On the design of Zeth transaction relays

Antoine Rondelet* and Duncan Tebbs*

*Clearmatics, UK

2020-12-21

Abstract

Privacy preserving protocols suffer from the need to pay transaction fees on blockchain systems. While such fees constitute a sound economic barrier to a wide class of Denial of Service attacks, they also represent impediments to the design of state transitions with anonymous initiators. In this paper, we navigate the design space for "`Zeth` cryptographic relays". These enable `Zeth` users to carry out `Zeth` payments anonymously by relying on an extra party – a relay – to settle and execute state transitions on the blockchain.

Keywords— Privacy, Ethereum, Zeth, Relays, Sender Anonymity, Digital Cash

Contents

1 Preliminaries **3**
- 1.1 Prerequisites . . . 3
- 1.2 Notation . . . 3
- 1.3 Terminology . . . 4
- 1.4 Introduction . . . 4
  - 1.4.1 Turning “front-runners” into relays . . . 5
  - 1.4.2 Relay incentives and risks . . . 5
  - 1.4.3 Assumptions . . . 6

2 Relays with Proof of Permission **8**
- 2.1 Protocol Overview . . . 9
- 2.2 Protocol . . . 10
  - 2.2.1 Relay Request and Permission Data . . . 10
  - 2.2.2 User Operations . . . 11
  - 2.2.3 Relay Operations . . . 12
  - 2.2.4 **RelayExec** Contract . . . 15

3 Relays with Private Fees **18**
- 3.1 Introduction . . . 18
- 3.2 Protocol overview . . . 18
  - 3.2.1 Relay-originated mix transactions . . . 19
- 3.3 Limitations and extensions to the protocol . . . 19
  - 3.3.1 Limitation of output notes . . . 19
  - 3.3.2 Increasing `JSOUT` . . . 19
  - 3.3.3 Support for `Ether` output . . . 20
  - 3.3.4 Fees as `Ether` or `Zeth` notes . . . 21

A Relays with Stake **22**
- A.1 Introduction . . . 22
- A.2 Protocol overview . . . 23
  - A.2.1 Stake contract . . . 23
  - A.2.2 Relay . . . 24
  - A.2.3 User . . . 25

A.2.4 Reclaiming stake 25
A.3 Remarks 26

B Network structure **28**
B.1 Binding requests to relays 28
B.1.1 Background 28
B.1.2 Emulating “free” relay requests 28
B.2 Unicast vs broadcast networks 29
B.3 Network anonymity 30

Chapter 1

Preliminaries

1.1 Prerequisites

This document assumes familiarity with `Ethereum` and `Zeth`. It does not, in any way, aim to replace the Ethereum yellow paper [Woo19] or the Zeth specifications [Cle20]. The reader is strongly advised to read about `Ethereum` and `Zeth` before delving into this document.

1.2 Notation

Unless stated otherwise, this document follows the notation of the `Zeth` protocol specifications [Cle20]. We note in particular the following notations (some of which originate from [Cle20]), used throughout the document.

$\widetilde{\textbf{Mixer}}$ A deployed instance of the Zeth contract.

$\mathcal{U}$ A user, with identity $\mathcal{U}_{\mathcal{E}}$ on the `Ethereum` network, and/or `Zeth` identity $\mathcal{U}_{\mathcal{Z}}$.

$\mathcal{R}$ An entity operating a relay, with identity $\mathcal{R}_{\mathcal{E}}$ on the `Ethereum` network, and/or `Zeth` identity $\mathcal{R}_{\mathcal{Z}}$.

$\widetilde{\textbf{RelayExec}}$ A contract deployed on the blockchain acting as a proxy to $\widetilde{\textbf{Mixer}}$ (removing the need for trust between users and relays).

$fee_{\mathcal{R}}$ The fee of the relay service $\mathcal{R}$.

$\mathsf{genCallTx}$ Algorithm which creates, signs and broadcasts a transaction to call a contract entry point with specific parameters. That is, given a contract method call of the form $\widetilde{\textbf{Contract}}.\mathsf{method}(\mathsf{param}_1, \ldots)$, a secret key sk for some `Ethereum` address, a gas price $gasP$ and gas limit $gasL$, the algorithm $\mathsf{genCallTx}(\widetilde{\textbf{Contract}}.\mathsf{method}(\mathsf{param}_1, \ldots), sk, gasP, gasL)$ creates a signed transaction that performs the given contract call.

broadcastTx Algorithm that accepts a signed `Ethereum` transaction and broadcasts it to the network, returning the transaction ID. The caller (in this document, the relay) can use the transaction ID to monitor the asynchronous completion of the transaction. The exact details will depend on the relay implementation, but once the transaction is complete, the relay can retrieve its result and update any internal state.

1.3 Terminology

The key words `MUST`, `MUST NOT`, `SHOULD`, `SHOULD NOT`, `MAY`, and `RECOMMENDED` in this document are to be interpreted as described in [Bra97] when they appear in `ALL CAPS`. These words may also appear in this document in lower case as plain English words, absent their normative meanings.

1.4 Introduction

The `Zeth` protocol allows users to carry out privacy-preserving state transactions on "smart-contract enabled blockchains" such as `Ethereum` or `Autonity`[1]. Like all `Ethereum` transactions, `Zeth` transactions require a fee to be paid. This is inherited from the base protocol, which uses transaction fees as a security mechanism against Denial of Service (DoS) attacks.

As pointed out in the `Zeth` paper [RZ19], the need to pay transaction fees represents a challenge for designers of privacy preserving protocols, since transaction originators must carry out `Zeth` contract calls from a funded `Ethereum` address (which in turn must have been funded by other user(s) on the system[2], and for which the "controlling user identity" must be known by at least one network member). As such, while `Zeth` provides strong privacy guarantees (recipient anonymity, private payment amount etc.), sender anonymity remains hard to achieve.

This document proposes some designs for "cryptographic relays" and investigates the space of tradeoffs for both users and relay operators. The protocols enable `Ethereum` peer-to-peer (P2P) nodes to act as *relays*, receiving requests (off-chain), and signing and broadcasting transactions (which incorporate these requests) on behalf of `Zeth` users. In exchange for this service, relays receive some fee from the original users.

As described below, relay fees are of paramount importance in establishing a sound incentive structure, which in turn is necessary for the overall robustness of the system. The primary goal of this study on "cryptographic relays" is to achieve `Zeth` sender anonymity on blockchain systems, and the proposals in this document suggest multiple ways in which relay fees can be paid while maintaining this anonymity. Furthermore, under additional network assumptions (e.g. namely that `Zeth` users and relay nodes communicate via an Anonymous Communication (AC) network, e.g. [PHE$^+$17]), `Zeth`

[1] `https://github.com/clearmatics/autonity`

[2] Unless the user is a miner.

users can make use of relay nodes without revealing any identifying information to the relay. See Appendix B.3 for further discussion.

1.4.1 Turning "front-runners" into relays

As noted in [DGK$^+$19] and [lsa20] (among others), on blockchains such as `Ethereum`, so-called "front-runners" actively seek out transactions that are profitable for the sender and attempt to replace them with modified versions, in order to steal the profit from the original sender. Front-running strategies leverage the mempool ordering policy adopted by miners. Namely, they set higher gas prices in order to overtake the targeted transactions.

With the proliferation of "bots"[3] inspecting the mempool and front-running profitable transactions (see e.g. [RK20]), it becomes key for "layer 2"-protocol designers to design state transitions that are secure against such replay attacks. As presented in [Cle20, Section 2.3], the `Zeth` protocol prevents "front-running"/"replay" attacks by design (see derivation of *hsig* and *dataToBeSigned*).

While "front-runners" present a threat to users of Decentralized Applications (DApps), they can potentially be leveraged to act as transaction relays [lsa20]. Since front-runners examine the mempool, looking for profitable transactions to overtake (by extracting the transaction payload, creating a new transaction, signing it and broadcasting it on the network with a higher gas price), a user may exploit this behavior by voluntarily broadcasting a transaction with low gas price on the network, in the hope that a front-runner will replay/overtake it. By doing so, the user may thus trigger a state transition on the blockchain without paying the associated transaction fees. Nevertheless, "front-runners" should be modelled as rational agents, meaning that such transactions must be profitable to them. As such, for users to leverage "front-runners" as "relays" in practice, transactions must be crafted such that "front runners" receive a fee in exchange for replacing them.

Finally, in order for a user's transactions to be added to a miner's transaction pool, it is necessary for the transaction to pass the "initial tests of intrinsic validity" [Woo19, Section 6] (see also transaction pool implementation in Geth[4]). This means that users who wish to have their transactions "front-run"/"relayed" must hold a funded `Ethereum` account. This may not be desirable in all scenarios (especially in settings where sender anonymity is a primary motivation).

1.4.2 Relay incentives and risks

Besides the potential profitability of "front-running", mentioned above, relaying transactions that haven't been added to a miner's mempool is inherently risky, and sound incentive structures must reward such risk appropriately.

