}_{rwd}$ is the old state of the reward tree; therefore, being able to prove the membership of this root, one can prove that their deposit has not been withdrawn. Finally, the AMR contract updates the Merkle tree with the new commitment, cm_{new}. This update is similar to the deposit phase, and it allows the client to refresh their original commitment to be eligible for future redemptions or withdrawal.

Issuing Withdraw. Upon receiving withdrawal transaction, the contract verifies the validity of the proof, π_{wdr} and the freshness of the nullifier sn_{wdr} (i.e.,sn $\notin$ NullifierList). To prevent future double-withdrawal, the AMR contract then appends sn_{wdr} to NullifierList. Then, the AMR contract redeems all deposited funds from lending platform Σ along with the accrued interest R. Finally, the AMR deposits amt to the user's address and redeposits leftover funds into lending platforms. To avoid leakage in distributing accrued interest, AMR deposits the interest into a separate pool, Γ_{AMR}. Only users who hold the governance tokens obtained from rewards can later obtain this interest.

Figure 3, Figure 4, and Figure 5 formally describe the interactions between the clients and the smart contract in AMR.

Distributing Accrued Interest. We adapt the time-weight voting proposed by Curve [3] for this distribution step. In particular, in AMR, the pool, Γ_{AMR}, receives a portion of the total accrued interest upon each withdrawal. Clients need to lock governance tokens to the pool to be eligible for redeeming this interest. The AMR pool periodically distributes the total accrued interest proportionally to clients based on their voting power.

Client's voting power is calculated based on their amount of governance tokens and how long they are willing to lock those tokens in the distribution pool. The pool requires two main functionalities:

Reward-Redeeming Interactions

Client(sk', wit) $\xrightarrow{\mathsf{tx}_{rwd}}$ AMR **Contract**

```
CreateRedeemTx($\mathsf{sk}'$, wit) :
Parse wit = $(k_{dep}, r)$
Sample wit' = $(k'_{dep}, r') \xleftarrow{\$} \{0,1\}^{\lambda}$
Obtain $\mathsf{pp}^h$ from the contract
Compute $\mathsf{sn}_{rwd} = H_p(k_{dep})$
Compute $\mathsf{cm}_{old} = H_p(k_{dep}||r)$ and $\mathsf{cm}_{new} = H_p(k'_{dep}||r')$
Get index $i$ of $\mathsf{cm}_{old}$ from $\mathsf{DepositList}^h$
Compute $\mathsf{path}_i^h$ such that:
    - $T_{rwd}.\textsc{Verify}(i, \mathsf{cm}_{old}, \mathsf{root}_{rwd}^{\mathsf{curr}}, \mathsf{path}_i^h) = 1$
Form $\mathsf{wit}_{rwd} = (\mathsf{sk}', k_{dep}, r, \mathsf{path}_i^h)$
$\pi_{rwd} \leftarrow \Pi.\textsc{Prove}(\mathsf{ek}_{rwd}, st[\mathsf{pk}', \mathsf{sn}_{rwd}, \mathsf{root}_{rwd}^{\mathsf{curr}}], \mathsf{wit}_{rwd})$
return $\mathsf{tx}_{rwd} = (\mathsf{sn}_{rwd}, \mathsf{root}_{rwd}^{\mathsf{curr}}, \pi_{rwd}, \mathsf{cm}_{new})$, wit' = $(k'_{dep}, r')$
```

```
IssueReward($\mathsf{tx}_{rwd}$) :
Parse $\mathsf{tx}_{rwd} = (\mathsf{sn}_{rwd}, \mathsf{root}'_{dep}, \pi_{rwd}, \mathsf{cm}_{new})$
Require:
    - $\mathsf{root}_{rwd}^{\mathsf{curr}} = \mathsf{root}'_{dep}$
    - $\mathsf{sn}_{rwd} \notin \mathsf{NullifierList}$
    - $\Pi.\textsc{Verify}(\mathsf{vk}_{rwd}, \pi_{rwd},$
          $st[\mathsf{msg.sender}, \mathsf{sn}_{rwd}, \mathsf{root}_{rwd}^{\mathsf{curr}}]) = 1$
Append $\mathsf{sn}_{rwd}$ to NullifierList
/* Refresh the old commitment */
Append $\mathsf{cm}_{new}$ to DepositList
Increment index = index + 1
Compute
    - $\mathsf{root}_{new} = T_{dep}.\textsc{Update}(\mathsf{index}, \mathsf{cm}, \mathsf{DepositList})$
Append $\mathsf{root}_{dep}$ to $\mathsf{RootList}_{wdr,k}$
/* Update reward roots*/
If $\mathsf{Block.Height} - \mathsf{root}_{rwd}^{\mathsf{next}}.\mathsf{blockheight} \geq t_{con}$ :
    - Set $\mathsf{root}_{rwd}^{\mathsf{curr}} = \mathsf{root}_{rwd}^{\mathsf{next}}$
    - Set $\mathsf{root}_{rwd}^{\mathsf{next}} = \mathsf{root}_{new}$
Do $\mathsf{tx}_{rwd}.\mathsf{sender.transfer}(\mathsf{amt}_{rwd})$
return 1
```

Figure 4: AMR's reward-redeeming interactions between the client (CreateRedeemTx algorithm) and AMR contract (IssueReward algorithm). pp^h **denotes the state of the contract at block height** h**. The reward-redeeming transaction,** tx_{rwd}**, contains the proof** π_{rwd} **that proves to the AMR contract the client's knowledge of a** t_{con}**-blocks old deposit,** $\mathsf{cm}_{old} = H_p(k_{dep}||r)$ **which is a valid member of the Merkle tree with the root,** $\mathsf{root}_{rwd}^{\mathsf{curr}}$**.** sn_{rwd} **is used to nullify the old commitment, and** cm_{new} **is used to refresh the old commitment.** tx_{rwd} **is signed by** sk'**.** Block.Height **denotes the block height of the block containing transactions** tx_{rwd}

CreateLock and Claim. The pool is initialized with a value t_{max} denoting the maximum amount of blocks that client can lock their governance tokens.

