Zef: Low-latency, Scalable, Private Payments

Mathieu Baudet
mathieu.baudet@zefchain.com
Facebook and Zefchain Labs

Alberto Sonnino
asonnino@fb.com
Facebook

Mahimna Kelkar
mahimna@cs.cornell.edu
Cornell University

George Danezis
g.danezis@ucl.ac.uk
University College London (UCL) and Mysten Labs

ABSTRACT

We introduce Zef, the first Byzantine-Fault Tolerant (BFT) protocol to support payments in anonymous digital coins at arbitrary scale. Zef follows the communication and security model of FastPay [5]: both protocols are asynchronous, low-latency, linearly-scalable, and powered by partially-trusted sharded authorities. In contrast with Fastpay, user accounts in Zef are uniquely-identified and safely removable. Zef coins are bound to an account by a digital certificate and otherwise stored off-chain by their owners. To create and redeem coins, users interact with the protocol via privacy-preserving operations: Zef uses randomized commitments and NIZK proofs to hide coin values; and, created coins are made unlinkable using the blind and randomizable threshold anonymous credentials of Coconut [30]. Besides the detailed specifications and our analysis of the protocol, we are making available an open-source implementation of Zef in Rust. Our extensive benchmarks on AWS confirm textbook linear scalability and demonstrate a confirmation time under one second at nominal capacity. Compared to existing anonymous payment systems based on a blockchain [23, 36], this represents a latency speedup of three orders of magnitude, with no theoretical limit on throughput.

1 INTRODUCTION

Anonymous payment systems have been an exciting research area in cryptography since Chaum's seminal work [13] on e-cash. Early e-cash schemes [12, 13, 28] however required a centralized issuer to operate, usually in the form of a trusted commercial bank, which hampered their adoption. In recent years, the advent of networks like Bitcoin has sparked renewed interest in privacy-preserving decentralized payment systems. A number of protocols [7, 22, 23] focusing on anonymous payments are now deployed as permissionless blockchains.

Compared to traditional global payment infrastructures (aka. RTGS systems [6]), however, decentralized anonymous payment systems have not yet reached performance levels able to sustain large-scale adoption. For instance, due to high computational costs, only 2% of Zcash [36] transactions typically take full advantage of the privacy features offered by the platform [1].

At the other end of the performance spectrum, the FastPay [5] protocol does not support anonymous payments, but offers low-latency transfers (in the range of 100-200 ms) and arbitrary (linear) scalability while operating in the Byzantine-Fault-Tolerant (BFT) model with an asynchronous network. This model makes FastPay suitable for a deployment as a high-performance sidechain of an existing blockchain. Remarkably, in order to scale linearly, FastPay is built solely on reliable broadcast—as opposed to using a BFT consensus (see e.g. [10]).

In this work, we revisit the FastPay design with privacy, storage costs, and extensibility in mind. In effect, we propose Zef, the first linearly-scalable BFT protocol for anonymous payments with sub-second confirmation time.

The Zef Protocol. Zef implements digital coins that are opaque and unlinkable (in short *anonymous*) by combining multiple techniques: (i) randomized commitments and Non-Interactive Zero-Knowledge (NIZK) proofs (e.g., [22]) to provide *opacity*, that is, to hide payment values, together with (ii) blind and randomizable signatures (e.g., [30]) to provide *unlinkability*, meaning that the relation between senders and receivers is hidden.

While these privacy-preserving techniques apply well to the communication model of FastPay, the natural approach to preventing double spending requires authorities to keep track of all the coins that have been spent. In the long run, maintaining such an ever-growing *spent list* is bound to create performance and storage bottlenecks. We note that both Monero and Zcash nevertheless follow this approach, suggesting theoretical limitations in throughput and/or system lifetime. In contrast, in a non-anonymous payment system such as Bitcoin, it is sufficient to keep track of the currently unspent coins—known as *Unspent Transaction Outputs* (UTXOs).

A key contribution of Zef is to remove the need for authorities to maintain a list of spent coins. In particular, Zef proposes a new concept of accounts supporting both (i) account-oriented operations, such as transferring ownership, receiving funds repeatedly, and creating unique identifiers, and (ii) UTXO-like operations, such as spending and deactivating accounts. Deactivated accounts may be safely cleaned up from the local storage of each authority.

Importantly, all the features introduced by Zef still rely exclusively on client-driven broadcast (reliable or not) and do not require a consensus protocol.

Contributions. (1) To support spendable assets such as digital coins, we start by revisiting the design of FastPay accounts: we propose a unified protocol for scalable accounts operations where accounts are addressed by unique, non-replayable identifiers (UIDs) and support a variety of operations such as account creation, deactivation, transparent payments, and ownership transfer. (2) Building on these new foundations, we describe the first asynchronous BFT protocol for opaque, unlinkable payments with linear (aka "horizontal") scalability and sub-second latency. (3) Finally, we are making available an open-source prototype implementation of Zef in Rust and provide extensive benchmarks to evaluate both the scalability and the latency of anonymous payments.

2 BACKGROUND AND RELATED WORK

Fastpay. FastPay [5] was recently proposed as a sidechain protocol for low-latency, high-throughput payments in the permissioned model with asynchronous communication.

- Sidechain protocol: FastPay is primarily meant as a scalability solution on top of an existing blockchain with smart contracts (e.g. Ethereum [33]).
- Permissioned model (Byzantine-Fault Tolerance): $N = 3f + 1$ replicas called *authorities* are designated to operate the system and process the clients' requests. A fixed set of at most f authorities may be *malicious* (i.e. deviate from the protocol).
- Low latency: Authorities do not interact with each other (e.g. running a mempool or a consensus protocol). Client operations succeed predictably after a limited number of client/authorities round trips. Notably, in FastPay, a single round-trip with authorities suffices to both initiate a payment and obtain a certificate proving that the transfer is final.
- Scalable: Each authority operates an arbitrary number of logical shards, across many physical hosts. By design, each client request is processed by a single shard within each authority. Within an authority, communication between shards is minimal and never blocks a client request.
- Asynchronous communication: Malicious nodes may collude with the network to prioritize or delay certain messages. Progress is guaranteed when messages eventually arrive.

In a nutshell, the state of the Fastpay accounts is replicated on a set of authorities. Each account contains a public key that can authorize payments out, a sequence number and a balance. Account owners authorize payments by signing them with their account key and including the recipient amount and payment value. An authorized payment is sent to all authorities, who countersign it if it contains the next sequence number; there are enough funds; and, it is the first for this account and sequence number. A large enough number (to achieve quorum intersection) of signatures constitute a certificate for the payment. Obtaining a certificate ensures the payment can eventually be executed (finality). Anyone may submit the certificate to the authorities that check it and update the sender account and recipient balance.

FastPay does not rely on State-Machine Replication (SMR) in the sense that it does not require authorities to agree on a single global state—as one could expect from a traditional sidechain. Doing so, the protocol avoids performance considerations commonly associated with SMR. Notably, FastPay does not incur the end-to-end latency cost of gathering, disseminating, and executing large blocks of transactions, a de-facto requirement for high throughput with SMR solutions [14, 18, 31, 34].

Despite the benefits listed above, until now, the FastPay protocol has been limited to transparent payments, that is, without any privacy guarantees. In fact, to ensure fund availability in worst-case scenarios, FastPay requires all past money transfers to be publicly available in clear text. This contrasts negatively with traditional retail payments (e.g. credit cards) where individual transactions remain within a private banking network. Another technical limitation of FastPay is that unused accounts cannot be deleted. In a privacy-sensitive setting where users would never re-use the same account twice, this means that storage cost of authorities would grow linearly with the number of past transactions.

Existing private payment schemes. Compared to payment channels (e.g. [26]), safety in FastPay and Zef does not require any upper bound on network delays and clients to stay connected (aka. a *synchrony* assumption [16]). Furthermore, the reliability of the lighting network depends on the existence of pairwise channels, with the success of a payment between two random nodes being at most 70%[1]. In contrast, coins delegated to a FastPay instance are always immediately transferable to any recipient that possesses a public key (resp. an account identifier in Zef).

Several privacy-preserving payment systems have been proposed in the past, each based on a blockchain consensus and therefore not linearly scalable: Zcash, based on Zerocash [7], uses a zero-knowledge proof of set inclusion which is expensive to compute instead of an efficient threshold issuance credential scheme. As a result most transactions are unshielded, leading to a degradation in privacy [19]. Monero [23] uses ring signatures to ensure transactions benefit from a small anonymity set. However, intersections attacks and other transaction tracing heuristics are applicable. This results in an uneven degree of privacy [24].

3 OVERVIEW

We present Zef, an evolution of FastPay [5] designed to support high-volume, low-latency payments, both anonymous and transparent, on top of a *primary blockchain*. To do so, Zef introduces a new notion of accounts, indexed by a unique identifier (UID) and able to support both account-oriented updates and UTXO-like deactivation when anonymous coins are spent.

Authorities and quorums. We assume a primary blockchain which supports smart contracts (e.g., Ethereum [33]). In a typical deployment, we expect Zef to be "pegged" to the primary chain through a smart contract, thereby allowing transfers of assets in either direction [3]. The Zef smart contract holds the reserve of assets (e.g., coins) and delegates their management to a set of external nodes called *authorities*. For brevity, in the rest of this paper, we focus on the Zef system and omit the description of transfers between the primary blockchain and Zef. The mechanics of such transfers is similar to "funding" and "redeeming" operations in FastPay [5].

Zef is meant to be *Byzantine-Fault Tolerant (BFT)*, that is, tolerate a subset of authorities that deviate arbitrarily from the protocol. We assume an *asynchronous* network that may collude with malicious authorities to deliver messages in arbitrary order. The protocol makes progress when message are eventually delivered.

We assume that authorities have shared knowledge of each other's signing public keys. Each authority is also assigned a *voting power*, which indicates how much control the authority has within the system. N denotes the total voting power, while f denotes the power held by adversarial authorities. In the simplest setting where each authority has a voting power of 1 unit, N denotes the total number of authorities and f denotes the number of adversarial authorities tolerated by the system. In general, unequal voting powers may be used to reflect different *stakes* locked by the authorities on the main blockchain. Similar to standard protocols, we require

[1]https://diar.co/volume-2-issue-25/

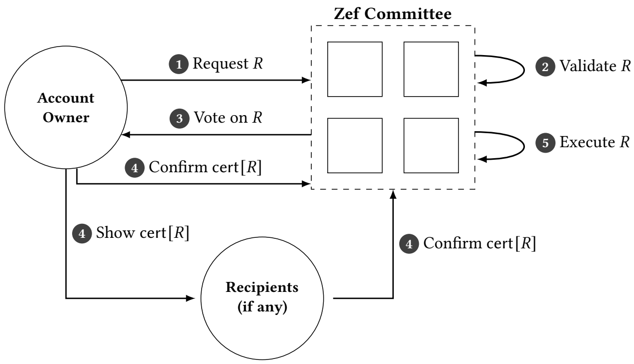

Figure 1: Request and execution of a regular account operation

$0 \leq f < \frac{N}{3}$. The system parameters N and f, as well as the public key and voting power of each authority are included in the Zef smart contract during setup.

We use *quorum* to refer to a set of signatures by authorities with a combined voting power of at least $N - f$. An important property of quorums, called *quorum intersection*, is that for any two quorums, there exists an honest authority α that is present in both.

Cryptographic primitives. We assume a collision-resistant hash function hash as well as a secure public-key signature scheme. Informally, a *random commitment* $\mathsf{cm} = \mathsf{com}_r(v)$ is an expression that provides a commitment over the value v (in particular, is collision-resistant) without revealing any information on v, as long as the random seed r is kept secret. A signing scheme supports *blinding* and *unblinding* operations iff (i) a signature of a *blinded message* $B = \mathsf{blind}(M; u)$ with *blinding factor* u can be turned into a valid signature of M by computing the expression $\mathsf{unblind}(B; u)$, and (ii) provided that u is a secret random value, an attacker observing B learns no information on M.

Blind signatures will be used for anonymous coins in Section 5 together with an abstract notion of Non-Interactive Zero-Knowledge (NIZK) proof of knowledge. We will also further assume that a public key $\mathsf{pk}_{\mathsf{all}}$ is set up between authorities in such a way that any quorum of signatures on M may be aggregated into a single, secure *threshold signature* of M, verifiable with $\mathsf{pk}_{\mathsf{all}}$. (See Appendix B for a concrete instantiation.)

Clients, requests, certificates, and coins. Clients to the Zef protocol are assumed to know the public configuration of the system (see above) including networking addresses of authorities. Network interactions are always initiated by a client request. We distinguish *account-based* requests, i.e., those targeting a specific account, noted R, from *free requests* R^*. In what follows, all requests are account-based unless mentioned otherwise. Free requests will be used for coin creation in Section 5.

As illustrated in Figure 1, clients may initiate a particular *operation* O on an account that they own as follows: (i) broadcast a request R containing the operation O and authenticated by the client's signature to the appropriate logical *shard* of each authority α (1); and (ii) wait for a quorum of responses, that is, sufficiently many answers so that the combined voting power of responding authorities reaches $N - f$.

An authority responds to a valid request R by sending back a signature on R, called a *vote*, as acknowledgment (3). After receiving votes from a quorum of authorities, a client forms a *certificate* C, that is, a request R together with a quorum of signatures on R. In the rest of this paper, we identify certificates on a same value V and simply write $C = \mathsf{cert}[V]$ when C is a certificate on V. Depending on the nature of value V (e.g., anonymous coins in Section 5) and implementation choices, the quorum of signatures in C may be aggregated into a single threshold signature σ.