Firstly, it must be noted that relay nodes may be vulnerable to DoS attacks. In such attacks, malicious clients "flood" targeted relays with an overwhelming stream of

[3] see, for instance, `https://github.com/Uniswap/uniswap-interface/issues/248`

[4] `https://github.com/ethereum/go-ethereum/blob/master/core/tx_pool.go#L578-L583`

transactions. While such attacks may be mitigated by relay operators using existing network monitoring techniques (e.g. packet filtering, rate limiting etc.), it is also important for relay operators to assess the profitability of the transactions that they relay to the blockchain. In fact, running a relay may quickly become a "money drain" if the cost of operating the relay service (i.e. infrastucture costs, transaction fees etc.) outweighs the relay fees received. As such, it is necessary for relays to have an efficient way to gauge the on-chain cost and profitability of a transaction. (Carrying out this operation may also exacerbate the DoS vector on relays, since a flood of maliciously crafted transactions – such as transactions that take a long time to execute, but fail to release any funds – may cause a relay node to spend all of its resources on transaction verification in return for no income[5].) Additionally, it is worth remembering that "front runners" and "relayers" may in turn be front-run by other competitors. As a consequence, allocating non-negligible computation resources to "assessing the profitability" of transactions represents a risk – other "front runners" may overtake the relay's (verified) transaction to avoid carrying out this verification work locally.

1.4.3 Assumptions

Based on the remarks given in the previous sections, we make the following assumptions in the rest of the document.

Transactions to be relayed are immune to front-running. We assume that all relayed transactions are inputs to the `Zeth` contract Mix function. As such, the inputs prevent front-running by construction.

Transactions to be relayed target specific relays. As mentioned above, `Zeth` is designed to avoid "front-running"/"malleability" attacks. To achieve this, several parameters (e.g. *dataToBeSigned*) are derived using the address of the `Ethereum` user that must send the transaction. As such, we assume that users *choose* a relay service to process their request. Note that, as discussed in Appendix B.1.2, users are free to target multiple relay services by creating multiple requests, but the underlying assumption is there exists a market of competing "relays", from which users can select the service that best suits their needs. For example, different relays may offer different trade-offs between settlement latency and fees, while others may offer "aggregation services" (see, e.g. [Ron20])

Relays are "discoverable". Discovery of relay nodes by users may be achieved through several possible mechanisms. For example, relay nodes may publish their IP address, current fees and `Ethereum` address (some of which may potentially be published on-chain), allowing users to discover their services. Overall, we assume that relay operators take the necessary steps to be "discoverable" by users.

[5] Additional security measures may alleviate such issues. For example, using properly crafted verification thresholds, discarding transactions that take too long to be verified. Note however that such mechanisms "specialize" a relay into relaying only certain classes of transactions. Again, proper tradeoffs need to be adopted depending on the use-cases and threat model.

In the remainder of this document, we propose a set of protocols to relay `Zeth` transactions, each with their own trade-offs and specific goals.

Note

Importantly, it is worth keeping in mind that in most scenarios, sound "relay economics" will imply that for a given state transition, relay fees are greater than the on-chain fees normally paid by blockchain users. Hence, we stress that using relays should not be seen as a "cheap way" to transact on a blockchain, but rather as a way to achieve otherwise impossible objectives on the system (e.g. to achieve sender anonymity).

Chapter 2

Relays with Proof of Permission

In this section, we propose a protocol with the aim of preserving the anonymity of `Zeth` transaction senders. This allows a user $\mathcal{U}_{\mathcal{Z}}$ to interact (anonymously) with a `Zeth` deployment to either "pour" the value of some of his *ZethNotes* into new ones (i.e. carry out a private payment), or withdraw some value *vout* from **Mixer** to a newly created `Ethereum` address $Addr_{new}$.

We assume that $\mathcal{U}_{\mathcal{Z}}$ is willing to pay a fee to achieve this anonymity, and that at least one party $\mathcal{R}_{\mathcal{E}}$ is willing to act as a "relay" in return for this fee.

By providing a mechanism to carry out private withdrawals of `Zeth` *funds to a newly generated* `Ethereum` *address, we allow* `Ethereum` *users to manipulate* Ether *"privately" in future transactions - either "plain EOA-to-EOA transactions" or smart-contract calls.*

We list below the set of characteristics that are desired in this setting.

From the user's perspective

- The user must be able to leverage relays to anonymously carry out a `Zeth` state transition on-chain. This includes, anonymous private transfer (i.e. "pouring" the value of existing *ZethNotes* to new ones), and anonymous withdrawals to a newly created `Ethereum` account.
- Not only must the user be anonymous with regard to the blockchain network, but he must also remain anonymous to the relay[1] for the mechanism to be robust against malicious/compromised relays. The user must not be required to reveal any identifying information, including any pre-funded `Ethereum` addresses.
- The user must be guaranteed (up to some negligible probability) that the relay will only call the Mix function, on-chain, with the *correct* inputs (i.e. the relay may not execute any state transition on behalf of the user that the user did not request).

[1]Further assumptions need to be made about the underlying network. The user must be able to communicate with the relay without revealing any network-layer identifying information.

From the relay's perspective

- The relay must be guaranteed (up to some negligible probability) that he will receive the agreed upon fee in exchange for carrying out his role in the protocol (and therefore he will not incur costs such as transaction fees for no reward)[2].

2.1 Protocol Overview

We start by assuming that a `Zeth` user $\mathcal{U}_{\mathcal{Z}}$ has chosen a relay service $\mathcal{R}$ (with `Ethereum` account $\mathcal{R}_{\mathcal{E}}$) which relays transactions for a fee $fee_{\mathcal{R}}$. We further assume that $\mathcal{U}_{\mathcal{Z}}$ knows the address of the $\widetilde{\mathbf{RelayExec}}$ contract.

The protocol consists of the following steps:

Step 1 (User creates the Mix **parameters).** $\mathcal{U}_{\mathcal{Z}}$ creates the Mix parameters *mixParams* that spend her note(s), including the public output value *vout*. *mixParams* are generated such that only $\widetilde{\mathbf{RelayExec}}$ can successfully use them. (This is achieved by leveraging the property of Mix parameters, described in [Cle20, Sections 2.4, 2.5], which restricts the `Ethereum` address of the caller, possibly a contract[3].)

Step 2 (User generates a *proof-of-relay-permission*). With *mixParams* properly created, $\mathcal{U}_{\mathcal{Z}}$ generates a *proof-of-relay permission* $\pi_{\mathcal{R}}^{(mixParams)}$ (described in further detail below) for *mixParams*. This proves that the owner of the `Zeth` notes to be spent by *mixParams* agrees that $\mathcal{R}_{\mathcal{E}}$ may relay the Mix parameters via $\widetilde{\mathbf{RelayExec}}$ for a fee $fee_{\mathcal{R}}$. $\mathcal{U}_{\mathcal{Z}}$ also specifies the address *outAddr* to which the remaining balance $vout - fee_{\mathcal{R}}$ (if any) should be sent. (In general, *outAddr* is expected to be a newly generated address $Addr_{new}$).

Step 3 (User sends parameters to the relay). $\mathcal{U}_{\mathcal{Z}}$ sends a "relay request" *req* to the chosen relay $\mathcal{R}$, containing *mixParams*, $\pi_{\mathcal{R}}^{(mixParams)}$ and other data such as *outAddr*. Note that, as long as *outAddr* is a newly generated address (with no history on the blockchain), this request contains no information that identifies $\mathcal{U}_{\mathcal{Z}}$ as the originator. $\mathcal{U}_{\mathcal{Z}}$ is also expected to leverage anonymising mechanisms to avoid revealing any identifying information at the transport level.

Step 4 (Relayer verifies and broadcasts the received request). Upon receipt of *req*, the relay performs a set of checks to gain confidence that *mixParams* and $\pi_{\mathcal{R}}^{(mixParams)}$ are valid, and indeed grant the relay fee to $\mathcal{R}_{\mathcal{E}}$. (Note that the relay has some scope to choose the extent of such checks, trading off the cost of checking against the risk of losing money by broadcasting an invalid transaction.) If the relay is satisfied that *req* is valid, he signs (using the $\mathcal{R}_{\mathcal{E}}$ identity) and broadcasts a transaction that calls $\widetilde{\mathbf{RelayExec}}$.

[2] This would question the profitability of operating a relay node and would jeopardize the "relay network" as well as the viability of the "relaying activity".

[3] This is currently the mechanism used to prevent front-runners from claiming *vout*.

Step 5 (The intermediary contract checks all parameters and executes the `Zeth` mixer). The $\widetilde{\textbf{RelayExec}}$ contract acts as an intermediary, trusted by both users and relays. It first checks $\pi_{\mathcal{R}}^{(mixParams)}$ to ensure that the caller ($\mathcal{R}_{\mathcal{E}}$) has indeed been granted permission to use the *mixParams* in exchange for $fee_{\mathcal{R}}$. $\widetilde{\textbf{RelayExec}}$ then calls $\widetilde{\textbf{Mixer}}$ with parameters *mixParams* and checks that the call succeeds. The transaction is aborted if any of these checks fail.

