Fast, private and regulated payments in asynchronous networks

Maxence Brugeres
LTCI, Telecom Paris, Institut Polytechnique de Paris
Paris, France
Forvis Mazars
Paris, France
maxence.brugeres@telecom-paris.fr

Victor Languille
LTCI, Telecom Paris, Institut Polytechnique de Paris
France
EDF
Palaiseau, France
victor.languille@telecom-paris.fr

Petr Kuznetsov
LTCI, Telecom Paris, Institut Polytechnique de Paris
Paris, France

Hamza Zarfaoui
LTCI, Telecom Paris, Institut Polytechnique de Paris
Paris, France

ABSTRACT

We propose a decentralized asset-transfer system that enjoys *full privacy*: no party can learn the details of a transaction, except for its issuer and its recipient. Furthermore, the recipient is only aware of the amount of the transaction. Our system does not rely on consensus or synchrony assumptions, and therefore, it is *responsive*, since it runs at the actual network speed. Under the hood, every transaction creates a consumable *coin* equipped with a non-interactive zero-knowledge proof (NIZK) that confirms that the issuer has sufficient funds without revealing any information about her identity, the recipient's identity, or the payment amount. Moreover, we equip our system with a *regulatory enforcement* mechanism that can be used to regulate transfer limits or restrict specific addresses from sending or receiving funds, while preserving the system's privacy guarantees.

Finally, we report on *Paxpay*, our implementation of Fully Private Asset Transfer (FPAT) that uses the Gnark [8] library for the NIZKs. In our benchmark, Paxpay exhibits better performance than earlier proposals that either ensure only partial privacy, require some kind of network synchrony or do not implement regulation features. Our system thus reconciles privacy, responsiveness, regulation enforcement and performance.

KEYWORDS

Anonymous, Asset Transfer, Asynchronous System, Responsiveness, Byzantine-Fault Tolerant, CBDC, NIZK, Payment System, Privacy, Regulation, Scalability, zk-SNARK

1 INTRODUCTION

Bitcoin [32] revolutionized the world of finance by proposing a fully decentralized asset-transfer system that allows an open set of users to consistently exchange assets without any mutual trust. Instead of relying on a central authority, users repeatedly engage in a form of *consensus* [16, 20] to maintain a replicated ledger—an ever-growing record of all transactions. It was later observed [23, 24] that, strictly speaking, global consensus, a notoriously hard task that requires partial synchrony [11, 14, 15, 20], is not required in a large array of applications. In most common cases, when every account is maintained by a dedicated user, asset transfer can be implemented in an asynchronous, *responsive* way on top of the *reliable-broadcast* primitive [10].

Reliable broadcast is weaker than consensus as it allows the correct users to eventually agree on the *set* of issued transactions but not on their order. However, it turned out to be sufficient to prevent *double spending*, a major issue in asset-transfer systems. This observation gave rise to a series of purely asynchronous, *consensus-free* asset-transfer systems [5, 13, 30].

However, a decentralized asset-transfer system, as any replicated service, still faces a vital challenge of preserving privacy of its users. Indeed, in all the mentioned payment protocols, all transactions are inherently *public* and every user's activities are traceable.

A number of proposals addressed the issue of privacy in payment systems. Zerocash [7] was one of the first protocol to address this issue. Using NIZK (non-interactive zero-knowledge proof), it makes sure that transactions' details, such as the amount and the identities of the sender and receiver, remain unknown to other users. Hiding the amount offers *confidentiality* and hiding the identities of the sender and receiver offers *anonymity*. *Private* payment systems ought to offer both confidentiality and anonymity. Zerocash is based on a blockchain and is, thus, consensus-based. Its low transaction throughput, high latency, and high computational overhead for crafting transactions make it less attractive as a practical payment protocol. Some protocols get rid of consensus to decrease transaction latency at the expense of providing weaker anonymity guarantees [6].

Additionally, many important use cases, such as Central Bank Digital Currency (CBDC) [4], require mechanisms for *regulatory enforcement*. Several proposals describe payment systems that provide privacy and regulatory compliance, while maintaining decent performance [35, 36, 40, 41]. However, this comes with limitations: some require a synchronous network [35, 40, 41], others offer weaker anonymity guarantees [35, 40], and some rely on additional trust assumptions about system participants to ensure privacy [35, 36]. All the existing distributed asset transfer systems arbitrate a trade-off between the four following aspects: **(a) Privacy**, **(b) Model assumptions**, **(c) Regulation enforcement**, **(d) Performance**. Table 1 provides a detailed comparison of several payment systems across these dimensions. This table and its content are detailed and discussed in Section 8. Reconciling these four aspects is a complex challenge, as strengthening one aspect often weakens another. In this paper, we address this reconciliation.

We formalize the problem by introducing the *Fully Private Asset Transfer* abstraction (FPAT) that maintains conventional safety and liveness properties of a payment system (informally, no asset

is spent twice and every transaction takes effect), while at the same time making sure that no transaction's detail is leaked to a non-involved party. We then describe the *Paxpay* protocol, a FPAT implementation over an asynchronous network. Paxpay has been implemented in Golang using Gnark library [8].

Paxpay leverages several cryptographic primitives such as hash functions, blind signatures, and non-interactive zero-knowledge proofs (NIZK). As in UTXO transactions [32], to execute a transfer, the sender *spends* a set of *old coins* it has received earlier and *generates* a set of *new coins* of the same total value. Every coin is uniquely identified by its *serial number*. Every coin should be signed by a sufficiently large set (a $2/3$ *quorum*) of *validators*, dedicated parties that maintain lists of spent coins and make sure that no user spends a same coin twice. The protocol is proved to be correct in an asynchronous network, even if any number of users and less than one third of validators are *Byzantine* (deviating arbitrarily from its algorithm). The rest of the validators are considered *semi-honest* (follow its algorithm but may share any data they receive during its execution with the adversary). To ensure that the transfer is legal, several assertions have to be checked: the total value of the spent coins equals the total value of the new coins, the user that calls the transfer is the owner of the spent coins, all the coins are properly signed, etc. To provide full privacy, each of these assertions is verified in an NIZK.

We ensure that transfers are *unlinkable*: in particular, given two transfers, we cannot say that they come from the same sender, destined to the same recipient or carry the same amount. This is achieved by using a *blind* signature scheme at the validators' side and verifying these signatures within an NIZK, allowing not to reveal the coins or their signatures when spending them. To verify these signatures efficiently, we implemented the NIZK using the Groth16 [22] scheme and employed a pairing-based signature scheme [34]. We then selected a pair of elliptic curves that form a chain to instantiate these cryptographic primitives, allowing efficient verification of the signature within the NIZK (we refer to Section 7 for more details).

Furthermore, we describe and implement a regulation enforcement mechanism that can be built on top of our system. This mechanism introduces an additional coin, referred to as the *compliance coin*. At each transfer, the compliance coin of the sender must be spent with the old coins and recreated as a new compliance coin that caries the update of the transfer. To obtain an initially signed compliance coin, each user must register with a regulatory authority. We make sure that a user can only have one valid compliance coin at a time.