We further distinguish *regular* operations from *locking* operations. In the most common case, a request R contains a regular operation O meant to be executed once. The certificate $C = \mathsf{cert}[R]$ is meant to be broadcast back to authorities as a *confirmation* (4), thereby triggering the one execution of O (5) and allowing the account behind R to process further requests. A confirmation certificate C also acts as a *proof of finality*, that is, a verifiable document proving that the transaction (e.g., a payment) can be driven to success. In the case of payments, recipients should obtain and verify the certificate themselves before accepting the payment.

In contrast, locking operations cannot be confirmed and executed. If a request R contains a locking operation, a certificate on R is called a *locking certificate* and written $L = \mathsf{cert}[R]$. Such certificate L serves as a proof that the account is locked and cannot process further account operations.

Finally, a third type of certificates associates a *coin* to an account identifier id. Section 5 introduces *anonymous coins* of the form $A = \mathsf{cert}[(\mathsf{id}, \mathsf{cm})]$ for some appropriate commitment cm on the value v of the coin. In Appendix A, we also introduce *transparent coins* $T = \mathsf{cert}[(\mathsf{id}, v, r)]$.

Accounts and unique identifiers. Zef accounts are replicated across all authorities. For a given authority α, we use the notation $X(\alpha)$ to denote the current view of α regarding some replicated data X. At a high level, Zef improves upon the notion of a FastPay account and provides the following important features:

- A Zef account is addressed by an *unique identifier* (or UID for short) designed to be non-replayable. We use id, $\mathsf{id}_1, \ldots$ to denote account identifiers.
- Every account includes an optional public key $\mathsf{pk}^{\mathsf{id}}(\alpha)$ to authenticate their owner, if any. When $\mathsf{pk}^{\mathsf{id}}(\alpha) = \perp$, the account is said to be *inactive*.
- Identifiers are created whenever the owner of an account id requests a fresh identifier, for themselves or for a third-party. In practice, we define the next available identifier as the concatenation of the account address id and its current sequence number $n = \mathsf{next\_sequence}^{\mathsf{id}}(\alpha)$.
- Zef makes it possible to safely and verifiably transfer the control of an account to another user by changing the key $\mathsf{pk}^{\mathsf{id}}(\alpha)$.
- An account can be *deactivated* by setting $\mathsf{pk}^{\mathsf{id}}(\alpha) = \perp$. This operation is final and effectively consumes the assets controlled by the account. Because identifiers id are never reused for new accounts, accounts that are deactivated may be optionally deleted by authorities to reclaim storage (see discussion in Section 4).
- In addition to the public balance $\mathsf{balance}^{\mathsf{id}}(\alpha)$, the owner of an account id may possess a number of anonymous coins $A = \mathsf{cert}[(\mathsf{id}, \mathsf{cm})]$.

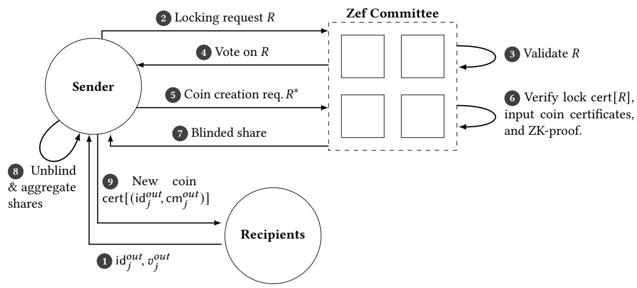

Figure 2: An anonymous payment

Sharding and cross-shard queries. In order to scale the processing of client requests, each Zef authority α may be physically divided in an arbitrary number of *shards*. Every request R sent to an account id in α is assigned a fixed shard as a public function of id and α. If a request requires a modification of another *target* account id', the shard processing the confirmation of R in α must issue an internal *cross-shard* query to the shard of id'. Cross-shard queries in Zef are asynchronous messages within each authority. They are assumed to be perfectly reliable in the sense that are they are never dropped, duplicated, or tampered with.

Transfer of anonymous coins. In Zef, anonymous coins are both (i) *unlinkable* and (ii) *opaque* in the sense that during an anonymous payment: (i) authorities cannot see or tracks users across coins being created; (ii) authorities cannot see the values behind the commitments cm of the coin being consumed or created.

Specifically, as illustrated in Figure 2, the owner of an account id may spend all the anonymous coins $A_i^{in} = \mathrm{cert}[(\mathrm{id}, \mathrm{cm}_i^{in})]$ linked to id altogether and create new anonymous coins A_j^{out} using two communication round-trips as follows:

- Provided the recipient accounts id_j^{out} and desired coin values v_j^{out} (1), compute fresh random commitments cm_j^{out} for v_j^{out} and fresh blinded messages $B_j = \mathrm{blind}((\mathrm{id}_j^{out}, \mathrm{cm}_j^{out}); u_j)$.
- Using the knowledge of the seed and coin value behind each random commitment cm_i^{in}, construct an NIZK proof π that the B_j are well-formed—in particular, that the values v_j^{out} are non-negative and have the expected sum.
- Broadcast a locking request R on the account id, containing the hash of the proof π and its public inputs A_i^{in} and B_j (2).
- Aggregate the responses from a quorum of authorities into a locking certificate $L = \mathrm{cert}[R]$.
- Broadcast a suitable request R^* containing the proof π together with L, the coins A_i^{in}, and the blinded messages B_j (5).
- Obtain signature shares from a quorum of authorities for each B_j (7), then unblind and aggregate the signatures shares to form new coins $A_j^{out} = \mathrm{cert}[(\mathrm{id}_j^{out}, \mathrm{cm}_j^{out})]$ (8).
- Communicate each new coin A_j^{out}, as well as its commitment seed and value, privately to the owner of id_j^{out} (9).

Section 5 further elaborates on the creation of coins from public balances $\mathrm{balance}^{\mathrm{id}}(\alpha)$ and supporting multiple source accounts. The Zef protocol also supports the converse operation consisting in transferring private coins into a public balance. Appendix B provides more details on an efficient cryptographic instantiation of blind signatures and NIZK proofs using the Coconut scheme [27, 30].

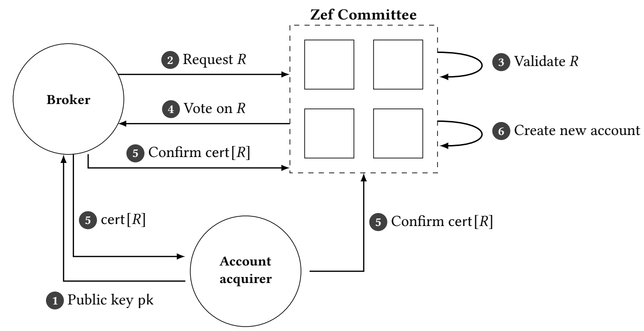

Figure 3: Request and creation of a new account

For comparison, we also describe a simplified protocol for transparent coins (i.e., without blinding and ZK-proofs) in Appendix A.

Bootstrapping account generation. In Zef, creating a new account requires interacting with the owner of an existing *parent* account. New identifiers are derived by concatenating the identifier of such a parent account with its current sequence number. This derivation ensures that account identifiers are unique while avoiding the communication overhead and the complexity of distributed random coin generation (see e.g., [11]).

While Zef lets any user derive new identifiers from an existing account that they possess, for privacy reasons, we expect certain entities to specialize in creating fresh identifiers on behalf of other users. We call them *brokers*. The role of brokers may also be assumed by authorities or delegated to third parties. In what follows, we assume that clients have a conventional way to pick an available broker and regularly create many UIDs for themselves ahead of time. The resulting interactions are summarized in Figure 3. To protect their identity, clients may also wish to interact with brokers and Zef privately, say, using Tor[2] or Nym[3] (1 and 5).

The fact that the role of broker can be delegated without risking account safety is an important property of the Zef protocol discussed in Section 4. The solution relies on the notion of certificate for account operations—here used to prove finality of account creation, initialized with an authentication key chosen by the client. In practical deployments, we expect authorities to charge a fee for account creation and brokers to forward this cost to their users plus a small margin. Discussing the appropriate pricing and means of payment is out of scope of this paper.

Finally, a Zef system must be set up with a number of *root* accounts (i.e., account without a parent). In the rest of the paper, we assume that the initial configuration of a Zef system always includes one root account id_α per authority α.

Transfers of account ownership. An interesting benefit of using unique identifiers as account addresses is that the authentication key $\mathrm{pk}^{\mathrm{id}}(\alpha)$ can be changed. Importantly, the change of key can be certified to a new owner of the account. This unlocks a number of applications:

- **Anonymous coins**. Anonymous coins (see Section 5) are defined as certificates of the form $A = \mathrm{cert}[(\mathrm{id}, \mathrm{cm})]$ for some

[2]https://www.torproject.org
[3]https://nymtech.net

commitment value cm. Spending A to create new coins is an unlinkable operation but it reveals the existence of coins controlled by the account id. Account transfers provide an alternative way to transfer anonymous coins that is linkable (and cannot divide coins) but delays revealing the existence of coins altogether.
- **Lower account-generation latency.** While transferring ownership of an account id requires the same number of messages as creating a new account, we will see in Section 4 that it only involves executing an operation within the shard of id itself (i.e., no cross-shard requests are used). Hence, brokers who wish to provide new accounts with the lowest latency may create a pool of accounts in advance then re-assign UIDs to clients as needed.

4 ACCOUNT MANAGEMENT PROTOCOL

We now describe the details of the Zef protocol when it comes to account operations. An upshot of our formalism is that it also naturally generalizes the FastPay transfer protocol [5]. Notably, in Zef, the one-time effect of a transaction consists in one of several possible operations, instead of transparent payments only. Additionally, in order to support deletion of accounts, Zef must handle the fact that a recipient account might be deleted concurrently with a transfer.

Unique identifiers. A *unique identifier* (UID) is a non-empty sequence of numbers written as $\mathsf{id} = [n_1, \dots, n_k]$ for some $1 \leq k \leq k_{\mathsf{max}}$. We use :: to denote the concatenation of one number at the end of a sequence: $[n_1, \dots, n_{k+1}] = [n_1, \dots, n_k] :: n_{k+1}$ ($k < k_{\mathsf{MAX}}$). In this example, we say that $\mathsf{id} = [n_1, \dots, n_k]$ is the *parent* of $\mathsf{id} :: n_{k+1}$. We assume that every authority α possesses at least one *root* identifier of length one: $\mathsf{id}_\alpha = [n_\alpha]$ such that the corresponding account is controlled by α at the initialization of the system (i.e., for every honest α', $\mathsf{pk}^{\mathsf{id}_\alpha}(\alpha') = \alpha$).

Protocol messages. A *message* $\langle \mathsf{Tag}, \mathsf{arg}_1, \dots, \mathsf{arg}_n \rangle$ is a sequence of values starting with a distinct marker Tag and meant to be sent over the network. In the remainder of the paper, we use capitalized names to distinguish message markers from mathematical functions (e.g. hash) or data fields (e.g. $\mathsf{pk}^{\mathsf{id}}(\alpha)$), and simply write $\mathsf{Tag}(\mathsf{arg}_1, \dots, \mathsf{arg}_n)$ for a message.

Account operations. A *regular* operation is a message O meant to be executed once on a *main* account id, with possible effects on an optional *recipient* account id'. The regular operations supported by Zef include the following messages:

- $\mathsf{OpenAccount}(\mathsf{id}', \mathsf{pk}')$ to activate a new account with a fresh identifier id' and authentication key pk'—possibly on behalf of another user who owns pk';
- $\mathsf{Transfer}(\mathsf{id}', \mathsf{value})$ to transfer value coins in a transparent way to an existing account id';
- $\mathsf{ChangeKey}(\mathsf{pk}')$ to transfer the ownership of an account;
- CloseAccount to remove the account id.

In Section 5, we introduce additional regular operations as well as a non-regular (i.e., *locking*) operation Spend that is not meant to be executed but instead locks an account until it is eventually deactivated. We introduce the operation CloseAccount now to mimic locking operations introduced later on.

Account states. The state of every authority α includes a map $\mathsf{accounts}(\alpha)$ that contains the states of the accounts present in α, indexed by their UID. The state of such an account id includes the following data:

- An optional public key $\mathsf{pk}^{\mathsf{id}}(\alpha)$ registered to control id, as seen before.
- A transparent (i.e., public) amount of coins, noted $\mathsf{balance}^{\mathsf{id}}(\alpha)$ (initially equal to $\mathsf{balance}^{\mathsf{id}}(\mathsf{init})$, where $\mathsf{balance}^{\mathsf{id}}(\mathsf{init})$ is 0 except for some special accounts created at the beginning).
- An integer value, written $\mathsf{next\_sequence}^{\mathsf{id}}(\alpha)$, tracking the expected sequence number for the next operation on id. (This value starts at 0.)
- $\mathsf{pending}^{\mathsf{id}}(\alpha)$, an optional request indicating that an operation on id is pending confirmation (the initial value being $\bot$).
- A list of certificates, written $\mathsf{confirmed}^{\mathsf{id}}(\alpha)$, tracking all the certificates C_n that have been confirmed by α for requests issued from the account id. One such certificate is available for each sequence number n ($0 \leq n < \mathsf{next\_sequence}^{\mathsf{id}}(\alpha)$).
- A second list of certificates, written $\mathsf{received}^{\mathsf{id}}(\alpha)$, tracking all the certificates that have been confirmed by α and involving id as a recipient account.

Operation safety and execution. Importantly, account operations may require some validation before being accepted. We say that an operation O is *safe* for the account id in α if one of the following conditions holds:

- $O = \mathsf{OpenAccount}(\mathsf{id}', \mathsf{pk}')$ and $\mathsf{id}' = \mathsf{id} :: \mathsf{next\_sequence}^{\mathsf{id}}(\alpha)$;
- $O = \mathsf{Transfer}(\mathsf{id}', \mathsf{value})$ and $0 \leq \mathsf{value} \leq \mathsf{balance}^{\mathsf{id}}(\alpha)$;
- $O = \mathsf{ChangeKey}(\mathsf{pk}')$ or $O = \mathsf{CloseAccount}$ (no additional verification).