Step6 (The intermediary contract distributes the value ***vout*****).** If the Mix call is successful, $\widetilde{\textbf{RelayExec}}$ now holds the *vout* from $\widetilde{\textbf{Mixer}}$. From this, it pays $fee_{\mathcal{R}}$ to $\mathcal{R}_{\mathcal{E}}.Addr$ and the remainder $vout - fee_{\mathcal{R}}$ to $outAddr$.

Remark 1. *Using* $\widetilde{\textbf{RelayExec}}$ *to distribute the fee and fund the user-specified outAddr address, gives* $\mathcal{U}_{\mathcal{Z}}$ *confidence that outAddr will receive the correct output for the agreed fee. Further,* $\mathcal{R}_{\mathcal{E}}$ *can be sure that no other relay can use the same set of* Mix *parameters and forge a request from* $\mathcal{U}_{\mathcal{Z}}$ *to receive the relay fee.*

2.2 Protocol

Below we give further details of the protocol outlined above, enabling anonymous `Zeth` transfers via relayed transactions. The protocol leverages specific characteristics of the `Zeth` protocol design (some of which were described above). In particular, it builds on the fact that Mix parameters can be generated without owning an `Ethereum` account (see derivation of *hsig* and related discussion [Cle20, Remark A.2.2]), and that Mix parameters are "bound" to the address of the `Ethereum` account that must call the $\widetilde{\textbf{Mixer}}$ contract (see derivation of *dataToBeSigned* [Cle20, Section 2.3]).

2.2.1 Relay Request and Permission Data

We use `MixInputDType` and related datatypes from [Cle20, Section 2.1], and define the following new data type to represent a relay request with proof of relay permission.

Definition 1. *The datatype* `RelayRequest` *is defined as:*

Field	Description	Value
mixParams	Parameters to the Mix call	`MixInputDType`
outAddr	`Ethereum` address credited with the funds withdrawn from **Mixer**	$\mathbb{B}^{\texttt{ADDRLEN}}$
relayAddr	`Ethereum` address allowed to relay *mixParams* (and perceive *fee*)	$\mathbb{B}^{\texttt{ADDRLEN}}$
fee	Fee to pay (out of *vout*) to the authorized relay	$\mathbb{N}_{\texttt{ETHWORDLEN}}$
permission	Signature proving the authenticity of the request	`SigOtsDType`

Table 2.1: `RelayRequest` type

The *permission* attribute is used to indicate that the user has given permission for the relay to forward the specific Mix call. We reuse the signature key $mixParams.otsvk$ for the scheme $\mathsf{SigSch}_{\mathsf{OT\text{-}SIG}}$, used to create $mixParams.otssig$ (see [Cle20, Section 2.3]), since only the author of the Mix call parameters can generate valid signatures.

TODO

Tighten the security requirements of the signature scheme used. For now, this proposal uses $\mathsf{SigSch}_{\mathsf{OT\text{-}SIG}}$ to create a second signature with the same private key. However, $\mathsf{SigSch}_{\mathsf{OT\text{-}SIG}}$ is "one-time". Additionally, the security claim on the nested signature is only that it is UF-CMA (see specs), relying on the fact that the wrapping signature scheme (signing the transaction object) is SUF-CMA. Here however, since the relay request does not contain a signed blockchain transaction object, an adversary may be able to maul the signature and pass the UF-CMA game. More consideration of the threat model and the security requirements is required here.

As described below, the user must sign the attributes *relayAddr*, *fee* and *outAddr* for the signature to be considered valid.

2.2.2 User Operations

We assume that user $\mathcal{U}_{\mathcal{Z}}$ has decided to use $\mathcal{R}$ (controlling `Ethereum` account $\mathcal{R}_{\mathcal{E}}$) to relay her `Zeth` transaction in order to either withdraw some value *vout* to `Ethereum` address *outAddr* and/or carry out a private transfer. We further assume that $\mathcal{U}_{\mathcal{Z}}$ agrees to pay a fee $fee_{\mathcal{R}}$ to $\mathcal{R}_{\mathcal{E}}$ in order to achieve this. Here $fee_{\mathcal{R}}$ is the fee that $\mathcal{R}$ is willing to accept in exchange for relaying a single Mix call.[4].

$\mathcal{U}_{\mathcal{Z}}$ executes the following steps:

[4]This fee should be strictly greater than the gas cost of Mix in order for the relay to be profitable (more refined profitability forecasts would internalize the infrastructure operational costs to adjust the fee). Additionally, the relay may adjust and republish $fee_{\mathcal{R}}$ in light of gas price fluctuations on the blockchain, other changes to cost and risk, or to compete with other relays.

1. Create a valid $\mathit{mixParams} \in$ `MixInputDType`, where:

   - $\mathit{mixParams}$ spends previously unspent *ZethNotes* owned by $\mathcal{U}_{\mathcal{Z}}$, with a public output of *vout*.
   - $\mathit{mixParams.otssig}$ is created using the address of $\widetilde{\textbf{RelayExec}}$, as described in [Cle20, Section 2.3]
   - The one-time signing key $sk_{\mathsf{OT\text{-}SIG}}$, used to create $\mathit{mixParams.otssig}$, can be (securely) extracted for use in the following step.

2. Use the signing key $sk_{\mathsf{OT\text{-}SIG}}$ to create a proof of relay permission:

$$\begin{aligned}
&\mathit{data}_{\mathcal{R}} \leftarrow \mathsf{encode}(\mathcal{R}_{\mathcal{E}}.\mathit{Addr}) \parallel \mathsf{encode}(\mathit{fee}_{\mathcal{R}}) \parallel \mathsf{encode}(\mathit{outAddr}) \\
&\pi_{\mathcal{R}}^{(\mathit{mixParams})} \leftarrow \mathsf{SigSch}_{\mathsf{OT\text{-}SIG}}.\mathsf{Sig}(sk_{\mathsf{OT\text{-}SIG}}, \mathsf{CRH}^{\mathsf{ots}}(\mathit{data}_{\mathcal{R}}))
\end{aligned}$$

3. Create a relay request $\mathit{req} \in$ `RelayRequest`:

$$\begin{aligned}
&\mathit{req} \leftarrow \{ \\
&\quad \mathit{mixParams} : \mathit{mixParams}, \\
&\quad \mathit{outAddr} : \mathit{outAddr}, \\
&\quad \mathit{relayAddr} : \mathit{relayEthAccount.Addr}, \\
&\quad \mathit{fee} : \mathit{fee}_{\mathcal{R}}, \\
&\quad \mathit{permission} : \pi_{\mathcal{R}}^{(\mathit{mixParams})} \\
&\}
\end{aligned}$$

Remark 2. *If the user simply wants to leverage a relay to carry out a private* `Zeth` *transfer (without withdrawing funds to a newly created address outAddr),* $\mathcal{U}_{\mathcal{Z}}$ *can set* $\mathit{outAddr} \leftarrow$ `0x0` *(see Fig. 2.3 for more details).*

The user then sends this request to the relay $\mathcal{R}$, via a secure (anonymous) communication channel.

2.2.3 Relay Operations

Let $\mathcal{R}_{\mathcal{E}}.sk$ be the secret key corresponding to the address $\mathcal{R}_{\mathcal{E}}.\mathit{Addr}$ of `Ethereum` account $\mathcal{R}_{\mathcal{E}}$. In what follows, we assume that $\mathcal{R}_{\mathcal{E}}.\mathit{Addr}$ is funded with enough Ether to pay the gas required to call $\widetilde{\textbf{RelayExec}}$ on-chain.

Let $\mathcal{R}_{\mathcal{E}}.\mathit{Addr}$ be the `Ethereum` address of $\mathcal{R}$ charging relay fee $\mathit{fee}_{\mathcal{R}}$. Further, assume that $\widetilde{\textbf{RelayExec}}$ and $\widetilde{\textbf{Mixer}}$ are deployed with addresses $\widetilde{\textbf{RelayExec}}.\mathit{Addr}$ and $\widetilde{\textbf{Mixer}}.\mathit{Addr}$ respectively.

Given the current `Ethereum` state ς (or a copy holding at least $\varsigma[\widetilde{\textbf{RelayExec}}.\mathit{Addr}]$ and $\varsigma[\widetilde{\textbf{Mixer}}.\mathit{Addr}]$) and a relay request $\mathit{req} \in$ `RelayRequest`, the algorithm $\mathsf{RelayCheckRequest}$ (see Fig. 2.1) succeeds if the request req is valid and will result in $\mathcal{R}_{\mathcal{E}}$ receiving $\mathit{fee}_{\mathcal{R}}$.

Note that we assume the existence of an algorithm RelayCheckMixParams which checks whether a given set of Mix parameters *mixParams* result in a successful Mix call in the context of the current blockchain state $\varsigma[\widetilde{\mathbf{Mixer}}.Addr]$. That is, given the state $\varsigma[\widetilde{\mathbf{Mixer}}.Addr]$ of the $\widetilde{\mathbf{Mixer}}$ contract, an `Ethereum` address $Addr_{caller} \in \mathbb{B}^{\texttt{ADDRLEN}}$ and Mix parameters $mixParams \in \texttt{MixInputDType}$,

$$\mathsf{RelayCheckMixParams}(\varsigma[\widetilde{\mathbf{Mixer}}.Addr], Addr_{caller}, mixParams)$$

returns the result of $\mathsf{ZethVerifyTx}(tx)$ (see [Cle20, Section 2.5]) where tx is a transaction that calls $\mathsf{Mix}(mixParams)$ from address $Addr_{caller}$, and ZethVerifyTx executes in the context of $\widetilde{\mathbf{Mixer}}$ with state $\varsigma[\widetilde{\mathbf{Mixer}}.Addr]$.

