MixEth: efficient, trustless coin mixing service for Ethereum

István András Seres[1], Dániel A. Nagy[1], Chris Buckland[2], and Péter Burcsi[1]

[1]Department of Computer Algebra, Eötvös Loránd University
[2]King's College London

March 30, 2019

Abstract

Coin mixing is a prevalent privacy-enhancing technology for cryptocurrency users. In this paper, we present MixEth, which is a trustless coin mixing service for Turing-complete blockchains. MixEth does not rely on a trusted setup and is more efficient than any proposed trustless coin tumbler. It requires only 3 on-chain transactions at most per user and 1 off-chain message. It achieves strong notions of anonymity and is able to resist denial-of-service attacks. Furthermore the underlying protocol can also be used to efficiently shuffle ballots, ciphertexts in a trustless and decentralized manner.

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

1 Introduction

Bitcoin [21] 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 [18, 20, 19]. 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 cannot be linked to them.

Several coin mixing protocols were proposed in the literature both centralized [7, 27, 14] and decentralized [15, 26, 2, 17, 5]. 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 cannot be guaranteed [7, 27]. 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 [28].

There is no doubt that there exists a tremendous need for privacy overlays for Ethereum as new tools for transaction deanonymization are getting developed and used [9]. This need of the Ethereum community for privacy was spectacularly embodied in September, 2017 when for several days 68% of all the transaction volume was controlled by a centralized coin mixing service [23].

The two major techniques to provide decentralized mixing services for Ethereum are Möbius, a ring-signature-based solution [17] and Miximus, a zkSNARK-based proposal [2]. 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 [3]. As the Ethereum network is congested, ie. blocks were full during 2018[1],

[1]https://etherscan.io/chart/gasused

we argue that it is essential for the network scalability to aim to create protocols and applications that burn as few gas as possible.

Even though (Ethereum) users and transactions can be deanonymized already on the network layer [22], we consider network anonymity an orthogonal problem to that of anonymity on the transaction ledger.

Our contributions. In this paper, we present a trustless and efficient mixing protocol for Turing-complete blockchains. This protocol can be used to shuffle ciphertexts, ballots or public keys. The protocol have many use cases: shuffling ElGamal-ciphertexts, decentralized mixnets, e-voting.

To show the practicality of the protocol, we introduce MixEth, a privacy-enhancing protocol and a practical tool for Ethereum, to overcome the above mentioned efficiency issues of Ethereum-based coin-mixers while retaining strong notions of anonymity, mixer availability and theft prevention already achieved by previous proposals [17, 2]. MixEth requires as few off-chain messages and on-chain transactions as Möbius and Miximus, meanwhile it burns significantly less gas.

The intuition behind MixEth is to apply Neff's verifiable shuffles [24] in the context of coin mixing. Participants of the tumbler shuffle their public keys in order to break links between sender and recipient public keys. The key insight is that verifying on-chain a Neff proof about the correctness of a shuffle would be too gas-inefficient, therefore we require receivers to be online to issue fraud proofs, if and only if an incorrect shuffle was made. Whenever recipients consider that enough shuffling was executed, they can withdraw their funds from the mixer.

We also implement MixEth in a state channel to leverage the scalability and instant finality of off-chain scaling solutions. Furthermore, the MixEth protocol could be used in any state channel application to mix funds before going back on-chain.

2 Background

In this section we introduce the building bocks required to create our mixing protocol and MixEth, the trustless coin tumbler.

2.1 Notations

In most cases if it is possible we will stick to the notations used in [17] 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 they needs to generate a secret key $s \xleftarrow{\$} \mathbb{Z}_n$, where n is the order of secp256k1 over a finite prime 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 [28].

2.3 Verifiable shuffle

Neff introduced the notion of verifiable shuffle [24]. 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 multiplied by the same secret multiplier 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 without disclosing the secret multiplier.

Neff's mathematical construct is extremely powerful, since it only relies on the intractability of the Decision 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 [4].

So far verifiable shuffles were only applied in voting schemes, we argue that they are useful in trustless coin mixers as well. The key insight in order to be able to apply verifiable shuffles in a decentralized, computational-resource-constrained environment, for instance Ethereum smart contracts, is to dismiss the proof generation for the correctness of the shuffle, rather we request users to give more succinct proofs for the incorrectness of the shuffle, if applicable.

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 [6], 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 one 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 [10]. 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 [13]. 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 usual 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* [13], i.e. no efficient adversary could forge a signature on any given message with non-negligible probability.

Note that although Ethereum does not support natively the verification of ECDSA signatures with respect to arbitrary generators, however it can easily be implemented in a smart contract.

2.6 Ethereum

Ethereum is a cryptocurrency built on top of a blockchain. Similarly to Bitcoin, network participants broadcast transactions in a peer-to-peer network, where transactions are bundled together into blocks that are appended to a public ledger called blockchain. Only those specific nodes can append new blocks to the blockchain who previously solved a difficult cryptographic puzzle. The state of the system consists of the state of different accounts populating it.

In Ethereum currently there are two types of accounts. The first account type is called *externally owned account*. It owns an ECDSA keypair controlled by its user. Private keys are used to sign transactions. On the other hand there are *contract accounts*, often smart contracts, that additionally have persistent storage and contract's code. Both of the account types have Ether balances, which is the native currency of the Ethereum network. Ether is denominated in wei, where 1 $ETH = 10^{18}$ *wei*.

Transactions can alter the system's state by either creating a contract account or by calling to an existing account. Transactions to externally owned accounts can only transfere Ether, while transactions to contract accounts can additionally execute the code associated with them. Codes are executed in a quasi-Turing complete execution environment, called Ethereum Virtual Machine (EVM).