When a regular operation O for an account id is confirmed (i.e. a suitable certificate C is received), we expect every authority α to *execute* the operation O in following way:

- if $O = \mathsf{OpenAccount}(\mathsf{id}', \mathsf{pk}')$, then the authority α uses a cross-shard request to set $\mathsf{pk}^{\mathsf{id}'}(\alpha) = \mathsf{pk}'$; if necessary, a new account id' is created first;
- if $O = \mathsf{Transfer}(\mathsf{id}', \mathsf{value})$, the authority subtracts value from $\mathsf{balance}^{\mathsf{id}}(\alpha)$ and uses a cross-shard request to add value to $\mathsf{balance}^{\mathsf{id}'}(\alpha)$; if necessary, the account id' is created first using an empty public key $\mathsf{pk}^{\mathsf{id}'}(\alpha) = \bot$;
- if $O = \mathsf{ChangeKey}(\mathsf{pk}')$, then the authority sets $\mathsf{pk}^{\mathsf{id}}(\alpha) = \mathsf{pk}'$;
- if $O = \mathsf{CloseAccount}$, then the authority remove the account id.

These definitions translate to the pseudo-code in Algorithm 1. The pseudo-code also includes the logging of certificates with $\mathsf{confirmed}^{\mathsf{id}}(\alpha)$ and $\mathsf{received}^{\mathsf{id}}(\alpha)$ as well as additional operations Spend and SpendAndTransfer that will be described in Section 5.

Account management protocol. We can now describe the protocol steps for executing a regular operation O on an account id:

(1) A client knowing the signing key of id and the next sequence number n signs a request $R = \mathsf{Execute}(\mathsf{id}, n, O)$ and broadcasts it to every authority in parallel, waiting for a quorum of responses.
(2) Upon receiving an authenticated request $R = \mathsf{Execute}(\mathsf{id}, n, O)$, an authority α must verify that R is authenticated for the current account key $\mathsf{pk}^{\mathsf{id}}(\alpha)$, that $\mathsf{next\_sequence}^{\mathsf{id}}(\alpha) = n$, that the operation O is safe (see above), and that $\mathsf{pending}^{\mathsf{id}}(\alpha) \in \{\bot, R\}$.

Algorithm 1 Account operations (internal functions)

```
function InitAccount(id, pk)
    pk^id ← pk
    next_sequence^id ← 0
    balance^id ← balance^id(init)  ▷ Set to 0 except for special accounts
    confirmed^id ← [ ]
    received^id ← [ ]

function ValidateOperation(id, n, O)
    switch O do
        case OpenAccount(id′, pk′):
            ensure id′ = id :: next_sequence^id
        case Transfer(id′, value):
            ensure 0 < value ≤ balance^id
        case ChangeKey(pk′) | CloseAccount:
            pass
        case Spend(value, h):
            return Lock(id, n, O)                ▷ O is valid and locking.
        case SpendAndTransfer(id′, value, σ_1 ... σ_ℓ, v_1 ... v_ℓ, r_1 ... r_ℓ):
            ensure 0 ≤ value ≤ balance^id
            for i = 1..ℓ do
                let cm_i = com_{r_i}(v_i)
                ensure cm_i ∉ {cm_k}_{k<i}
                ensure σ_i is a valid coin signature for (id, cm_i)
    return Execute(id, n, O)  ▷ If we reach this, O is valid and regular.

function ExecuteOperation(id, O, C)
    switch O do
        case OpenAccount(id′, pk′):
            do asynchronously                    ▷ Cross-shard request to id′
                run Init(id′, pk′)                ▷ Create new account
                received^id′ ← received^id′ :: C  ▷ Update receiver's log
        case Transfer(id′, value):
            balance^id ← balance^id − value  ▷ Update sender's balance
            do asynchronously                    ▷ Cross-shard request to id′
                if id′ ∉ accounts then
                    run Init(id′, ⊥)              ▷ Create receiver's account
                balance^id′ ← balance^id′ + value  ▷ Receiver's balance
                received^id′ ← received^id′ :: C
        case ChangeKey(pk′):
            pk^id ← pk′                          ▷ Update authentication key
        case CloseAccount:
            pk^id ← ⊥                            ▷ Make account inactive
        case SpendAndTransfer(id′, value, σ_1 ... σ_ℓ, v_1 ... v_ℓ, r_1 ... r_ℓ):
            pk^id ← ⊥                            ▷ Deactivate sender's account
            do asynchronously                    ▷ Cross-shard request to id′
                if id′ ∉ accounts then
                    run Init(id′, ⊥)
                balance^id′ ← balance^id′ + value + ∑_i v_i
                received^id′ ← received^id′ :: C
```

Algorithm 2 Account service (message handlers)

```
function HandleRequest(auth[R])
    let Execute(id, n, O) | Lock(id, n, O) = R    ▷ Allow regular and locking operations
    ensure pk^id ≠ ⊥                        ▷ The account must be active
    verify that auth[R] is valid for pk^id    ▷ Check authentication
    if pending^id ≠ R then
        ensure pending^id = ⊥
        ensure next_sequence^id = n
        ensure ValidateOperation(id, n, O) = R
        pending^id ← R                       ▷ Lock the account on R
    return Vote(R)         ▷ Success: return a signature of the request

function HandleConfirmation(C)
    verify that C = cert[R] is valid
    match Execute(id, n, O) = R          ▷ Allow regular operations only
    ensure pk^id ≠ ⊥                      ▷ Make sure the account is active
    if next_sequence^id = n then
        run ExecuteOperation(id, O, C)
        next_sequence^id ← n + 1            ▷ Update sequence number
        pending^id ← ⊥                  ▷ Make the account available again
        confirmed^id ← confirmed^id :: C      ▷ Log certificate
```

Then, it sets $\text{pending}^{\text{id}}(\alpha) = R$ and returns a signature on R to the client.

(3) The client aggregates signatures into a confirmation certificate $C = \text{cert}[R]$.

(4) The client (or another stakeholder) broadcasts $\text{Confirm}(C)$.

(5) Upon receiving $\text{Confirm}(C)$ for a valid certificate C of value $R = \text{Execute}(\text{id}, n, O)$ when O is a regular operation, each authority α verifies that $\text{pk}^{\text{id}}(\alpha) \neq \bot$, $\text{next\_sequence}^{\text{id}}(\alpha) = n$, then increments $\text{next\_sequence}^{\text{id}}(\alpha)$, sets $\text{pending}^{\text{id}}(\alpha) = \bot$, adds C to $\text{confirmed}^{\text{id}}(\alpha)$, and finally executes the operation O once (see above).

The corresponding pseudo-code for the service provided by each authority α is summarized in Algorithm 2. Importantly, *inactive* accounts, i.e., those accounts id satisfying $\text{pk}^{\text{id}}(\alpha) = \bot$, cannot accept any request (step (2)) or execute any confirmed operation (step (5)). The protocol to submit locking operations (such as Spend in Section 5) is similar except that there is no final execution step (5), and by convention, the request is written $R = \text{Lock}(\text{id}, n, O)$ and its certificate $L = \text{cert}[R]$. Note that step (1) above implicitly assumes that all authorities are up-to-date with all past certificates. In practice, a client may need to provide each authority with missing confirmation certificates for past sequence numbers. (See also "Liveness considerations" below.)

Agreement on account operations. When it comes to the operations executed from one account id, the Zef protocol guarantees that authorities execute the same sequence of operations in the same order. Indeed, the quorum intersection property entails that two certificates C and C' must contain a vote by a same honest authority α. If they concern the same account id and sequence number n, the verification by α in step (2) above and the increment of $\text{next\_sequence}^{\text{id}}(\alpha)$ in step (5) implies that C and C' certifies the same (safe) request R.

It is easy to see by induction on the length of $\text{id} = [n_1, \ldots, n_k]$ that each authority can only execute certified operations for a given id by following the natural sequence of sequence numbers (i.e., $\text{next\_sequence}^{\text{id}}(\alpha) = 0, 1, \ldots$). Indeed, by the induction hypothesis (resp. by construction for the base case), at most one operation of the form $O = \text{OpenAccount}(\text{id}, ..)$ can ever be executed by α on the parent account of id (resp. as part of the initial setup if id has no parent). We also note that due to the checks in step (5), no operation can be executed from id while the account id is locally absent or if $\text{pk}^{\text{id}}(\alpha) = \bot$. Account creation executed by the parent account of id is the only way for $\mathsf{pk}^{\mathsf{id}}(\alpha)$ to be updated from an empty value $\perp$. Therefore, if an account id is deleted by α due to an operation CloseAccount, it is necessarily so after OpenAccount(id, ..) was already executed once. The account id may be created again by some operation Transfer(id, value) after deletion, but since OpenAccount(id, ..) is no longer possible, $\mathsf{pk}^{\mathsf{id}}(\alpha)$ will remain empty, thus no more operations will be executed from id at this point. Therefore, due to the checks in step (5), the operations executed on id while $\mathsf{pk}^{\mathsf{id}}(\alpha) \neq \perp$ follows the natural sequence of sequence numbers.

Agreement on account states. Let α be authority and id be an account such that $\mathsf{pending}(\alpha) = \perp$ and α has not executed an operation CloseAccount on id yet. We observe that the state of id seen by α is a deterministic function of the following elements:

- the sequence of operations previously executed by α on id, that is, the content of $\mathsf{confirmed}^{\mathsf{id}}(\alpha)$, and
- the (unordered) set of operations previously executed by α that caused a cross-shard request to id as recipient, that is, the content of $\mathsf{received}^{\mathsf{id}}(\alpha)$.

Indeed, operations issued by id are of the form ChangeKey(pk), Transfer(.., value_j^{out}), and OpenAccount(..). Similarly, possible operations received by id are of the form OpenAccount(id, pk) and Transfer(id, value_i^{in}). We can determine the different components of the account id as seen by α as follows:

- $\mathsf{next\_sequence}^{\mathsf{id}}(\alpha)$ will be the size of $\mathsf{confirmed}^{\mathsf{id}}(\alpha)$;
- $\mathsf{pk}^{\mathsf{id}}(\alpha)$ will be the last key set by OpenAccount(id, pk) (or an equivalent initial setup for special accounts) then subsequent ChangeKey(pk) operations, and otherwise $\mathsf{pk}^{\mathsf{id}}(\alpha) = \perp$;
- $\mathsf{balance}^{\mathsf{id}}(\alpha) = \sum_i \mathsf{value}_i^{in} - \sum_j \mathsf{value}_j^{out} + \mathsf{balance}^{\mathsf{id}}(\mathsf{init})$, where $\mathsf{balance}^{\mathsf{id}}(\mathsf{init})$ denotes a possibly non-zero initial balance for some special accounts. (In the presentation of FastPay [5], additionally terms account for external transfers with the primary blockchain in replacement of $\mathsf{balance}^{\mathsf{id}}(\mathsf{init})$.)

The agreement property on account operations (see above) entails that whenever two honest authorities have executed the same operations, they must also agree on the current set of active accounts and their corresponding states. In other words, if for all id, $\mathsf{confirmed}^{\mathsf{id}}(\alpha) = \mathsf{confirmed}^{\mathsf{id}}(\alpha')$, then for all id such that $\mathsf{pk}^{\mathsf{id}}(\alpha) \neq \perp$ or $\mathsf{pk}^{\mathsf{id}}(\alpha') \neq \perp$, we have $\mathsf{next\_sequence}^{\mathsf{id}}(\alpha) = \mathsf{next\_sequence}^{\mathsf{id}}(\alpha')$, $\mathsf{pk}^{\mathsf{id}}(\alpha) = \mathsf{pk}^{\mathsf{id}}(\alpha')$, and $\mathsf{balance}^{\mathsf{id}}(\alpha) = \mathsf{balance}^{\mathsf{id}}(\alpha')$.

In particular, similar to the proof of FastPay [5], $\mathsf{balance}^{\mathsf{id}}(\alpha) \geq 0$ holds for every id once every certified operations has been executed. Indeed, consider an honest authority which accepted to vote at step (2) for the last transfer Transfer(.., value_j^{out}) from id.

Liveness considerations. Zef guarantees that conforming clients may always (i) initiate new valid operations on their active accounts and (ii) confirm a valid certificate of interest as a sender or as a recipient. We note that question (i) is merely about ensuring that the sequence number of an active sender account can advance after a certificate is formed at step (3). This reduces to the question (ii) of successfully executing step (5) for any honest authority, given a valid certificate C.

If the client, an honest authority α, or the network was recently faulty, it is possible that (a) the sender account id may not be active yet at α, or (b) the sequence number of id may be lagging behind compared to the expected sequence number in C. In the latter case (b), similarly to Fastpay, the client should replay the *previously confirmed certificates* C_i of the same account—defined as $C_i \in \mathsf{confirmed}^{\mathsf{id}}(\alpha')$ for some honest α'—in order to bring an authority α to the latest sequence number and confirm C. In the case (a) where id is not active yet at α, the client must confirm the creation certificate C' of id issued by the parent account $\mathsf{id}' = \mathsf{parent}(id)$. This may recursively require confirming the history of C'. Note however that this history is still sequential (i.e. there is at most one parent per account) and the number of parent creation certificates is limited by k_{MAX}.

Importantly, a certificate needs only be confirmed once per honest authority on behalf of all clients. Conforming clients who initiate transactions are expected to persist past certificates locally and pro-actively share them with all responsive authorities.

In practice, the procedure to bring authorities up-to-date can be implemented in a way that malicious authorities that would always request the entire history do not slow down the protocol. (See the discussion in FastPay [5], Section 5.)