$\mathsf{RelayCheckRequest}(\varsigma, req)$

```
// Check the fee and relay address in the request
if (req.fee ≠ fee_R) ∨ (req.relayAddr ≠ R_E.Addr) then
  return false
endif
// Check that vout can pay the fee and reject deposits
if (req.mixParams.vout < fee_R) ∨ (req.mixParams.vin ≠ 0) then
  return false
endif
// Check the proof-of-relay-permission
data_R ← encode (R_E.Addr)|| encode(fee_R) || encode(req.outAddr)
if ¬SigSch_OT-SIG.Vf(req.mixParams.otsvk, CRH^ots(data_R), req.permission) then
  return false
endif
return RelayCheckMixParams(ς[Mixer~.Addr], R_E.Addr, req.mixParams)
```

Figure 2.1: RelayCheckRequest algorithm. The relay address $\mathcal{R}_{\mathcal{E}}.Addr$, desired relay fee $fee_{\mathcal{R}}$ and contract addresses $\widetilde{\mathbf{RelayExec}}.Addr$ and $\widetilde{\mathbf{Mixer}}.Addr$ are implicitly available as variables.

Remark 3. *For this check to be meaningful, we assume here that $\mathcal{R}_{\mathcal{E}}$ has access to $\varsigma[\widetilde{\mathbf{Mixer}}.Addr]$ for some recent block height*[5]*.*

Further, we introduce the relay logic in Fig. 2.2 which illustrates the RelayHandleRequest algorithm that processes a received `RelayRequest` request object *req* and relays it on the blockchain network. Here again, we assume that the tuple $(\mathcal{R}_{\mathcal{E}}.Addr, fee_{\mathcal{R}},$

[5]Implementations are not necessarily expected to track $\varsigma[\widetilde{\mathbf{Mixer}}.Addr]$ by themselves, but rather to leverage an existing `Ethereum` full node implementation. RelayCheckMixParams can then be implemented as queries to the node (e.g. via RPC).

$\widetilde{\textbf{RelayExec}}.Addr, \widetilde{\textbf{Mixer}}.Addr)$ is implicitly available to the algorithm. In $\mathsf{RelayHandleRequest}$ a gas price $gasP$ and gas limit $gasL$ are (possibly dynamically) set at the relay's discretion (based on its "relaying service and strategy", defining the trade-off between relay fees, settlement latency, etc.) and passed as explicit parameters to $\mathsf{RelayHandleRequest}$.

$\mathsf{RelayHandleRequest}(\varsigma, req)$

// Check the request
if $\neg\mathsf{RelayCheckRequest}(\varsigma, req)$ **then**
  abort
endif
// Create and broadcast the relay Ethereum transaction
$tx_{\mathcal{R}} \leftarrow \mathsf{genCallTx}(\widetilde{\textbf{RelayExec}}.\mathsf{ProcessRequest}(req), \mathcal{R}_{\mathcal{E}}.sk, gasP, gasL)$
$txid \leftarrow \mathsf{broadcastTx}(tx_{\mathcal{R}})$
// Record the transaction id
$\mathsf{recordTransaction}(txid)$
return

Figure 2.2: Algorithm to process relay requests. $\mathsf{recordTransaction}$ represents the relay-specific handling of the transaction id.

Remark 4. *The checks in* $\mathsf{RelayCheckRequest}$ *provide some level of assurance that* $tx_{\mathcal{R}}$ *will be successfully executed on chain. However, the relay cannot rule out the possibility that a conflicting transaction* $tx^*_{\mathcal{R}}$ *exists on the network, such that, if* $tx^*_{\mathcal{R}}$ *were mined first it would alter the blockchain state and affect the execution of* $tx_{\mathcal{R}}$*. For example, in the case of* `Zeth`*, if some transaction* $tx^*_{\mathcal{R}}$ *which spends the same notes as* $tx_{\mathcal{R}}$ *were mined first,* $tx_{\mathcal{R}}$ *would fail and no payment would be made to the relay.*

Relays may perform further checks to increase their level of confidence that $tx_{\mathcal{R}}$ *will execute as expected (such as examining their own mempool for conflicting transactions) but all such checks add to the cost of running a relay, which may be reflected in the relay fees. It is expected that relays will compete with each other on the basis of their fees. Consequently, relays are expected to adopt some strategy that trades off risk, validation costs and competitiveness, and thereby determine an appropriate price range for relaying.*

Note

Most relay implementations are likely to be able to receive and process multiple requests simultaneously, and at times may receive requests at a faster rate than they can be processed. In this case, the relay has scope to prioritise certain requests over others. In the simplest case, requests may be handled in the order in which they are received (for example via a FIFO queue). More sophisticated relays may employ a strategy allowing them to accept "relay bribes", in which case $fee_{\mathcal{R}}$ is composed of a "base fee" covering the cost of relaying the request, complemented by a relaying premium/bribe to be processed ahead of other requests. Additionally, we note that, the strategy adopted by the relays for ordering and processing relay requests is likely to be impacted by the economics of the underlying platform (see e.g. [BCD+19, Rou20]), and so, can be adjusted at the relay's discretion.

2.2.4 $\widetilde{\textbf{RelayExec}}$ Contract

$\widetilde{\textbf{RelayExec}}$ is a smart contract, deployed to the blockchain, with knowledge of the address of the $\widetilde{\textbf{Mixer}}$ contract to which transactions are to be forwarded. Any participant (relay or potential user) can verify its byte code, and be sure that it cannot be modified by any other party. It executes the Mix call on behalf of the relay, distributing fees as described in the user's request, thereby allowing user and relay to interact in a trustless way. That is, neither the user or the relay are required to trust the the other party to behave in a certain way – $\widetilde{\textbf{RelayExec}}$ constrains how a relay request will be handled once both parties have "agreed" to it.

Relays create transactions that call the ProcessRequest method on $\widetilde{\textbf{RelayExec}}$. This method carries out processing of relay requests, and distribution of *vout*, as defined in Fig. 2.3

$\widetilde{\textbf{RelayExec}}.\mathsf{ProcessRequest}(req)$

```
// Check that the fee is redeemable
if req.mixParams.vout < req.fee then
  abort
endif
// Check the relay permission
data_R ← encode(req.relayAddr) ‖ encode(req.fee) ‖ encode(req.outAddr)
if ¬SigSch_OT-SIG.Vf(req.mixParams.otsvk, CRH^ots(data_R), req.permission) then
  abort
endif
// Cross-contract call to the zeth mixer
mixSuccess ← Mixer~.Mix(req.mixParams)
if ¬mixSuccess then
  abort
endif
// Distribute the fee and the withdrawn funds
send(req.relayAddr, req.fee)
funds ← safeSub(req.mixParams.vout, req.fee)
if req.outAddr ≠ 0x0 ∧ funds ≠ 0 then
  send(req.outAddr, funds)
endif
return
```

Figure 2.3: $\mathsf{ProcessRequest}$ function, where $\mathsf{send}(addr, amt)$ refers to the `Ethereum` operation that sends funds amt to some address $addr$, and where $\mathsf{safeSub}\colon \mathbb{N}_{\texttt{ETHWORDLEN}} \times \mathbb{N}_{\texttt{ETHWORDLEN}} \to \mathbb{N}_{\texttt{ETHWORDLEN}}$, such that $\mathsf{safeSub}(x, y) = x - y$ if $x > y$, 0 otherwise.

Note that the definition of $\mathsf{ProcessRequest}$ does not check that the transaction has originated from $req.relayAddr$. Hence, it is technically possible for a third party to "front-run" this transaction. However any value returned from the transaction is always explicitly distributed to $req.relayAddr$ and $req.outAddr$, and so any other party signing this would essentially be paying the transaction fee on behalf of $\mathcal{R}_{\mathcal{E}}$, while still allowing $\mathcal{R}_{\mathcal{E}}$ to keep the relay fee.

The system would still function if $\mathsf{ProcessRequest}$ did perform such a check (i.e. that $req.relayAddr$ signed the transaction). However, omitting such a check allows the relay some flexibility when sending the transaction. For example, the $\widetilde{\text{relay}}$ may wish to pay the transaction fee from another account, or may wish to call $\widetilde{\textbf{RelayExec}}$ from another contract of some kind.

Finally, we note that users create relay requests such that only **RelayExec** may successfully pass *mixParams* to **Mixer**, however there is no mechanism to ensure that *mixParams* may only be used by a specific *method* of **RelayExec**. Thus, before entering into the protocol, the user should convince himself that **RelayExec** cannot be called in such a way as to violate the protocol (consider the case of a malicious deployer colluding with a relay to provide a second method on **RelayExec** which returns all value to *relayAddr*). For simplicity, we stipulate that **RelayExec** `MUST NOT` have methods other than ProcessRequest.

