MixEth: efficient trustless coin mixing service for Ethereum

István András Seres[1], Dániel A. Nagy[1], and Péter Burcsi[1]

[1]Department of Computer Algebra, Eötvös Loránd University

November 3, 2018

Note: this is an early-stage work-in-progress! Security proofs, (state channel) implementation and many more are yet to come!

Abstract

Cryptocurrencies enable users to transact with each other without relying on trusted parties or intermediaries. These transactions are recorded in an immutable, publicly verifiable ledger. Due to this transparent nature of the ledger, privacy is notably reduced. If the link between users' public key and their physical identity is exposed their pseudonymity is lost. One way to increase users' privacy is to deploy coin mixing services. In this paper, we present MixEth, which is a trustless coin mixing service. MixEth is more efficient than any proposed trustless coin tumbler. It requires only 3 on-chain transactions at most per user and 1 off-chain. It achieves strong notions of anonymity and is able to resist denial-of-service attacks.

Keywords: Cryptography, Verifiable shuffle, Cryptocurrency, Ethereum, Coin mixer

1 Introduction

Bitcoin [16] and other cryptocurrencies are pseudonymous. Users' public keys are used as pseudonyms in these systems. Transactions essentially record a flow of cryptocurrency from one (or more) public keys to another public key (or more). Flow of cryptocurrency can be easily tracked due to the open and transparent nature of cryptocurrencies' transaction ledger. Moreover, coherent public keys, which are used by the same user, can be clustered merely by analyzing the ledger. Recently several tools and algorithms were proposed to diminish users' privacy ([13],[15], [14]). Such deanomyzation attacks are extremely harmful to user privacy, especially in the case when any of the users' pseudonyms, public keys, are linked to their real world identity.

One of the methods to increase users' privacy is coin mixing or tumbling. This technique provides *k-anonymity* or *plausible deniability*. The idea is that k users deposit 1 coin each and then in the course of a coin shuffling protocol either a centralized trusted third party or a smart contract mixes the coins and redistributes them to designated fresh public keys. This powerful technique gives users superior privacy and anonimity since their new received coins can not be linked to them.

Several coin mixing protocols were proposed in the literature both centralized ([6], [19], [10]) and decentralized ([11], [18], [1], [12], [4]). A major drawback of centralized coin mixing is that the availability of the tumbler is entirely dependent on the trusted party and in most cases theft prevention can not be guaranteed ([6], [19]). On the other hand decentralized tumblers achieve availability, theft prevention and satisfy strong notions of anonymity although they are considerably heavier computationally. In the following we will solely focus on the problem of coin mixing on Ethereum [20].

The two major techniques to provide mixing services for Ethereum are Möbius, a ring-signature-based solution [12] and Miximus, a zkSNARK-based proposal [1]. Both of them burn tremendous amounts of gas to withdraw funds, which could be prohibitive for many use cases. Möbius requires $335{,}714n$ gas (n is the ring size) while Miximus consumes 1,903,305 gas to verify a zkSNARK proof [2].

Our contributions. In this paper, we present MixEth, to overcome the above mentioned efficiency issues while retaining strong notions of anonymity already achieved by previous proposals. MixEth requires as few off-chain messages and on-chain transactions as Möbius and Miximus,

meanwhile it burns significantly less gas. Game-theoretical analysis of incentives in MixEth is also enclosed.

2 Background

2.1 Notations

In most cases if it is possible we will stick to the notations used in [12] for sake of uniformity. Let [] denote the empty tuple. For a tuple $t = (x_1, \ldots, x_n)$ we denote as $t[x_i]$ the value stored at x_i. The cardinality of a finite set X is denoted as $|X|$. In the following let $\lambda \in \mathbb{N}$ be the security parameter and its unary representation is 1^λ. If x is uniformly randomly sampled from a set A we write $x \xleftarrow{\$} A$. The symmetric group of degree n is written as S_n. In a cyclic group $\mathbb{G}$, the standardized generator is denoted as G and we use the additive notation. Secret keys and public keys are denoted as sk and pk respectively (or often times s and sG), while the user the corresponding key belongs to is indicated in subscript. Let PK_i denote the set of public keys belonging to receivers at a particular shuffling round i.

We use games in definitions and proofs of security. At the end of each game, the main procedure of game G outputs a single bit. $\Pr(\mathsf{G})$ denotes the probability that the output is 1.