Our regulation framework incurs very low impact on the transaction throughput. It mainly affects the transaction proving time, which we do not consider a primary performance factor. Similar to PEReDi and PARScoin [35, 36], our approach allows setting limits on individual transfer amounts or the cumulative amount transferred by a user since they joined the system.

Regulations for CBDCs mandate stringent measures for Anti-Money Laundering (AML) and Countering the Financing of Terrorism (CFT). These measures go beyond limiting transaction amounts. PEReDi and PARScoin [35, 36] address these requirements and enable validators to reveal the details of a transfer. However, this approach relies on additional trust assumptions for the validators.

To avoid such trust assumptions, we equip users with a mechanism to generate cryptographic proofs summarizing the entirety of their transaction histories, giving them the capability to prove arbitrary statements on the histories. Additionally, we incorporate a *sanction list* mechanism, empowering the regulator to impose sanctions on specific addresses. This type of mechanism is important to comply with traditional financial sanctions emitted by the central banks or intergovernmental organizations, such as the UN. Furthermore, it is explicitly required by the European Central Bank for the digital Euro [4]. Addresses under sanctions are prevented from engaging in any interactions with other users, ensuring compliance without undermining the privacy of non-sanctioned participants.

Finally, we prove that our system performs well compared to other distributed payment systems that provide either regulatory enforcement or privacy-preserving features. In our benchmark (detailed in Section 7.2), we witness that Paxpay can process at least 8 times more transactions per second than any of theses systems under similar conditions. The benchmark on AWS EC2 instances demonstrates a throughput of 925 transactions per second. This performance gain can be explained by the use of Groth16 as a succinct NIZK with low verification cost. This comes at the cost of heavy computations required to generate a proof. The time required to create a transfer request with limited computational resources in Paxpay on a one-core benchmark is up to $7s$ for example. However, given the throughput gains, this appears to be a good trade-off. Moreover, Paxpay's design allows for pushing the throughput even further by parallelizing the validator's load on several machines.

Contribution. To sum up our contribution:

- We formalize the problem by introducing the *Fully Private Asset Transfer* abstraction (FPAT).
- We propose Paxpay, a protocol that implements a FPAT object, and prove its correctness.
- We implement Paxpay in Golang, mainly leveraging on the Gnark [8] library.
- Paxpay achieves a true reconciliation of **(a) Privacy**, **(b) Model Assumptions** (network synchrony and trust assumptions for validators), **(c) Regulation Enforcement**, and **(d) Performance**. Unlike previous works that propose trade-offs, Paxpay excels in each of these aspects simultaneously, bringing together the best of all these dimensions:
  - Regarding **Privacy**, transactions in Paxpay are *confidential*, *fully anonymous*, and *unlinkable*, ensuring the most robust privacy guarantees.
  - Regarding **Model Assumptions**, the system tolerates up to one-third Byzantine validators, with the remaining validators only required to be *semi-honest*, and does not rely on the network synchrony.
  - Regarding **Regulation**, Paxpay provides a comprehensive enforcement framework by enabling users to generate proofs about their transaction history, setting spending limits, and incorporating a sanction mechanism preventing sanctioned addresses to receive or spend funds.
  - Regarding **Performance**, our implementation outperforms all existing protocols. Also, the transaction throughput scales with the computational power of

validators. The system can thus increase its supported throughput by distributing validators across additional machines.

- Paxpay can find a variety of use cases such as a standalone payment system, a layer 2 solution on top of a blockchain (brings scalability, compliance, or privacy) or a CBDC.

Table 1: Paxpay vs. Zcash [17], Lelantus [26], Quisquis [19], Zef [6], PRCash [40], PEReDi [35] and PARScoin [36].

	Zcash	Lelantus	Quisquis	Zef	PRCash	PEReDi	PARScoin	Paxpay
PRIVACY PROPERTIES								
Confidential transfers: Yes ‖ No ‖ Partial	●	●	●	◑	◑	●	●	●
Sender-anonymous transfers: Yes ‖ No ‖ Partial	●	●	●	❍	◑	◑	◑	●
Receiver-anonymous transfers: Yes ‖ No	●	●	●	●	●	●	●	●
Unlikable transfers: Yes ‖ No	●	●	●	❍	◑	●	●	●
Anonymity strategy: Full ‖ AS (Anonymity Set)	Full	AS	AS	Full	Full	Full	Full	Full
MODEL ASSUMPTIONS								
Asynchronous network: Yes ‖ No	❍	❍	❍	●	❍	❍	●	●
Correct validators model: H (Honest) ‖ SH (Semi-Honest)	SH	SH	SH	SH	SH	H	H	SH
REGULATION FEATURES								
Limited held amount per user	❍	❍	❍	❍	❍	●	❍	❍
Limited spendable amount per tx	❍	❍	❍	❍	◑	●	●	●
Limited spendable amount in total	❍	❍	❍	❍	❍	●	●	●
Full asset tracing	❍	❍	❍	❍	❍	●	●	❍
Sanction list	❍	❍	❍	❍	❍	❍	❍	●
Provable transaction history	❍	❍	❍	❍	❍	❍	❍	●
PERFORMANCE								
Transaction throughput (tx/s)	25	N/A	N/A	88	N/A	N/A	N/A	925
Transaction latency (s)	1000	N/A	N/A	< 1	N/A	N/A	N/A	< 1
NIZK Proving time (ms)	21K	2378	2110	438	100	3100	392	6959
NIZK Verification time (ms)	9	40	251	142	96	518	159	5

Road map. The rest of the paper is organized as follows. Section 2 overviews the related work and Section 3 recalls our model assumptions. Section 4 introduces the specification of the FPAT abstraction. Section 5 presents Paxpay protocol, our FPAT implementation. Section 6 builds a regulatory enforcement on top of our protocol. Section 7 describes Paxpay implementation in Golang and discusses its performance. In Section 8, we elaborate on some choices made in our protocol design, discuss its use cases and make a comparison with other protocols. Section 9 concludes the paper.

2 RELATED WORK

Consensus-free payment systems. Guerraoui *et al.* [23] proposed a payment system that relies on the secure broadcast primitive [10, 31] instead of consensus. The key property exported by secure broadcast is *source ordering*: if two correct validators deliver two messages from the same sender, then they should deliver them in the same order. To prevent double spending, the transactions are broadcast with sequence numbers, incremented each time the user issues a new transaction. Astro [13], FastPay [5] and Zef [6] use the same idea to build a payment system without consensus, on top of an asynchronous network.

Private payment systems without regulation. Zerocash [7], Zexe [9], Monero [33], Quisqus [19] and Lelantus [26] are examples of payment systems that provide *privacy* on top of blockchains (and thus require consensus and synchronous network). They rely on NIZK or ring signatures to ensure anonymity of transactions, hiding the sender, the receiver and the amount of payments.

In Zerocash and Zexe [7, 9], users can exchange coins that are represented as secret *commitments*. Theses coins can be seen as private unspent transactions (UTXO) in Bitcoin. These commitments are organized in a shared Merkle tree. If a user wants to spend a coin, she provides an NIZK that confirms the inclusion of her commitment in the Merkle tree. Then the user reveals the *nullifier* used in the commitment, which is added to the list of spent nullifiers. This list ensures that double spending cannot take place. The users engage in consensus to agree both on the list of valid commitments and the list of nullifiers.