- CreateLock(tx_{lock}) takes as input the locking transaction, tx_{lock}, from the client. The locking transaction contains the governance tokens, γ_{rwd}, and t_{lock} specify the number of blocks that the client will lock γ_{rwd} in the pool. At any given point in time, the voting power of the client is $\gamma_{rwd} \cdot \frac{t}{t_{max}}$ where t is the time left to unlock $t \leq t_{lock}$.
- Claim() is a contract function that can be periodically triggered by the clients. When a client triggers this function, the pool calculates the client's current voting weight as $w = \gamma_{rwd} \cdot \frac{t}{t_{max}}$, and the total voting power of all users, W. Finally, the pool distributes the accrued interest proportionally to the client according their voting power and the total voting power, $\frac{w}{W}$.

Similar to Curve [3], the main goal of time-weighted voting is to distribute more reward to users who *contribute* (i.e., having more governance tokens) and *commit* more to the system (i.e., locking their stakes for a longer period).

6 SYSTEM ANALYSIS

In this section, we informally discuss how AMR achieves the security goals mentioned in Section 4.4. As mentioned in Section 4.5, the underlying cryptographic primitives (i.e., zk-SNARK, commitment scheme, hash functions) are assumed to be secure, and AMR's depositing and withdrawing functionalities can be thought as the shielding and de-shielding transactions in ZCash with a fixed denomination. Therefore, the security of AMR follows from the security of zk-SNARK-based applications like ZCash [46]. In particular, the malicious outsider will not learn any information from the public data. However, adversaries can still guess the pair-wise link between a withdrawing, a depositing, and reward-redeeming transactions. The probability of guessing correctly largely depends on the number of deposits, redemptions, and withdrawals issued to AMR. Thus, we need to understand this adversarial probability to quantify the privacy offered by AMR.

6.1 Privacy Metric

We let h be the height of the blockchain, we define: $\mathsf{AnomSet}^h$ be the set of commitments issued to the AMR contract until block height h by *honest* users, and the adversary does not know the preimages of those commitments. $\mathsf{NullifierSet}^h$ be the set of nullifiers appeared

Withdraw Interactions

Client(sk′, wit)

```
CreateWithdrawTx(sk′, wit) :
    Parse wit = $(k_{dep}, r)$
    Obtain $\mathsf{pp}^h$ from the contract
    Compute $\mathsf{sn}_{wdr} = H_p(k_{dep})$
    Compute $\mathsf{cm} = H_p(k_{dep}||r)$
    Get index $i$ of cm from $\mathsf{DepositList}^h$
    Choose $\mathsf{root}_{dep} \in \mathsf{RootList}_{wdr,k}$
    Compute $\mathsf{path}^h_{dep,i}$ such that
        - $T$.Verify($i$, cm, $\mathsf{root}_{dep}$, $\mathsf{path}^h_{dep,i}$) = 1
    Form: $\mathsf{wit}_{dep} = (\mathsf{sk}', k_{dep}, r, \mathsf{path}^h_{dep,i})$
    $\pi_{wdr} \leftarrow \Pi$.Prove($\mathsf{ek}_{dep}$, $st$[pk′, $\mathsf{sn}_{wdr}$, $\mathsf{root}_{dep}$], $\mathsf{wit}_{dep}$)
    return $\mathsf{tx}_{wdr} = (\mathsf{sn}_{wdr}, \mathsf{root}_{dep}, \pi_{wdr})$
```

$\xrightarrow{\mathsf{tx}_{wdr}}$

AMR **Contract**

```
IssueWithdraw($\mathsf{tx}_{wdr}$) :
    Parse $\mathsf{tx}_{wdr} = (\mathsf{sn}_{wdr}, \mathsf{root}_{dep}, \pi_{wdr})$
    Require
        - $\mathsf{root}_{dep} \in \mathsf{RootList}_{wdr,k}$
        - $\mathsf{sn}_{wdr} \notin \mathsf{NullifierList}$
        - $\Pi$.Verify($\mathsf{vk}_{dep}$, $\pi_{wdr}$,
              $st$[msg.sender, $\mathsf{sn}_{wdr}$, $\mathsf{sn}_{rwd}$, $\mathsf{root}_{dep}$]) = 1
    Let $\mathsf{balance}_\Sigma$ be the $\Sigma$ token balance
    Let NoDepositsLeft be the number of deposits remained
    /* Redeeming from the lending platform */
    Redeem balance + $R$ by executing $\Sigma$.Redeem($\mathsf{balance}_\Sigma$)
        where balance = NoDepositsLeft · amt
    Append $\mathsf{sn}_{wdr}$ to NullifierList
    /* Send the accrued interest to the distribution pool */
    Do $\Gamma_{\mathsf{AMR}}$.transfer($R$/NoDepositsLeft)
    /* Send the original deposit to sender*/
    Do $\mathsf{tx}_{wdr}$.sender.transfer(amt)
    /* Reinvest into the lending platform */
    Let removedFund = amt + $R$/NoDepositsLeft
    Execute $\Sigma$.Deposit(balance + $R$ − removedFund)
    return 1
```

Figure 5: AMR's deposit interactions between the client (Client's CreateWithdrawTx algorithm) and AMR contract (AMR's IssueWithdraw algorithm). pp^h **denotes the state of the contract at block height** h**. The withdrawing transaction,** tx_{wdr}**, contains the proof** π_{wdr}**, that proves to the AMR contract the client's knowledge of a commitment,** cm**, which is a valid member of the Merkle tree with the root,** root_{dep}**.** sn_{wdr} **is used to nullify the old commitment,** cm**. Finally,** tx_{wdr} **is signed by** sk′**.**

in either reward-redeeming or withdrawing transactions issued to the AMR until block height h by *honest* users. AnomSet and NullifierSet are always available to the adversary. We also assume that $|\mathsf{AnomSet}^h| - |\mathsf{NullifierSet}^h| > 0$ for all h.