Chapter 3

Relays with Private Fees

3.1 Introduction

Chapter 2 describes a protocol that allows users to to carry out `Zeth` transactions while remaining anonymous (i.e. make Mix calls without controlling an `Ethereum` account in ς). Under this scheme, relays receive their fee as part of the public output *vout* withdrawn from the `Zeth` mixer contract. It is clear that, under the assumptions described in Section 1.4.3, fees paid publicly *in real time* in this way can be detected by observers of the `Ethereum` network. Such observers may then learn about the profits made by each relay service, which may not be desirable.

In this chapter we introduce a protocol in which relays can receive fees of hidden denominations. This setting is of particular interest in the context of a "relaying market" in which a set of competing relay services operate with the aim of capturing as much bandwidth[1] (on the "relay network") as possible, in order to maximize their revenue.

3.2 Protocol overview

As in Chapter 2, we propose a protocol built on top of `Zeth` which allows holders of `Zeth` notes to securely spend their notes without needing to hold Ether. We assume that relays are willing to sign and broadcast Mix calls, and therefore pay for the gas, in exchange for fee payment in the form of `Zeth` notes. To do so, relays must publish their public `Zeth` address $\mathcal{R}_{\mathcal{Z}}.pub$ and their fee $fee_{\mathcal{R}}$, as well as additional network information such as endpoints which accept relay requests.

Users create parameters *mixParams* to the Mix call, such that one of the newly created notes corresponds to payment of $fee_{\mathcal{R}}$ to $\mathcal{R}_{\mathcal{Z}}.pub$. These parameters are then sent to the relay via an established communication channel. Users must create the signature $mixParams.otssig$ using the relay's `Ethereum` address $\mathcal{R}_{\mathcal{E}}.Addr$, to allow $\mathcal{R}_{\mathcal{E}}$ to use $mixParams$. (In some simple scenarios, $\mathcal{R}_{\mathcal{E}}.Addr$ may be made public alongside other relay information. Here, however, we assume that $\mathcal{R}_{\mathcal{E}}.Addr$ is passed securely from the

[1] i.e. "market share"

relay to the user client as part of the protocol, as this gives the relay more flexibility – see Section 3.3.3 for discussion of how this may enable further relay privacy.)

When the relay receives *mixParams* from the user, it checks to ensure that *mixParams* is valid and indeed contains an output note that pays their fee. Relays can then sign and broadcast a transaction directly calling $\mathsf{Mix}(\mathit{mixParams})$ on the **Mixer** contract. Note that $\mathit{mixParams}.\mathit{otssig}$ ensures that the transaction cannot be front-run.

3.2.1 Relay-originated mix transactions

One potential advantage of relays receiving their fees in the form of `Zeth` notes is that they maintain a level of privacy with respect to their fees. Observers that know the relay's `Ethereum` address can tell that a given transaction is likely to be on behalf of some user, and therefore that one of the output notes is likely to be addressed to the relay (although they will be unable to see any amounts). However, relays can generate their own Mix transactions (which increase privacy by mixing their notes). These Mix transactions are indistinguishable from regular relay transactions created on behalf of other users.

3.3 Limitations and extensions to the protocol

3.3.1 Limitation of output notes

The **Mixer** contract is deployed with a hard-coded number of inputs and outputs (denoted `JSIN` and `JSOUT` respectively). In any Mix call that is anonymized using the relay system described above, one of the outputs must be used to pay the relay. For the case where $\mathtt{JSOUT} = 2$ (a reasonable default value when relays are not considered), the utility of the system is greatly reduced since users may only set the one remaining output freely. In this case, users are able to combine multiple of their notes into another, but are unable to "split" input notes into multiple output notes. In particular, they are unable to pay a specific amount to another `Zeth` user and receive change.

3.3.2 Increasing `JSOUT`

To address the issues of limited output notes, `Zeth` could be instantiated with different parameters (in particular $\mathtt{JSOUT} \leftarrow 3$), in order to support "note-splitting" and relay fee payment in a single transaction. However, such a change to the configuration may have several consequences for the protocol.

To support more output notes, each transaction requires more data to be transferred and processed. This increases the storage and processing requirements of the **Mixer** contract (increasing transaction gas costs), and in turn decreases the lifetime of a `Zeth` deployment for a given Merkle tree size (note that the Merkle tree size is defined when **Mixer** is deployed). Furthermore, the `Zeth` statement must be made more complex (in order to handle more commitments, and possibly to accommodate a deeper Merkle tree), increasing the cost of generating zero-knowledge proofs.

Note also that if `JSOUT` is increased, there may be a tendency for each user's funds to become distributed over more notes. If `JSIN` is not also increased, and the ability of users to recombine `Zeth` notes is not balanced with this, users may more frequently be required to issue multiple transactions when spending their funds (to "recombine" their funds spread across many notes).

Hence, adjusting `JSOUT` may have important consequences which should be considered very carefully, especially if the extra output notes are unlikely to be used outside of the relay system.

3.3.3 Support for Ether output

By default, the $\widetilde{\textbf{Mixer}}$ contract will return any Ether value *vout* to the calling address which, in the protocol described here, would be $\mathcal{R}_{\mathcal{E}}.Addr$, belonging to the relay. Thus, the protocol as presented does not allow users to withdraw value as Ether while using a relay, unless he is willing to trust the relay to forward the Ether in a later transaction. Our aim is to remove any need for trust within the protocol, and a trustless way to withdraw Ether could be valuable in several scenarios.

As in Chapter 2, users could withdraw Ether to previously unseen `Ethereum` addresses. Such anonymous addresses could then be used to pay for `Zeth` transactions, apparently disconnected from any other transactions in the blockchain history. Note that this provides a means for users to anonymously perform `Zeth` transactions that utilize all `JSOUT` output notes, without changing the `Zeth` configuration. Clearly, a user performing two transactions (one to withdraw and one to execute the original `Zeth` transaction) must pay the relay fee *and* the transaction fee for his subsequent transaction. This may have an impact on the economic model for relay fees.

It is clear that relays will be required to regularly withdraw `Zeth` notes as Ether, in order to continue to pay transaction fees. They can, of course, simply issue `Zeth` transactions to withdraw to $\mathcal{R}_{\mathcal{E}}.Addr$. However, if the relay protocol supports output to Ether, relays could also use this mechanism to withdraw to new `Ethereum` addresses. To an observer, such transactions would appear to be standard relay transactions on behalf of some user, but would provide the relay with anonymous Ether, further increasing their privacy.

We next identify two approaches to supporting withdrawals to Ether using the protocol given here.

Arbitrary *vout* address in `Zeth` protocol

The `Zeth` protocol could be slightly modified so that *mixParams* contains an explicit output address *mixParams.outAddr* to which *vout* should be sent by $\widetilde{\textbf{Mixer}}$. This new field $\widetilde{\textbf{Mixer}}.outAddr$ must be included in the data signed by *mixParams.otssig*, ensuring that it cannot be altered by front-runners or malicious relays. This approach adds a small overhead to the generation of *mixParams*, and to the cost of Mix calls, since this output address must be passed as an extra parameter and used in signature validation.

However, supporting this would add versatility to the Zeth protocol and may allow other applications to be built on top of it.

Note that this new address *outAddr* is distinct from the *sender's address* already included in *mixParams.otssig*. *mixParams* must ensure that the encapsulating Ethereum transaction originates from $\mathcal{R}_{\mathcal{E}}.Addr$, and that *vout* Ether are paid to *mixParams.outAddr*.

Relay via intermediary contract

An alternative approach to support secure withdrawal of Ether via relays is to use an intermediary contract, as described in Chapter 2. This change to the relay protocol has the benefit that *vout* can be distributed to one or more parties in a trustless way. However, it does have a potential down-side in terms of privacy – namely that it is trivial for observers to distinguish between transactions issued by relays, and regular transactions issued by users, even if the observer does not know any Ethereum addresses owned by relays.

3.3.4 Fees as Ether or Zeth notes

In order to address the problem of limited output notes in Zeth (see Section 3.3.1), the protocol could allow the user to choose between 2 fee payment methods: as a Zeth note (as described here) or as Ether via *mixParams.vout*. In this case, users can use *all* JSOUT outputs from the Mix call for their own purposes, potentially avoiding the need to adjust JSOUT in the Zeth configuration, and all the associated problems (as described in Section 3.3.2).

A simple way to pay fees as Ether is for users to set $vout = fee_{\mathcal{R}}$ when creating *mixParams*. Upon receiving *mixParams*, relays then check for *either* a Zeth note *or* *mixParams.vout* that pays their fees. The Zeth protocol extension described in Section 3.3.3 to add *mixParams.outAddr* would then be desirable, to prevent front-runners from claiming the relay fee.

An alternative approach would be to use an intermediary contract as described in Chapter 2 (already partially mentioned in Section 3.3.3 to support Ether withdrawals). The protocol would then require the extra complexity of a request structure and *proof-of-relay-permission*, but would provide maximal flexibility for users. A single Mix call could withdraw Ether to a new user address, use all output Zeth notes *and* pay the relay fee (in Ether).