Deletion of deactivated accounts. We have seen that once deactivated, an account id plays no role in the protocol and that id will never be active again. Therefore, it is always safe for an authority α to delete a once-deactivated account.

A simple strategy for an authority α to take advantage of this fact and reclaim *some* local storage consists in deleting the account id whenever $\mathsf{pk}^{\mathsf{id}}(\alpha)$ changes its value to $\perp$. We note however that this strategy is only a best effort. Effectively reclaiming the maximum amount of storage available in the system requires addressing two questions:

(1) If an honest authority α deletes id, how to guarantee that the account is not recreated later by α.
(2) If an honest authority α deletes id, how to guarantee that every other honest authority $\alpha' \neq \alpha$ eventually deletes id;

Regarding (1), when a cross-shard request is received for an operation Transfer(id, value), the current version of the protocol may indeed require re-creating an empty account id. This storage cost can be addressed by modifying Zef so that an authority α does not re-create id (or quickly deletes it again) if it determines that no operation $O = \mathsf{OpenAccount}(\mathsf{id}, ..)$ can occur any more. This fact can be tested in background using $|\mathsf{id}| \leq k_{\max}$ cross-shard queries. Indeed, consider the opposite fact: an inactive account $\mathsf{id} = \mathsf{id}_0 :: n$ *can become active* in α iff it holds that (i) $\mathsf{next\_sequence}^{\mathsf{id}_0}(\alpha) \leq n$ and (ii) the parent account id_0 is either active or can become active.

Regarding (2), we note that sending and receiving clients in payment operations have an incentive to fully disseminate the confirmation certificates to all authorities—rather than just a quorum of them—whenever possible. (The incentives are respectively to fully unlock the sender's account and to fully increase the receiver's balance in the eventuality of future unresponsive authorities.) However, such an incentive does not exist in the case of the CloseAccount operation (resp. the message CreateAnonymousCoins of Section 5). Therefore, in practical deployments of Zef, we expect authorities to either communicate with each other a minima in background, or to incentivize clients to continuously disseminate missing certificates (resp. missing free queries R^* of Section 5)) between authorities.

Safety of delegated account generation. In the eventuality of malicious brokers, a client must always verify the following properties before using a new account id':

- The certificate C returned by the broker is a valid certificate $C = \mathsf{cert}[R]$ such that $R = \mathsf{Execute}(\mathsf{id}, n, O)$ and $O = \mathsf{OpenAccount}(\mathsf{id}', \mathsf{pk})$ for the expected public key pk. (Under BFT assumption, this implies $\mathsf{id}' = \mathsf{id} :: n$.)
- If the client did not pick a fresh key pk, it is important to also verify that C is not being replayed.

To avoid revealing which accounts they own, clients should use a fresh authentication key pk every time and consider communicating with brokers privately (e.g. over Tor). How a client may anonymously purchase their first UID from a broker raises the interesting question of how to effectively bootstrap a fully anonymous payment system. (For instance, a certain number of fresh key-less accounts could be given away regularly for anyone to acquire and reconfigure them over Tor before receiving their very first anonymous payment.) Yet, we should point out that revealing the owner of an account does not contradict the confidentiality and the unlinkability properties of the anonymous coins described in Section 5.

Further comparison with FastPay. In FastPay, accounts are indexed by the public key pk that controls payment transfers from the account. Such a key is also called a *FastPay address*. The state of an account pk is replicated by every authority α and includes notably a balance $\mathsf{balance}^{\mathsf{pk}}(\alpha)$ and a sequence number $\mathsf{next\_sequence}^{\mathsf{pk}}(\alpha)$ used to prevent replay of payment certificates.

The definition of FastPay addresses entails that an account pk (even with balance 0) can never be removed from the system. Indeed, after the information on the sequence number $\mathsf{next\_sequence}^{\mathsf{pk}}(\alpha)$ is lost, the account owner may re-create an account for the same public key pk and exploit $\mathsf{next\_sequence}^{\mathsf{pk}}(\alpha) = 0$ to replay all past transfers originating from pk. In a context of privacy-aware applications, we note that user are less likely to re-use a same account pk many times, thus amplifying the storage impact of unused accounts.

In Zef, accounts are indexed by a UID and can be deleted, thus enabling a variety of applications (see Section 3 and 5). On the down side, new users must interact with a broker or an authority ahead of time to obtain new Zef accounts. Existing users have the additional choice to trade some privacy and derive UIDs from their existing account(s).

Cross-shared queries in both FastPay and Zef are asynchronous in the sense that they do not block a client request to confirm a certificate (see Algorithm 1). This is crucial to guarantee that an authority with a lagging view on a particular account can be brought up to date by providing missing certificate history for this account and its parents only—as opposed to exponentially many accounts. In Zef, this property results from a careful design of the protocol allowing missing recipient accounts to be (re)created with an empty public key $\mathsf{pk}^{\mathsf{id}}(\alpha) = \bot$ whenever needed. The uniqueness property of identifiers guarantees that a deleted account can never be reactivated later on.

5 ANONYMOUS PAYMENTS

We now describe the Zef protocol for anonymous payments using generic building blocks. In particular, we use a blind signature scheme, random commitments, and Zero-Knowledge (ZK) proofs in a black-box way. A more integrated realization of the protocol suitable for an efficient implementation is proposed in Appendix B.

Anonymous coins. An anonymous coin is a triplet $A = (\mathsf{id}, \mathsf{cm}, \sigma)$ where id is the unique identifier (UID) of an active account, cm is a random commitment on a value $v \in [0, v_{\max}]$ using some randomness r, denoted $\mathsf{cm} = \mathsf{com}_r(v)$, and σ is a threshold signature from a quorum of authorities on the pair $(\mathsf{id}, \mathsf{cm})$. Following the notations of Section 3, an anonymous coin A can also be seen as a certificate $A = \mathsf{cert}[(\mathsf{id}, \mathsf{cm})]$. To spend a coin, a client must know the value v, the randomness r, and the authentication key controlling id. Importantly, authorities do not need to store commitments cm or signatures σ but only manage the active accounts id. Looking ahead, spending the same coin twice will be prevented by making sure that the id is only spent once, or in other words, by removing id from the list of active accounts.

New account operation. We extend the account operations of Section 4 with a locking operation $O = \mathsf{Spend}(\mathsf{value}, \mathsf{hash}(P))$ meant to be included in a request $\mathsf{Lock}(\mathsf{id}, n, O)$ in order to prepare some payment P, withdraw value coins publicly, and eventually deactivate the corresponding account. (See details below and algorithm 1.)

Creating anonymous coins. Suppose that a user owns ℓ coins $A_i^{in} = (\mathsf{id}_i^{in}, \mathsf{cm}_i^{in}, \sigma_i^{in})$ $(1 \le i \le \ell)$ such that all id_i^{in} are registered with the authorities as active, the cm_i^{in} are ℓ mutually distinct random commitments, and σ_i^{in} is a blind rand randomizable signature on $(\mathsf{id}_i^{in}, \mathsf{cm}_i^{in})$ supporting threshold issuance [30]. Let v_i^{in} be the value of the coin A_i^{in}. Let $\mathsf{value}_i \ge 0$ be a value that the user wishes to withdraw publicly from the account id_i^{in}.

Importantly, we require commitments cm_i^{in} to be distinct but not the identifiers id_i^{in}. This allows several coins to be linked to the same account — assuming that their owner agrees to spend them simultaneously. In practice, we expect $\sum_i \mathsf{value}_i$ to be the entire balance of the set of accounts $\{\mathsf{id}_i^{in}\}$ about to be closed, to the user's knowledge.

We define the total input value of the transfer as $v = \sum_i (v_i^{in} + \mathsf{value}_i)$. To spend the coins into d new coins with values v_j^{out} $(1 \le j \le d)$ such that $\sum_j v_j^{out} = v$, the sender requests an UID id_j^{out} from each recipient, then proceeds as follows:

(1) First, the sender constructs a payment description P as follows:
   (a) For $1 \le j \le d$, sample randomness r_j^{out} and set $\mathsf{cm}_j^{out} = \mathsf{com}_{r_j^{out}}(v_j^{out})$.
   (b) For $1 \le j \le d$, sample random blinding factor u_j and let $B_j = \mathsf{blind}((\mathsf{id}_j^{out}, \mathsf{cm}_j^{out}); u_j)$.
   (c) Construct a zero-knowledge proof π for the following statement: I know v_i^{in}, r_i^{in} for each $1 \le i \le \ell$ and $v_j^{out}, r_j^{out}, u_j, \mathsf{id}_j^{out}$ for each $1 \le j \le d$ such that

- $\mathsf{cm}_i^{in} = \mathsf{com}_{r_i^{in}}(v_i^{in})$ and $\mathsf{cm}_j^{out} = \mathsf{com}_{r_j^{out}}(v_j^{out})$
- $B_j = \mathsf{blind}((\mathsf{id}_j^{out}, \mathsf{cm}_j^{out}); u_j)$
- $\sum_i v_i^{in} + \sum_i \mathsf{value}_i = \sum_j v_j^{out}$
- Each value v_i^{in} and v_j^{out} is in $[0, v_{\max}]$

(d) Let $P = (\pi, A_1^{in}, \ldots, A_\ell^{in}, \mathsf{value}_1, \ldots, \mathsf{value}_\ell, B_1 \ldots B_d)$.

(2) For every distinct id $\in \{\mathsf{id}_i^{in}\}$, the sender broadcasts an authenticated request $R_i = \mathsf{Lock}(\mathsf{id}, n, O)$ where $O = \mathsf{Spend}(\mathsf{value}, \mathsf{hash}(P))$, n is the next available sequence number for the account id, and $\mathsf{value} = \sum_{\mathsf{id}_i^{in}=\mathsf{id}} \mathsf{value}_i$.

(3) Upon receiving an authenticated request $R = \mathsf{Lock}(\mathsf{id}, n, \mathsf{Spend}(\mathsf{value}, h))$ from the owner of id, an authority α verifies that $\mathsf{next\_sequence}^{\mathsf{id}}(\alpha) = n$, $\mathsf{pending}^{\mathsf{id}}(\alpha) = \perp$, and $0 \leq \mathsf{value} \leq \mathsf{balance}^{\mathsf{id}}(\alpha)$. Then, α sets $\mathsf{pending}^{\mathsf{id}}(\alpha) = R$ and responds with a signature on R.

(4) The sender collects a quorum of signatures for each R_i sent above, thus forming a certificate $L_i = \mathsf{cert}[R_i]$ for every i. It now sends a free request $R^* = \mathsf{CreateAnonymousCoins}(P, L_1, \ldots, L_\ell)$ to all authorities and waits for a quorum of responses.

(5) Upon receiving a free request of the form $R^* = \mathsf{CreateAnonymousCoins}(P, L_1, \ldots, L_\ell)$ where

$$P = (\pi, A_1^{in}, \ldots, A_\ell^{in}, \mathsf{value}_1, \ldots, \mathsf{value}_\ell, B_1 \ldots B_d)$$

and $A_i^{in} = (\mathsf{id}_i^{in}, \mathsf{cm}_i^{in}, \sigma_i^{in})$, each authority α verifies the following:
- The values cm_i^{in} are mutually distinct.
- For every i, σ_i^{in} is a valid signature on $(\mathsf{id}_i^{in}, \mathsf{cm}_i^{in})$.
- Every L_i is a valid certificate for a request of the form $R_i = \mathsf{Lock}(\mathsf{id}, n, O)$ and such that we have $\mathsf{id} = \mathsf{id}_i^{in}$, $O = \mathsf{Spend}(\mathsf{value}, \mathsf{hash}(P))$, and $\mathsf{value} = \sum_{\mathsf{id}_k^{in}=\mathsf{id}} \mathsf{value}_k$.
- The proof π is valid.

The authority then deactivates each account id_i^{in} (if needed), and responds with d signature shares, one for each $B_j = \mathsf{blind}((\mathsf{id}_j^{out}, \mathsf{cm}_j^{out}); u_j)$.

(6) For every j, the sender finally combines the signature shares received by a quorum of authorities, then uses unblind to obtain a signature σ_j^{out} on $(\mathsf{id}_j^{out}, \mathsf{cm}_j^{out})$.

(7) The j^{th} recipient receives $(\mathsf{id}_j^{out}, \mathsf{cm}_j^{out}, v_j^{out}, r_j^{out}, \sigma_j^{out})$. She verifies that the values and UIDs are as expected, that the commitments cm_j^{out} are mutually distinct, that the signatures σ_j^{out} are valid.

We note that finality is achieved as soon as the request R^* is formed by the sender. Importantly, the operation Spend is a locking operation (Section 4), meaning that a certificate $L_i = \mathsf{cert}[R_i]$ alone is not sufficient to be executed and deactivate the account id_i^{in}. Appendix B provides a concrete instantiation of this scheme.

The corresponding pseudo-code for coin creation is presented in Algorithm 3.