We say cm originates sn, when k_{dep} is used to compute both cm in tx_{dep} and sn_{wdr} in tx_{wdr} or tx_{rwd}. We define:

- $\mathsf{cm} \overset{link}{\leftarrow} \mathsf{sn}$: if the value k used to compute $\mathsf{cm} = H_p(k||r) \in \mathsf{tx}_{dep}$ or $\in \mathsf{tx}_{rwd}$ is equal to the value k used to compute $\mathsf{sn} = H_p(k) \in \mathsf{tx}_{rwd}$ or $\in \mathsf{tx}_{wdr}$.

The reward linking advantage is the probability that an adversary can output the correct commitment that originates the nullifier value appeared in reward-redeeming transactions. We define that probability is as follow:

Definition 5. *(Reward Linking Advantage) Let $\mathcal{A}$ be the PPT adversary, tx^{h+1}_{rwd} be the only valid reward-redeeming transaction issued at block $h+1$ from an honest user. Let sn^{h+1}_{rwd} be the nullifier appeared in tx^{h+1}_{rwd} We define the adversarial advantage as follow:*

$$\mathsf{Adv}^h_{\mathcal{A},rwd} = \Pr[\mathcal{A}(tx^{h+1}_{rwd}) \to \mathsf{cm} \in \mathsf{AnomSet}^h \ s.t. \ \mathsf{cm} \overset{link}{\leftarrow} \mathsf{sn}^{h+1}_{wdr}]$$

Similarly, the adversarial advantage in linking withdrawing transaction to other transactions is the probability that an adversary can guess correctly the commitment that originates the nullifier value appeared in withdrawing transaction.

Definition 6. *(Withdraw Linking Advantage) Let $\mathcal{A}$ be the PPT adversary, tx^{h+1}_{wdr} be the only valid withdrawing transaction issued at block $h+1$ from an honest user. Let sn^{h+1}_{wdr} be the nullifier appeared in tx^{h+1}_{wdr}. We define the adversarial advantage as follow:*

$$\mathsf{Adv}^h_{\mathcal{A},wdr} = \Pr[\mathcal{A}(tx^{h+1}_{wdr}) \to \mathsf{cm} \in \mathsf{AnomSet}^h \ s.t. \ \mathsf{cm} \overset{link}{\leftarrow} \mathsf{sn}^{h+1}_{wdr}]$$

We assume that the deposit addresses are independent and unlinkable accounts for our privacy metric to hold. If the same entity deposits from different addresses, but a blockchain analysis allows to link those addresses, the anonymity set would only grow by at most 1 deposit.

6.2 Privacy Analysis

Systems without reward. In a vanilla AMR system that only supports depositing and withdrawing functionalities, a withdrawal transaction can be at the origin of any deposit transactions of honest users before the withdrawal transaction , under the assumption that all underlying cryptographic primitives are secure. The adversarial advantage in linking withdrawing transaction to the original deposit transaction is: $\mathsf{Adv}^h_{\mathcal{A},wdr} = 1/|\mathsf{AnomSet}^h| + \mathsf{negl}(\lambda)$ where

$\mathsf{negl}(\lambda)$ is the adversarial advantage in breaking the underlying cryptographic primitive.

System with reward. Because AMR involves redeeming transactions, we need to analyze the adversarial advantages under different scenarios. In the following, we show the adversarial advantage in linking different transactions through the following claims.

Claim 1. *Assuming that all underlying cryptographic primitives are secure, the adversarial advantage in linking reward-redeeming transaction to other transactions as defined in Definition 5 is less than* $1/|\mathsf{AnomSet}^{h-t_{con}}| + \mathsf{negl}(\lambda)$

Sketch. AMR is parameterized with the value t_{con}, the number of blocks that a client needs to wait to be eligible for a reward. When the adversary observes a redeeming transaction issued to the AMR contract after block height $h+1$ from an honest user, a valid redeeming transaction indicates that the sender has issued commitment into the system at least $h - t_{con}$ blocks ago. Therefore, the probability that the adversary links the redeeming transaction to the correct commitment is hence: $\mathsf{Adv}^{h}_{\mathcal{A},rwd} \leq 1/|\mathsf{AnomSet}^{h-t_{con}}| + \mathsf{negl}(\lambda)$ where $\mathsf{negl}(\lambda)$ is the adversarial advantage in breaking the underlying cryptographic primitive.

Claim 2. *Assuming that all underlying cryptographic primitives are secure, the adversarial advantage in linking between withdrawing transaction to other transactions as defined in Definition 6 is* $1/|\mathsf{AnomSet}^{h}| + \mathsf{negl}(\lambda)$.

Sketch. Since we assume that the underlying cryptographic primitives are secure, the adversarial advantage in guessing correctly by breaking those primitives is negligible. Moreover, since each deposit and reward-redeeming transaction in AMR adds another leaf to the Merkle tree, the probability of guessing a correct leaf is equal to the number of Merkle leaves that are not controlled by the adversary. In another word, the probability is $1/(|\mathsf{AnomSet}^{h}|)$. Therefore, the adversarial advantage, $\mathsf{Adv}^{h}_{\mathcal{A},wdr} \leq 1/|\mathsf{AnomSet}^{h}| + \mathsf{negl}(\lambda)$ where $\mathsf{negl}(\lambda)$ is the adversarial advantage in breaking the underlying cryptographic primitive.

In summary, in AMR, given a reward-redeeming transaction, to guess the correct commitment, the adversary can reduce the size of the anonymity set by narrowing the search window to t_{con} blocks before the block containing the reward-redeeming transaction. On the other hand, given a withdrawing transaction, the adversary's advantage in guessing the correct commitment is still the same as the adversarial advantage in the system without reward, $1/|\mathsf{AnomSet}^{h}| + \mathsf{negl}(\lambda)$. The main reason is that, in AMR, beside each deposits, each reward-redeeming transaction also adds one additional commitment to the Anonymity Set, AnomSet. Therefore, AMR offers a bigger anonymity set than system without reward.