EVM is quasi-Turing complete, since smart contract's code cannot run indefinitely due to the so called *gas mechanism*. In every transaction the sender needs to pay upfront for the execution of the contract's code. The computational complexity of a transaction is measured in gas, which can be bought for Ether on a price set by the transaction originator, so called gas price. Therefore the transaction fee is the gas cost multiplied by the gas price. One needs to specify a gas limit, meaning that they do not allow their transaction to burn more gas than the limit. If a transaction during execution runs out of gas, then all the state changes are reverted, while the transaction fee is paid to the miner. If there is gas left after successful execution, transaction originator is reimbursed. Additionally there exists a block gas limit, which limits the number of computational steps fitting in one block. Currently the block gas limit is cca. 8,000,000 gas. Naturally users of Ethereum are very much incentivized to minimize the gas cost of their transactions in order to spend as little as possible on transactions fees. Small gas costs are also crucial from a scalability point of view, since the less gas burnt for each transaction, the more transaction can fit into a single block.

2.6.1 State channels

Public blockchain's decentralization comes at an inherent cost in regard to scalability, since currently each full node verifies the full state of the public ledger. Often times Bitcoin's ($\sim$ 7 transactions per second (tps)) or Ethereum's ($\sim$ 15 tps) throughput is compared to that of Visa's ($\sim$ 45,000 tps). Blockchains' scalability issues became an increasingly growing problem as more and more users adopted the technology. One remarkable example was the launch of the Cryptokitties game

in 2017, when the Ethereum network was congested for a few hours due to the enormous popularity of the game. Therefore several solutions were proposed to alleviate aforementioned scalability issues.

One of the major class of these techniques is called off-chain solutions. The insight of these proposals is that it is not needed to conduct all transactions on the blockchain, since participants could lock funds on-chain and afterwards securely issue transactions off-chain, for instance micropayments, with high degree of security and finality. Participants only need to get back on-chain if there is a dispute about what happened precisely off-chain or they would like to lock up funds and redeem them on-chain. The first implementation of this idea was a payment channel network for Bitcoin called Lightning network [25]. The advantage of the Lightning network is that participants can issue several payments without sending transactions to the blockchain and paying the sometimes costly transaction fees. Furthermore users are guaranteed to have instant finality instead of waiting several blocks to confirm their payments.

State channels are the more general form of payment channels, they can be used not only for payments, but for any arbitrary state updates on a blockchain, like changes inside a smart contract. State channels were first described in detail by Jeff Coleman et al.[11]. Since then several other frameworks for generalised state channels were elaborated [12, 16]. Recently a case study of the Battleship game was published by Patrick McCorry, Chris Buckland et al. to evaluate how state channels could contribute in scaling blockchain-based applications [16].

Later in this paper, in Section 6.2 we argue that MixEth can be made more scalable by implementing shuffling in a state channel.

2.6.2 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 cannot 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 cannot 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 [8]. 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. The list of contract identifiers associated with distinct sessions is denoted as $tumblers$. 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:

$tx \xleftarrow{\$} Deposit(tumblers, sk_A, pk_B)$: 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.

Let PK_0 denote the set of public keys after the depositing period to be mixed. Furthermore let us set $C_0^* = G$, the standard generator point of the secp256k1 curve. Anyone is allowed to shuffle the public keys at most once by paying some deposit to the tumbler contract:

$PK_{i+1}, C_{i+1}^* \xleftarrow{\$} Shuffle(PK_i, C_i^*, c_i, \pi_i)$. Let us call C_i^* 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.

$tx \xleftarrow{\$} WithdrawShufflingDeposit(sk_B)$: after a challenging period a shuffler can withdraw their shuffling deposit from the tumbler contract.

$0 \vee 1 \leftarrow VerifyWitdhrawShufflingDeposit(pk_B)$: if there was no successful challenges against the shuffler, i.e. their deposit is not slashed, they can withdraw their shuffling deposit.

$tx \xleftarrow{\$} Withdraw(sk_B, C_{final}^*)$: 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_{final}^*, 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 [17], namely anonymity, availability and theft prevention. These notions of anonymity, availability and theft prevention were introduced in [17], which are included in this section for sake of self-containedness.

We are going to assume that at most $n-2$ recipients are malicious (n 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 [17] adversary $\mathcal{A}$ might have access to the following oracles. CORR enables $\mathcal{A}$ to corrupt a sender 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.

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 a specific key in a given session as as $tumbler.keys_B$.

3.2.1 Anonymity

Sender anonymity is achieved if an adversary cannot 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.

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:

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

$(pk_i, sk_i) \xleftarrow{\$} \mathsf{KGen}(1^\lambda)\ \forall i \in [n]$

$\mathsf{PK}_A \leftarrow \{pk_i\}_{i=1}^n; C, H_d, tumblers \leftarrow \emptyset$

$b \xleftarrow{\$} \{0,1\}$

$(state, j, pk, i_0, i_1) \xleftarrow{\$} \mathcal{A}^{CORR,AD,HD,AW}(1^\lambda, \mathsf{PK}_A)$

$tx \xleftarrow{\$} Deposit(tumblers[j], sk_{A_{l_b}}, pk)$

$b' \xleftarrow{\$} \mathcal{A}^{CORR,AD,HD,AW}(state, tx)$

return $b = b'$

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

$(pk_i, sk_i) \xleftarrow{\$} \mathsf{KGen}(1^\lambda)\ \forall i \in [n]$

$\mathsf{PK}_B \leftarrow \{pk_i\}_{i=1}^n; C, H_d, tumblers \leftarrow \emptyset$