Monero [33] describes a payment system designed for private transactions. Like Zerocash, Monero operates on a blockchain where users manage UTXOs. However, while Zerocash conceals the sender or receiver's identity among all users of the system, Monero limits this obfuscation to a smaller group, known as the *anonymity set*. Monero achieves this through the use of ring signatures. These signatures enable any member of the anonymity set (referred to as the *ring*) to sign a transaction, confirming that she belongs to the set without disclosing her identity. To prevent double spending, the ring signature scheme incorporates a feature called *linkability*, which ensures that if the same member of the ring signs multiple transactions originating from the same ring, it can be detected.

Quisquis [19] presents another private payment system built on a blockchain. Similar to Monero, Quisquis leverages anonymity sets to hide user identities during transactions. A distinguishing feature of Quisquis is its use of *updatable keys*, which enable users to update public keys without altering the associated secret keys, so that the updated public keys are indistinguishable from ones that are freshly generated with new private keys. To initiate a payment, the sender updates the input keys and provides an NIZK that verifies, among others, the correctness of the update. This mechanism ensures that a public key cannot be used as an input more than once, effectively preventing double spending. Compared to Monero, Quisquis offers stronger privacy guarantees in certain scenarios. Indeed, there are specific transaction configurations in Monero where an attacker could potentially de-anonymize the transaction, by comparing intersections of anonymity sets. In contrast, such de-anonymizations are precluded in Quisquis thanks to the updatable public keys, although it also relies on anonymity sets.

Lelantus [26, 27] proposes a private payment system built on a blockchain, and uses anonymity sets to provide privacy. Lelantus uses NIZK to provide privacy but requires less advanced cryptographic assumptions than Zerocash. For instance, it does not require any trusted setup. In the protocol, validators can verify *batches* of proofs, which help maintain a good overall performance by limiting the average verification time of a single proof.

All payment systems mentioned so far are built on a blockchain and, thus, require consensus. Baudet et al. [6] describe Zef, a private payment system, that assumes an asynchronous network but only provides *partial* anonymity and confidentiality: when a payment is made, the receiver and the amount are hidden but the sender is known (thus, no sender-anonymity). Also, Zef does not guarantee that payments amount are entirely confidential. Indeed, when a user makes a payment, she reveals coin commitments that were previously sent by other users, and thus known by other users. These previous senders can, therefore, deduce some information about the amount of the payments being made if they recognize a spent coin that they created. To improve performance, the transaction data in Zef is distributed over a set of *authorities*. Each authority can be sharded — *i.e.* distributed over several machines as in Astro [13]. The throughput of Zef grows linearly with the number of shards for each authority, which allows it to be "arbitrarily" scalable.

Our FPAT (Fully-Private Asset Transfer) specification state is inspired by Albouy et al. [1].

Private payment systems with regulation. PRCash [40], Platypus [41], PEReDi [35], and PARScoin [36] incorporate certain regulatory features into private payment systems. A common element in their designs is the reliance on NIZK.

PRCash [40] introduced one of the first regulated private payment systems built on blockchains. Similar to some of the systems mentioned above, PRCash operates with commitments that are consumed and created during each transaction. PRCash only offers partial privacy. Like Zef, it provides only partial confidentiality, as some details about the payment amount can leak to the users who created the input commitments. Also, PRCash only ensures partial anonymity, as the sender of a payment is known to the receiver, making anonymous donations impossible. Validators can also link certain transactions, violating transaction unlinkability. Since PRCash is blockchain-based, it incorporates a concept of time defined by the number of blocks. Regulation features proposed by PRCash are limited to one: users cannot spend more than a predefined amount set by the regulator within a certain time window.

Platypus [41] introduces regulations that includes features such as holding limits, receiving limits, and spending limits. It utilizes a type of NIZK with low verification costs. However, unlike other payment systems described in this section, Platypus is a centralized payment system rather than a distributed one.

Sarencheh et al. introduced PEReDi [35] which provides a more comprehensive regulatory framework compared to PRCash. It operates as an account-based system: users have blinded accounts that are signed by a quorum of validators. Transfers are executed by creating a joint transaction between the sender ant receiver. They compute their new account state after the transfer, proving through an NIZK that the accounts are updated correctly. The regulation framework enforces spending and receiving limits for each transaction and imposes restrictions on the maximum amount a user can hold. Additionally, the system allows validators to trace funds and reveal details of certain payments by encrypting the transaction details with a threshold encryption scheme. Validators can reconstruct the transaction details using their shares of the encrypted transaction. However, this approach comes with several drawbacks: it assumes a synchronous network and provides only partial anonymity, as the receiver of a payment knows the sender. Moreover, the system tolerates fewer Byzantine validators, allowing less than $1/5$ validators to be Byzantine, compared to $1/3$ in most of other systems we discussed. Also, due to the tracing capability granted to validators, $4/5$ of the *correct* (*i.e.*, non-Byzantine) validators must also be *honest* (*i.e.*, strictly follow the protocol and avoid seeking additional information). In contrast, most systems discussed here (as well as ours) are designed to tolerate *semi-honest* correct validators. Another noticeable drawback is the high verification time of a transaction, which limits the overall transaction throughput.

Concurrently with this submission, the same authors have recently proposed PARScoin [36], a payment system designed to handle stable coins exchange in a private and regulated manner. Its construction is close to PEReDi's but mitigates some of its drawbacks. PARScoin allows to issue payments without the help of the receiver to form a transaction, which turns the interaction between the sender and the receiver asynchronous. However, PARScoin still requires the non-Byzantine validators to be strictly honest to preserve user's privacy.

The verification time of the NIZK is also reduced. Also, neither PRCash, PARScoin nor PEReDi have conducted latency tests to evaluate the system's maximum transaction throughput or assess its scalability.

3 MODEL

3.1 Participants and adversary

The system is composed of two types of participants:

- A set $\mathcal{U}$ of U *users* of the payment system.
- A set $\mathcal{V}$ of $N = 3f + 1$ *validators*.

Here f denotes the maximum number of validators that can go *Byzantine*, i.e., deviate from the protocol. A non-Byzantine (faithful following the protocol) is called *correct*. Correct validators may, however, be *semi-honest* [18]: a correct party may try to learn as much as possible from the messages they receive from other parties, which may involve colluding and pooling their views together. In contrast, *honest* validators that are not trying to learn any information. The correct participants are thus following the protocol but they may exchange information with any other participant (Byzantine or not) to learn as much as possible.

As we shall see, Byzantine participants pose challenges to all correctness aspects of our system (safety, liveness and privacy), while semi-honest participants affect only privacy.

The adversary $\mathcal{A}$ thus controls any number of users and all validators, but at most f validators can deviate from the protocol. $\mathcal{A}$ can be seen as a hybrid adversary between honest and semi-honest. We assume that $\mathcal{A}$ is *static* and *network-ignorant*: the set of Byzantine participants controlled by $\mathcal{A}$ is chosen a priori, and $\mathcal{A}$ has no information about message delay between the validators and the other participants. It can delay the messages but must eventually deliver them. Conventionally, the adversary $\mathcal{A}$ is computationally bounded. More precisely, $\mathcal{A}$ is *probabilistic polynomial-time* (PPT).