2.2 Cryptographic keys in Ethereum

Ethereum uses Elliptic Curve Cryptography (ECC) to secure users' funds. More specifically, it uses the secp256k1 curve, the same one as used in Bitcoin. If a user wants to create an Ethereum address, first she needs to generate a secret key $s \xleftarrow{\$} \mathbb{Z}_n$, where n is the order of secp256k1 over a finite field $\mathbb{F}_p$. The corresponding public key will be sG. Note that any multiples of G is also a generator of curve points since n, the order of the group is also a prime. Accounts in Ethereum are identified by their addresses which can be obtained by taking the right most 20 bytes of the Keccak hashed public key [20].

2.3 Verifiable shuffle

Neff introduced the notion of verifiable shuffle [17]. It is a cryptographic protocol allowing a party to verifiably shuffle a sequence of k modular integers. The output of the shuffle is another k modular integers raised to the same secret exponent only known to the shuffler. The shuffler can generate a publicly verifiable zero-knowledge proof to convince the public that the shuffle was done correctly.

Neff's mathematical construct is extremely powerful, since it only relies on the intractability of the Decisional Diffie-Hellman (DDH) problem. Therefore, Neff's verifiable shuffle can also be applied in groups over elliptic curves.

Verifiable shuffle can be used to shuffle a set of public keys, $PK = (s_1G, s_2G \ldots, s_kG)$. Note that secret keys are not known to the shuffler.

1. Shuffler commits to $C = cG$, publishes

$$PK^* = (c(s_{\pi^{-1}(1)}G), c(s_{\pi^{-1}(2)}G), \ldots, c(s_{\pi^{-1}(k)}G))$$

   where π is a random permutation. Shuffler additionally computes and publishes a zero-knowledge proof about the correctness of the shuffle. This proof can be made non-interactive via the Fiat-Shamir heuristic. Let us call C as the shuffling constant.

2. Assuming the proof verifies users gain new public keys with respect to another generator element, namely cG.

For verifying the proof one needs to compute $8k + 5$ exponentiations, however later this result was ameliorated to $3,5k$ exponentiations by Bayer and Groth [3].

So far verifiable shuffles were only applied in voting schemes, we argue that they are useful in trustless coin mixers as well.

2.4 Decision Diffie-Hellman Problem and Chaum-Pedersen Protocol

The Decision Diffie-Hellman assumption (DDH) is a standard cryptographic hardness assumption which underlies the security of many cryptographic protocols. Roughly speaking DDH states that no efficient algorithm can distinguish between the two distributions (aG, bG, abG) and (aG, bG, cG), where $a, b, c \xleftarrow{\$} \mathbb{Z}_{|\mathbb{G}|}$. It is believed that the DDH assumption holds for elliptic curves with prime order over a prime field with large embedding factor [5], specifically DDH holds for the secp256k1 curve, which is used to generate accounts and sign transactions in Bitcoin and Ethereum among other cryptocurrencies.

Although it is hard to decide whether a triplet is a DDH-triplet without knowing the multipliers, one could convince anyone in zero-knowledge that a tuple is indeed a DDH-tuple if possesses the multipliers

The language $\mathcal{L}_{DDH}$ is defined to be the set of all tuples (G, aG, bG, abG) where $G \in \mathbb{G}$ is of order prime q. The Chaum-Pedersen protocol enables a prover $\mathcal{P}$ to prove to a verifier $\mathcal{V}$ that $(G, A, B, C) \in \mathcal{L}_{DDH}$ in zero-knowledge for groups of prime order [8]. The protocol is organized as follows:

1. $\mathcal{V}$: $s \xleftarrow{\$} \mathbb{Z}_q$, then sends $commit(s)$
2. $\mathcal{P}$: $r \xleftarrow{\$} \mathbb{Z}_q$, then sends $y_1 = rG$, $y_2 = rB$.
3. $\mathcal{V}$ opens commitment by sending s
4. $\mathcal{P}$ sends $z = r + as \pmod q$
5. $\mathcal{V}$ checks $zG = y_1 + sA \pmod q \wedge zB = y_2 + sC \pmod q$

Note that in the following a non-interactive version of this protocol will only be considered that can be achieved by applying the Fiat-Shamir heuristic.

2.5 ECDSA with arbitrary generator element

Elliptic Curve Digital Signature Algorithm (ECDSA) is a key component of MixEth. ECDSA is widely deployed in practice, where in most cases signatures are generated and verified with respect to a fixed generator element of the underlying group [9]. Since all generators are equal from a security point of view, a single generator element is usually fixed in order to promote standardization and assists usability.