$b \xleftarrow{\$} \{0,1\}$

$(state, j, pk, l_0, l_1) \xleftarrow{\$} \mathcal{A}^{CORR,AD,HD,AW}(1^\lambda, \mathsf{PK}_B)$

$\mathsf{PK} \leftarrow tumblers[j].keys_B$

$if(pk_{B_{l_0}} \notin \mathsf{PK}) \vee (pk_{B_{l_1}} \notin \mathsf{PK})\ return\ 0$

$tx \xleftarrow{\$} Withdraw(tumblers[j], sk_{B_{l_b}})$

$b' \xleftarrow{\$} \mathcal{A}^{CORR,AD,HD,AW}(state, tx)$

$if(pk_{l_b} \in C\ for\ b \in \{0,1\})\ return\ 0$

$if((j, l_b, \cdot)) \in H_w\ for\ b \in \{0,1\})\ return\ 0$

return $b = b'$

Then the tumbler satisfies sender or recipient anonymity if for all PPT 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, even if senders and all but one recipients are compromised.

Adversary $\mathcal{A}$ wins the availability security game if they manage to get the tumbler into a state where honest recipient cannot withdraw their funds.

Definition 3.2. Define $\mathbf{Adv}^{avail}_{mix,\mathcal{A}}(\lambda) = \Pr[\mathsf{G}^{avail}_{mix,\mathcal{A}}(\lambda)]$, where the game is defined as follows:

$\underline{\text{MAIN } \mathsf{G}^{avail}_{mix,\mathcal{A}}(\lambda)}$

$(pk_i, sk_i) \xleftarrow{\$} \mathsf{KGen}(1^\lambda)\ \forall i \in [n]$

$\mathsf{PK}_B \leftarrow \{pk_i\}_{i=1}^n; C, H_w \leftarrow \emptyset$

$(l, j) \xleftarrow{\$} \mathcal{A}^{CORR,AD,HW,AW}(1^\lambda, \mathsf{PK}_B)$

$b \leftarrow \mathsf{VerifyWithdraw}(\mathsf{tumblers}[j], \mathsf{Withdraw}(sk_l))$

$if((pk_l \in C)\ \vee\ ((j, l, \cdot) \in H_w))\ return\ 0$

return $(b = 0) \wedge (pk_l \in tumblers[j].keys_B)$

Then the tumbler satisfies availability if for all PPT adversaries $\mathcal{A}$ there exists a negligible function $\nu(\cdot)$ such that $\mathbf{Adv}^{avail}_{mix,\mathcal{A}}(\lambda) < \nu(\lambda)$.

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. Theft prevention is defined formally as follows:

Definition 3.3. Define $\mathbf{Adv}^{theft}_{mix,\mathcal{A}}(\lambda) = \Pr[\mathsf{G}^{theft}_{mix,\mathcal{A}}(\lambda)]$, where the game is defined as follows:

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

$(pk_i, sk_i) \xleftarrow{\$} \mathsf{KGen}(1^\lambda)\ \forall i \in [n]$

$\mathsf{PK}_B \leftarrow \{pk_i\}_{i=1}^n; C, H_w, \mathsf{contract} \leftarrow \emptyset$

$(\mathsf{tx}, j) \xleftarrow{\$} \mathcal{A}^{CORR,AD,AW,HW}(1^\lambda, \mathsf{PK}_B)$

$if(\mathsf{tumblers}[j].\mathsf{keys}_B \not\subset \mathsf{PK}_B \setminus C)\ return\ 0$

return $\mathsf{VerifyWithdraw}(\mathsf{tumblers}[j], \mathsf{tx})$

Then the tumbler satisfies theft prevention if for all PPT adversaries $\mathcal{A}$ there exists a negligible function $\nu(\cdot)$ such that $\mathbf{Adv}^{theft}_{mix,\mathcal{A}}(\lambda) < \nu(\lambda)$.

4 MixEth

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

4.1 Initializing the tumbler and depositing period

A MixEth contract living on the Ethereum blockchain at $id_{contract}$ address must be initialized with the *amt* parameter, which denotes the denomination of ether to be mixed. Every sender must deposit exactly *amt* ether to a specific public key. Deposits with incorrect ether value or invalid public key are rejected. Public keys in subsequent deposit transactions are written into the $initPubKeys[]$ array.

4.2 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 and given to the challenger, their 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 multiply all the public keys with a secret multiplier 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.

Computing the 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)

4.3 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.

In Procedure 2 $verifyChaumPedersen(proof_{DDH})$ denotes a deterministic polynomial-time algorithm which verifies the correctness of a Chaum-Pedersen zero-knowledge proof. The algorithm outputs 1 if the proof is verified, otherwise 0.

4.4 Withdrawing

Let C_{final}^* 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 $s_B C_{final}^*$. The recipient can prove to MixEth that she knows secret key s_B by signing their public key using a modified ECDSA, which uses C_{final}^* as the generator element instead of the standardized G.