Every participant is provided with a pair of distinct public and secret addresses denoted *apk* and *ask*. These addresses are used to identify users and validators. The public addresses of the validators

are known to every participant. Otherwise, a user only needs to know the identifiers of the users she engages in transfers with.

3.2 Network and communications

Concerning the network, we assume *asynchronous* but *reliable* communication. The participants can communicate via *anonymous*, *secure*, and *asynchronous network channels*. The channels do not modify or create messages. If a correct participants sends a message to a correct one, the message is eventually received, though we assume no bounds on the communication delays. The sender's identity is not known to the receiver, but the receiver can still respond to the sender. No other participant can tell who is the sender or the receiver, or tamper with the message content. We could use a network based on Syverson et al. [38] work. We consider that the latency distribution is the same for all the channels.

3.3 Cryptographic tools

Our protocol makes use of several cryptographic tools that we list below. Due to the space constraints, we only give informal definitions and refer to [21] for more details. Also, in the algorithms mentioned below, we omit the security parameters taken as inputs. Some primitives require a setup phase, which will be handled by a trusted third party $\mathcal{T}$. Note that these setups could be carried out via MPC [18] between the validators.

(k, N)-Threshold Blind Signature Scheme . Tuple of algorithms Π_{sig}= ($\text{SETUP}_{\text{Sig}}$, BLIND, SIGN, UNBLIND, AGGREGATE, $\text{VERIFY}_{\text{Sig}}$) allowing to divide a secret key sk in n fragments $[sk_i]_{i=1}^n$ between n signers, such that valid signatures from any subset of k signers can be aggregated into a valid signature on behalf of the corresponding public key pk_{agg}. Each fragment sk_i has a corresponding public key pk_i so that a partial signature σ_i generated with sk_i can be verified with respect to pk_i. Moreover, the signers sign a blinding version $\tilde{m}$ of a message m such that no information on m can be derived from $\tilde{m}$. A valid signature with respect to m can then be computed. Signature are unforgeable, which means that no PPT adversary can forge a valid partial signature σ_i of a message m that correctly verifies against pk_i without the knowledge of sk_i. As a blind signaure, this should not also be possible for a signer to eventually make the link between the signature is has issued, and the final signature (after UNBLIND and AGGREGATE). The following details Π_{sig}:

- $\text{SETUP}_{\text{Sig}}(k, N) \rightarrow ([sk_i]_{i=1}^n, [pk_i]_{i=1}^N, pk_{agg})$: Randomised algorithm run by a trusted party. Takes as input threshold parameters (k, N) and returns a list of N signing keys $[sk_i]_{i=1}^N$ and one corresponding verification key pk_{agg}.
- $\text{BLIND}(m, b) \rightarrow (\tilde{m})$: Takes as input a message m and a blinding factor b, returns a blinded message $\tilde{m}$.
- $\text{SIGN}(\tilde{m}, sk_i) \rightarrow \tilde{\sigma_i}$: Takes as input a blinded message $\tilde{m}$ and a secret key sk_i, and returns a partial blinded signature $\tilde{\sigma_i}$.
- $\text{UNBLIND}(\tilde{m}, \tilde{\sigma}_i, b) \rightarrow \sigma_i$: Takes as input a blinded message $\tilde{m}$, the blinding factor b used to blind m and the corresponding partial blinded signature $\tilde{\sigma_i}$, and returns a valid partial signature σ_i for the original message m.
- $\text{AGGREGATE}([\sigma_i]_{i=1}^k) \rightarrow \sigma$: Takes as input a list of k signatures $[\sigma_i]_{i=1}^k$ and produce a signature σ such that σ is a valid signature on behalf of the public key pk_{agg} if and only if all σ_i are valid signatures for distinct public keys pk_i.
- $\text{VERIFY}_{\text{Sig}}(m, \sigma, pk) \rightarrow b$: Takes as input a partial or aggregated signature σ of a message m and returns a bit b of value 1 if σ is valid with respect to pk and 0 otherwise. The signature can be a partial or aggregated signature.

***Collision-Resistant* and *Preimage-Resistant* Pseudorandom Function Family [21].** A family of functions $\text{PRF} = \{\text{PRF}_s : \{0,1\}^* \rightarrow \{0,1\}^{O(|s|)}\}_s$, where s denotes a seed, computationally indistinguishable from a random function family. *Collision-Resistance* means here that it is computationally infeasible to find couples $(s, x) \neq (s', x')$ such that $\text{PRF}_s(x) = \text{PRF}_{s'}(x')$. *Preimage-Resistance* means that it is computationally infeasible, given y, to find (s, x) such that $\text{PRF}_s(x) = y$.

Non-Interactive Zero-Knowledge Proof (NIZK) [21]. A tuple of algorithms $\Pi_{\text{proof}} = (\text{SETUP}_{\text{NIZK}}, \text{PROVE}, \text{VERIFY}_{\text{NIZK}})$ allowing a *prover* to prove to a *verifier* that, given a statement defined by an NP relation $\mathcal{R}(a, b)$ and an instance public_input, she knows a witness private_input such that $\mathcal{R}$(public_input, private_input).

- $\text{SETUP}_{\text{NIZK}}(\mathcal{R}) \rightarrow \text{pp}_{\text{NIZK}}$: Randomised algorithm run by a trusted party. Takes as input a relation $\mathcal{R}$ and outputs public parameters pp_{NIZK} (also known as common reference string). These public parameters are taken as input by the two following algorithms PROVE and VERIFY, but we omit it to lighten the notation.
- $\text{PROVE}(\text{public\_input}, \text{private\_input}, \text{pp}_{\text{NIZK}}) \rightarrow \pi$: Randomised algorithm. Takes as inputs instance and witness. Outputs a proof π such that : $\mathcal{R}$(public_input, private_input).
- $\text{VERIFY}_{\text{NIZK}}(\text{public\_input}, \text{pp}_{\text{NIZK}}, \pi) \rightarrow$ b: Takes as input an instance public_input and a proof π. Outputs 1 if the proof is valid and 0 otherwise.

Incremental commitment scheme: A tuple of algorithms Π_{com} = (COM, INCR) that allows us, given a sequence of messages $[m_i]_{i=1}^n$ and a random sequence $[r_i]_{i=1}^n$, to produce a *commitment* c. The commitment is *hiding*, i.e., no information on $[m_i]_{i=1}^n$ can be derived from c without prior knowledge on $[r_i]_{i=1}^n$. The commitment is also *binding*, meaning that given a sequence of couples $[(m_i, r_i)]_{i=1}^n$ that commits to a value c, it should be computationally infeasible for a PPT adversary to compute $[(m'_i, r'_i)]_{i=1}^n$ such that $[(m'_i, r'_i)]_{i=1}^n \neq [(m_i, r_i)]_{i=1}^n$ also commits to c. The commitment scheme is also *incremental*: let us consider a sequence $[(m_i, r_i)]_{i=1}^n$, its commitment c and a couple (m', r'). Given the knowledge of c, m' and r', it is possible to compute c' that commits to $[(m_1, r_1), (m_2, r2), ..., (m_n, r_n), (m', r')]$:

- $\text{COM}([(m_i, r_i)]_{i=1}^n) \rightarrow c$ takes as input a sequence of messages and randomness and returns the corresponding commitment. COM is hiding and binding.
- $\text{INCR}(c, m', r') \rightarrow c'$ takes as input a commitment c of a sequence $[(m_i, r_i)]_{i=1}^n$, a new message m and a randomness r'. It returns c', the commitment of the sequence $[(m_1, r_1), (m_2, r2), ..., (m_n, r_n), (m', r')]$.