However, in MixEth, we deploy a somewhat loosened version of ECDSA, where we allow arbitrary generator elements to be used. Such an extension is indeed needed for withdrawing funds from the mixer, because shuffled public keys remain public keys with respect to non-standardized generator elements. Therefore the usial Sig and Vf algorithms for signing and verifying a messages gets an additional parameter G', which is not necessarily the standardized generator element. Key generation algorithm works as usual $(pk, sk) \xleftarrow{\$} \mathsf{KGen}(1^\lambda)$, on the other hand $\sigma \xleftarrow{\$} \mathsf{Sig}(G', sk, m)$ and $0/1 \leftarrow \mathsf{Vf}(G', pk, \sigma, m)$ accept new generators.

In our security proofs we will be relying on the fact that ECDSA is *existentially unforgeable* [9], i.e. no efficient adversary could forge a signature on any given message with non-negligible probability.

2.6 Ethereum

2.7 Ethereum account abstraction

Unfortunately, neither Möbius nor Miximus can be deployed on the present-day Ethereum. When users of the coin mixing contract, either Möbius or Miximus would like to withdraw their funds they can not do this from a fresh address, since it does not hold any ether. Since as of now only the sender of a transaction can pay for the gas fee, users can not withdraw their funds unless they ask someone to fund their fresh address.

Another solution for this problem is the Ethereum Improvement Proposal (EIP) 86 suggested by Nick Johnson and Vitalik Buterin [7]. EIP86 permits receivers of a transaction paying the gas fee. This would certainly enable a functional Möbius and Miximus as well, since the tumbling

contract could pay for the withdrawal transactions' gas fee, eliminating the previous workaround to unlinkably fund freshly mixed addresses. Additionally, EIP86 also allows contracts and accounts to define their own digital signature algorithms. This means that users are no longer required to sign transactions with Elliptic Curve Digital Signature Algorithm (ECDSA). Moreover if EIP86 or something similar is implemented, which is expected in 2019, MixEth is also made viable.

3 Threat model

3.1 Participants and interactions

In a decentralized tumbler, we have 3 distinct entities: the tumbling smart contract, a set of senders and a set of receivers. A sender, whom we will call Alice, sends funds to the receiver, Bob, through the mixer contract in order to break direct links between their public keys. In all the following interactions and algorithms we assume that the public state of the tumbler is implicitly given as input. Interactions of these entities can be summarized as follows:

$tx \xleftarrow{\$} Deposit(sk_A, pk_B)$: The sender runs this algorithm to deposit a predefined amount of ether to the receiver's public key.

$0/1 \leftarrow VerifyDeposit(tx)$: The tumbler contract checks the validity of senders' deposits.

$ProcessDeposit(tx)$: upon receiving a valid deposit transaction, the mixing contract updates its internal state accordingly.

After some period of time no more deposits are allowed to the tumbling contract. Let PK_0 denote the set of public keys after the depositing period to be mixed. Every recipient of the mix is allowed to shuffle the public keys at most once:

$PK_{i+1}, C^*_{i+1} \xleftarrow{\$} Shuffle(PK_i, C^*_i, c_i, \pi_i)$. Let us call C^* as the shuffling accumulated constant, which can be obtained by $C^*_{i+1} = c_i C^*_i$. The shuffling accumulated constant is needed for receivers to audit shuffling and to collect their funds at the end of the final shuffling period. The permutation π and the secret multiplier c_i from the new shuffling accumulated constant should be kept private after shuffling, otherwise it is trivial to track how public keys are shuffled. All the outputs of the $Shuffle$ algorithm are public and written into the tumbling contract.

$0 \vee 1 \leftarrow ChallengeShuffle(PK_i, C^*_i, PK_{i-1}, C^*_{i-1}, pk_B)$: receiver B with public key $pk_B = s_B G$ can challenge an incorrect shuffle at the ith round by giving a Chaum-Pedersen zero-knowledge proof that the following tuple is DDH-tuples: $(C^*_{i-1}, s_B C^*_{i-1}, C^*_i, s_B C^*_i)$. If the proof verifies and $s_B C^*_i \notin PK_i$, while $s_B C^*_{i-1} \in PK_{i-1}$, then the challenge is accepted, otherwise rejected. This proof and checks allow one to be certain that indeed the ith round is the first round in which the corresponding public key to s_B is shuffled incorrectly.

$tx \xleftarrow{\$} Withdraw(sk_B, C^*)$: after the end of the shuffling period users are allowed to withdraw their funds. Note that here withdraw transactions will be signed with a modified version of ECDSA, where not the original generator element G is used as generator rather C^*, the final shuffling accumulated constant.