4.5 MixEth formal specification

Deposit(sk_A, pk_B)
return FormTx$(sk_A, id_{\text{contract}}, amt, pk_B)$

VerifyDeposit(tx)
$if\ (tx[amt] \neq id_{\text{contract}}[amt])\ return\ 0$
$if\ (pk_B \notin \mathbb{G})\ return\ 0$
return VerifyTx(tx)

ProcessDeposit(tx)
add addr(tx[pk]) to initPubKeys[]

Shuffle(PK, C^*)
$c \xleftarrow{\$} \mathbb{Z}_{|\mathbb{G}|}$
$C^{'*} \leftarrow cC^*$
$PK^{'} = \{pk_i^{'} | pk_i^{'} = c * pk_i \forall i \in [1..n]\}$
return $(PK^{'}, C^{'*})$

UploadShuffle(sk_A, PK, C^*)
return FormTx$(sk_A, id_{contract}, (PK, C^*))$

Challenge$(proof_{ChP}(C^*, sC^*, C'^*, sC'^*), PK, PK^{'})$
$if\ (\neg Verify(proof_{ChP}))\ return\ 0$
$if\ (sC^* \notin PK)\ return\ 0$
$delete(C^{'}, PK^{'})$
$slashDeposit(shuffler(C^{'}, PK^{'}))$
return 1

WithdrawShufflingDeposit(sk_A)
return FormTx$(sk_A, id_{contract}, 0)$

VerifyWithdrawShufflingDeposit(pk_A)
return $shufflingDepositSlashed(pk_A)$

Withdraw(s, C^*)

$\sigma \leftarrow \mathsf{Sig}(C^*, s, sC^*||msg.sender)$
return $\mathsf{FormTx}(s, C^*, id_{\mathsf{contract},0}, \sigma)$

VerifyWithdraw(tx)

$if\ (sC^* \notin id_{\mathsf{contract}}[lastShuffle])\ return\ 0$
$if\ (\neg\mathsf{Vf}(C^*, sC^*, \sigma, sC^*||tx.sender))\ return\ 0$
return VerifyTx(tx)

ProcessWithdraw(tx))

$delete\ sC^*\ from\ id_{\mathsf{contract}}[lastShuffle]$
$tx' \xleftarrow{\$} \mathsf{FormTx}(id_{contract}, tx.sender, amt)$

5 Security

This section provides informal ideas to the security proofs for the notions of security introduced in Section 3.2. Formal reductions are enclosed in the Appendices.

5.1 Recipient anonymity

The withdrawing transaction for recipient B sends funds to the public key s_BC^*. This public key does not reveal any links to the original s_BG 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. See reduction proof in Appendix A.

5.2 Availability

If an adversary is able to destroy an honest recipient's funds' availability, it implies that adversary $\mathcal{A}$ either breaks the completeness of the Chaum-Pedersen protocol or successfully launched an eclipse attack against the honest recipient, who cannot send any transactions to honest Ethereum peers.

5.3 Theft prevention

If an adversary is able to steal funds from other users than it would imply that they managed to create a valid message/signature, (m, σ) pair for the final shuffled public key of an honest recipient without having access to the secret key of the honest recipient. This contradicts to the assumption that ECDSA is existentially unforgeable. Reduction proof is enclosed in Appendix C.

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

	Anonimity against			Availability		Theft prevention
	outsiders	senders	recipients	sender	tumbler	
Centralized						
Mixcoin [7]	TTP	✗	✓	✓	✗	TTP
Blindcoin [27]	✓	✗	✓	✓	✗	TTP
TumbleBit [14]	✓	✓	✓	✓	✗	✓
Decentralized						
Coinjoin [15]	✓	✗	✓	✗	n.a.	✓
Coinshuffle [26]	✓	✗	✓	✗	n.a.	✓
XIM [5]	✓	✗	✓	✓	n.a.	✓
Möbius [17]	✓	✓	✗	✓	✓	✗
Miximus [2]	✓	✗	✗	✓	✓	✓
MixEth	✓	✓	✗	✓	✓	✓

6 Implementation

We implemented MixEth with two different approaches. 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, storage and gas, 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. Both of the implementations allow users to mix Ether or other ERC20-compatible, a popular Ethereum token standard, tokens.

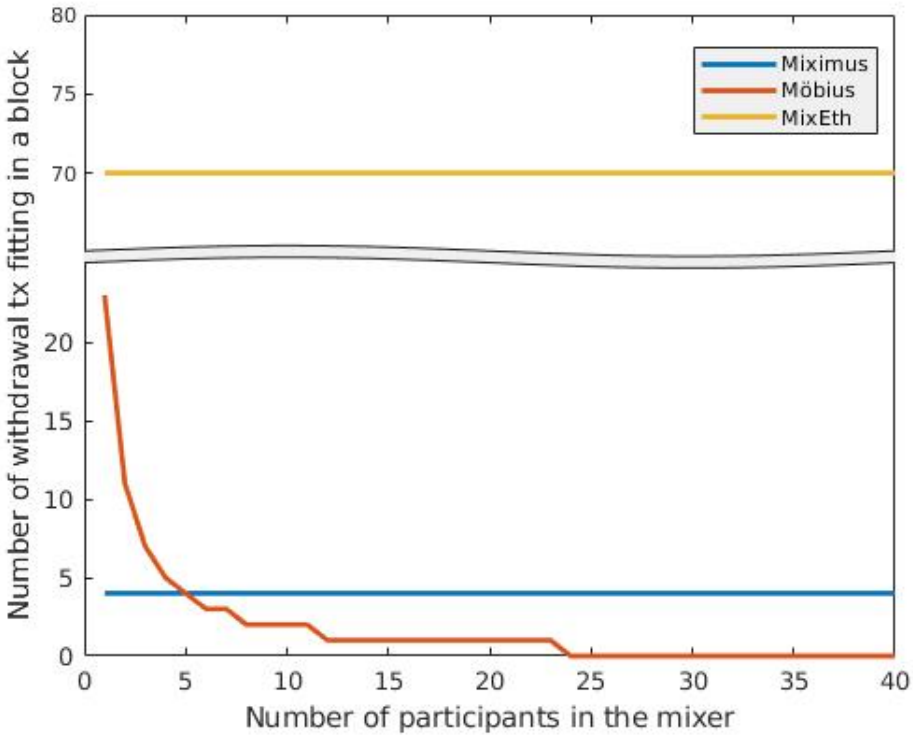

Figure 1: Gas complexity of withdraw transactions for various trustless Ethereum coin mixers

One 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 contract linearly increases with the numbers of participants.