4 FULLY PRIVATE ASSET TRANSFER (FPAT)

State and interface. At the abstract level, the state of the FPAT object is represented as an array of U positive integer values $[v_k]_{k=1}^{U}$, one for each user, interpreted as the current balances of the users' accounts. Let $[v_k^{init}]_{k=1}^{U}$ be the *initial state* of the object. The object exports two operations: *transfer* and *balance*. Assuming that a user u invokes an operation, they are defined as follows:

- $transfer_u(v, w)/r$ takes as inputs a value v and a user identifier w. It transfers the amount v from u to w: updates a state $[v_k]_{k=1}^{U}$ to the state $[v'_k]_{k=1}^{U}$ where $v'_k = v_k + v$ if $k = w \wedge u \neq w$, $v'_k = v_k - v$ if $k = u \wedge u \neq w$ and $v'_k = v_k$ otherwise. It returns a response $r = \text{confirm}$ if it succeeds and $r = \text{fail}$ otherwise.
- $balance_u()/v_u$ takes no inputs and returns the value v_u stored at location u of the state $[v_k]_{k=1}^{U}$.

Let O be a set of FPAT *operations*—invocations of *transfer* and *balances* provided with matching responses, each associated with a distinct user. Let $transfer_u(*, *)/C$ (resp., $transfer_u(*, *)/F$) denotes a transfer operation invoked by a user u that returns *confirm* (resp. *fail*). For each user u, we define a function $total_i(O)$ as follows:

$$total_u(O) = v_u^{init} + \sum_{transfer_*(v,u)/C \in O} v \quad - \sum_{transfer_u(v,*)/C \in O} v$$

$total_u(O)$ is thus defined as the current *balance* of u after all successful transfers in O complete: the initial amount owned by u, minus all the funds sent by u plus all the funds received by u in the set of operation O.

A *sequential history* S is a totally ordered set of FPAT operations, let $\prec_S$ be this order. Let $S|u$ denotes the subsequence of S consisting of the events of user u. We say that S is *legal* if:

(1) $\forall o = transfer_u(v, w)/C \in S \quad v \leq total_u(\{o' \in S : o' \prec_S o\})$
(2) $\forall o = balance_u()/v \in S \quad v \leq total_u(\{o' \in S : o' \prec_S o\})$
(3) If an operation $balance_u()$ returning v_1 directly precedes a $transfer_u(v_2, w)$ operation in $L|_u$, and $v_2 \leq v_1$, then the transfer operation cannot return fail.

Histories and serializations. Consider an *execution* of a FPAT algorithm: a sequence of *events* produced by the algorithm, such as invocations and responses of FPAT operations, sending and receiving messages, etc. A *local history* L_i of a user i is the sequence of *operations* invoked by i in that execution. We assume that correct users are *well-formed*—they never invoke a new operation before the previous one returns, and thus if u is correct, then L_u is sequential. In case the last operation of L_u is *incomplete* (not followed by a response), we can add any matching response and get a *completion* of L_u. Now a *history* H is a vector $(L_1, L_2, ..., L_U)$ of local histories, one for each user. Notice that if the history is produced by an execution of a FPAT algorithm, only the local histories of correct users are of interest for us: a Byzantine user is not obliged to follow the protocol.

A sequential history S is a *serialization* of a history $H = (L_1, L_2, ..., L_U)$ if for each *correct* user u, there exists $\hat{L}_u$, a completion of L_u, such that $S|u = \hat{L}_u$ ($\prec_S$ respects the local order $\prec_{\hat{L}_u}$). S can be seen as a *global interpretation* of the local histories of correct users. Notice that we allow any operations to be executed by Byzantine users in S, as long as it "makes sense" to the correct ones.

FPAT-Safety. An implementation of a FPAT-object is *FPAT-Safe* if and only if, for any finite history of execution H it produces, there exists a serialisation S of H which is legal.

FPAT-Liveness. Liveness ensures that (1) all operations invoked by a correct user eventually terminates and (2) considering two correct users u and w, if w transfers money to u, then u eventually receives this money. Let us consider the existence of a global time during the execution of an implemented FPAT object. An implementation of a FPAT-object is *FPAT-Live* if:

(1) All *transfer* and *balance* operations terminate for correct users.
(2) Consider a correct user u. For all operation $transfer_*(*, u)$ completed during the execution at time t, there exists a time $t' \geq t$ such that any operation *balance* inserted at time $t'' \geq t'$ will return a value v such that:

$$v \geq \sum_{\substack{transfer_*(v_+,u) \\ \text{invoked by correct users} \\ \text{completed before time } t}} v_+ \quad - \sum_{\substack{transfer_u(v_-,*) \\ \text{completed before time } t''}} v_-$$

FPAT-privacy. We capture the privacy guarantees of FPAT through a *distinguishing game* $\mathcal{G}^{priv}$ defined as an interaction between the following entities:

- Two copies of an oracle, O_0 and O_1. Each oracle takes as input an *abstract transfer* query tx, which specifies a sender u, a receiver w, and a transferred value v. In response, the oracle outputs the execution traces of all the involved parties that would result if the sender u had executed the *transfer* operation with value v and receiver w.
- An adversary $\mathcal{A}$ trying to gain information by making abstract transfer queries.
- A challenger C, initializing the cryptographic primitives used by the oracle, that acts as an interface between $\mathcal{A}$ and copies of O.

At the beginning of the game, C samples a random bit b. Then $\mathcal{A}$ sends a pair (tx_0, tx_1) of *abstract transfers* to C. C checks the following consistency conditions: if the receiver of one of the transfer is controlled by $\mathcal{A}$, the receiver of the other transfer must also be controlled by $\mathcal{A}$, and the transferred value v must same in both transfers.

If the consistency checks succeed, C forwards tx_0 to O_0 and tx_1 to O_1. The two oracles respectively send back execution traces Tr_0 and Tr_1. The challenger C then computes $(Tr_0^{|\mathcal{A}}, Tr_1^{|\mathcal{A}})$, which are the restrictions of Tr_0 and Tr_1 to the execution traces of the parties controlled by the adversary. C provides the adversary with the couple $(Tr_b^{|\mathcal{A}}, Tr_{1-b}^{|\mathcal{A}})$. Finally, $\mathcal{A}$ outputs a guess b' about the bit b sampled by the challenger. Intuitively, *FPAT-privacy* requires that the probability $Pr(b = b')$ (i.e., the probability that $\mathcal{A}$ wins the game) is only negligibly higher than $\frac{1}{2}$. More details about the distinguishing game $\mathcal{G}^{priv}$ are given in Appendix C.3.