$0/1 \leftarrow VerifyWithdraw(tx)$: tumbler checks the validity of a recipient's withdrawal transaction.

$ProcessWithdraw(tx)$: upon receiving a valid withdrawal transaction, mixing contract updates its internal state accordingly.

3.2 Security goals

We are aiming to achieve and prove the same notions of security as the ones defined in [12], namely anonymity, availability and theft prevention. We are going to assume that at most $k-2$ recipients are malicious (k is the number of recipients). Otherwise, no meaningful notion of security can be achieved. Furthermore we presume that participants are on-line during the entire course of mixing in order to be able to monitor and potentially challenge any incorrect shuffle. Finally we assume that honest recipients will always exercise their rights to shuffle and they do not disclose any private information used in their shuffles.

In the security definitions and games introduced by [12] adversary $\mathcal{A}$ might have access to the following oracles. CORR enables $\mathcal{A}$ to corrupt a sencer or receiver by learning the secret key of any party l of their choice. Oracle access to AD or AW allow $\mathcal{A}$ to deposit or withdraw respectively from tumbler session j. Furthermore $\mathcal{A}$ might instruct honest senders or receivers to deposit or

withdraw from tumbler session j by using oracles HD and HW. In the following C denotes the set of corrupted parties, while the list of honest deposits and withdrawals are denotes as H_d and H_w respectively The list of contract identifiers associated with distinct sessions is denoted as $tumblers$.

These oracles are formally defined as follows:

AD(tx,j)

$b \leftarrow VerifyDeposit(tumblers[j], tx)$
$if\ (b)\ ProcessDeposit(tumblers[j], tx)$
return b

AW(tx,j)

$b \leftarrow VerifyWithdraw(tumblers[j], tx)$
$if\ (b)\ ProcessWithdraw(tumblers[j], tx)$
return b

CORR(l)

$C = C.push(pk_{B_l})$
return sk_{B_l}

HD(i,j,l)

$tx \xleftarrow{\$} Deposit(sk_{A_i}, pk_{B_l})$
$H_d = H_d.push(tx)$
$ProcessDeposit(tumblers[j], tx)$
return tx

HW(j,l)

$if\ (pk_{B_l} \notin tumblers[j].keys_B)\ return\ \perp$
$tx \xleftarrow{\$} Deposit(sk_{A_i}, pk_{B_l})$
$H_w = H_w.push(j, l, tx)$
$ProcessWithdraw(tumblers[j], tx)$
return tx

From now on, we will denote the number of senders and receivers in the mixer as n, while a specific key in a given session is denoted as $tumbler.keys_B$.

3.2.1 Anonymity

Sender anonymity is achieved if an adversary can not determine to whom honest senders are sending funds, assuming that honest senders' deposits are indistinguishable.

Recipient anonymity is achieved if honest recipients withdrawal transactions are indistinguishable. These notions of anonymity were introduced in [12], which are included here for sake of self-containedness.

Definition 3.1. Define $\mathbf{Adv}^{d-anon}_{mix,\mathcal{A}}(\lambda) = 2\Pr[\mathsf{G}^{d-anon}_{mix,\mathcal{A}}(\lambda)] - 1$ $for\ d \in \{dep, with\}$, where these games are defined as follows:

MAIN $\mathsf{G}^{dep-anon}_{mix,\mathcal{A}}(\lambda)$

$(pk_i, sk_i) \xleftarrow{\$} \mathsf{KGen}(1^\lambda)\ \forall i \in [n]$
$\mathsf{PK}_A \leftarrow \{pk_i\}_{i=1}^n; C, H_d, tumblers \leftarrow \emptyset$
$b \xleftarrow{\$} \{0,1\}$
$(state, j, pk, i_0, i_1) \xleftarrow{\$} \mathcal{A}^{CORR,AD,HD,AW}(1^\lambda, \mathsf{PK}_A)$
$tx \xleftarrow{\$} Deposit(tumblers[j], sk_{A_{l_b}}, pk)$
$b' \xleftarrow{\$} \mathcal{A}^{CORR,AD,HD,AW}(state, tx)$
return $b = b'$