Privacy of the Accrued Interest Distribution. One naïve way to distribute the accrued interest is to split the total accrued interest equally among withdrawing addresses. This approach reveals nothing about the link between deposits and withdrawals. However, it introduces an unfair allocation of interest as users depositing into the system later receive the same amount of interest as users joining the system earlier.

In AMR, to achieve fairness in the accrued interest distribution, AMR allows users with governance tokens to lock their tokens in a separated pool (i.e., Γ_{AMR}). This pool receives a portion of the accrued interest from the AMR contract upon each withdrawal, and it periodically distributes the total accrued interest to addresses that lock their governance token into the pool. The amount each address receives is based on the number of governance tokens and how long those tokens are locked. It is not difficult to see that AMR ensures the fairness in the accrued interest allocation because only users contributing more to the anonymity set of AMR, can redeem more governance tokens; therefore, they can obtain more accrued interest. Moreover, under the assumption that clients use different addresses for each reward and withdraw, the unlinkability between deposit, redeem, and withdraw should be preserved.

6.3 Other goals achieved by AMR

In addition to the privacy goal, we briefly explain how AMR achieves the other goals defined in Section 4.4.

Correctness. AMR satisfies correctness. If an adversary can provide a withdrawal transaction that verifies without depositing any coins into the system, there are two possible scenarios: First, the adversary can derive a new valid transaction for the current state of the contract (i.e. observing commitment list), or it intercepts a withdrawal transaction and replaces the recipient address with its address. However, in the first case, it implies that the adversary breaks the preimage-resistant security of the underlying hash function $H_p(\cdot)$, and the second case implies that the adversary breaks the security of the zk-SNARK instance.

Availability. We argue that AMR satisfies availability. Unlike existing centralized tumbler designs [28], the availability of the system relies on the fact that the tumbler has to stay online. Similar to Möbius [35], AMR is a smart contract that executes autonomously on the blockchain, so adversary cannot prevent clients from interacting (i.e., reading and writing) with the blockchain.

Front-Running Resilience. Recall that the AMR contract stores a list of k recent roots. To invalidate a withdrawal transaction, an adversary needs to "front-run" at least k deposit transactions before a withdrawal transaction. Thus, one can choose the value k to be sufficiently large so that the cost of attacking is too expensive for the adversary to carry out. More specifically, to invalidate a single deposit transaction, the amount of token an adversary needs to have are at least $k \times (\mathsf{amt} + \mathsf{fee}_{dep})$ where amt is the fixed denomination specified in section 5.2 and fee_{dep} is the deposit fee. For example, if we set $k = 1000$, $\mathsf{amt} = 10$, assuming $\mathsf{fee}_{dep} = 0.02$, and let the token be *ether*, the adversary needs at least $k \times (\mathsf{amt} + \mathsf{fee}_{dep}) = 10,020$ *ethers* ($38m$ USD) to carry out the attack, and the adversary will lose at least 20 *ethers* (76,000 USD) in term of fee.

7 EVALUATION

7.1 Parameters

Choice of cryptographic primitives. We use Groth's zk-SNARK [27] as our instance of zk-SNARK due to its efficiency in term of proofs' size and verifier's computations. For cryptographic hash functions, we use a Pedersen hash function [37] for H_p and evaluate AMR using two different choices of hash functions for H_{2p}: the MiMC [8] and the Poseidon hash function [26]. Arithmetic circuits using MiMC and Poseidon hash yield a lower number

Tree Depth	# Constraints		Setup Time		Keys Size	
	C_{wdr}		t_{wdr}		$(\mathsf{ek}_{wdr}, \mathsf{vk}_{wdr} = 640B)$	
	Poseidon	MiMC	Poseidon	MiMC	Poseidon	MiMC
10	$4,245$	$15,045$	$86.56s$	$246.99s$	4.3MB	7.3MB
15	$5,460$	$21,660$	$107.34s$	$377.44s$	5.8MB	11.1MB
20	$6,675$	$28,275$	$126.51s$	$465.27s$	7.3MB	13.8MB
25	$7,890$	$34,890$	$146.41s$	$642.04s$	8.8MB	18.6MB
30	$9,105$	$41,505$	$185.64s$	$729.03s$	10.8MB	21.4MB

Table 1: zk-SNARK Setup Cost

of constraints and operations when compared to arithmetic circuits relying on other hash functions [4, 8] (i.e. SHA-256, Keccak). Moreover, both MiMC and Poseidon hash functions are not only designed specifically for SNARK applications, but also highly efficient for Ethereum smart contract applications in terms of gas costs. Finally, as discussed in Section 5.1, the commitment scheme and the Merkle tree can be directly instantiated using Pedersen and MiMC/Poseidon hash functions.

Software. For the arithmetic circuit construction, we use the `Circom` library [29] to construct the withdrawing circuit, C_{wdr} for the relation described in Equation (1). We use Groth's zk-SNARK proof system implemented by the `snarkjs` library [30] to develop the client's algorithms (cf. Section 4.2), and to perform the trusted setup for obtaining the proving and evaluation keys for the AMR contract and clients. We deploy AMR to the Ethereum Kovan testnet [4] [5]. AMR contract consists 1013 lines of Solidity code.

Hardware. We conducted our experiment on a commodity desktop machine, which is equipped with an Intel Core i5-7400 @3.800GHz CPU, 32GB RAM.

7.2 Performance

We measure the performance and the cost of AMR using the following tree depths $d = 10, 15, 20, 25, 30$.