FPAT-privacy has important implications that can be expressed as a collection of properties. Let ϵ be a negligible function and λ

the security parameter. Consider a protocol execution, let H be its history and $\mathcal{U}_{Byz}$ the set of users controlled by the adversary, among the U users of the system. Let $o = transfer_u(v, w) \in H$ be any *transfer* operation such that u is honest. Then for each guess (u', v', w') made by an adversary with no prior knowledge, where u' is the payment sender, v' is the value, and w' is the payment recipient, the following properties hold:

(1) **Sender-anonymity**: $\mathbb{P}(u' = u) \leq \frac{1}{U - |\mathcal{U}_{Byz}|} + \epsilon(\lambda)$, i.e., the adversary only knows that the sender is not itself.
(2) **Receiver-anonymity**: If w is honest, then $\mathbb{P}(w' = w) \leq \frac{1}{U - |\mathcal{U}_{Byz}|} + \epsilon(\lambda)$, i.e., the adversary only knows that the receiver is not itself.
(3) **Confidentiality**: If w is honest, then $\mathbb{P}(v' = v) \leq \epsilon(\lambda)$, i.e.,the adversary cannot guess the amount of the transaction.

These three properties together constitute **full privacy**. Additionally, *FPAT-privacy* implies **unlinkability**, meaning that, given two transfers, no adversary can determine whether the sender (or the receiver) is the same in both transfers.

5 PAXPAY PROTOCOL

We overview our FPAT implementation below and delegate detailed algorithms and proofs of correctness to Appendices A and C.

Setup. Every user is identified by her public address *apk* that is derived from a secret address *ask* as follows: $apk = PRF_{ask}(0)$ [1]

The protocol uses a $(2f+1, N)$-threshold blind signature scheme, as defined in Section 3.3. The secret signing keys $[sk_i]_{i=1}^{N}$ are held by each of the validators. The aggregated public key is denoted by pk_{agg} The protocol will also make use of NIZK, as defined in Section 3.3. The setup for these primitives is described in Algorithm 1 in Appendix A.

Coin structure A *coin* is a tuple $\mathbf{c} = (v, apk, \rho)$ with:

- v the (integer) value of the coin.
- apk the public address of the owner of the coin.
- ρ the seed of the coin, from which the serial number is derived.

Coins are similar to unspent transactions (UTXO) in Bitcoin: to make a payment, a user "spends" *old* coins and creates *new* coins whose owners are the payment recipients.

Coin validity Using a quorum of validators signatures (inspired by Byzantine Consistent Broadcast), we decide whether a coin is valid or not. In concrete terms, a coin is valid if it has been signed by $2f + 1$ validators. During a transfer, several old coins $[\mathbf{c}_i^{old}]_{i\in[1,n]}$ are spent to create new coins $[\mathbf{c}_j^{new}]_{j\in[1,m]}$. To make sure that a coin it not spent twice, a unique *serial number* is derived from the coin's seed ρ as $sn = \mathsf{PRF}_{ask}(\rho)$. Each validator maintains a list snList of the old coins' serial numbers. The coins $[\mathbf{c}_i^{old}]_{i\in[1,n]}$ are considered spent when a quorum of $2f+1$ validators have appended the corresponding serial numbers $[sn_i^{old}]_{i\in[1,n]}$ to their snList.

Intuitively, as the computation of sn_i^{old} is using the PRF and the secret address of the payer, no party can link sn_i^{old} to ρ_i^{old} or apk_i^{old}, and thus the sender-anonymity property is preserved. Algorithm 3 initializes the balances for the initial set of users by creating coins for each of them.[2]

Local variables. User storage consists of:

- apk/ask: its public/secret address pair
- coinList $= [(\mathbf{c}_i, \sigma_i)]_i$: the set of coins owned by the user associated with the aggregated signature of each coin.
- rhoList: the list of all seed ρ corresponding to coins received by the user

Validator storage consists of:

- sk: its secret signing key
- snList: the list of serial numbers identifying spent coins.
- signedCoinList: the list of all blinded coins signed by the replica.

Transfer. Assuming a user has enough funds, she can issue payments to several recipients $[apk_j^{new}]_{j=1}^{m}$ in one transfer by sending v_j^{new} to apk_j^{new}. For more details, see Algorithm 6 in Appendix A. The user first chooses n (old) coins denoted $[\mathbf{c}_i^{old}]_{i=1}^{n}$ and their signatures $[\sigma_i^{old}]_{i=1}^{n}$ from her coinList, where $\mathbf{c}_i^{old} = (v_i^{old}, apk_i^{old}, \rho_i^{old})$, such that the total value of the old coins does not fall below the total value payed in the transfer: $\sum_{i=1}^{n} v_i^{old} \geq \sum_{j=1}^{m} v_j^{new}$. (Otherwise, the transfer operation returns fail.) The user then computes the serial number $[sn_i^{old}]_{i=1}^{n}$ of every old coin as follows: $sn_i^{old} = \mathsf{PRF}_{ask_i^{old}}(\rho_i^{old})$.

The j new coins are then produced by the user according to the values and addresses to pay: $\forall j \in [1, m]$, $\mathbf{c}_j^{new} = (v_j^{new}, apk_j^{new}, \rho_j^{new})$. ρ_j^{new} is derived as $\rho_j^{new} = \mathsf{PRF}_{\rho_{seed}}(sn_1^{old}||...||sn_n^{old}||j)$, where ρ_{seed} is sampled randomly. This binds each new coin to the old coins spent for its creation. This way, we make sure that the validators signing $\mathbf{c}^{new}$ mark the same old coins as spent. Since the value of the chosen old coins might exceed the value of the new coins, the user also produces a *redeem coin* $\mathbf{c}_{m+1}^{new} = (v_{m+1}^{new}, apk_{m+1}^{new}, \rho_{m+1}^{new})$, with $apk_{m+1}^{new} = apk$ and v_{m+1}^{new} the exceeding value. The coins $[\mathbf{c}_j^{new}]_{j=1}^{m+1}$ are then *blinded*. The blinding of $\mathbf{c}_j^{new}$ is denoted $\tilde{\mathbf{c}}_j^{new} = \mathsf{BLIND}(\mathbf{c}_j^{new}, b_j)$, where b_j is a randomly sampled blinding factor.

The sender of the payment then computes an NIZK with:

- public_input : $([sn_i^{old}]_{i=1}^{n}, [\tilde{\mathbf{c}}_j^{new}]_{j=1}^{m+1})$
- private_input : $([\mathbf{c}_i^{old}]_{i=1}^{n}, [\sigma_i^{old}]_{i=1}^{n}, [ask_i^{old}]_{i=1}^{n}, [\mathbf{c}_j^{new}]_{j=1}^{m+1}, [b_j]_{j=1}^{m+1}, \rho_{seed})$

This NIZK $\pi_{transfer}$ proves the following relations:

(1) Serial numbers $[sn_i^{old}]_{i=1}^{n}$ are correctly derived from the old coins $[\mathbf{c}_i^{old}]_{i=1}^{n}$.
(2) New coins $[\mathbf{c}_j^{new}]_{j=1}^{m+1}$ are correctly derived from the old coins $[\mathbf{c}_i^{old}]_{i=1}^{n}$.
(3) Blinded coins $[\tilde{\mathbf{c}}_j^{new}]_{j=1}^{m+1}$ are correctly derived from $[\mathbf{c}_j^{new}]_{j=1}^{m+1}$ and $[b_j]_{j=1}^{m+1}$.
(4) Signatures $[\sigma_i^{old}]_{i=1}^{n}$ of the coin $[\mathbf{c}_i^{old}]_{i=1}^{n}$ are correct.