Redeeming anonymous coins. Suppose that a user owns ℓ coins $A_i = (\mathsf{id}, \mathsf{cm}_i, \sigma_i)$ $(1 \leq i \leq \ell)$ attached to the same active account id. We define a new (regular) account operation

$$O = \mathsf{SpendAndTransfer}(\mathsf{id}', \mathsf{value}, \sigma_1, \ldots, \sigma_\ell, v_1, \ldots, v_\ell, r_1, \ldots, r_\ell)$$

Algorithm 3 Coin creation service

```
function HandleCoinCreationRequest(R*)
    let CreateAnonymousCoins(P, L_1, ..., L_ℓ) = R*
    let (π, A_1, ..., A_ℓ, value_1, ..., value_ℓ, B_1 ... B_d) = P
    let (id_i, cm_i, σ_i) = A_i for each i = 1..ℓ
    for i = 1..ℓ do
        ensure cm_i ∉ {cm_k}_{k<i}
        ensure σ_i = cert[(id_i, cm_i)] is a valid coin signature
        ensure L_i = cert[R_i] is a valid certificate
        match Lock(id, n, Spend(value, h)) = R_i
        ensure h = hash(P)
        ensure id = id_i
        ensure value = Σ_{id_k=id} value_k
    let v = Σ value_i
    verify the ZK-proof π on inputs (cm_1 ... cm_ℓ, v, B_1 ... B_d)
    for i = 1..ℓ do
        pk^{id_i} ← ⊥                        ▷ Deactivate sender accounts
    let s_j = SignShare(B_j) for each j = 1..d
    return (s_1, ..., s_d)   ▷ Return a blinded signature for each output
```

meant to be included in a request $R = \mathsf{Execute}(\mathsf{id}, n, O)$. Following the framework of Section 4:

- O is *safe* iff $\mathsf{id} \in \mathsf{accounts}(\alpha)$, $0 \leq \mathsf{value} \leq \mathsf{balance}^{\mathsf{id}}(\alpha)$, and every signature σ_i is a valid signature of a distinct pair $(\mathsf{id}, \mathsf{cm}_i)$ with $\mathsf{cm}_i = \mathsf{com}_{r_i}(v_i)$.
- Upon receiving a valid confirmation certificate $C = \mathsf{cert}[R]$, the execution of O consists in removing the account id and sending a cross-shard request to add the value $v = \mathsf{value} + \sum_i v_i$ to $\mathsf{balance}^{\mathsf{id}'}(\alpha)$ (possibly after creating an empty account id').

The pseudo-code for redeeming operations is presented in Algorithm 1.

Safety of the protocol. We say that an account id is *locked-or-deleted* iff if there exists a valid certificate of the form

- $L = \mathsf{cert}[R]$ where $R = \mathsf{Lock}(\mathsf{id}, n, O)$ for some locking operation O, or
- $C = \mathsf{cert}[R]$ where $R = \mathsf{Execute}(\mathsf{id}, n, O)$ such that O is an operation SpendAndTransfer or CloseAccount.

If O is a transfer operation we write $\mathsf{value}(O)$ for the value of the transfer, $\mathsf{source}(O)$ for the main account, $\mathsf{recipient}(O)$ for the recipient account. By extension, we write $\mathsf{value}(C)$ for value of a valid confirmation certificate containing such operation O. Summations over certificates range over all valid certificates for distinct requests or coins.

We define the *spendable value* of an account id as $\mathsf{spendable}^{\mathsf{id}} = 0$ if id is locked-or-deleted, and otherwise:

$$\begin{aligned} \mathsf{spendable}^{\mathsf{id}} &= \mathsf{balance}^{\mathsf{id}}(\mathsf{init}) + \sum_{\mathsf{recipient}(C)=\mathsf{id}} \mathsf{value}(C) \\ &\quad - \sum_{\mathsf{source}(C)=\mathsf{id}} \mathsf{value}(C) + \sum_{\mathsf{id}(A)=\mathsf{id}} \mathsf{value}(A) \end{aligned}$$

We note that if an authority α is perfectly up to date with the certificates related to id and the account id is not locked-or-deleted:

$$\mathsf{spendable}^{\mathsf{id}} = \mathsf{balance}^{\mathsf{id}}(\alpha) + \sum_{\mathsf{id}(A)=\mathsf{id}} \mathsf{value}(A)$$

The protocol for account operations entails that a locked-or-deleted account id can never produce new certificates C (in particular with recipient$(C) = \mathsf{id}' \neq \mathsf{id}$). Indeed, the case of account deletion was analyzed in Section 4. Similarly, in the case of a locking request $R = \mathsf{Lock}(\mathsf{id}, n, O)$, at least a quorum of authorities α have processed R. From this point on, the account id held by such authority α can only (i) process the same locking request R and return the same vote, or (ii) become deleted or inactive. By quorum intersection, no new certificate can be produced.

By inspection of the protocol, we deduce that the total spendable value over all accounts $S = \sum_{\mathsf{id}} \mathsf{spendable}^{\mathsf{id}}$ never increases during regular account operations, coin creation, and redeeming of anonymous coins:

- Regular account operations have been studied in Section 4.
- Redeeming coins with SpendAndTransfer increases the balance of a recipient but changes the source account to be locked-or-deleted, effectively "burning" at least an equivalent amount. (To this effect, note that the definition of spendable only counts distinct coins and so does the protocol.)
- Creating coins with a free request $R^* = \mathsf{CreateAnonymousCoins}(P, L_1, \ldots, L_\ell)$ requires changing all the source accounts in $L_1, \ldots, L_\ell$ to be locked-or-deleted. Importantly, L_i contains a commit of P and cannot re-used for another payment $P' \neq P$. We also note that replaying R^* produces the same coins A_j and does not increase S.

Liveness considerations. For coin creation, we used $\mathsf{hash}(P)$ instead of P in each R_i both to minimize communication and for liveness. Indeed, it is important for the initiator to keep P secret before collecting all the certificates L_i. We note that otherwise an attacker knowing P may intercept the requests R_i, use them to obtain the certificates L_i, then send R^* to kill the input accounts id_i^{in} before the sender has a chance to replay the requests R_i. Depending on the archiving policy for L_i, this may leave the sender unable to replay R^* to obtain the signature shares of step (5).

Privacy properties. The protocol to create anonymous coins guarantees the following privacy properties.

- Opacity: Except for the ZK proofs π, the coin values under the commitments cm_i^{in} and cm_j^{out} are never communicated publicly.
- Unlinkability: Assuming that the sender during coin creation is honest, authorities cannot trace back to the origin of an anonymous coin when it is spent.

Regarding unlinkability, we note indeed that the receiver information id_j^{out} and cm_j^{out} are only communicated to authorities in blinded form. Besides, after unblinding, the threshold signature σ_j does not depend on values controlled by authorities, therefore is not susceptible to tainting.

To prevent double spending, the protocol must reveal the identifiers id_i^{in} of the coins being spent. This means that the sender who initially created the coins linked to id_i^{in} must be trusted for unlinkability to hold. To mitigate this concern, it is recommended that receivers of anonymous coins re-send their coins anonymously to themselves sometime before spending their coins.

In general, the account identifiers id_j^{out} needed for anonymous transfers should be obtained ahead of time and using anonymity solution such as Tor. For simplicity, we have only described account authentication based on a transparent public key $\mathsf{pk}^{\mathsf{id}}(\alpha)$. In practice, using a random commitment of a public key and a proof of knowledge of the corresponding secret would allow using the same authentication key for several accounts without degrading privacy.

6 IMPLEMENTATION

We now sketch our prototype implementation of a multi-core, multi-shard Zef authority in Rust. Our implementation is based on the existing FastPay codebase[4] which already implemented the Byzantine reliable broadcast primitive needed for Zef. In particular, we were able to re-use modules based on Tokio[5] for asynchronous networking and cryptographic modules based on ed25519-dalek[6] for elliptic-curve-based signatures. For simplicity, data-structures in our Zef prototype are held in memory rather than persistent storage. Our prototype supports both TCP and UDP for transport. The core of Zef is idempotent to tolerate retries in case of packet loss. Each authority shard is a separate native process with its own networking and Tokio reactor core. We are open-sourcing Zef[7] along with any measurements data to enable reproducible results[8].

Cryptographic primitives for anonymous coins. We have chosen Coconut credentials [30] to implement the blind randomizable threshold-issuance signatures σ_i^{in} and σ_i^{out} of Section 5. Zero-Knowledge proofs are constructed using standard sigma protocols, made non-interactive through the Fiat-Shamir heuristic [17]. As a result, our implementation of Zef assumes the hardness of LRSW [21] and XDH [8] (required by Coconut), and the existence of random oracles [17]. Appendix B presents this protocol in details. Our implementation of Coconut is inspired from Nym's[9] and uses the curve BLS12-381 [35] as arithmetic backend.

We have implemented all range proofs using Bulletproofs [9] as they only rely on the discrete logarithm assumption (which is implied by XDH) and do not require a trusted setup. Unfortunately, we couldn't directly use Dalek's implementation of Bulletproofs[10] as it uses Ristretto [15] as arithmetic backend. Ristretto is incompatible with Coconut (which requires a pairing-friendly curve). Therefore, we have modified Dalek's implementation to use curve BLS12-381. This required significant effort as the curve operations are deeply baked into the library. Our resulting library is significantly slower than Dalek's for two reasons: operations over BLS12-381 are slower than over Ristretto, and we couldn't take advantage of the parallel formulas in the AVX2 backend present in the original library. We are open-sourcing our Bulletproof implementation over BLS12-381[11].

7 EVALUATION

We now present our evaluation of the performance of our Zef prototype based on experiments on Amazon Web Services (AWS). Our focus was to verify that (i) Zef achieves high throughput even for large committees, (ii) Zef has low latency even under high load and within a WAN, (iii) Zef scales linearly when adding more shards,

[4]https://github.com/novifinancial/fastpay
[5]https://tokio.rs
[6]https://github.com/dalek-cryptography/ed25519-dalek
[7]https://github.com/novifinancial/fastpay/tree/extensions
[8]https://github.com/novifinancial/fastpay/tree/extensions/benchmark_scripts
[9]https://github.com/nymtech/coconut
[10]https://github.com/dalek-cryptography/bulletproofs
[11]https://github.com/novifinancial/fastpay/tree/extensions/bulletproofs

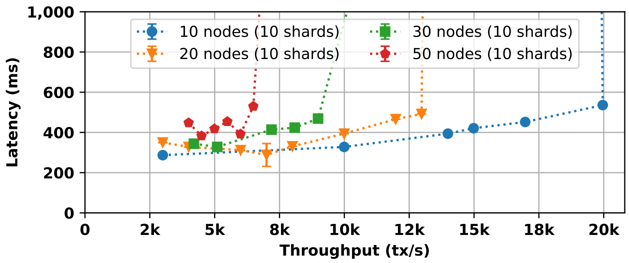

Figure 4: Throughput-latency graph for regular transfers. WAN measurements with 10, 20, 30 authorities; 10 collocated shards per authority. No faulty authorities.

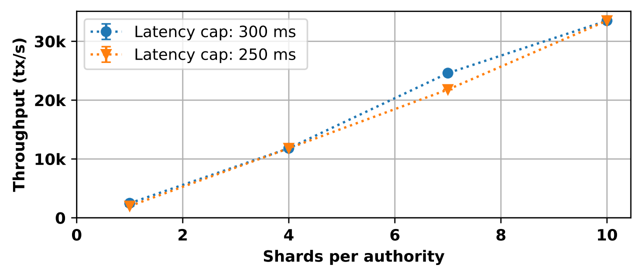

Figure 5: Maximum achievable throughput for regular transfers, keeping the latency under 250ms and 300ms. WAN measurements with 4 authorities; 1 to 10 shards per authority running on separate machines. No faulty authorities.

and (iv) Zef is robust when some parts of the system inevitably crash-fail. Note that evaluating BFT protocols in the presence of Byzantine faults is still an open research question [4].

We deployed a testbed on AWS, using `m5.8xlarge` instances across 5 different AWS regions: N. Virginia (us-east-1), N. California (us-west-1), Sydney (ap-southeast-2), Stockholm (eu-north-1), and Tokyo (ap-northeast-1). Authorities were distributed across those regions as equally as possible. Each machine provided 10Gbps of bandwidth, 32 virtual CPUs (16 physical core) on a 2.5GHz, Intel Xeon Platinum 8175, 128GB memory, and ran Linux Ubuntu server 20.04. We selected these machines because they provide decent performance and are in the price range of "commodity servers".

In the following sections, each measurement in the graphs is the average of 2 independent runs, and the error bars represent one standard deviation[12]. We set one benchmark client per shard (collocated on the same machine) submitting transactions at a fixed rate for a duration of 5 minutes.

7.1 Regular Transfers

We benchmarked the performance of Zef when making a regular transfer as described in Section 4. When referring to *latency* in this section, we mean the time elapsed from when the client submits the request (Step 1 in Figure 1) to when at least one honest authority processes the resulting confirmation certificate (Step 5 in Figure 1). We measured it by tracking sample requests throughout the system.

Benchmark in the common case. Figure 4 illustrates the latency and throughput of Zef for varying numbers of authorities. Every authority ran 10 collocated shards (each authority ran thus a single machine). The maximum throughput we observe is 20,000 tx/s for a committee of 10 nodes, and lower (up to 6,000 tx/s) for a larger committee of 50. This highlight the important of sharding to achieve high-throughput. This reduction is due to the need to transfer and check transfer certificates signed by $2f + 1$ authorities; increasing the committee size increases the number of signatures to verify since we do not use threshold signatures for regular transfers.

Scalability. Figure 5 shows the maximum throughput that can be achieved while keeping the latency under 250ms and 300ms. The committee is composed by 4 authorities each running a data-center; each shard runs on a separate machine. Figure 5 clearly supports our scalability claim: the throughput increases linearly with the number of shards, ranging from 2,500 tx/s with 1 shard per authority to 33,000 tx/s with 10 shards per authority.

[12]Error bars are absent when the standard deviation is too small to observe.

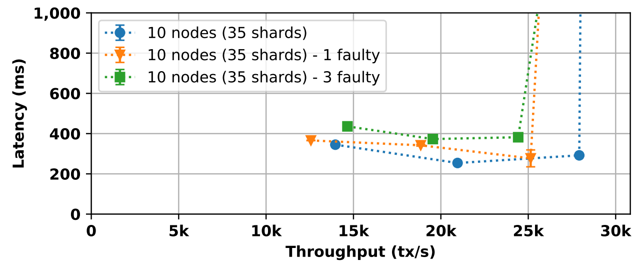

Figure 6: Throughput-latency graph for regular transfers under crash-faults. WAN measurements with 10 authorities; 35 collocated shards per authority; 0, 1, and 3 crash-faults.