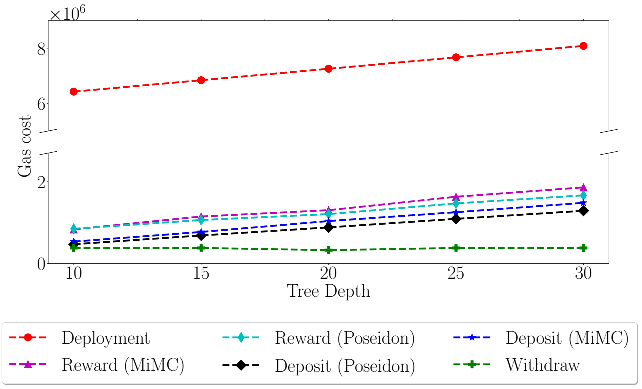

Figure 6: On-chain Costs of Deployments, Deposit, Withdrawal, and Reward Redemption for Different Tree Depths and Hash Functions.

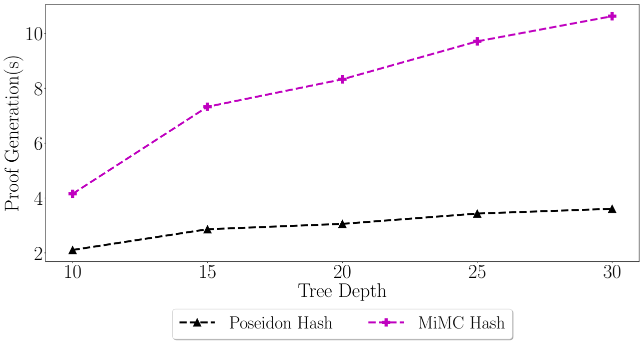

Figure 7: zkSnark Proof Generation Time for Poseidon and MiMC hash functions.

zk-SNARK Setup. Table 1 presents an overall performance of the zk-SNARK setup for the withdraw circuit. For the MiMC hash function, for a tree of depth d, the withdraw circuit has $1,815 + 1,323 \times d$ constraints. For the Poseidon hash function, the withdraw circuit has $1815 + 243 \times d$ constraints.

Onchain Costs. Figure 6 provides the overall costs of deployment, deposit, reward, and withdraw for different tree depths. The cost of deploying the contract is the most expensive operation, accounting from $\approx 6m$ gas for $d = 10$ to $\approx 8m$ gas for $d = 30$ for both the MiMC and Poseidon hash functions. However, we note that the deployment cost is a one-time cost which is amortized over the lifetime of the contract. The cost of the depositing transaction depends on the depth of the tree, which is approximately $43,000 + 51,000 \times d$ for the MiMC hash and approximately $43,000 + 41,000 \times d$ for the Poseidon hash function. The gas cost for verifying a withdrawing transaction is approximately $320,000$ for all tree depths and both choices of hash functions. The gas cost for a reward-redeeming transaction is equal to a total gas cost of a deposit and a withdraw as the AMR contract needs to verify the zkSnark proof as well as to update the Merkle tree.

zk-SNARK Proof Generation. As the Poseidon hash function generates less constraints for the arithmetic circuit than the MiMC hash function (i.e. 243 vs 1323), we observe a reduction of 3× for the clients' proof generation time with an AMR system using the Poseidon hash function. Figure 7 presents the time for a client to generate the zkSnark proofs.

Lending Platforms' Additional Costs. In additional to the cost of executing cryptographic functions in the AMR contract, we also need to consider the cost of other interactions with decentralized lending platforms such as Aave [1] or Compound [2]. These costs are the gas cost of depositing into and redeeming from lending platforms. We estimate the costs of these interactions using data from Etherscan [6] and Compound developer documentations [7]. Thus, depositing into these lending platforms takes approximately $0.3m$ gas (for both Aave and Compound), and redeeming from these platforms takes less than $0.2m$ gas for Aave and less than $0.1m$ gas

[4] AMR's address: 0xdE992c4fBd0f39E5c0356e6365Bcfafa1e94970b

[5] A demo video AMR can be found at the following URL: https://youtu.be/-oAQlsRTF08

[6] https://etherscan.io/

[7] https://compound.finance/docs#networks

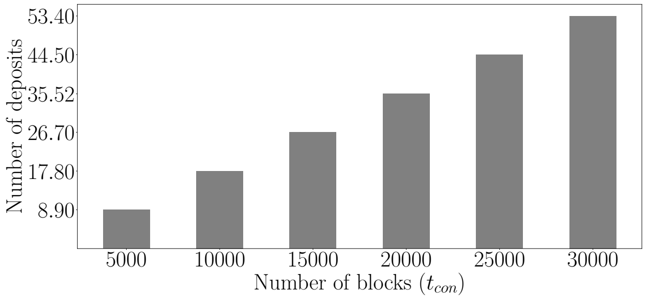

Figure 8: Average number of deposit transactions issued to the contract over the span of $5,000$**,** $10,000$**,** $15,000$**,** $20,000$**,** $25,000$**,** $30,000$ **blocks.**

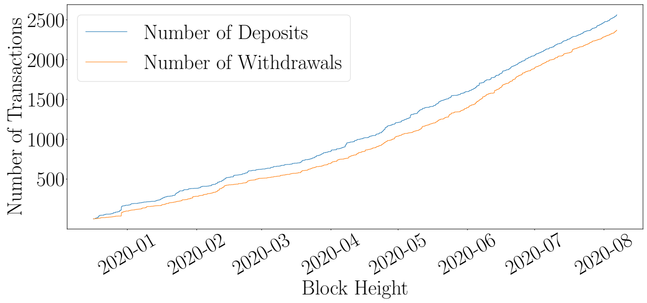

Figure 9: Number of deposits and withdrawals issued to the tornado cash 10 **ETH pool.**

for Compound. Therefore, depending on the choice of lending platforms, we would expect additional $0.3m$ gas for AMR's depositing function and additional $0.2m$ gas for AMR's withdrawing function.

7.3 Empirical analysis on Tornado Cash

To become eligible for a reward payment, clients need to keep their deposit in the contract locked for a predefined period (i.e. t_{con} blocks). Thus, one needs to decide a suitable value for t_{con}.