MAIN $\mathsf{G}^{with-anon}_{mix,\mathcal{A}}(\lambda)$

$(pk_i, sk_i) \xleftarrow{\$} \mathsf{KGen}(1^\lambda)\ \forall i \in [n]$
$\mathsf{PK}_B \leftarrow \{pk_i\}_{i=1}^n; C, H_d, tumblers \leftarrow \emptyset$
$b \xleftarrow{\$} \{0,1\}$
$(state, j, pk, l_0, l_1) \xleftarrow{\$} \mathcal{A}^{CORR,AD,HD,AW}(1^\lambda, \mathsf{PK}_B)$
$\mathsf{PK} \leftarrow tumblers[j].keys_B$
$if(pk_{B_{l_0}} \notin \mathsf{PK}) \vee (pk_{B_{l_1}} \notin \mathsf{PK})\ return\ 0$
$tx \xleftarrow{\$} Withdraw(tumblers[j], sk_{B_{l_b}})$
$b' \xleftarrow{\$} \mathcal{A}^{CORR,AD,HD,AW}(state, tx)$
$if(pk_{l_b} \in C\ for\ b \in \{0,1\})\ return\ 0$
$if((j, l_b, \cdot)) \in H_w\ for\ b \in \{0,1\})\ return\ 0$
return $b = b'$

Then the tumbler satisfies sender or recipient anonymity if for all PT adversaries $\mathcal{A}$ there exists a negligible function $\nu(\cdot)$ such that $\mathbf{Adv}^{dep-anon}_{mix,\mathcal{A}}(\lambda) < \nu(\lambda)$ or $\mathbf{Adv}^{with-anon}_{mix,\mathcal{A}}(\lambda) < \nu(\lambda)$ respectively.

3.2.2 Availability

It is essential for a coin mixer to provide availability, meaning that honest recipients can always withdraw their money from the mixer.

3.2.3 Theft prevention

We would like to ensure that neither coins can be withdrawn twice, nor withdrawn by anyone other but the intended recipient.

4 MixEth

4.1 Overview

MixEth is a coin mixing smart contract allowing parties to efficiently tumble coins in a trustless manner on Ethereum.

4.2 Initializing the tumbler

A MixEth contract living on the Ethereum blockchain at $id_{contract}$ address must be initialized with the following parameters:

- amt: denomination of ether to be mixed;
- $senders[]$: list of all the sender addresses;
- $initPubKeys[]$: list of all the recipients' public keys;
- $shuffles[][]$: list of all the shuffles with corresponding shuffling accumulated constants.
- $withdrawn[]$: list of all the shuffled public keys who had already withdrawn their coins.

4.3 Depositing period

Every sender must deposit exactly amt ether to a specific public key. Deposits with incorrect ether value are rejected.

4.4 Shuffling period

After the depositing round, shuffling and challenging rounds are coming after in turns. Each shuffling round is followed by a challenging round when the correctness of the preceding shuffle can be challenged by anyone. If a challenge is accepted, then shuffler's deposit is lost, her shuffle is discarded and shuffling continues from the set of public keys prior to the discarded shuffle. In the course of a shuffle an honest shuffler should raise all the public keys to a secret exponent c and then permute all the transformed public keys. Honest shuffler commits to c by sending back to MixEth the new shuffling accumulated constant and the shuffled public keys.

Shuffle is done off-chain, however the result and the updated shuffling accumulated constant is loaded into the MixEth contract enabling anyone to verify the shuffle's correctness and to continue public key shuffling after the corresponding challenging round.

Procedure 1 Off-chain public key shuffling algorithm for the ith shuffling round

1: $PK_i \leftarrow []$
2: $c \xleftarrow{\$} \mathbb{Z}_n$
3: $C^*_{i-1} \leftarrow read\ from\ MixEth\ contract$
4: $PK_{i-1} \leftarrow read\ from\ MixEth\ contract\ the\ current\ sequence\ of\ shuffled\ public\ keys$
5: $\pi \xleftarrow{\$} S_{|PK_{i-1}|}$
6: **for** $j = 0; j < |PK_{i-1}|; j++$ **do**
7: $\quad PK_i[\pi(j)] = c * PK_{i-1}[j]$
8: **end for**
9: $C^*_i = cC^*_{i-1}$

Output: (PK_i, C^*_i)

4.5 Challenging period

Every participant should check the correctness of incoming shuffles, therefore sufficient time should be provided for each challenging round. These are the actions Bob as a receiver needs to perform to check the correctness of the shuffle at ith round if Bob has secret key s_B. In this case Bob should check whether $s_B C_i^* \in PK_i$ or not. If not, Bob should prove to MixEth that the ith round is indeed the first round, where the shuffled public key corresponding to s_B is compromised. The Chaum-Pedersen proof in the challenge transaction ensures that the integrity of the shuffled public key in round $i-1$st is intact, while shuffled public key is compromised in the ith round.