Benchmark under crash-faults. Figure 6 depicts the performance of Zef when a committee of 10 authorities suffers 1 to 3 crash-faults (the maximum that can be tolerated in this setting). Every authority runs 35 collocated shards (each authority runs thus a single machine). Contrarily to BFT consensus systems [20], Zef maintains a good level of throughput under crash-faults. The underlying reason for the steady performance under crash-faults is that Zef doesn't rely on a leader to drive the protocol. The small reduction in throughput is due to losing the capacity of faulty authorities. To assemble certificates, the client is now required to wait for all the remaining $2f + 1$ authorities and can't simply select the fastest $2f + 1$ votes; this accounts for the small increase of latency. Note that the performance shown in Figure 6 are superior to those shown in Figure 4 because the authorities run more shards.

7.2 Anonymous Payments

We benchmarked the performance of Zef when spending two opaque coins into two new ones, as described in Section 5. When referring to *latency* in this section, we mean the time elapsed from when the client submits the request (Step 2 in Figure 2) to when it assembles the new coins (Step 8 in Figure 2). We measured it by tracking sample requests throughout the system.

Microbenchmarks. We report on microbenchmarks of the single-CPU core time required execute the cryptographic operations. Table 1 displays the cost of each operation in milliseconds (ms); each measurement is the result of 100 runs on a AWS `m5.8xlarge` instance. The first 3 rows respectively indicate the time to (i) produce a coin creation request meant to spend two opaque coins into two new ones, (ii) verify that request, and (iii) issue a blinded coin share.

Measure	Mean (ms)	Std. (ms)
(User) Generate coin create request	438.35	1.10
(Authority) Verify coin creation request	142.31	0.24
(Authority) Issue a blinded coin share	4.90	0.01
(User) Unblind a coin share	3.37	0.05
(User) Verify a coin share	9.62	0.04
(User) Aggregate 3 coin shares	1.70	0.00

Table 1: Microbenchmark of single core CPU costs of anonymous coin operations; average and standard dev. of 100 measurements.

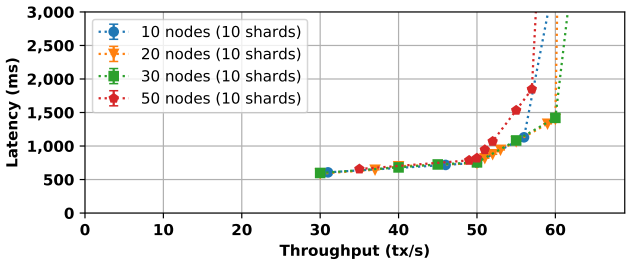

Figure 7: Throughput-latency graph for anonymous coins. WAN measurements with 10, 20, 30 authorities; 10 collocated shards per authority. No faulty authorities.

The last 3 rows indicate the time to unblind a coin share, verify it, and aggregate 3 coin shares into an output coin. The dominant CPU cost is on the user when creating a coin request (438.35ms), which involves proving knowledge of each input coins (1 Bulletproof per coin). However, verifying coin requests (142.31ms) is also expensive: it involves verifying the input coins (1 pairing check per input coin) and the output coins request (1 Bulletproof per coin). Issuing a blinded coin share (1 Coconut signature per output coin) is relatively faster (4.90ms). Unblinding (3.37ms), verifying (9.62ms) and aggregating (1.70ms) coin shares take only a few milliseconds. These results indicate that a single core shard implementation may only settle just over 7 transactions per second—highlighting the importance of sharding to achieve high-throughput.

Benchmark in the common case. Figure 7 illustrates the latency and throughput of Zef for varying numbers of authorities. Every authority runs 10 collocated shards. The performance depicted in Figure 7 (anonymous payments) are 3 order of magnitude lower than those depicted in Figure 4 (regular transfers); this is due to the expensive cryptographic operations reported in Table 1. We observe virtually no difference between runs with 10, 20, 30, or even 50 authorities: Zef can process about 50 tx/s while keeping latency under 1s in all configurations. This highlights that anonymous payments operations are extremely CPU intensive and that bandwidth is far from being the bottleneck.

Scalability. Figure 8 shows the maximum throughput that can be achieved while keeping the latency under 500ms and 1s. The committee was composed by 4 authorities each running a data-center; each shard runs on a separate machine. Figure 5 demonstrates our scalability claim: throughput increases linearly with the number of shards, ranging from 5 tx/s with 1 shard per authority to 55 tx/s with 10 shards per authority (with a latency cap of 1s).

Benchmark under crash-faults. Figure 9 depicts the performance of Zef when a committee of 10 authorities suffers 1 to 3

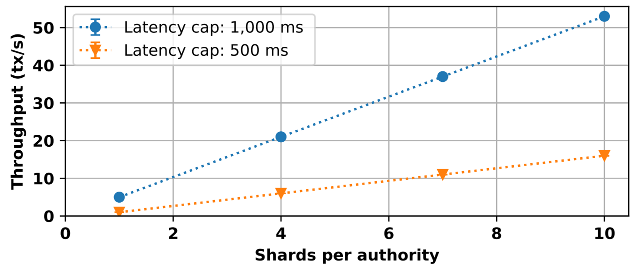

Figure 8: Maximum achievable throughput for anonymous coins while keeping the latency under 500ms and 1s. WAN measurements with 4 authorities; 1 to 10 shards per authority running on separate machines. No faulty authorities.

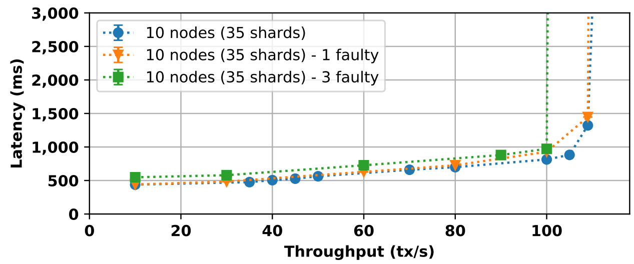

Figure 9: Throughput-latency graph for anonymous coins under crash-faults. WAN measurements with 10 authorities; 35 collocated shards per authority; 0, 1, and 3 crash-faults.

crash-faults. Every authority ran 35 collocated shards (each authority ran thus a single machine). There is no noticeable throughput drop under crash-faults, and Zef can process up to 100 tx/s within a second with 0, 1, or 3 faults. The performance of Zef shines compared to Zcash [7] which is known to process about 27 tx/s with a 1 hour latency [2]. Similarly, Monero [23] processes about 4 tx/s with a 30 minute latency [2].

8 CONCLUSION

Zef is the first linearly-scalable BFT protocol for anonymous payments with sub-second latency. Zef follows the FastPay model [5] by defining authorities as sharded services and by managing singly-owned objects using reliable broadcast rather than consensus. To support anonymous coins without sacrificing storage costs, Zef introduces a new notion of uniquely-identified, spendable account. Users can bind new anonymous coins to their accounts and spend coins in a privacy-preserving way thanks to state-of-the-art techniques such as the Coconut scheme [30].

Despite the CPU-intensive cryptographic operations required to preserve opacity and unlinkability of digital coins, our experiments confirm that anonymous payments in Zef provides unprecedentedly quick confirmation time (sub-second instead of tens of minutes) while supporting arbitrary throughput thanks to the linearly-scalable architecture.

In future work, we wish to explore applications of Zef beyond payments. To this end, one may consider generalizing account balances using Commutative Replicated Data Types (CmRDTs) [29]. Alternatively, one could introduce short-lived instances of a BFT consensus protocol whenever agreements on multi-tenant objects are needed by the system.

REFERENCES

[1] [n.d.]. https://explorer.zcha.in/statistics/usage.

[2] Alphazero. 2022. What Is The Fastest Blockchain And Why? Analysis of 43 Blockchains. https://alephzero.org/blog/what-is-the-fastest-blockchain-and-why-analysis-of-43-blockchains.

[3] Adam Back, Matt Corallo, Luke Dashjr, Mark Friedenbach, Gregory Maxwell, Andrew K. Miller, A. Poelstra, Jorge Timón, and Pieter Wuille. 2014. Enabling Blockchain Innovations with Pegged Sidechains.

[4] Shehar Bano, Alberto Sonnino, Andrey Chursin, Dmitri Perelman, and Dahlia Malkhi. 2021. Twins: BFT Systems Made Robust. In *Principles of Distributed Systems*.

[5] Mathieu Baudet, George Danezis, and Alberto Sonnino. 2020. FastPay: High-Performance Byzantine Fault Tolerant Settlement. In *ACM AFT*. 163–177.

[6] Morten L Bech and Bart Hobijn. 2006. Technology diffusion within central banking: the case of real-time gross settlement. *FRB of New York Staff Report* (2006).

[7] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. 2014. Zerocash: Decentralized Anonymous Payments from Bitcoin. In *IEEE SP*. 459–474.

[8] Dan Boneh, Ben Lynn, and Hovav Shacham. 2001. Short signatures from the Weil pairing. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 514–532.

[9] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. 2018. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy (SP)*. Ieee, 315–334.

[10] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. 2011. *Introduction to reliable and secure distributed programming*. Springer Science & Business Media.

[11] Christian Cachin, Klaus Kursawe, and Victor Shoup. 2000. Random Oracles in Constantinople: Practical Asynchronous Byzantine Agreement using Cryptography. In *PODC*. 123–132.

[12] Jan Camenisch, Susan Hohenberger, and Anna Lysyanskaya. 2005. In *EUROCRYPT*. 302–321.

[13] David Chaum. 1982. In *CRYPTO*. 199–203.

[14] George Danezis, Eleftherios Kokoris Kogias, Alberto Sonnino, and Alexander Spiegelman. 2021. Narwhal and Tusk: A DAG-based Mempool and Efficient BFT Consensus. *arXiv preprint arXiv:2105.11827* (2021).

[15] H. de Valence, J. Grigg, G. Tankersley, F. Valsorda, and I. Lovecruft. 2022. The ristretto255 Group. http://www.watersprings.org/pub/id/draft-hdevalence-cfrg-ristretto-00.html.

[16] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)* 35, 2 (1988), 288–323.

[17] Amos Fiat and Adi Shamir. 1986. How to prove yourself: Practical solutions to identification and signature problems. In *Theory and Application of Cryptographic Techniques*. Springer, 186–194.

[18] Bingyong Guo, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. 2020. Dumbo: Faster asynchronous bft protocols. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 803–818.

[19] George Kappos, Haaroon Yousaf, Mary Maller, and Sarah Meiklejohn. 2018. An Empirical Analysis of Anonymity in Zcash. In *USENIX Security*. 463–477.

[20] Hyojeong Lee, Jeff Seibert, Md. Endadul Hoque, Charles Edwin Killian, and Cristina Nita-Rotaru. 2014. Turret: A Platform for Automated Attack Finding in Unmodified Distributed System Implementations. In *ICDCS*. IEEE Computer Society, 660–669.

[21] Anna Lysyanskaya, Ronald L Rivest, Amit Sahai, and Stefan Wolf. 1999. Pseudonym systems. In *International Workshop on Selected Areas in Cryptography*. Springer, 184–199.

[22] Ian Miers, Christina Garman, Matthew Green, and Aviel D. Rubin. 2013. Zerocoin: Anonymous Distributed E-Cash from Bitcoin. In *IEEE SP*. 397–411.

[23] Monero. 2014. Monero. https://www.getmonero.org.

[24] Malte Möser, Kyle Soska, Ethan Heilman, Kevin Lee, Henry Heffan, Shashvat Srivastava, Kyle Hogan, Jason Hennessey, Andrew Miller, Arvind Narayanan, and Nicolas Christin. 2018. An Empirical Analysis of Traceability in the Monero Blockchain. *Proc. Priv. Enhancing Technol.* 2018, 3 (2018), 143–163.

[25] David Pointcheval and Olivier Sanders. 2016. Short Randomizable Signatures. In *CT-RSA*. 116–126.

[26] Joseph Poon and Thaddeus Dryja. 2015. The Bitcoin lightning network. *Scalable o-chain instant payments* (2015).

[27] Alfredo Rial and Ania M Piotrowska. 2022. Security Analysis of Coconut, an Attribute-Based Credential Scheme with Threshold Issuance. *Cryptology ePrint Archive* (2022).

[28] Tomas Sander and Amnon Ta-Shma. 1999. In *CRYPTO*. 555–572.

[29] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. A comprehensive study of Convergent and Commutative Replicated Data Types. (2011).

[30] Alberto Sonnino, Mustafa Al-Bassam, Shehar Bano, Sarah Meiklejohn, and George Danezis. 2019. Coconut: Threshold issuance selective disclosure credentials with applications to distributed ledgers. In *NDSS*.

[31] Chrysoula Stathakopoulou, Tudor David, Matej Pavlovic, and Marko Vukolić. 2019. Mir-BFT: High-Throughput Robust BFT for Decentralized Networks. *arXiv preprint arXiv:1906.05552* (2019).

[32] Brent Waters. 2005. Efficient identity-based encryption without random oracles. In *EUROCRYPT*. 114–127.

[33] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151 (2014), 1–32.

[34] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. 347–356.

[35] S. Yonezawa, Lepidum, S. Chikara, NTT TechnoCross, T. Kobayashi, and T. Saito. 2022. Pairing-Friendly Curves. https://tools.ietf.org/id/draft-yonezawa-pairing-friendly-curves-00.html.

[36] ZCash. 2016. ZCash. https://z.cash.