Although MixEth is more gas-efficient than Möbius or Miximus, it incurs a higher time-complexity, ie. recipients need to expect longer delays for funds to arrive since each challenging period lasts a few blocks of time. Furthermore MixEth requires users to be online during the course of mixing, in some scenarios this might be a demanding requirement.

All MixEth smart contracts were written in the Solidity language, which is currently the dominant language for developing Ethereum smart contracts. MixEth contracts are compiled using the latest version of the Solidity compiler in order to take advantage of the latest language features such as the possibility to return dynamic-sized arrays from contract functions. Note that these implementations are only proof-of-concept implementations, expect further improvements. All MixEth contracts are available online[2].

[2]https://github.com/seresistvanandras/MixEth

6.1 Fully on-chain implementation

Conceivably mixers would like to minimize off-chain coordination, therefore in our first implementation of the MixEth protocol, we assumed that all transactions will take place on-chain. There is only a single off-chain message from receiver to sender, where receiver delivers their public key to the sender. The rest of the protocol happens entirely on-chain.

On-chain storage is extremely expensive: it requires 20,000 gas to store a 256-bit number, however if a particular storage slot is already taken and one wants to overwrite it with a non-zero element then storing only consumes 5000 gas. To minimize on-chain storage costs, only the last two list of shuffled public keys are stored in the MixEth contract's permanent storage. Note that storing only the latest list of shuffled public keys would not be enough, since honest receivers could not prove to the contract that their shuffled public key is compromised unless also the last but first list of shuffled keys is also available for the contract to check the Chaum-Pedersen proof against. Such a storage structure implies that after uploading the new list of shuffled public keys, a challenging period should proceed in order to let receivers check the correctness of the shuffle and whether their shuffled public key is stored in the smart contract, see Figure 2. Furthermore we also allow senders to shuffle and deposit new public keys at the same time, meaning that only 3 on-chain transactions (shuffle, withdraw mixed coins and withdraw shuffling deposit) are sufficient to complete the protocol. We left it as a future work to determine the optimal ratio of mixed coins and shuffling deposits in order to incentivise correct shuffling.

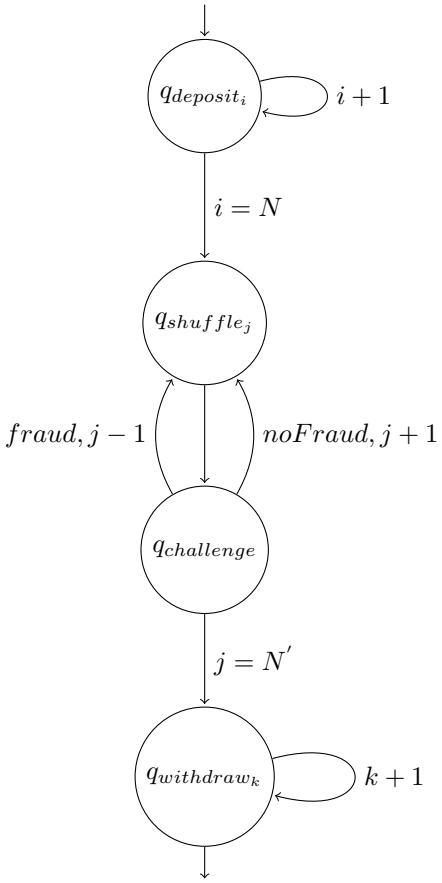

Figure 2: Shuffle-and-challenge paradigm implemented for the fully on-chain version of MixEth

A great advantage of the fully on-chain version of MixEth is that it allows dynamic anonymity sets. One could potentially deposit funds to the contract and shuffle public keys and leave funds in the mixing contract for indefinite amount of time. As soon as the anonymity set is large enough a receiver could withdraw their assets. The whole process is represented as a finite automata as shown in Figure 2. A receiver in a MixEth contract with N senders could withdraw their funds after $N^{'}$ shuffling rounds, where $N^{'}$ is arbitrary. This dynamic nature of the contract could even lead to a single monolithic MixEth contract instead of having multiple MixEth contracts with significantly fragmented anonymity sets. A single MixEth contract is able to support the mixing of ether and ERC-20 compatible tokens as well. However note that the gas complexity of shuffling transactions grows linearly in the number of participants, therefore the fully on-chain implementation is not capable to support extremely large anonymity sets with participants more than cca. 800.

Also note that the on-chain cost of a shuffle transaction could be amortized among participants. Specifically if there is a group of senders and/or receivers who trust each other they could collectively charge any of them to shuffle once. Since they trust their peer, all the rest of the group does not need to shuffle anymore. Later non-shufflers could reimburse the only shuffler for their services either on-chain or off-chain.