Procedure 2 On-chain verification algorithm of incoming shuffle challenges

Input$(PK_i, PK_{i-1}, proof_{DDH}(C_{i-1}^*, s_B C_{i-1}^*, C_i^*, s_B C_i^*)$

$b \leftarrow verifyChaumPedersen(proof_{DDH}(C_{i-1}^*, s_B C_{i-1}^*, C_i^*, s_B C_i^*)$
$b^* \leftarrow 0$
if $b \wedge s_B C_{i-1}^* \in PK_{i-1} \wedge s_B C_i^* \notin PK_i$ **then**
    $b^* \leftarrow 1$
else
    $b^* \leftarrow 0$
end if **Output:** b^*

Note that every recipient should perform this check after each shuffling. Noone can check the inclusion and correctness of shuffled public keys for recipients other than themselves. This task is non-outsourcable unless one reveals her own private key, which would obviously lead to loss of funds at the end of the MixEth protocol, since anyone can claim the funds knowing the corresponding secret key.

4.6 Withdrawing

Let C^* be the final shuffling accumulated constant. For a recipient B, whose public key $s_B G \in initPubKeys[]$, in the final shuffle there will be $s_B C^*$. The recipient can prove to MixEth that she knows secret key s_B by using a modified ECDSA, which uses C^* as the generator element instead of the standardized G.

4.7 MixEth pseudocode

```
contract MixEth {
  uint256 amt;
  address[] senders;
  CurvePoint[] initPubKeys;
  Shuffle[] Shuffles;

  struct CurvePoint {
    uint256 x; //x coordinate
    uint256 y; //y coordinate
  }
  struct Shuffle {
    //describes a shuffle
    // contains the shuffled pubKeys and the shuffling accumulated constant
  }

  constructor() public {
   //sets amt
  }

  function uploadShuffle(Shuffle newShuffle) public {
    //needs to be ensured that recipient has not shuffled yet
    Shuffles.append(newShuffle);
  }

  function challengeShuffle(CurvePoint[] challenge, uint256 i) public {
    bool accepted;
    //contract checks the correctness of the i-th shuffle which is stored at
        Shuffles[i]. If challenge accepted malicious shuffler's deposit is slashed.
```

```
28      return accepted;
29    }
30
31    function withdraw() public {
32      //receivers can withdraw funds at most once
33    }
34  }
```

Figure 1: MixEth contract pseudocode

5 Security

This section is going to demonstrate security proofs for the notions of security introduced in Section 3.2.

In this early version of the MixEth paper, we only restrict ourselves to give intuition and informal proofs for certain security properties.

5.1 Recipient anonymity

The withdrawing transaction for recipient B sends funds to the public key $s_B C^*$. This public key does not reveal any links to the original $s_B G$ in case if at least one honest sender shuffled and the DDH assumption holds. Adversary can only distinguish between honest recipients public keys with negligible probability.

5.2 Availability

If an adversary is able to destroy an honest recipient's funds' availability, it implies that adversary either breaks the soundness of the Chaum-Pedersen protocol or successfully launched an eclipse attack against the honest recipient, who can not send any transactions to the Ethereum transaction ledger.

5.3 Theft prevention

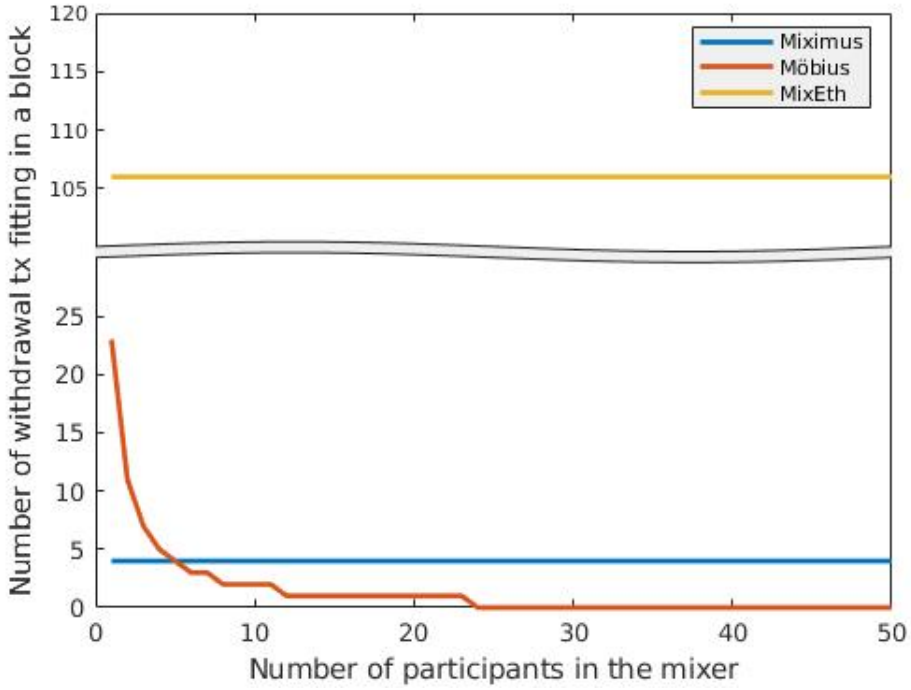
If an adversary is able to steal funds from other users than it would imply that they broke discrete logarithm problem on the secp256k1 curve.