We perform an empirical analysis on the tornado cash system [5] which is, to the best of our knowledge, the only zk-SNARK-based mixer deployed to the Ethereum main net. Tornado cash supports two operations: deposit and withdraw. We analyzed their 10.0 ETH denomination deposit pool [8] from block $9,161,895$ (25 December 2019) to block $10,726,597$ (25 August 2020) to understand how frequent clients deposit to the tornado cash systemreviewer. This frequency allows us to derive an appropriate value for how long client should keep their funds in AMR contract to be eligible for a reward. For example, Figure 8 suggests that for the waiting period of $t_{con} = 30,000$ (approximately 4.5 days), we can expect an additional 52 deposit transactions issued to the contract intermittently, and the more deposit transactions reach the contract, the higher the anonymity set becomes.

Moreover, over the course of 8 months (Cf. Figure 9), we observe a total of $2,810$ deposit transactions, and $2,606$ withdrawing transactions on the tornado cash contract. We note that if the number of withdrawing transactions equals to the number of deposit transactions at any point, the size of the anonymity set is reduced to zero. Thus, in contrast to Tornado cash, the reward mechanism in AMR is used to incentivise clients to keep a deposit in the system to help maintain a healthy gap between the number of deposits and withdrawals.

8 DISCUSSION AND APPLICATIONS

Trusted Setup in zk-SNARK. As discussed in Section 2, a zk-SNARK requires a trusted setup to generate the evaluation and proving key for each circuit. While one can assume that there exists a trusted third party which helps run the setup, this trust assumption is typically not welcome by the blockchain community, because if such third party can maliciously generate the keys (or the common reference string), it can form a valid proof and steal the contract funds.

To remove the trusted third party assumption, one can run a multi-party computation (MPC) setup where users can contribute a share to the trusted setup. Several works [12, 17, 18] proposed different protocols for such trusted setup, and they showed that as long as one participant is honest, the zk-SNARK instance will be secure. In particular, the Zcash team has performed such MPC setup for their protocol parameters in 2017 [39]. However, the MPC setup may need to be carried out independently for different circuits and related works [20, 21, 24, 33] have proposed several zk-SNARK constructions that utilizes a universal setup that can be used for *any* circuits with a bounded size. These zk-SNARK constructions can be easily integrated into AMR in the future.

Sender Outsources Transaction Fee Payment. Issuing a transaction requires the payment of fees, and clients should not use the same address for such payment; otherwise, their addresses can be linked. In practice, users can choose to use a relayer that broadcasts transactions and is paid from a fraction of the withdraw or reward transaction. In addition, the relayer can receive the corresponding client proof through a side channel. In term of privacy, the relayer does not learn anything beyond the client's receiving address and the validity of the proof because of the security guarantees of zkSnark (i.e., Zero-Knowledge Property). However, outsourcing transaction fee payments to relayers may compromise availability as relayers can refuse to relay clients' transactions.

Transferring Arbitrary Denomination. The current version of AMR does not allow clients to transfer an arbitrary amount of coins among clients privately. To achieve such property, one either needs an out-of-band communication channel between a sender and a recipient to transfer secret notes, or the sender can spend more fees to store additional encrypted data on-chain. Moreover, to prevent a sender from stealing coins from the recipient, one could use a similar commitment scheme and encryption as used in Zcash [7]; however, the use of these primitives will increase the cost of the on-chain verification. Nevertheless, adding the transferring functionality to AMR can be an interesting future work.

Limitation of a Fixed Waiting Period. The current design of AMR requires a fixed waiting period for all clients. This fixed waiting period is necessary in order to offer a fair privacy guarantee for all users. However, with this information, the adversary can

[8] Address: 0x910Cbd523D972eb0a6f4cAe4618aD62622b39DbF

potentially separate profit-seeking users from anonymity-seeking users because the former will tend to redeem/withdraw immediately when the waiting period expires. Therefore, one potential approach is to use pseudo-random function (PRF) with deposit and block data to assign a random waiting period for each participant when they deposit. Still, with the use of PRF and public data, the waiting period will be plaintext; hence, based on on-chain public data, the adversary can still separate two sets of users. Thus, autonomously verifying that clients have waited for the assigned waiting time can be challenging. Nevertheless, addressing the fixed waiting period problem can be interesting future work.

Constant Querying State. Most blockchain clients (e.g. MetaMask) outsource their blockchain information to centralized services such as Infura. Those centralized services are aware of, the clients' blockchain address(es), IP address as well of the fact that the client queried the AMR contract state. These services are therefore privacy critical, as they may be able to link different addresses from the same client. We hence recommend a privacy aware client to operate an independent validating full blockchain client, or use oblivious light client solutions [31, 40] or network-level anonymity solutions such as Tor or Virtual Private Network (VPN) before connecting to these centralized services.

Decentralized governance. Once deployed, AMR's system parameters, will likely need to be adjusted during its lifetime. One could chose an admin key to govern AMR, for the sake of decentralization, however, we believe that a decentralized approach would be beneficial. A governance token is hence the natural choice, whereby AMR can itself distribute those tokens to the clients participating in the protocol. We have identified the following parameters that should be governed: *(i)* new relayer addresses, *(ii)* condition for client reward, *(iii)* the amount of the reward. Once a new version of AMR is developed, the governance mechanism could vote to *(iv)* migrate deposits to a new contract with new features/bug fixes.

9 RELATED WORK ON ADD-ON PRIVACY SOLUTIONS

Add-on privacy solutions for smart contract-enabled blockchains. While we are not aware of any academic works that propose a zk-SNARK-based mixing system as ours, Tornado cash [5] appears to be the first system deployed which allows clients to deposit and withdraw fixed amount of coins. Our work is to the best of our knowledge the first academic work which presents such a system, formalizes the privacy and security properties, and adds a novel privacy preserving reward mechanism.

Meiklejohn *et al.* [35] propose an Ethereum-based tumbler called Möbius. The construction of Möbius relies on the linkable ring signature primitive and stealth address mechanism used in Monero [9] 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, AMR offers a bigger anonymity set over time while operating at constant system costs.