[1] Algorithm 2 in Appendix A shows the address generation process by sampling a random *ask* and deriving the *apk*.

[2] As discussed in Section 8, depending on the use cases of the system, this algorithm might not be executed. Instead, users might freely join the system and call a Mint algorithm that provides them with new coins.

(5) Private addresses $[ask_i^{old}]_{i=1}^n$ match the public address $[apk_i^{old}]_{i=1}^n$.
(6) The sum of the values of the old coins $[\mathbf{c}_i^{old}]_{i=1}^n$ equals the sum of the values of the new coins $[\mathbf{c}_j^{new}]_{j=1}^{m+1}$.

The user can now send the NIZK $\pi_{transfer}$ along with the public inputs to all the validators. Algorithm 12 describes the algorithm run by the validators. Once a validator receives the proof:

(1) It checks that none of the $[sn_i^{old}]_{i=1}^n$ appears in its snList;
(2) It checks that the proof π is correct;
(3) If the last two conditions are fulfilled, it adds all $[sn_i^{old}]_{i=1}^n$ to snList, add all $[\tilde{\mathbf{c}}_j^{new}]_{j=1}^{m+1}$ to signedCoinList, it signs all the coins $[\tilde{\mathbf{c}}_j^{new}]_{j=1}^{m+1}$ and returns the blinded partial signatures.
(4) If any of the previous conditions is not fulfilled, the validator checks if the blinded coins $[\tilde{\mathbf{c}}_j^{new}]_{j=1}^{m+1}$ appear in signedCoinList. If they all do, then the validator still sends back the signatures. This is done because a Byzantine validator p might receive a transfer request from a user u and send it to other validators using a private channel. As a result, other validators might answer p before answering u, preventing u from receiving the signatures while the old coins are actually spent (their serial numbers would have been added to snList already).

Once the user receives $2f+1$ valid partial signatures for the coins $[\tilde{\mathbf{c}}_j^{new}]_{j=1}^{m+1}$ from $2f+1$ validators, she can unblind and aggregate them to form $m+1$ signatures $[\sigma_j]_{j=1}^{m+1}$ that are valid signatures for $[\mathbf{c}_j^{new}]_{j=1}^{m+1}$. She can now send each couple $(\mathbf{c}_j^{new}, \sigma_j^{new})$ to apk_j^{new}. The user apk_j^{new} only accepts the payment if $\rho_j^{new} \notin \text{rhoList}_{apk_j^{new}}$ and $\text{VERIFY}_{\text{Sig}}(\mathbf{c}_j^{new}, \sigma_j^{new}, pk_{agg}) == 1$.[3]

Double spending. Consider a coin $\mathbf{c}^{\text{old}}$ that has been spent. Hence, a set $\mathcal{V}_1$ of $2f+1$ validators have claimed to have added sn^{old} to their snList, and at least $f+1$ out of them are correct. If a Byzantine user tries to spend $\mathbf{c}^{\text{old}}$ again, it should collect confirmations from a set $\mathcal{V}_2$ of $2f+1$ validators that will have to add sn^{old} in their snList again. Since there are $3f+1$ validators in total, $\mathcal{V}_1$ and $\mathcal{V}_2$ have at least one correct validator in common. It must refuse to sign the second transaction because sn^{old} is already in its snList, so double spending is prevented and FPAT-Safety guaranteed (see Appendix C for the proof).

Example. Figure 1 depicts a transfer:

(1) Alice wants to pay 3 recipients with 2 coins. She generates 3 blinded coins and a matching NIZK, and sends them to the validators.
(2) Each validator checks the validity of the NIZK and that the old coins are not in its snList, and, if so, sign the blinded coins and add the serial numbers to their snList.
(3) Alice receives the signatures.
(4) For each coin: once $2f+1$ blinded partial signatures are received, Alice unblinds and then aggregates them into one aggregated signature.
(5) Alice sends the coin tuples and their signatures to their recipients.

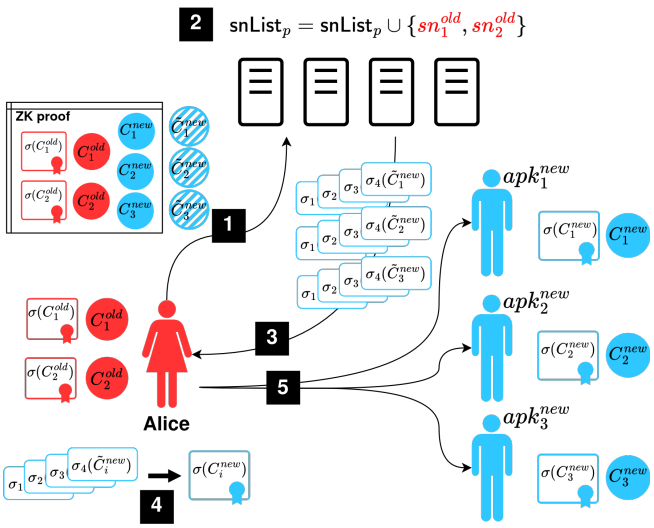


Figure 1: Paxpay example: Alice pays 3 recipients with 2 old coins

6 REGULATORY ENFORCEMENT

Paxpay is designed as a private payment system. As in PEReDi [35], PRCash [40] or Platypus [41], the protocol can be enhanced to be *regulation-compiant*. The use of succinct proofs limits the impact of regulatory enforcement on the system's performances. This section describes how to build such a regulatory enforcement on top of the protocol described above.

Our regulatory enforcement is achieved via *compliance coins*. Each user owns its unique compliance coin. The coin commits to some data related to the user and all his transactions. The user must attach the compliance coin to each of it transactions as an input and the validators will sign the updated compliance coin. Well-formedness of the updated compliance coin is proved through NIZK. As for a classical coin, the old compliance coin is spent by revealing its serial number. As a result, a user has one one valid compliance coin at a time and this coin acts as an append-only tracing mechanism.

The European Central Bank (ECB) has published requirements for the digital Euro, its future CBDC. Inspired by this, our regulatory construction enforce the following for each transfer:

- The amount transferred in the given transfer does not exceed V_{sent}^{max};
- The total amount of money transferred so far by the user does not exceed V_{total}^{max};
- The receivers of the transfer are not in the sanction list $SancList$.

V_{sent}^{max}, V_{total}^{max} and $SancList$ are public parameters chosen and potentially updated by the regulator. $SancList$ is represented as a sorted Merkle tree, which allows efficient proof of non-membership.

[3] For convenience, we allow the `Transfer` algorithm to send money to several recipients in a single call, in contrast with the specification, which allows only one recipient at a time. As explained in the proof in Appendix C, this more generic transfer can be seen as a batch of successive transfer calls, each destined to a single user.