6.2 State channel implementation

We have also adapted MixEth to operate within a state channel. We wrote the implementation within the guidelines of the Counterfactual framework [11]. This allowed us to delegate the processes of setup, liveness disputes and finalisation to the framework so that we could focus on adapting the application logic. Unlike the on-chain implementation the state channel implementation requires that the set of participants be agreed upon up-front. In state channels each update to the state needs to be signed by all other participants, this means that state channel applications are inherently at least $\mathcal{O}(n)$. To co-ordinate these off-chain updates the Counterfactual framework enforces that all applications be turn based, introducing a turn taker for each turn who may propose a new state. The original MixEth implementation was not turn based so we have adapted the application to this constraint, an example of this adaptation is the challenge round. In the on-chain implementation a time period is allowed during which any participant may challenge, we have adapted this by proceeding turn-based through the participants offering each the chance to either challenge or pass. In the case of a breakdown in cooperation in the channel, a liveness fault, it has been shown that all operations succeeding the cooperation breakdown must proceed on chain[16] or be abandoned at some financial cost specified by the application, meaning that if every shuffle were to be succeeded by a challenge round each participant would be forced, by threat this lost deposit, to make an on-chain transaction after each shuffle, incurring $\mathcal{O}(n)$ on chain operations. To mitigate this we removed the challenge after each round and instead introduced a challenge round that takes place after all shuffles have completed, during this round any of the preceding shuffles may be challenged.

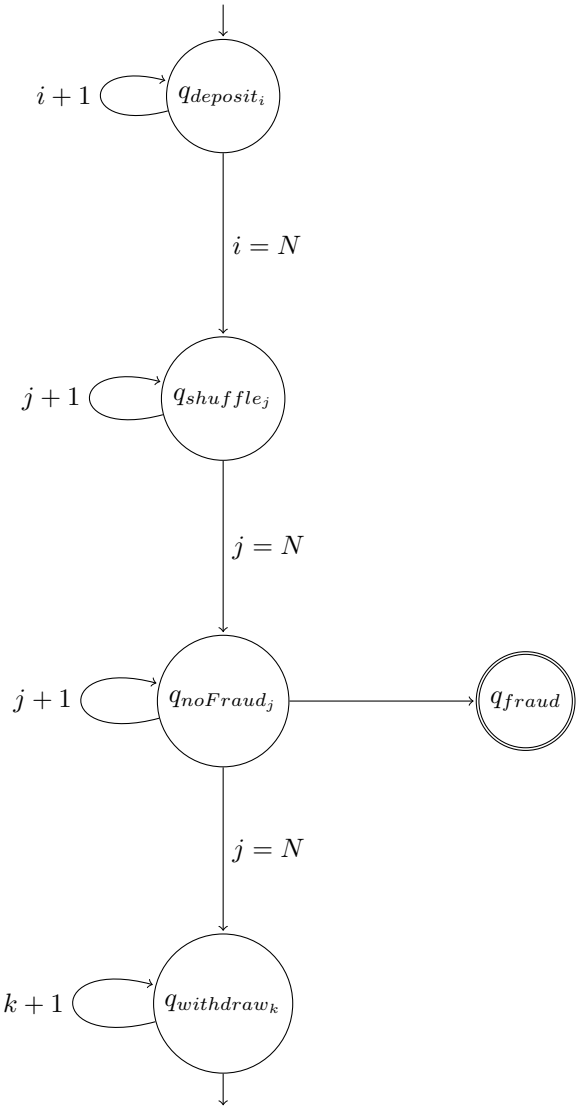

Figure 3: Shuffle-then-challenge paradigm implemented for the channelised version of MixEth

Given these adaptations the application proceeds as follows, all participants including senders, shufflers and receivers, deposit funds in a mutli-signature wallet compatible with the Counterfactual framework, they then follow the installation protocols specified by the framework to install the adapted MixEth logic. Afterwards each participant signs a transaction that transfers an equal amount to each withdrawer from the multi-sig, dependent on correct execution of the channelised MixEth application logic. This application logic proceeds as follows: each sender names a public key of a shuffler as in the deposit stage of the on-chain application, then each shuffler takes it in turn to shuffle. After all shuffles have taken place each withdrawer is given a turn to either declare fraud or no-fraud on any shuffle round. Finally each withdrawer then provides proof of ownership by submitting a valid signature on the modified ECDSA scheme. If any of these steps does not occur, or does not occur correctly, the protocol aborts and the conditional transfer does not occur. In this case the perpetrator loses a deposit, either through fraud proof or through failure to take their turn when state is published on-chain. A further modification would be to distribute the slashed deposit to each of the other participants, compensating them for their lost time and the gas costs associated with proving the fault of the other party. Following this protocol the on-chain transactions are now reduced to: one transaction from each participant to deposit funds into the multi-sig, and a set of transactions that send funds from the multi-sig to each of the withdrawers and deposits back to each of the other participants. The modified procedure of the MixEth protocol is illustrated as a finite automate in Figure 3. Note that all three Ns are the same. In a channelised MixEth contract there are exactly

N depositors, N shufflers and N receivers.

Table 2: Proof-of-concept implementation gas cost results. Expect further improvements. MixEthChannel refers to the implementation which leverages state channels for shuffling and challenging periods

	Deployment	Deposit	Shuffle		Withdraw
			Shuffle upload	Challenge	
Möbius [17]	1,046,027	76,123	0	0	335,714n
Miximus [2]	1,751,378	732,815	0	0	1,903,305
MixEth	5,395,945	99,254	$138{,}653 + 10{,}000n$	227,563	113,265
MixEthChannel	672,276	21,000	0	0	26,749

7 Related work

Möbius was the very first trustless coin mixer designed for Ethereum[17]. Authors of Möbius provided formal definitions of various notions of security such as anonymity, theft prevention and mixer availability. These properties could be used to evaluate and compare existing and future proposals from a security perspective. Möbius is a ring-signature-based trustless coin mixer with minimal on-chain transaction complexity: users of Möbius just need to create a deposit and a withdraw transaction. However the gas cost of the withdrawal transaction increases linearly in the number of receivers, which limits the size of possible anonymity sets. No more than 24 reciever could use Möbius with current cca.8,000,000 block gas limit. If more people tried to use the mixer funds would be stucked in the mixer contract, since the gas costs of withdrawal transactions would be greater than the block gas limit. Another, maybe even more severe problem with Möbius' implementation is that withdrawal transactions can be frontrunned by anyone in the network, meaning that anyone could steal receivers' funds from the Möbius smart contract. When an honest receiver broadcasts a Möbius withdrawal transaction, other peers in the network might just create another withdrawal transaction with the very same valid linkable ring signature but with a considerably larger gas price and with their own address as recipient. Miners will act rationally and pick the transaction with the higher gas price, therefore theft-prevention in Möbius' implementation is not guaranteed.