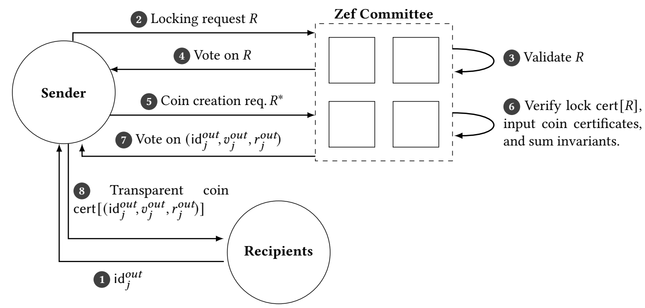

Figure 10: A payment with transparent coins

A TRANSPARENT COINS

For comparison purposes, we sketch a simplified version of anonymous coins (Section 5) without opacity and unlinkability. At a high level, the protocol is similar to anonymous coins in terms of communication (Figure 10). Due to the absence of blinding and random commitments, communication channels and validators must be trusted for the privacy of every coin operation.

Transparent coins. A *transparent coin* is a certificate $T = \mathsf{cert}[S]$ on a triplet $S = (\mathsf{id}, v, r)$ where id is the UID of an active account, $v \in [0, v_{\max}]$, and r is some random seed value.

Seed values r are used to distinguish coins of the same value attached to the same id. In what follows, we identify certificates with the same content, that is: $\mathsf{cert}[S] = \mathsf{cert}[S']$ iff $S = S'$.

To spend a transparent coin T, a client must possess the authentication key controlling id. Importantly, authorities do not need to store T themselves—although they will observe such certificates occasionally in clear.

New account operation. Similar to Section 5, we assume a locking account operation $O = \mathsf{Spend}(\mathsf{value}, \mathsf{hash}(P))$ meant to prepare some payment P (see below), withdraw value coins publicly, and eventually destroy the corresponding account.

Transparent coin payment protocol. Suppose that a user owns ℓ mutually distinct transparent coins $T_i^{in} = \mathsf{cert}[S_i^{in}]$ where $S_i^{in} = (\mathsf{id}_i^{in}, v_i^{in}, r_i^{in})$ $(1 \leq i \leq \ell)$. Let $\mathsf{value}_i \geq 0$ be a value that the user wishes to withdraw publicly from the account id_i^{in}.

Similar to Section 5, we require certificates T_i^{in} to be distinct but not the UIDs id_i^{in}. In practice, we expect $\sum_i \mathsf{value}_i$ to be the entire balance of the set of accounts $\{\mathsf{id}_i^{in}\}$.

We define the total input value of the transfer as $v = \sum_i (v_i^{in} + \mathsf{value}_i)$. To spend the coins into d new coins with values v_j^{out} (1 ≤

$j \le d$) such that $\sum_j v_j^{out} = v$, the sender requests an UID id_j^{out} from each recipient, then proceeds as follows:

(1) First, the sender constructs a payment description P as follows:
  (a) For every $1 \le j \le d$, sample randomness r_j^{out}.
  (b) Let $P = (T_1^{in}, \dots, T_\ell^{in}, \mathsf{value}_1, \dots, \mathsf{value}_\ell, S_1^{out}, \dots, S_d^{out})$ where $S_j^{out} = (\mathsf{id}_j^{out}, v_j^{out}, r_j^{out})$.
(2) For every distinct id_i^{in}, the sender broadcasts an authenticated request $R_i = \mathsf{Lock}(\mathsf{id}_i^{in}, n_i, O)$ where $O = \mathsf{Spend}(\mathsf{value}, \mathsf{hash}(P))$, n_i is the next available sequence number for the account id_i^{in}, and $\mathsf{value} = \sum_{\mathsf{id}_k^{in} = \mathsf{id}_i^{in}} \mathsf{value}_k$.
(3) Upon receiving an authenticated request $R = \mathsf{Lock}(\mathsf{id}, n, \mathsf{Spend}(\mathsf{value}, h))$ from the owner of id, an authority α verifies that $\mathsf{next\_sequence}^{\mathsf{id}}(\alpha) = n$, $\mathsf{pending}^{\mathsf{id}}(\alpha) = \bot$, and $0 \le \mathsf{value} \le \mathsf{balance}^{\mathsf{id}}(\alpha)$. Then, α sets $\mathsf{pending}^{\mathsf{id}}(\alpha) = R$ and responds with a signature on R.
(4) The sender collects a quorum of signatures for each R_i sent above, thus forming a locking certificate $L_i = \mathsf{cert}[R_i]$ for every i. It now sends a free request
$$R^* = \mathsf{CreateTransparentCoins}(P, L_1, \dots, L_\ell)$$
to all authorities and waits for a quorum of responses.
(5) Upon receiving a free request of the form $R^* = \mathsf{CreateTransparentCoins}(P, L_1, \dots, L_\ell)$ such that
$$P = (T_1^{in}, \dots, T_\ell^{in}, \mathsf{value}_1, \dots, \mathsf{value}_\ell, S_1^{out}, \dots, S_d^{out})$$
where $T_i^{in} = \mathsf{cert}[(\mathsf{id}_i^{in}, v_i^{in}, r_i^{in})]$ and $S_j^{out} = (\mathsf{id}_j^{out}, v_j^{out}, r_j^{out})$, each authority α verifies the following:
  - The certificates T_i^{in} are valid and mutually distinct.
  - Every L_i is a valid certificate for some request $R = \mathsf{Lock}(\mathsf{id}, n,)$ where $\mathsf{id} = \mathsf{id}_i^{in}$, $O = \mathsf{Spend}(\mathsf{value}, \mathsf{hash}(P))$, and $\mathsf{value} = \sum_{\mathsf{id}_k^{in} = \mathsf{id}_i^{in}} \mathsf{value}_k$.
  - $\sum_i v_i^{in} + \sum_i \mathsf{value}_i = \sum_j v_j^{out}$.

  The authority then destroys each account id_i^{in} (if needed) and responds with d signatures, one for each S_j^{out}.
(6) For every j, the sender finally combines a quorum of signatures on S_j^{out} into a new coin T_j^{out}.
(7) The j^{th} recipient receives $T_j^{out} = \mathsf{cert}[(\mathsf{id}_j^{out}, v_j^{out}, r_j^{out})]$. She verifies that the values and UIDs are as expected, that the random seeds r_j^{out} are mutually distinct, and that the certificates T_j^{out} are valid.

Redeeming transparent coins. Suppose that a user owns ℓ transparent coins T_i ($1 \le i \le \ell$) linked to the same active account id. We define a new account operation
$$O = \mathsf{SpendAndTransfer}(\mathsf{id}', \mathsf{value}, T_1, \dots, T_\ell)$$
meant to be included in a request $R = \mathsf{Execute}(\mathsf{id}, n, O)$ and following the framework of Section 4:

- O is *safe* iff $\mathsf{id} \in \mathsf{accounts}(\alpha)$, $0 \le \mathsf{value} \le \mathsf{balance}^{\mathsf{id}}(\alpha)$, and every T_i is a valid certificate of some distinct triplet $S_i = (\mathsf{id}, v_i, r_i)$.
- Upon receiving a valid certificate $C = \mathsf{cert}[R]$, the execution of O consists in removing the account id and sending a cross-shard request to add the value $v = \mathsf{value} + \sum_i v_i$ to $\mathsf{balance}^{\mathsf{id}'}(\alpha)$ (possibly after creating an empty account id').

B NIZK PROTOCOL

In this section, we show one possible efficient instantiation of the anonymous payment protocol from Section 5 by opening up the cryptographic primitives used. Our protocol here makes use of the Coconut threshold credential scheme [30], which is based on the work of Pointcheval and Sanders [25]. Informally, Coconut allows users to obtain credentials on messages with private attributes in a distributed setting using a threshold t out of n authorities.

B.1 Coconut++

We start by giving an overview of a suitable variant of the Coconut scheme, nicknamed Coconut++. This variant of Coconut is formally proven secure by Rial and Piotrowska [27]. At a high level, Coconut allows a user to obtain, from a threshold number of authorities, an anonymous credential on a private attribute m showing that it satisfies some application-specific predicate $\phi(m) = 1$. Later, the user can anonymously prove the validity of this credential to any entity in possession of the verification key. While the standard Coconut scheme works for a single attribute, [30] also includes an extension that allows for credentials on a list of q integer-valued attributes $\bar{m} = (m_1, \dots, m_q)$.

Below, we use the notation $\bar{X} = (X_1, \dots, X_q)$ for any list of q variables X_i ($1 \le i \le q$). The scheme Coconut++ consists of the following algorithms:

❖ $\mathsf{Setup}(1^\lambda) \to (pp)$**:** Choose groups $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T)$ of order p (a λ-bit prime) with a bilinear map $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$. Let $H : \mathbb{G}_1 \to \mathbb{G}_1$ be a secure hash function. Let $g_1, h_1, \dots, h_q$ be generators of $\mathbb{G}_1$ and let g_2 be a generator of $\mathbb{G}_2$. The system parameters are given as $pp = (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_3, p, e, H, g_1, g_2, \bar{h})$. Parameters are implicit in the remaining descriptions.

❖ $\mathsf{KeyGen}(t, n) \to (\mathsf{sk}, \mathsf{vk})$**:** Pick $q + 1$ polynomials $u, w_1, \dots, w_q$ each of degree $t - 1$ with coefficients in $\mathbb{F}_p$ and set $\mathsf{sk} = (x, \bar{y}) = \big(u(0), w_1(0), \dots, w_q(0)\big)$. Publish the verification key $\mathsf{vk} = (\bar{\gamma}, \alpha, \bar{\beta}) = (g_1^{y_1}, \dots, g_1^{y_q}, g_2^x, g_2^{y_1}, \dots, g_2^{y_q})$. Also issue to each authority $j \in \{1, \dots, n\}$, the secret key $\mathsf{sk}_j = (x_j, \bar{y}_j) = (u(j), w_1(j), \dots, w_q(j))$ and publish the corresponding verification key $\mathsf{vk}_j = (\bar{\gamma}_j, \alpha_j, \bar{\beta}_j) = (g_1^{y_{j,1}}, \dots, g_1^{y_{j,q}}, g_2^{x_j}, g_2^{y_{j,1}}, \dots, g_2^{y_{j,q}})$.

❖ $\mathsf{PrepareBlindSign}(\bar{m}, \phi) \to (\bar{r}, \Lambda)$**:** Pick a random $o \in \mathbb{F}_p$. Compute the commitment $c_{\bar{m}}$ and group element h as
$$c_{\bar{m}} = g_1^o \prod_{i=1}^{q} h_i^{m_i} \qquad \text{and} \qquad h = H(c_{\bar{m}})$$
For all $i = 1 \dots q$, pick a random $r_i \in \mathbb{F}_p$ and compute the blinded value c_i as follows:
$$c_i = h^{m_i} g_1^{r_i}$$
Output $(\bar{r}, \Lambda)$ where $\Lambda = (c_{\bar{m}}, \bar{c}, \pi_s)$ where π_s is defined as:
$$\pi_s = \mathrm{NIZK}\{(\bar{m}, o, \bar{r}) : \ \forall i, c_i = h^{m_i} g_1^{r_i} \ \wedge \ c_{\bar{m}} = g_1^o \prod_{i=1}^{q} h_i^{m_i} \wedge \ \phi(\bar{m}) = 1\}$$

- ❖ $\mathsf{BlindSign}(\mathsf{sk}_j, \Lambda, \phi) \to (\widetilde{\sigma}_j)$: The authority j parses $\Lambda = (c_{\bar{m}}, \bar{c}, \pi_s)$, and $\mathsf{sk}_j = (x_j, \bar{y}_j)$. Recompute $h = H(c_{\bar{m}})$. Verify the proof π_s using $\bar{c}, c_{\bar{m}}$ and ϕ; if the proof is valid, compute $\widetilde{s}_j = h^{x_j} \prod_{i=1}^{q} c_i^{y_{j,i}}$ and output $\widetilde{\sigma}_j = (h, \widetilde{s}_j)$; otherwise output $\perp$.
- ❖ $\mathsf{Unblind}(\widetilde{\sigma}_j, \bar{r}, \bar{\gamma}) \to (\sigma_j)$: Parse $\widetilde{\sigma}_j = (h, \widetilde{s}_j)$, let $s_j = \widetilde{s}_j \prod_{i=1}^{q} \gamma_i^{-r_i}$, and output $\sigma_j = (h, s_j)$.
  This results in $\sigma_j = (h, s_j)$ where $s_j = h^{x_j} \prod_{i=1}^{q} c_i^{y_{j,i}} \prod_{i=1}^{q} \gamma_i^{-r_i} = h^{x_j + \sum_{i=1}^{q} y_{j,i}\, m_i}$.
  This is similar to a Waters signature [32] related to the public key of each authority. Verification of partial coins is used in the implementation of Zef for clients to validate a quorum of answers received in parallel from authorities and discard erroneous values before running the aggregation step.
- ❖ $\mathsf{AggCred}(\{\sigma_j\}_{j \in J}) \to (\sigma)$: Return $\perp$ if $|J| \neq t$. Parse each σ_j as (h, s_j). Output $\sigma = (h, \prod_{j \in J} s_j^{\ell_j})$, where each ℓ_j is the Lagrange coefficient given by:

$$\ell_j = \left[\prod_{k \in I \setminus \{j\}} (0 - k)\right] \left[\prod_{k \in I \setminus \{j\}} (j - k)\right]^{-1} \bmod p$$