Bünz *et al.* proposed a private payment protocol for the Ethereum blockchain called Zether [19]. However, the cost of Zether transactions (i.e. $7.8m$ gas) is expensive for Ethereum, and Zether does not hide the link between the receiver and recipient. Diamond proposed Anonymous Zether [22] to address linkability problem, but the cost of Anonymous Zether is still expensive for Ethereum. For the maximum anonymity set of size 64 reported in the paper, the gas cost of a single transferring call in Anonymous Zether is $48.7m$ gas which is approximately 32 times the cost of an AMR deposit and 130 times the cost of an AMR withdrawal for $h = 30$.

Rondelet and Zajac proposed Zeth [42], which implements all functionalities of ZeroCash [46] as an Ethereum smart contract. While Zeth allows different functionalities, such as transferring arbitrary denomination of notes, it comes with the cost of using a bigger zk-SNARK circuit than in AMR. We expect that the zk-SNARK proof generation time in Zeth is an order of magnitude larger than in AMR. For the zkSnark proof verifcation, the authors of Zeth also report that an estimated cost of verifying a zk-SNARK proof is approximately $2m$ gas (5× the cost in AMR).

Other Tumbler Designs. The community proposes several centralised tumbler designs [16, 28, 47, 48]. The main essence of those designs relies on a centralised offchain server to mix users' funds, e.g., Tumblebit [28] and A2L [47]. Both require less trust in the offchain server than solutions such as Mixcoin [16] and Blindcoin [48] by preventing the server from stealing funds from participants. However, centralised tumbling protocols cannot ensure the availability property, because the centralised system can always censor deposits from clients. Existing decentralized tumbler designs, such as Coinshuffle [44, 45] and Coinjoin [34], address the availability problem by proposing protocols allowing participants to interact and form transactions that helps hide the sender from the recipient. However, the availability of participants and the interactivity among them can be difficult to enforce and may lead to privacy leaking side channels.

10 CONCLUSION

Coin mixers allow alleviating to some degree the missing privacy properties of open and permissionless blockchains. Their operations are cost-intensive both from a transaction fee perspective and because "better" privacy is more expensive than "weaker" privacy when measuring privacy quality quantitatively with the anonymity set size.

In this work, we introduce a zk-SNARK-based coin mixer AMR. AMR is to our knowledge the first construction that allows to reward mixer participants which hold coins within the mixer for at least time t. Moreover, AMR allows users to earn interest on the deposited funds by leveraging popular DeFi lending platforms. This incentive mechanism should not only attract privacy-seeking users, but also participants that are interested in the underlying reward distribution. Therefore, we hope that such a system fundamentally broadens the diversity of the mixer user, improving the anonymity set quality for all involved users. Our implementation and evaluation shows that our mixer is practical by supporting anonymity set sizes beyond thousands of users.

Acknowledgement. The authors would like to thank Ben Livshits, Liyi Zhou, and the anonymous reviewers for all their insightful comments and suggestions. This work was partially supported by the Liquidity Network.

REFERENCES

[1] Aave: The money market protocol. https://aave.com/.

[2] Compound. https://compound.finance/.

[3] Curve dao. https://curve.fi/.

[4] Jubjub. Available at: https://z.cash/technology/jubjub/.

[5] Tornado cash. Available at: https://tornado.cash/.

[6] Yearn finance. https://yearn.finance/.

[7] Zcash. Available at: https://z.cash/.

[8] Martin Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and Tyge Tiessen. Mimc: Efficient encryption and cryptographic hashing with minimal multiplicative complexity. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology – ASIACRYPT 2016*, pages 191–219, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[9] Kurt M. Alonso. Zero to Monero: First edition. a technical guide to a private digital currency; for beginners, amateurs, and experts. https://web.getmonero.org/library/Zero-to-Monero-2-0-0.pdf.

[10] Elli Androulaki, Ghassan O Karame, Marc Roeschlin, Tobias Scherer, and Srdjan Capkun. Evaluating user privacy in bitcoin. In *International Conference on Financial Cryptography and Data Security*, pages 34–51. Springer, 2013.

[11] Shehar Bano, Alberto Sonnino, Mustafa Al-Bassam, Sarah Azouvi, Patrick McCorry, Sarah Meiklejohn, and George Danezis. Sok: Consensus in the age of blockchains. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, pages 183–198, 2019.

[12] E. Ben-Sasson, A. Chiesa, M. Green, E. Tromer, and M. Virza. Secure sampling of public parameters for succinct zero knowledge proofs. In *2015 IEEE Symposium on Security and Privacy*, pages 287–304, 2015.

[13] Josh Benaloh and Michael de Mare. One-way accumulators: A decentralized alternative to digital signatures. In Tor Helleseth, editor, *Advances in Cryptology — EUROCRYPT '93*, pages 274–285, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[14] Dan Boneh and Victor Shoup. A graduate course in applied cryptography, 2020.

[15] Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua A Kroll, and Edward W Felten. Sok: Research perspectives and challenges for bitcoin and cryptocurrencies. In *Symposium on Security and Privacy*, pages 104–121. IEEE, 2015.

[16] Joseph Bonneau, Arvind Narayanan, Andrew Miller, Jeremy Clark, Joshua A Kroll, and Edward W Felten. Mixcoin: Anonymity for bitcoin with accountable mixes. In *International Conference on Financial Cryptography and Data Security*, pages 486–504. Springer, 2014.

[17] Sean Bowe, Ariel Gabizon, and Matthew D. Green. A multi-party protocol for constructing the public parameters of the pinocchio zk-snark. In Aviv Zohar, Ittay Eyal, Vanessa Teague, Jeremy Clark, Andrea Bracciali, Federico Pintore, and Massimiliano Sala, editors, *Financial Cryptography and Data Security*, pages 64–77, Berlin, Heidelberg, 2019. Springer Berlin Heidelberg.