We expect that, given the choice, users would favour fee payment in Ether more often than payment in Zeth notes, since fee payment in Ether allows them to control all JSOUT output notes from the Zeth transaction. Further, it seems reasonable to assume that there will always be some relay operators willing to accept relay fees in Ether, and thereby users will have some element of choice in how fees are paid. Hence, we should expect some divergence between the relays fees paid in Ether and those paid in Zeth notes – namely that fees paid in Zeth notes will tend to be lower, in order to incentivise users to adopt this protocol.

Appendix A

Relays with Stake

Note

This section is concerned with a high-level description of a work-in-progress proposal. There are several issues still to be addressed before it can be considered complete. It is given here as a first step towards the goal of addressing possible relay DoS vectors, with the hope that it can be iterated and eventually turned into a practical solution.

A.1 Introduction

In the above proposals (Chapters 2 and 3), relays receive requests and perform "offline" checks to gain a high level of confidence that the transaction (which the relay must sign and therefore pay for) will "succeed" and the relay will receive the designated fee. Under these protocols, a relay is exposed to risk in two forms:

1. Users are free to make invalid relay requests with no consequences (we assume that they connect via anonymising networks). At the same time, relays are highly incentivised to filter out such invalid requests and avoid signing and broadcasting the corresponding transaction (which would result in the relay paying the invalid transaction's gas, without receiving their relay fee). In order to detect invalid relay requests, the relay must essentially simulate execution of the full transaction against the current state of the blockchain. Although must less costly than paying the corresponding gas, this may still require significant compute resources. If the relay request is judged to be invalid, the relay will necessarily receive no relay fee in exchange for these verification costs.

2. As mentioned in Remark 4, there is a chance that a transaction appears valid (i.e. it passes all checks performed by the relay), but later fails due to a conflicting transaction (unseen by the relay) being mined ahead of it. While this risk can be

reduced by more thorough checks, the relay can never rule out the possibility that a conflicting transaction exists somewhere on the network.

Both of these risks represent possible DoS attack vectors, especially 1, since clients can very easily craft invalid transactions with maximal cost of validation.

We consider a potential approach to address these problems. Specifically, we outline a protocol involving collateral staked by users and associated with some specific relay request. This supports a very lightweight check that relays can perform *before* they commit either Ether as gas for transactions, or compute resources to fully verify the validity of the request. If this fast upfront check passes, the relay is effectively guaranteed income as a result of relaying the transaction - whether or not the transaction is valid at the time it is mined. In this way, relays can avoid DoS attacks that force them to waste resources for no return.

The user's stake is pre-deposited with a contract $\widetilde{\textbf{RelayStake}}$ in such a way that it is bound to a specific relay request. On receipt of a relay request, relays need only confirm that an associated stake exists before creating the relay transaction and executing it via $\widetilde{\textbf{RelayStake}}$. The relay can be sure that $\widetilde{\textbf{RelayStake}}$ will release the stake to the relay, even if the associated relay transaction fails for any reason. In this way, relays can very quickly gain assurance (up to some negligible probability) that they will not lose any operating costs by proceeding with the request.

Since they are exposed to reduced risk, relays should be able to charge users lower fees for their services, while still allowing users to perform `Zeth` operations that are not directly connected to any of their `Ethereum` addresses. However, we note that this reduction in risk for the relay comes with a trade-off for the user, who must deposit Ether from some address in order to use the system. Although users do not reveal *which* `Zeth` operation they are performing, they reveal that the owner of the `Ethereum` address paying for the deposit *may* interact with `Zeth` at some future time.

While this does not provide as much anonymity as the systems described in Chapters 2 and 3, this does represent an improvement in anonymity over the plain `Zeth` protocol. (Note that users interacting directly with `Zeth` can employ strategies to obfuscate their actions, such as broadcasting "dummy" transactions which mix their commitments, however this may incur a relatively high cost and always reveal to observers the exact set of commitments they have created.) The nature of the improvement achieved by the stake system will depend on the details of any final design. We discuss possible trade-offs, and potential strategies to mitigate them, in Appendix A.3.

A.2 Protocol overview

We describe the components involved in the protocol, and their role in the full workflow.

A.2.1 Stake contract

We assume the existence of some contract $\widetilde{\textbf{RelayStake}}$ which performs multiple functions:

- Accept and hold stake as collateral against a specific relay request (and in turn specific relay). Users depositing stake should be able to generate a proof π_{req} that **RelayStake** holds some stake for a specific request *req*. Further, verifiers of π_{req} should not learn which transaction caused the deposit (and thereby the `Ethereum` address of the user who deposited it).

- Act as a relay intermediary, accepting relay requests and proof-of-relay-stake objects. If a valid request and proof are received, the sender has permission to act as a relay for the given request, and the stake is still unspent, **RelayStake** executes the relay request (calls **Mixer**.Mix(*req.mixParams*), using the notation of Chapter 2) and releases the stake to the relay, regardless of the outcome of the Mix call.

Note that this may be implemented as multiple interacting contracts.

A.2.2 Relay

Relays receive pairs (req, π_{req}) of relays requests and associated proof-of-relay-stake objects. When a relay $\mathcal{R}$ (with `Ethereum` address $\mathcal{R}_{\mathcal{E}}.Addr$) receives such a pair, the operations he performs are relatively straightforward:

Step 1. Check the correctness of *req* and π_{req}, namely that:

1. *req* names the relay's `Ethereum` address as the recipient of the relay fee
2. a valid stake exists in **RelayStake**, corresponding to *req* and π_{req}

The relay should not learn which transaction deposited the stake that corresponds to *req* and π_{req} (which would reveal an `Ethereum` account of the user).

Step 2. Create a transaction $tx_{\mathcal{R}}$ which calls **RelayStake** with parameters *req* and π_{req}.

Step 3. Broadcast $tx_{\mathcal{R}}$ to the network and asynchronously wait to receive the relay fee.

As described above, after the initial check of stake corresponding to *req*, the relay can be sure he will receive his fee even if the Mix call fails. Note also that the relay's transaction cannot be front-run, since **RelayStake** will only release the stake to the relay mentioned in *req*. A user may be able to spend the notes in *req.mixParams* via another transaction, but he will not be able to prevent the relay from claiming the stake he previously deposited.

For these reasons, the relay need only perform the checks listed above. There is little to be gained by checking any further details of *req*, including *req.mixParams* or the state of the `Zeth` mixer contract **Mixer**.

A.2.3 User

A user who wants to make use of a specific relay $\mathcal{R}$ with `Ethereum` address $\mathcal{R}_{\mathcal{E}}.Addr$, using Mix parameters $mixParams$, performs the following actions:

Step 1. Create the appropriate $mixParams$ and corresponding relay request req, bound to the relay's address $\mathcal{R}_{\mathcal{E}}.Addr$.

Step 2. Stake some Ether with **RelayStake** against req and generate a corresponding proof-of-relay-stake π_{req}. Note that this collateral is "bound" to req.

Step 3. Send req, π_{req} to the chosen relay (via an anonymous channel), and asynchronously wait for a corresponding transaction to be mined.

Note that the user must have a funded `Ethereum` account $\mathcal{U}_{\mathcal{E}}$ in order to stake collateral, impacting their anonymity to some extent.

A.2.4 Reclaiming stake

Under some circumstances, a user may wish to reclaim his stake after depositing it in **RelayStake**. In fact, if no mechanism were available to accomplish this, the protocol would be vulnerable to withholding attacks. That is, malicious relays could accept relay requests but not relay them within a reasonable time, essentially locking up the user's stake. Eventually, the victims of withholding attacks would be forced to forfeit their stake and find another means to carry out their `Zeth` operations (potentially via another relay). Once the operation has been completed via another transaction, the malicious relay (still holding a request with associated stake) can then claim the user's stake by broadcasting the relay transaction, which will now fail. The victim must pay for his transaction more than once, losing his stake (which, may be greater than the original relay fee - see Appendix A.3). This attack would also cause significant disruption to the relay network and associated market.

> **Note**
>
> Note that the affect of such an attack may be mitigated by some kind of reputation system alongside a relay market fee, so the threat posed by this may not be considered severe. In fact, a reputation system may be a vital part of any relay market.

If the user were allowed to reclaim their stake at any time, the transaction to reclaim it could be unseen by the relay, yet mined ahead of $tx_{\mathcal{R}}$. Thus, the relay would no longer have any guarantee that he would receive the stake in exchange for broadcasting the relay transaction.

It may be possible to facilitate reclaim of the stake by setting a "period of validity" for the stake, if this can be implemented such that:

- the user cannot reclaim the stake until the period of validity elapses,

- the relay can obtain some estimate regarding the period of validity (minimally, if the relay knows some lower bound on the remaining period of validity, it can gain a high degree of certainty that the relay transaction will be mined before the user is able to reclaim the stake),

- for a relay transaction $tx_{\mathcal{R}}$, observers (and relays) should not be able to infer significant information about the transaction that deposited the collateral for $tx_{\mathcal{R}}$,

- for a stake reclaim transaction, observers should not be able to infer significant information about the transaction that deposited the stake being reclaimed,

- the transaction to reclaim a stake cannot be front-run.