  This computation results in a value $\sigma = (h, h^{x + \sum_{i=1}^{q} y_i\, m_i})$ that does not depend on the set of authorities J.
- ❖ $\mathsf{ProveCred}(\mathsf{vk}, \bar{m}, \sigma, \phi') \to (\Theta, \phi')$: Parse $\sigma = (h, s)$ and $\mathsf{vk} = (\bar{\gamma}, \alpha, \bar{\beta})$. Pick at random $r, r' \in \mathbb{F}_p^2$, set $h' = h^{r'}$, $s' = s^{r'}(h')^r$, and $\sigma' = (h', s')$. Build $\kappa = \alpha\, g_2^r \prod_{i=1}^{q} \beta_i^{m_i}$. Output (Θ, ϕ'), where $\Theta = (\kappa, \sigma', \pi_v)$ and ϕ' is an application-specific predicate satisfied by $\bar{m}$, and π_v is:

$$\pi_v = \mathrm{NIZK}\{(\bar{m}, r) : \kappa = \alpha\, g_2^r \prod_{i=1}^{q} \beta_i^{m_i} \;\wedge\; \phi'(\bar{m}) = 1\}$$

- ❖ $\mathsf{VerifyCred}(\mathsf{vk}, \Theta, \phi') \to (true/false)$: Parse $\Theta = (\kappa, \sigma', \pi_v)$ and $\sigma' = (h', s')$; verify π_v using vk and ϕ'. Output $true$ if the proof verifies, $h' \neq 1$ and the bilinear evaluation $e(h', \kappa) = e(s', g_2)$ holds; otherwise output $false$.
  The bilinear evaluation is justified by the following equations:

$$e(h', \kappa) = e(h^{r'}, \alpha\, g_2^r \prod_{i=1}^{q} \beta_i^{m_i}) = e(h^{r'}, g_2^{x + r + \sum_i y_i\, m_i})$$
$$e(s', g_2) = e(s^{r'}(h')^r, g_2) = e(h^{r'(x + \sum_i y_i\, m_i)}\, h^{rr'}, g_2)$$

B.2 Anonymous Transfer Protocol

We now instantiate the anonymous transfer protocol from Section 5 using the Coconut scheme with three attributes $\bar{m} = (k, q, v)$ consisting of a key k, a random seed q, and a private coin value v. From the point of view of its owner, an opaque coin is defined as $A = (\mathsf{id}, x, q, v, \sigma)$ where id is the linked account, x is an unique index within the same account id, q is a secret random seed, v is the value of the coin, and σ denotes the Coconut credential for $k = \mathsf{hash}(\mathsf{id} :: [x])$, q and v. When a new opaque coin is created, the three attributes are hidden to authorities. The account id id and the index x of a coin are revealed when it is spent to verify coin ownership and prevent double-spending of coins within the same account. We use the third attribute q to guarantee the privacy of the value v even after k is revealed[13].

Suppose that a sender owns ℓ input coins $A_i^{in} = (\mathsf{id}_i^{in}, x_i^{in}, q_i^{in}, v_i^{in}, \sigma_i^{in})$ $(1 \leq i \leq \ell)$ and wishes to create d output coins of the form $(\mathsf{id}_j^{out}, x_j^{out}, q_j^{out}, v_j^{out}, \sigma_j^{out})$ $(1 \leq j \leq d)$. Let value_i^{in} denotes the public value associated with the Zef account id_i^{in} (possibly 0) as in Section 5. The sender must ensure that $\sum_i v_i^{in} + \sum_i \mathsf{value}_i^{in} = \sum_j v_j^{out}$ and that the coin indices $(\mathsf{id}_j^{out}, x_j^{out})$ are mutually distinct.

Using Coconut for opaque coin transfers. We present an overview of the changes to the anonymous transfer protocol from Section 5 to implement opaques coins. We present an intuition on how these changes use the Coconut primitives described in Appendix B.1); the next paragraphs provide detailed explanations of the opaque coin transfer protocol.

Recall that the sender first constructs a payment description P and uses it to lock the UIDs of the input coins. For this, the sender proceeds as follows. For every $1 \leq i \leq \ell$, it reveals k_i^{in}. Then, for every $1 \leq i \leq \ell$ and $1 \leq j \leq d$, it calls

$$\Theta_i \leftarrow \mathsf{ProveCred}(\mathsf{vk}, (q_i^{in}, v_i^{in}), \sigma_i^{in}, \phi')$$

and

$$((\mathsf{rk}_j, \mathsf{rq}_j, \mathsf{rv}_j), \Lambda_j) \leftarrow \mathsf{PrepareBlindSign}(k_j^{out}, q_j^{out}, v_j^{out}, \phi')$$

where ϕ' is a predicate satisfied by the input and output coin values and defined as follows: $\phi'(\bar{v}^{in}, \bar{v}^{out}) = \text{true}$ iff

$$\sum_i^l v_i^{in} + \sum_i^l \mathsf{value}_i^{in} = \sum_j^d v_j^{out} \quad \wedge \quad v_i^{out} \in [0, v_{\mathsf{max}}]$$

Effectively, the predicate ϕ' binds the NIZKs associated with all ProveCred proofs for the input coins and all PrepareBlindSign proofs for the output coins. It also shows that the value on both sides of the transfer is consistent. The payment description P reveals $\bar{\mathsf{id}}^{in}$ and $\bar{x}^{in}$ (and thus $\bar{k}^{in}$), and is now constructed as

$$P = (\bar{\mathsf{id}}^{in}, \bar{x}^{in}, \bar{\Theta}, \bar{\Lambda}, \phi', A_1^{in}, \ldots, A_l^{in}, \mathsf{value}_1, \ldots, \mathsf{value}_l).$$

Recall now that the sender uses this P to lock the input UIDs, and after retrieving a locking certificate for each UID from the authorities, it submits the request $R^* = \mathsf{CreateAnonymousCoins}(P, L_1, \ldots, L_l)$ where the L_i denote locking certificates. On receiving R^* from the sender, an authority χ now verifies the proofs $\bar{\Theta}$ and $\bar{\Lambda}$ and the predicate ϕ' by running $\mathsf{VerifyCred}(\mathsf{vk}, \Theta_i, \phi')$ for each i and $\widetilde{\sigma}_j^{out} = \mathsf{BlindSign}(\mathsf{sk}_\chi, \Lambda_j, \phi')$ for each j. If the proofs are valid, it returns $\bar{\widetilde{\sigma}}^{out}$ to the sender.

After collecting t such responses, the sender can now run Unblind and AggCred to obtain a valid credential on each created output coin. Finally, to complete the transfer, it can send the coin $(\mathsf{id}_j^{out}, x_j^{out}, q_j^{out}, v_j^{out}, \sigma_j^{out})$ to the j^{th} recipient.

[13] As noted in the original Coconut paper [30], if a credential contains a single attribute m of low entropy (such as a coin value), the verifier can run multiple times the verification algorithm making educated guesses on the value of m and effectively recover its value through brute-force.

Opaque coin construction. We present the cryptographic primitives used by the opaque coins transfer protocol. The Setup and KeyGen algorithms are exactly the same as Coconut.

❖ $\mathsf{CoinRequest}(\mathsf{vk}, \bar{\sigma}^{in}, \bar{q}^{in}, \bar{v}^{in}, \bar{k}^{out}, \bar{q}^{out}, \bar{v}^{out}, \mathsf{value}_1^{in}, \ldots, \mathsf{value}_l^{in}, \mathsf{value}_1^{out}, \ldots, \mathsf{value}_d^{out}) \rightarrow ((\bar{\mathsf{rk}}, \bar{\mathsf{rq}}, \bar{\mathsf{rv}}), \Gamma)$:
Parse $\mathsf{vk} = (\gamma_0, \gamma_1, \gamma_2, \alpha, \beta_0, \beta_1, \beta_2)$. For every input coin σ_i^{in} ($1 \leq i \leq \ell$), parse $\sigma_i^{in} = (h_i, s_i)$, pick at random $\mathsf{rh}_i, \mathsf{rs}_i \in \mathbb{F}_p^2$, and compute

$$h_i' = h_i^{\mathsf{rh}_i} \qquad \text{and} \qquad s_i' = s_i^{\mathsf{rh}_i}(h_i')^{\mathsf{rs}_i}$$

Then set $\sigma_i'^{in} = (h_i', s_i')$ and build:

$$\kappa_i = \alpha\, g_2^{\mathsf{rs}_i} \beta_1^{q_i^{in}} \beta_2^{v_i^{in}}$$

For every output coin j ($1 \leq j \leq d$), pick a random $o_j \in \mathbb{F}_p$, and compute the commitments cm_j and the group elements $\hat{h}_j$ as

$$\mathsf{cm}_j = g_1^{o_j} h_0^{k_j^{out}} h_1^{q_j^{out}} h_2^{v_j^{out}} \qquad \text{and} \qquad \hat{h}_j = H(\mathsf{cm}_j)$$

For all $1 \leq j \leq d$, pick a random $(\mathsf{rk}_j, \mathsf{rq}_j, \mathsf{rv}_j) \in \mathbb{F}_p^3$ and compute the commitments $(\mathsf{ck}_j, \mathsf{cq}_j, \mathsf{cv}_j)$ as follows:

$$\mathsf{ck}_j = \hat{h}_j^{k_j^{out}} g_1^{r_{k,j}} \quad \text{and} \quad \mathsf{cq}_j = \hat{h}_j^{q_j^{out}} g_1^{\mathsf{rq}_j} \quad \text{and} \quad \mathsf{cv}_j = \hat{h}_j^{v_j^{out}} g_1^{\mathsf{rv}_j}$$

Output $((\bar{\mathsf{rk}}, \bar{\mathsf{rq}}, \bar{\mathsf{rv}}), \Gamma)$ where $\Gamma = (\bar{\sigma}'^{in}, \bar{\kappa}, \bar{\mathsf{cm}}, \bar{\mathsf{ck}}, \bar{\mathsf{cq}}, \bar{\mathsf{cv}}, \pi_r)$ where π_r is defined as:

$$\begin{aligned}
\pi_r = \mathrm{NIZK}\{&(\bar{q}^{in}, \bar{v}^{in}, \bar{k}^{out}, \bar{q}^{out}, \bar{v}^{out}, \bar{\mathsf{rs}}, \bar{o}, \bar{r}_k, \bar{r}_q, \bar{r}_v) : \\
&\forall i, \kappa_i = \alpha\, g_2^{\mathsf{rs}_i} \beta_1^{q_i^{in}} \beta_2^{v_i^{in}} \\
&\wedge \quad \forall j, \mathsf{cm}_j = g_1^{o_j} h_0^{k_j^{out}} h_1^{q_j^{out}} h_2^{v_j^{out}} \\
&\wedge \quad \forall j, \mathsf{ck}_j = \hat{h}_j^{k_j^{out}} g_1^{\mathsf{rk}_j} \\
&\wedge \quad \forall j, \mathsf{cq}_j = \hat{h}_j^{q_j^{out}} g_1^{\mathsf{rq}_j} \\
&\wedge \quad \forall \mathsf{cv}_j = \hat{h}_j^{v_j^{out}} g_1^{\mathsf{rv}_j} \\
&\wedge \quad \sum_i^l v_i^{in} + \sum_i^l \mathsf{value}_i^{in} = \sum_j^d v_j^{out} + \sum_j^d \mathsf{value}_j^{out} \\
&\wedge \quad v_i^{out} \in [0, v_{\mathsf{max}}] \\
\}&
\end{aligned}$$

❖ $\mathsf{IssueBlindCoin}(\mathsf{sk}_\chi, \mathsf{vk}, \Gamma, \bar{k}^{in}, \mathsf{value}_1^{in}, \ldots, \mathsf{value}_l^{in}, \mathsf{value}_1^{out}, \ldots, \mathsf{value}_d^{out}) \rightarrow (\tilde{\bar{\sigma}})$: The authority χ parses $\mathsf{sk}_\chi = (x, y_0, y_1, y_2)$, $\mathsf{vk} = (\gamma_0, \gamma_1, \gamma_2, \alpha, \beta_0, \beta_1, \beta_2)$, and $\Gamma = (\bar{\sigma}'^{in}, \bar{\kappa}, \mathsf{cm}, \bar{\mathsf{ck}}, \bar{\mathsf{cq}}, \bar{\mathsf{cv}}, \pi_r)$. Recompute $\hat{h}_j = H(\mathsf{cm}_j)$ for each $1 \leq j \leq d$.
Verify the proof π_r using Γ, $\bar{h}_*$, vk, $\mathsf{value}_1^{in}, \ldots, \mathsf{value}_l^{in}$, and $\mathsf{value}_1^{out}, \ldots, \mathsf{value}_d^{out}$. For each $1 \leq i \leq \ell$, parse $\sigma_i'^{in} = (h_i', s_i')$, verify $h_i' \neq 1$, and that the following bilinear evaluation holds:

$$e(h_i', \kappa_i + \beta_0^{k_i^{in}}) = e(s_i', g_2)$$

If one of these checks fail, stop the protocol and output $\perp$. Otherwise, compute:

$$\tilde{s}_j = \hat{h}_j^x \mathsf{ck}_j^{y_0} \mathsf{cq}_j^{y_1} \mathsf{cv}_j^{y_2}$$

and output $\tilde{\sigma}_j = (\hat{h}_j, \tilde{s}_j)$.

❖ $\mathsf{PlainVerify}(\mathsf{vk}, \sigma, k, q, v) \rightarrow$ **(`true/false`):** Parse $\sigma = (h, s)$ and $\mathsf{vk} = (\gamma_0, \gamma_1, \gamma_2, \alpha, \beta_0, \beta_1, \beta_2)$. Reconstruct $\kappa = \alpha\beta_0^k \beta_1^q \beta_2^v$. output `true` if $h \neq 1$ and $e(h, \kappa) = e(s, g_2)$; otherwise output `false`.

The user then calls AggCred and Unblind over each $\tilde{\sigma}_j$ exactly as described in Appendix B.1.