Miximus is a zkSNARK-based mixer for Ethereum[2]. It uses zkSNARKs to conceal the mapping between depositors and recipients. A depositor creates a leaf in a Merkle-tree. A depositor needs to exchange the preimage of the leaf with the recipient. Later, a recipient could prove to the Miximus contract that they know one of the preimages of a certain, undisclosed leaf. So called nullifiers enable recipients to withdraw funds once and only once. The gas costs of depositing and withdrawing funds from a Miximus mixer is independent of the number of participants. However there are disadvantages of this approach; Miximus only provides anonymity against outsiders, since if Alice funds to Bob via Miximus, Alice will know when Bob made the withdrawal transaction. Another, more severe limitation of Miximus is the trusted setup required for generating the proving key for the zkSNARK. If this trusted setup is compromised, the deployer of the contract, who generated the proving key could potentially steal funds from the mixer. Although, this issue could be amended somehow via a multi-party computation (MPC) further increasing the off-chain communication complexitiy of Miximus.

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

Table 3: Number of on-chain transactions and off-chain messages per a single participant required to run a certain coin mixer protocol. Note that in case of Miximus if one wants to avoid the trusted setup for the zkSNARK, then they need to perform a secure multi-party computation protocol to trust-minimize the proving key generation.

	#Off-chain messages	#Transactions
Centralized		
Mixcoin [7]	2	2
Blindcoin [27]	4	2
TumbleBit [14]	12	4
Decentralized		
Coinjoin [15]	$\mathcal{O}(n^2)$	1
Coinshuffle [26]	$\mathcal{O}(n)$	1
XIM [5]	0	7
Möbius [17]	2	2
Miximus [2]	1+MPC	2
MixEth	1	3
MixEthChannel	$\mathcal{O}(n)$	2

7.1 Additional related work

Recently, in November 26, 2018 a new privacy-enhancing protocol and tool was released, called Aztec protocol [1]. Aztec allows its users to convert any type of existing digital asset on top Ethereum into Aztec notes. Later on Aztec notes can be traded in a join-split style confidential transaction, namely the value of the traded Aztec notes are encrypted and not visible to outsiders. A join-split style transaction can have multiple inputs and outputs. Aztec applies zero-knowledge proofs and range proofs in order to achieve that the sum of inputs traded in a confidential transaction is greater or equal than the sum of outputs. The gas cost of a confidential Aztec transaction is $219{,}000+121{,}000n+41{,}000m$, where n and m are the numbers of inputs and outputs respectively. Although Aztec is not a coin mixer, it can be used not only to provide confidential transaction, but also to increase anonymity. On the downside Aztec uses a trusted setup and considerably more expensive than MixEth to provide the same level of k-anonymity, however it is able to conceal transaction volumes. Most likely a combination of MixEth and Aztec would yield the most gas-efficient solution to achieve k-anonym and confidential transactions. For instance on top Aztec one could use MixEth to shuffle Aztec notes.

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.

In the current design of MixEth if sender, Alice and receiver, Bob would like to use the mixer several times, Bob needs to share his receiver address in a secure communication channel with Alice as many times as he would like to receive payments. This communication overhead could be overcome by applying stealth addresses, where Bob needs to share once his public master key with Alice in order to receive arbirtary number of payments from her.

9 Acknowledgements

Chris Buckland is supported by an Ethereum Foundation scaling grant and an Ethereum Community Fund grant. We would like to thank Liam Horne for helping with the state channel implementation, Barry WhiteHat, Dmitry Khovratovich and Sina Mahmoodi for the insightful comments and discussions.

References

[1] Aztec. The aztec protocol. https://github.com/AztecProtocol/AZTEC, 2018.

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

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

[4] 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.

[5] 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.

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

[7] 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.

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

[9] Wren Chan and Aspen Olmsted. Ethereum transaction graph analysis. In *2017 12th International Conference for Internet Technology and Secured Transactions (ICITST)*, pages 498–500. IEEE, 2017.

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

[11] Jeff Coleman, Liam Horne, and Li Xuanji. Counterfactual: Generalized state channels, 2018.

[12] Stefan Dziembowski, Lisa Eckey, Sebastian Faust, and Daniel Malinowski. Perun: Virtual payment channels over cryptographic currencies. Technical report, IACR Cryptology ePrint Archive, 2017: 635, 2017.

[13] 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.

[14] 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.

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

[16] Patrick McCorry, Chris Buckland, Surya Bakshi, Karl Wüst, and Andrew Miller. You sank my battleship! a case study to evaluate state channels as a scaling solution for cryptocurrencies.

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

[18] 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.

[19] 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.

[20] 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.

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

[22] Nchinda Nchinda. Exploring pseudonimity on ethereum. https://media.consensys.net/exploring-pseudonimity-on-ethereum-dda257019eb4, 2016.