Table 1: Security properties achieved by each coin mixing protocol.

	Anonimity against			Availability		Theft prevention
	outsiders	senders	recipients	sender	tumbler	
Centralized						
Mixcoin [6]	TTP	✗	✓	✓	✗	TTP
Blindcoin [19]	✓	✗	✓	✓	✗	TTP
TumbleBit [10]	✓	✓	✓	✓	✗	✓
Decentralized						
Coinjoin [11]	✓	✗	✓	✗	n.a.	✓
Coinshuffle [18]	✓	✗	✓	✗	n.a.	✓
XIM [4]	✓	✗	✓	✓	n.a.	✓
Möbius [12]	✓	✓	✗	✓	✓	✓
Miximus [1]	✓	✓	✗	✓	✓	✓
MixEth	✓	✓	✗	✓	✓	✓

6 Implementation

We envision two approaches for implementing MixEth. The first implementation of MixEth does not apply state channels, all the transactions are made on-chain. This could lead to unwanted gas costs as the number of corrupted shuffles increases. One of our main motivation with MixEth is to provide an efficient and scalable coin mixing protocol which uses as little blockchain resources as possible. Therefore we also implement and evaluate MixEth applying state channels, namely shuffling and challenging a shuffle occurs off-chain and only deposit and withdrawal transactions happen on-chain.



On of the main bottlenecks of coin mixing protocols is the withdrawal transactions' gas costs. A Miximus withdrawal transaction burns 1,903,305 gas, regardless of the number of participating parties. Since the block gas limit is 8,000,266 as of 2018, October 24 only 4 Miximus withdrawal transactions could fit in one Ethereum block. This is even worse for Möbius, since the gas cost for withdrawing coins from a Möbius mixer linearly increases with the numbers of participants.

6.1 Fully on-chain implementation

6.2 State channel implementation

Table 2: Preliminary implementation gas cost results. Expect further improvements. MixEth++ refers to the implementation which leverages state channels for shuffling and challenging periods

	Deployment	Deposit	Shuffle		Withdraw
			Shuffle upload	Challenge	
Möbius [12]	1,046,027	76,123	0	0	335,714n
Miximus [1]	1,751,378	732,815	0	0	1,903,305
MixEth	TBD	99,254	138,653+40,000n	210,220	113,265
MixEth++	TBD	TBD	0	0	TBD

7 Related work

As Table 3 demonstrates, both Möbius and Miximus require 2 on-chain transactions, while MixEth requires 3. In spite of this seemingly added complexity, we are confident that these 3 (deposit, shuffle, withdraw) transactions consume significantly less gas than those (deposit, verify linkable ring signature/zkSNARK) of Möbius and Miximus.

Table 3: Number of on-chain transactions and off-chain messages required to run a certain coin mixer protocol.

	#Off-chain messages	#Transactions
Centralized		
Mixcoin [6]	2	2
Blindcoin [19]	4	2
TumbleBit [10]	12	4
Decentralized		
Coinjoin [11]	$\mathcal{O}(n^2)$	1
Coinshuffle [18]	$\mathcal{O}(n)$	1
XIM [4]	0	7
Möbius [12]	2	2
Miximus [1]	1	2
MixEth	1	3

8 Extensions and improvements

MixEth is not fully compatible with the current EVM, however it could be deployed with a workaround. A recipient could ask another party or service to send a signed transaction including a signature which uses the modified version of ECDSA, where the generator element is the shuffling accumulated constant. MixEth could check this signature and send out funds to a fresh Ethereum address given in the withdraw transaction.

9 Conclusion

10 Future work

We are going to implement and deploy MixEth on Ethereum and evaluate its performance. We are also going to give formal security proofs for the claimed security properties. An important and crucial research question is whether it is possible to design and implement an efficient and trustless Ethereum-based coin mixer with strong security guarantees which do not rely on any workaround and upcoming EIP implementation and is ready to be deployed on present-day Ethereum.