A.3 Remarks

In this section we give some remarks about the proposed stake system above. Note that details are highly dependent on the specific cryptographic primitives used. The steps above give an outline of how a staking system may work, with the caveat that they are very likely to be modified in any fully specified protocol.

Stake deposit leaks information. The transaction to deposit the user's stake reveals an `Ethereum` address of the user, and the value of the stake. Similarly, the transaction to reclaim an unused stake is also associated with the user's `Ethereum` address. In particular, observers may learn about `Ethereum` addresses of users likely to interact with the relay system. Note that this could potentially be mitigated if users are able, via some other mechanism such as Chapter 2, to anonymously receive funds at new "unused" `Ethereum` addresses. However, in isolation, the stake system always requires an initial transaction from a funded address.

Relays with stake as part of a wider market. Reliance on external mechanisms (such as the ability to withdraw to new `Ethereum` addresses, as mentioned above), may not necessarily be a problem for a stake-based relay protocol. We note that there may be significant benefit to an ecosystem of various relay types (or relays supporting a variety of request types) including those discussed in this document. In particular, we may imagine a market in which users can choose between a range of relay request types - those supporting more privacy and not requiring Ether (with higher fees, as a consequence of the extra risk assumed by the relay), and relays with lower fees which place more requirements on users (able to mitigate much of their risk). Concretely, such a system may allow users to withdraw Ether to an anonymous address via an expensive relay, and then use this Ether as collateral for further relay transactions with lower relay fees. Obviously, in such a scenario, users (and their wallet software) must take a great deal of care not to reveal relationships between transactions.

Collateral value vs Relay Fee. The above outline suggests a very simple scenario in which the stake is unconditionally used as the relay fee (that is, the stake must have the same value as the relay fee, which cannot be negotiated after the stake has been deposited. We note that any concrete instantiation would require somewhat more flexibility, and would likely extend fee payment mechanisms in one of several ways.

- The user could be required to stake some fixed value from which fees are paid, with the "change" being paid either to the user, or as *vin* to the `Zeth` mixer. The fixed stake value would then define an upper bound on relay fees. This allows a lot more scope for relays to dynamically change their fees.
- We may prefer to distinguish between successful and failed Mix calls. On success, **RelayStake** may refund the user with the difference between the stake and the relay fee (as described above), while users could be punished for invalid relay requests by forfeiting their entire stake to the relay.
- Similarly, it may be preferable for relay fees to be paid directly from the Mix call in the case of successful transactions. That is, the Mix call parameters $req.mixParams$ must include an output paying the relay (either *vout* as in Chapter 2, or a `Zeth` note as in Chapter 3), and the stake is refunded to the user (either to an address of their choosing, or as *vin* to the Mix call. This allows relays to maintain privacy with respect to their fee payments, while minimizing risk through the stake system.
- It may be possible to allow users to deposit a single stake, bound to a specific relay, which can then be used for multiple relay requests. The relay could still claim the stake by presenting an invalid request from the user, but the user could make multiple relay requests (possibly within some time limit - see the following paragraph) without the need to redeposit. User anonymity would be improved because, while observers would still learn the `Ethereum` address depositing each stake, they would not be able to determine how many relay requests were carried out on behalf of each depositor.

Period of validity. The period of validity gives observers information about when the users relay transaction will be carried out. Depending on the frequency of relay transactions interacting with **RelayStake**, users should ensure that the period of validity is sufficiently long, to avoid compromising their anonymity. Similarly, a predictable interval between a stake being deposited and the corresponding relay transaction that claims it, would also reveal information about the user originating each relay transaction. To avoid this, the period of validity should be sufficiently long to allow some "noise" in the interval between stake deposits and relay transactions. Note that this presents a trade-off, since the user's funds are potentially "locked up" for a longer period.

Appendix B

Network structure

B.1 Binding requests to relays

B.1.1 Background

The protocols presented in this document require users to create relay requests that can only be successfully processed by a specific relay. This serves as a mechanism to prevent other network participants from "stealing" the relay requests or front-running relay transactions. The alternative to this would be to support "free" relay requests, not bound to specific relays, which could therefore be processed by any participant. If these "free" requests are made available (or "broadcast") to multiple relays, those relays must then "race" to process the request and broadcast a corresponding relay transaction. When the first relay transaction is accepted by the blockchain, the "winning" relay will receive the relay fee and later transactions from other relays will be rendered invalid (as a consequence of the nullifiers declared in the `Zeth` relay request being marked as used).

While such an approach is entirely feasible, it increases the risk for relays, making it much harder for them to hedge against lost fees and wasted compute resources. As a consequence, it becomes much more difficult for relays to assess the risk associated with a given request, which in turn is likely to result in an increase in relay fees. All resources used by "losers" of the race are wasted. In contrast, in the case where requests are bound to specific relays, these resources can be used to process multiple requests in parallel, increasing the efficiency of the system.

B.1.2 Emulating "free" relay requests

Despite the mechanisms to prevent front-running, the protocols presented in this document could be leveraged by some user (say $\mathcal{U}$) to force relays to "compete" for relay requests. Specifically, in order to call the `Zeth` mixer $\widetilde{\textbf{Mixer}}$ with parameters *mixParams*, $\mathcal{U}$ can run multiple instances of a protocol in parallel, generating N relay requests for *mixParams*, each targeting a different relay. If the user then sends each of these N requests to the targeted relay, the desired state transition will be carried out by the first relay transaction to be accepted into the blockchain, rendering the remaining $N-1$ requests invalid (by the nullifier mechanism cited previously).

As in the case of "free" requests, relays are exposed to extra risk for the reasons given above. However, this "emulating" approach does provide partial mitigation of this risk, due to the extra cost that the user $\mathcal{U}$ must incur. Under the protocols given in this document, in order to create N requests targeting different relays, the user must generate N zk-SNARKs, which is computationally demanding and therefore represents a cost to the user. Therefore, request generation may act as a user-side proof-of-work, preventing malicious messages from flooding the network (as originally designed for by Dwork et al. [DN92, JJ99]). This naturally leads to the following process by which relays can partially protect against some DoS vectors, by performing the following checks on relay requests:

1. Verify that none of the nullifiers in the request has been seen in previously received requests (inspect the mempool and the blockchain state). If one or more nullifiers is a duplicate then reject the request, else proceed.

2. Verify the zk-SNARK proof in the request. If the proof is invalid, reject the request. Otherwise the request can be considered for processing.

In this way, the cost of generating N proofs imposes some upper bound on the message output rate of a potential attacker.

B.2 Unicast vs broadcast networks

The discussions in this document assume only that some transport mechanism exists for users to send relay requests to specific relays. As noted in Section 1.4, users can achieve further anonymity if this transport mechanism does not require the user to reveal any identifying information at the network level. We now discuss some specific implementations of the transport layer (namely "unicast" vs "broadcast" networks), and their respective properties.

By design, relay requests are bound to specific relays, which intuitively implies a "unicast" style transport mechanism. That is, relays publish a network address of some form, and users send requests to this address. Observers of the physical network may determine that a message has been sent from the user to the relay, but the content of the message (i.e. the details of the relay request) is not visible to other participants. This is a natural choice given that relay request data cannot be used by parties other than the targeted relay. While not a requirement of any of these protocols, unicast channels (in particular point-to-point communication channels, which we assume to be encrypted by default) only reveal the relay request content to the relay itself. Adversaries able to gain control a physical network node along the route between user and relay (in general a limited set of nodes, which varies depending on user and relay) may learn that a message was sent from the user to the relay, but they will not learn anything about the message content. Such communication channels also allow for interaction between relay and user (for example, the relay could dynamically select an `Ethereum` or `Zeth` address to receive payment, or privately negotiate fees with the user).

Clearly it would be entirely possible to implement these protocols using a "broadcast" system, such as those employed by blockchains to propagate transactions and blocks. Requests could be broadcast unencrypted without impacting the reliability of the system, as long as messages were eventually seen by the target relay. Broadcast networks provide some inherent receiver anonymity, in the sense that it is more difficult to identify which *network node* is the recipient of a given message, however in this setting, all participants in the system would be able to see the content of relay requests and potentially determine the number of requests received by each relay identity (and, in turn, infer information about their profit). Despite relay requests being visible to other participants, the protocol would still prevent other relays from profiting from these requests, since they are bound to the target relay. Further, the content of requests could be hidden by encryption so that only the intended relay may read them. Instead of publishing a network "address" of some form, relays could publish an encryption key, with which users must encrypt requests before broadcasting them (although care must be taken to use a key-private encryption scheme to avoid leaking information about the recipient).

While broadcast networks could theoretically be used in these relay protocols, unicast networks are more bandwidth efficient (i.e. a given message needs only to find a path through the network in order to flow from the sender to the recipient). In contrast, broadcast networks may provide a level of sender anonymity in the face of network observers, although even in broadcast networks methods exist to infer the message originator (e.g. nodes with high degree[1] – also referred to as "supernodes" – can be used to infer the sender of a message on a broadcast channel by using timing information [KKM14]). Broadcast communication channels are also of great interest to achieve "recipient anonymity"(see [PW87] for more details on "anonymity").