[18] Sean Bowe, Ariel Gabizon, and Ian Miers. Scalable multi-party computation for zk-snark parameters in the random beacon model. Cryptology ePrint Archive, Report 2017/1050, 2017. https://eprint.iacr.org/2017/1050.

[19] Benedikt Bünz, Shashank Agrawal, Mahdi Zamani, and Dan Boneh. Zether: Towards privacy in a smart contract world. *IACR Cryptol. ePrint Arch.*, 2019:191, 2019.

[20] Matteo Campanelli, Dario Fiore, and Anaïs Querol. Legosnark: Modular design and composition of succinct zero-knowledge proofs. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 2075–2092, New York, NY, USA, 2019. Association for Computing Machinery.

[21] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas Ward. Marlin: Preprocessing zksnarks with universal and updatable srs. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology – EUROCRYPT 2020*, pages 738–768, Cham, 2020. Springer International Publishing.

[22] Benjamin E. Diamond. "many-out-of-many" proofs with applications to anonymous zether. Cryptology ePrint Archive, Report 2020/293, 2020. https://eprint.iacr.org/2020/293.

[23] Shayan Eskandari, Seyedehmahsa Moosavi, and Jeremy Clark. Sok: Transparent dishonesty: Front-running attacks on blockchain. In Andrea Bracciali, Jeremy Clark, Federico Pintore, Peter B. Rønne, and Massimiliano Sala, editors, *Financial Cryptography and Data Security*, pages 170–189, Cham, 2020. Springer International Publishing.

[24] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Report 2019/953, 2019. https://eprint.iacr.org/2019/953.

[25] Arthur Gervais, Srdjan Capkun, Ghassan O Karame, and Damian Gruber. On the privacy provisions of bloom filters in lightweight bitcoin clients. In *Computer Security Applications Conference*, pages 326–335, 2014.

[26] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. Poseidon: A new hash function for zero-knowledge proof systems. Cryptology ePrint Archive, Report 2019/458, 2019. https://eprint.iacr.org/2019/458.

[27] Jens Groth. On the size of pairing-based non-interactive arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology – EUROCRYPT 2016*, pages 305–326, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[28] Ethan Heilman, Leen Alshenibr, Foteini Baldimtsi, Alessandra Scafuro, and Sharon Goldberg. Tumblebit: An untrusted bitcoin-compatible anonymous payment hub. In *Network and Distributed System Security Symposium*, 2017.

[29] Iden3. Circom: Circuit compiler for zksnark. https://github.com/iden3/snarkjs.

[30] Iden3. Snarkjs: Javascript and pure web assembly implementation of zksnark schemes. https://github.com/iden3/snarkjs.

[31] Duc V. Le, Lizzy Tengana Hurtado, Adil Ahmad, Mohsen Minaei, Byoungyoung Lee, and Aniket Kate. A tale of two trees: One writes, and other reads. *Proceedings on Privacy Enhancing Technologies*, 2020(2):519–536, 2020.

[32] Jiangtao Li, Ninghui Li, and Rui Xue. Universal accumulators with efficient nonmembership proofs. In Jonathan Katz and Moti Yung, editors, *Applied Cryptography and Network Security*, pages 253–269, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[33] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. Sonic: Zero-knowledge snarks from linear-size universal and updatable structured reference strings. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 2111–2128, New York, NY, USA, 2019. Association for Computing Machinery.

[34] Greg Maxwell. Coinjoin: Bitcoin privacy for the real world. In *Post on Bitcoin forum*, 2013.

[35] Sarah Meiklejohn and Rebekah Mercer. Möbius: Trustless tumbling for transaction privacy. *Proceedings on Privacy Enhancing Technologies*, 2018(2):105–121, 2018.

[36] Ralph 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.

[37] Silvio Micali, Michael Rabin, and Joe Kilian. Zero-knowledge sets. In *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '03, page 80, USA, 2003. IEEE Computer Society.

[38] Ian Miers, Christina Garman, Matthew Green, and Aviel D Rubin. Zerocoin: Anonymous distributed e-cash from bitcoin. In *Symposium on Security and Privacy*, pages 397–411, 2013.

[39] Andrew Miller and Sean Bowe. Zcash MPC Setup. https://www.zfnd.org/blog/powers-of-tau/.

[40] Kaihua Qin, Henryk Hadass, Arthur Gervais, and Joel Reardon. Applying private information retrieval to lightweight bitcoin clients. In *2019 Crypto Valley Conference on Blockchain Technology (CVCBT)*, pages 60–72. IEEE, 2019.

[41] Phillip Rogaway and Thomas Shrimpton. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In *FSE 2004*, pages 371–388, 2004.

[42] Antoine Rondelet and Michal Zajac. Zeth: On integrating zerocash on ethereum, 2019.

[43] Tim Ruffing and Pedro Moreno-Sanchez. Valueshuffle: Mixing confidential transactions for comprehensive transaction privacy in bitcoin. In *International Conference on Financial Cryptography and Data Security*, pages 133–154. Springer, 2017.

[44] Tim Ruffing, Pedro Moreno-Sanchez, and Aniket Kate. Coinshuffle: Practical decentralized coin mixing for bitcoin. In *European Symposium on Research in Computer Security*, pages 345–364. Springer, 2014.

[45] Tim Ruffing, Pedro Moreno-Sanchez, and Aniket Kate. P2P mixing and unlinkable bitcoin transactions. In *Network and Distributed System Security Symposium*, 2017.

[46] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *Symposium on Security and Privacy*, pages 459–474. IEEE, 2014.

[47] Erkan Tairi, Pedro Moreno-Sanchez, and Matteo Maffei. A2l: Anonymous atomic locks for scalability and interoperability in payment channel hubs. Technical report, Cryptology ePrint Archive, Report 2019/589, 2019.

[48] Luke Valenta and Brendan Rowan. Blindcoin: Blinded, accountable mixes for bitcoin. In *International Conference on Financial Cryptography and Data Security*, pages 112–126. Springer, 2015.

[49] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151:1–32, 2014.