11 Acknowledgements

We would like to thank barryWhiteHat and Dmitry Khovratovich for the insightful comments and discussions.

References

[1] barryWhiteHat. Miximus. https://github.com/barryWhiteHat/miximus, 2018.

[2] barryWhiteHat. Miximus gas costs. https://www.reddit.com/r/ethereum/comments/8ss53z/miximus_zksnark_based_anonymous_transactions_is/, 2018.

[3] Stephanie Bayer and Jens Groth. Efficient zero-knowledge argument for correctness of a shuffle. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 263–280. Springer, 2012.

[4] George Bissias, A Pinar Ozisik, Brian N Levine, and Marc Liberatore. Sybil-resistant mixing for bitcoin. In *Proceedings of the 13th Workshop on Privacy in the Electronic Society*, pages 149–158. ACM, 2014.

[5] Dan Boneh. The decision diffie-hellman problem. In *International Algorithmic Number Theory Symposium*, pages 48–63. Springer, 1998.

[6] 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.

[7] Vitalik Buterin and Nick Johnson. Eip86: Account abstraction. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-86.md, 2017.

[8] David Chaum and Torben Pryds Pedersen. Wallet databases with observers. In *Annual International Cryptology Conference*, pages 89–105. Springer, 1992.

[9] Manuel Fersch, Eike Kiltz, and Bertram Poettering. On the provable security of (ec) dsa signatures. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1651–1662. ACM, 2016.

[10] 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.

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

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

[13] Sarah Meiklejohn, Marjori Pomarole, Grant Jordan, Kirill Levchenko, Damon McCoy, Geoffrey M Voelker, and Stefan Savage. A fistful of bitcoins: characterizing payments among men with no names. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 127–140. ACM, 2013.

[14] Pedro Moreno-Sanchez, Muhammad Bilal Zafar, and Aniket Kate. Listening to whispers of ripple: Linking wallets and deanonymizing transactions in the ripple network. *Proceedings on Privacy Enhancing Technologies*, 2016(4):436–453, 2016.

[15] Malte Moser, Rainer Bohme, and Dominic Breuker. An inquiry into money laundering tools in the bitcoin ecosystem. In *eCrime Researchers Summit (eCRS), 2013*, pages 1–14. IEEE, 2013.

[16] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[17] C Andrew Neff. A verifiable secret shuffle and its application to e-voting. In *Proceedings of the 8th ACM conference on Computer and Communications Security*, pages 116–125. ACM, 2001.

[18] 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.

[19] 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.

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

Appendices

A Proof of Anonymity

Hereby we show that if there exists an adversary $\mathcal{A}$ who is able to break withdrawal anonymity defined in 3.2.1, then there exists another adversary $\mathcal{B}$ who is able to break the DDH assumption.

Let us assume that the challenge to the DDH-adversary $\mathcal{B}$ is of the form (s_0G, C^*, s_bC^*). If original keys are (s_0G, s_1G) and shuffled keys are $\mathsf{PK}_B = (s_0C^*, s_1C^*)$, then $\mathcal{B}$ forwards PK_B to $\mathcal{A}$. Then a withdrawal transaction occurs from s_bC^*. After polynomial-time $\mathcal{A}$ outputs b' and $\mathcal{B}$ will outpu the very same bit. It is straightforward to see that $\mathcal{B}$ wins the DDH-game if and only if $\mathcal{A}$ wins the $\mathsf{G}^{with,anon}_{mix,\mathcal{A}}$ security game. Therefore, by the assumption that recipient anonymity does not hold we have that

$$\mathsf{non\text{-}negl}(\lambda) = \Pr[\mathsf{G}^{with,anon}_{mix,\mathcal{A}}] = \Pr[\mathsf{DDH}_{\mathcal{B}}],$$

which contradicts to the DDH assumption.

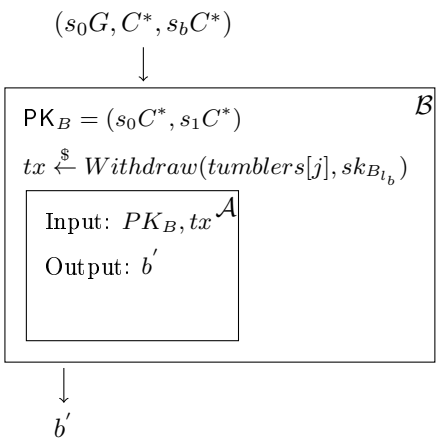


Figure 1: An illustration for the reduction of withdrawal anonymity to the DDH assumption