[23] Serge Nedashkovsky. Huge ethereum mixer. https://blog.cyber.fund/huge-ethereum-mixer-6cf98680ee6c, 2017.

[24] 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.

[25] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments. 2016.

[26] 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.

[27] 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.

[28] 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.

Towards contradiction let us assume that the recipient anonymity does not hold. Let us assume that the challenge to the DDH-adversary $\mathcal{B}$ is of the form (sG, cG, c_0G). In a DDH-game the adversary's goal is to decide whether c_0G is a random group element or it equals to scG. At the end of the DDH-game adversary outputs 1 if $c_0G = scG$ and 0 otherwise. Adversary $\mathcal{B}$ generates uniformly random public key c_1G and invokes $\mathcal{A}$ with the set $\mathsf{PK}_B = (c_0C^*, c_1C^*)$, then $\mathcal{B}$ forwards PK_B to $\mathcal{A}$. Then a withdrawal transaction occurs from c_bG. After polynomial-time $\mathcal{A}$ outputs $b^{'}$ and $\mathcal{B}$ will output $\neg b^{'}$. If $\mathcal{A}$ outputs 0 and $b = b^{'}$, this signals to $\mathcal{B}$ that c_0G might potentially be of the form $scG = c_0G$ i.e. it is a DDH tuple, therefore $\mathcal{B}$ outputs 1. In all the other cases $\mathcal{B}$ outputs a random bit. Therefore we have that $\mathcal{B}$ has an adventage in their DDH-game if and only if $\mathcal{A}$ wins their $\mathsf{G}^{with,anon}_{mix,\mathcal{A}}$ security game. Since we assumed that recipient anonymity does not hold we have that

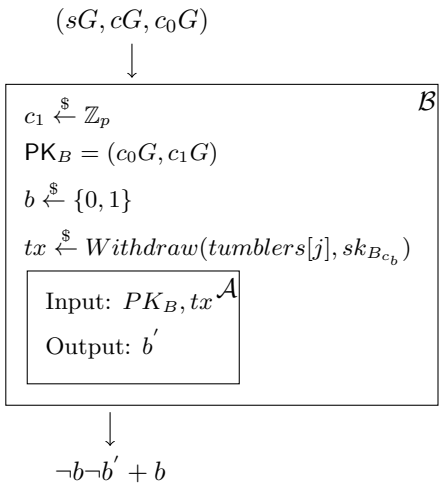

Figure 4: An illustration for the reduction of withdrawal anonymity to the DDH assumption

$$\Pr[\mathsf{DDH}_{\mathcal{B}}] = \frac{1}{2} + \frac{1}{2} * \Pr[\mathsf{G}^{with,anon}_{mix,\mathcal{A}}] = \frac{1}{2} + \frac{1}{2} * \mathsf{non\text{-}negl}(\lambda),$$

which contradicts to the DDH assumption.

B Proof of Availability

The only possibility for an adversary to threaten the availability of funds for an honest receiver if they create an incorrect shuffle, where honest receiver's shuffled public key is compromised. Since the Chaum-Pedersen zero-knowledge protocol is complete, an honest receiver is alway able to create a Chaum-Pedersen proof, which demonstrates to the contract that their shuffled public key is compromised. Therefore we have for any PPT $\mathcal{A}$ and $\forall\lambda \in \mathbb{N}$ that,

$$\Pr[\mathsf{G}^{avail}_{mix,\mathcal{A}}] = 0 < negl(\lambda).$$

C Proof of Theft Prevention

Towards contradiction we assume that there exists an adversary $\mathcal{A}$, who is able to break the theft prevention property introduced in Section 3.2.3 with non-negligible probability. Using such an adversary as a subroutine we could create another efficient adversary $\mathcal{B}$ who is able to break the existential unforgeability of ECDSA. The input of the forgeability game is the security parameter which is forwarded to $\mathcal{A}$ along with n randomly generated public keys. By assumption $\mathcal{A}$ outputs with non-negligible probability valid withdraw transaction belonging to one of the public keys in the mixer. A valid withdraw transaction is a (tx, σ_{tx}) pair, where σ_{tx} is a valid signature on transaction tx. Adversary $\mathcal{B}$ outputs the withdraw transaction and the ECDSA signature on it. $\mathcal{B}$ wins the forgeability game if and only if $\mathcal{A}$ wins their $\mathsf{G}^{theft}_{mix,\mathcal{A}}$ game:

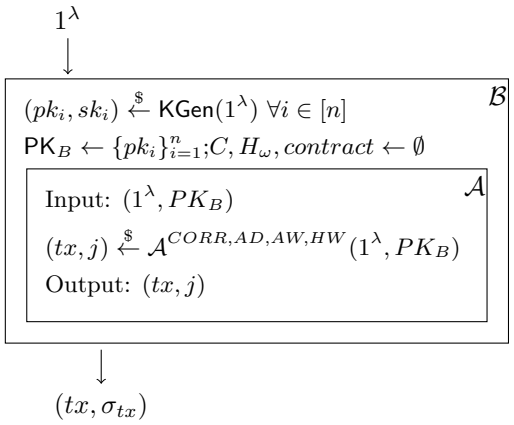

Figure 5: An illustration for the reduction of theft prevention to the existentially unforgeability of ECDSA.

$$\Pr[\mathsf{Forge}_{ECDSA,\mathcal{B}}] = \Pr[\mathsf{G}^{theft}_{mix,\mathcal{A}}] = \frac{1}{\lambda^\alpha},$$

for some fixed α. This contradicts to the assumption that ECDSA is existentially unforgeable.