At first sight, the use of a unicast transport may appear to increase the centralization of the system. However, this is demonstrably not the case for the protocols discussed here, which can (as described above) be implemented using a transparent broadcast network and do not inherently rely on any centralization.

Finally, we note that, although broadcast networks could theoretically be used, we suggest that unicast networks are likely to be more suitable, given their lower bandwidth and complexity.

B.3 Network anonymity

Relay protocol designers may choose to transmit requests via the method that best fits their needs, taking into consideration the tradeoffs mentioned in Appendix B.2 above. As well as overhead, network topology also has a strong influence on anonymity [DMT10]). However, in order to achieve strong privacy guarantees, anonymisation techniques (e.g. cover traffic, message padding etc.) must be used, to minimize communication leakages.

[1] In the graph theoretical sense.

While protocols like Dandelion [VFV17, FVB$^+$18] were initially introduced to improve *diffusion* mechanisms and improve network anonymity on Bitcoin, they could also be of interest in the context of relay request broadcasts (as alluded to in Appendix B.2). However, other techniques (providing different properties) may also be of interest for relay network. Some of these are given below:

- DC-nets [Cha88] provide strong guarantees with respect to the sender anonymity (but generally incur a big overhead and require large amounts of randomness).

- Crowds [RR98] follow a “blending into a crowd” approach (i.e. hiding one’s actions among the actions of many others), in which a user’s request is randomly circulated in a “crowd” (set of users) before being submitted – by a random member of the crowd – and sent to the destination[2]. Note that such approaches generally do not provide strong guarantees with respect to recipient anonymity[3].

- Mix networks (or *mixnets*) [Cha03], in which nodes (“mix nodes”) are routers that perform cryptographic operations (providing bit-wise unlinkability), and modify the order in which output messages are emitted. This hides any correspondence between input and output messages.

- Onion routing [GRS99] (which also underlies “garlic routing” [Din00]) consists of multiple layers of encryption (one per “hop” on the network). Requests are sent through a chosen set of routers (forming a “circuit”) in order to obfuscate the link between sender and recipient, as seen by non-global adversaries (i.e. those that do not control all nodes on the circuit[4]). This generally achieves low-latency relative to other approaches.

Importantly, modern protocols building on these techniques use additional mechanisms for enhanced robustness (e.g. “cover traffic” to prevent timing attacks etc.).

Remark 5. *We note that accountable anonymous communication networks [DP07] are also of great interest in the context of transaction relay protocols as a way to further prevent DoS attacks.*

[2] Crowd members *cannot* identify the initiator of the request. The initiator is indistinguishable from a member that forwards a request from another user.

[3] While relay anonymity is not our principal focus, it is worth keeping in mind the impact of side channel leakages which can be used to infer information about the sender. For instance, a powerful adversary – monitoring a big part of the Internet – may notice a client access the relay’s public information (such as the relay’s website) followed by a message to the relay from a crowd to which the client belongs. The adversary may then infer that the client was the relay user. Hence, additional care needs to be allocated to the relay discovery mechanism itself, and the right trade-offs must be made depending on the application and associated threat model.

[4] In some cases, controlling the “entry” and “exit” nodes (i.e. first and last nodes of the chain/circuit) is sufficient to carry out so-called “correlation attacks”. See `https://github.com/Attacks-on-Tor/Attacks-on-Tor` for a list of attacks on Tor [DMS04]

Bibliography

[BCD+19] Vitalik Buterin, Eric Conner, Rick Dudley, Matthew Slipper, Ian Norden, and Abdelhamid Bakhta. Fee market change for eth 1.0 chain. `https://github.com/ethereum/EIPs/blob/master/EIPS/eip-1559.md`, 2019.

[Bra97] S. Bradner. Key words for use in rfcs to indicate requirement levels. RFC 2119, RFC Editor, March 1997.

[Cha88] David Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *J. Cryptol.*, 1(1):65–75, 1988.

[Cha03] David Chaum. Untraceable electronic mail, return addresses and digital pseudonyms. In Dimitris Gritzalis, editor, *Secure Electronic Voting*, volume 7 of *Advances in Information Security*, pages 211–219. Springer, 2003.

[Cle20] Clearmatics. Zeth Protocol Specification, 2020.

[DGK+19] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. Flash boys 2.0: Frontrunning, transaction reordering, and consensus instability in decentralized exchanges, 2019.

[Din00] Roger Dingledine. The Free Haven Project: Design and Deployment of an Anonymous Secure Data Haven. MIT Master's Thesis. https://www.freehaven.net/papers.html, 06 2000.

[DMS04] Roger Dingledine, Nick Mathewson, and Paul F. Syverson. Tor: The second-generation onion router. In Matt Blaze, editor, *Proceedings of the 13th USENIX Security Symposium, August 9-13, 2004, San Diego, CA, USA*, pages 303–320. USENIX, 2004.

[DMT10] Claudia Díaz, Steven J. Murdoch, and Carmela Troncoso. Impact of network topology on anonymity and overhead in low-latency anonymity networks. In Mikhail J. Atallah and Nicholas J. Hopper, editors, *Privacy Enhancing Technologies, 10th International Symposium, PETS 2010, Berlin, Germany, July 21-23, 2010. Proceedings*, volume 6205 of *Lecture Notes in Computer Science*, pages 184–201. Springer, 2010.

[DN92] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In Ernest F. Brickell, editor, *Advances in Cryptology - CRYPTO '92, 12th Annual International Cryptology Conference, Santa Barbara, California, USA, August 16-20, 1992, Proceedings*, volume 740 of *Lecture Notes in Computer Science*, pages 139–147. Springer, 1992.

[DP07] Claudia Díaz and Bart Preneel. Accountable anonymous communication. In Milan Petkovic and Willem Jonker, editors, *Security, Privacy, and Trust in Modern Data Management*, Data-Centric Systems and Applications, pages 239–253. Springer, 2007.

[FVB+18] Giulia C. Fanti, Shaileshh Bojja Venkatakrishnan, Surya Bakshi, Bradley Denby, Shruti Bhargava, Andrew Miller, and Pramod Viswanath. Dandelion++: Lightweight cryptocurrency networking with formal anonymity guarantees. *Proc. ACM Meas. Anal. Comput. Syst.*, 2(2):29:1–29:35, 2018.

[GRS99] David M. Goldschlag, Michael G. Reed, and Paul F. Syverson. Onion routing. *Commun. ACM*, 42(2):39–41, 1999.

[JJ99] M. Jakobsson and A. Juels. Proofs of work and bread pudding protocols. In *Communications and Multimedia Security*, 1999.

[KKM14] Philip Koshy, Diana Koshy, and Patrick D. McDaniel. An analysis of anonymity in bitcoin using P2P network traffic. In Nicolas Christin and Reihaneh Safavi-Naini, editors, *Financial Cryptography and Data Security - 18th International Conference, FC 2014, Christ Church, Barbados, March 3-7, 2014, Revised Selected Papers*, volume 8437 of *Lecture Notes in Computer Science*, pages 469–485. Springer, 2014.

[lsa20] lsankar4033. Surrogeth: Tricking frontrunners into being transaction relayers. `https://ethresear.ch/t/surrogeth-tricking-frontrunners-into-being-transaction-relayers/6937`, 2020.

[PHE+17] Ania M. Piotrowska, Jamie Hayes, Tariq Elahi, Sebastian Meiser, and George Danezis. The loopix anonymity system. In Engin Kirda and Thomas Ristenpart, editors, *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, pages 1199–1216. USENIX Association, 2017.

[PW87] Andreas Pfitzmann and Michael Waidner. Networks without user observability. *Comput. Secur.*, 6(2):158–166, 1987.

[RK20] Dan Robinson and Georgios Konstantopoulos. Ethereum is a dark forest. `https://medium.com/@danrobinson/ethereum-is-a-dark-forest-ecc5f0505dff`, 2020.

[Ron20] Antoine Rondelet. Zecale: Reconciling privacy and scalability on ethereum. *CoRR*, abs/2008.05958, 2020.

[Rou20] Tim Roughgarden. Transaction Fee Mechanism Design for the Ethereum Blockchain: An Economic Analysis of EIP-1559. `https://timroughgarden.org/papers/eip1559.pdf`, 2020.

[RR98] Michael K. Reiter and Aviel D. Rubin. Crowds: Anonymity for web transactions. *ACM Trans. Inf. Syst. Secur.*, 1(1):66–92, 1998.

[RZ19] Antoine Rondelet and Michal Zajac. ZETH: On Integrating Zerocash on Ethereum. [Online; released April-2019], 2019.

[VFV17] Shaileshh Bojja Venkatakrishnan, Giulia C. Fanti, and Pramod Viswanath. Dandelion: Redesigning the bitcoin network for anonymity. *Proc. ACM Meas. Anal. Comput. Syst.*, 1(1):22:1–22:34, 2017.

[Woo19] Dr Gavin Wood. ETHEREUM: A Secure Decentralised Generalised Transaction Ledger Byz antium. `https://ethereum.github.io/yellowpaper/paper.pdf`, 2019. [VERSION 7e819ec - 2019-10-20].