Compliance coin The compliance coin of a user u is defined as a tuple $cc_u = (apk, \rho, v, com)$ where:

- apk is u's sole public address
- ρ the coin seed, as for a normal coin
- v is the total amount of money sent so far by u
- com is the commitment of the list of all the transfer done by u so far. For each coin created, it commits the tuple (apk, v) with apk the public key of the receiver and v the value sent. It uses the incremental commitment introduced in Section 3.3

com allows to trace u's activities and allows u to prove additional statements on her transaction history in case of an update of the regulation.

Registration Adding a user registration process is essential for regulatory enforcement, to comply with the *know-your-customer* (KYC) and *customer-due-diligence* (CDD) checks. Additionally, it ensures the uniqueness of the compliance coin for each physical user.

To this end, the validator storage is provided with a new list registeredList, the list of the enrolled users associated with their public key. registeredList is used to make sure that a physical user can only register once on the system. The registration process is as follows for the user u:

(1) Generate a random ρ and computes $cc_u = (apk_u, \rho, 0, 0)$ and its blinding $\tilde{cc}_u$ (with blinding factor b).
(2) Computes an NIZK $\pi_{register}$ that takes $(\tilde{cc}_u, apk_u)$ as public input and (ask_u, cc_u, b) as private input. It proves that:
   (a) $\tilde{cc}_u = \mathsf{BLIND}((apk_u, \rho, 0, 0), b)$.
   (b) ask_u and apk_u form a correct secret/public address pair.
(3) Send to each validator $\pi_{register}$ and $(\tilde{cc}_u, apk_u)$.
(4) Upon correct KYC and CDD proving u's identity, each validator check if $\pi_{register}$ is correct and that: $(u, *) \notin$ registeredList (u has registered no public address yet). If so, they sign $\tilde{cc}_u$, send it back to u, and add (u, apk_u) to registeredList.
(5) Once u has received $2f + 1$ partial signature, she can unblind and aggregates the partial signatures to form a valid signature for cc_u.

The detailed algorithm is described in Appendix A as Algorithm 11.

Transfer Section 5 described how a transfer is handled by a user u, by spending n old coins $[\mathbf{c}_i^{old}]_{i=1}^{n}$ and creating $m + 1$ new coins $[\tilde{\mathbf{c}}_j^{new}]_{j=1}^{m+1}$ to pay m recipients (m coins are created to the pay the recipient and a redeem coin is also created with index $m + 1$ to get the excess of money back to u). Now the user also has to update a compliance coin at each payment and prove that their transfer is compliant. The following additional steps are thus required. She generates m random values: $[r_j]_{j=1}^{m}$. The user provides an additional proof π_{comply}. This proof takes the following public inputs:

- $\tilde{cc}_u^{new}$ the blinding of the new compliance coin
- sn_{cc}^{old} the serial number of the old compliance
- V_{sent}^{max} the maximum amount that can be transferred in one transfer
- V_{total}^{max} the total amount of money that u can transfer in her use of the system
- $SancList$ the list of addresses under sanctions

π_{comply} takes the following as private input:

- $cc^{old} = (apk_{cc}^{new}, \rho_{cc}^{old}, v_{cc}^{old}, com^{old})$ the old compliance coin
- $cc^{new} = (apk_{cc}^{new}, \rho_{cc}^{new}, v_{cc}^{new}, com^{new})$ the new compliance coin
- ask_{cc}^{old} the secret address corresponding to apk_{cc}^{old}

π_{comply} should verify the following clauses:

(1) $\forall i \in [1, n], apk_i^{old} = apk_{cc}^{old}$
(2) $ask_{cc}^{old} = \mathsf{PRF}_{ask_{cc}^{old}}(0)$
(3) $apk_{m+1}^{new} = apk_{cc}^{old}$
(4) $apk_{cc}^{new} = apk_{cc}^{old}$
(5) $\rho_{cc}^{new} = \mathsf{PRF}_{\rho_{seed}}(sn_1^{old}||...||sn_n^{old}||m + 2)$
(6) $v_{cc}^{new} = v_{cc}^{old} + \sum_{1 \le j \le m} v_j^{new}$
(7) $com_0 = com^{old}$
$\wedge\ \forall j \in [1, m], com_j = \mathsf{INCR}(com_{j-1}, apk_j^{new}, v_j^{new}, r_j)$
$\wedge\ com^{new} = com_m$
(8) $\sum_{1 \le j \le m} v_j^{new} \le V_{sent}^{max}$
(9) $v_{cc}^{new} \le V_{total}^{max}$
(10) $apk_{cc}^{old} \notin SancList$
(11) $\forall j \in [1, m], apk_j^{new} \notin SancList$

Conditions (1) ensures that all the input coins belong to the owner of the input compliance coin. Condition (2) ensures that u, the user that invokes *transfer*, is the owner of the compliance coin. Condition (3) ensures that the redeem coin will belong to u. Conditions (4), (5), (6) and (7) ensure that the new compliance coin is well-formed. Condition (8), (9) and (10) ensure that the transfer is compliant. (8) verifies that the amount of the transfer does not exceed that maximum allowed for a single transfer. (9) verifies that the total amount of money sent by u does not exceed that maximum amount. (10) verifies that the sender is not a user under sanctions. (11) verifies that none of the receivers of the transfer is under sanction.

The user storage is also augmented with a list comList, in which she stores the tuples (apk, v, r) that she has used at each transfer. At the end of the transfer described above, she thus append to comList the following: $\forall j, j \in [1, m], (apk_j^{new}, v_j^{new}, r_j)$. She can later use comList and her compliance coin to prove any statement about her transaction.

Upon receiving correct proofs $(\pi_{comply}, \pi_{transfer})$ and their public parameters, the validators execute the same algorithm as for a non regulated transfer, except that they verify both π_{comply} and $\pi_{transfer}$. They can then sign the news coins and then send the signature back.

The detailed algorithm is described in Appendix A as Algorithm 9.

7 PAXPAY IMPLEMENTATION AND PERFORMANCE

Paxpay implementation in Golang is available on Github. We came up with two implementations, with and without regulation support. Below we overview the cryptographic tools we employed and report on the performance of our protocols.

7.1 Cryptographic building blocks

NIZK. NIKZ proofs are implemented using the Groth16 [22] protocol. Groth16 allows for constructing succinct non-interactive arguments of knowledge (SNARKs) with constant verification complexity, regardless of the complexity of the relation $\mathcal{R}$ it attests to. We instantiate this scheme with the BW6-761 [25] curve, designed to cycle with the BL12-377 [25] curve, allowing for efficient pairing verification over BL12-377 within the SNARK.

Groth16 is *weak simulation extractable* [3] and it allows us to prove the satisfiability of an arithmetic circuit that reflects the relation $\mathcal{R}$. It requires trusted setup that is specific to this arithmetic circuit. When it comes to proving relations on transfers with n old coins and m new coins, in building the circuit, we need to choose upper bounds for n and m. If a transfer involves more coins than these limits, a different circuit, and thus a new trusted setup, would be necessary. Adopting another SNARK protocol with a universal trusted setup, such as Marlin [12], would allow transfers with an arbitrary number of coins using a single trusted setup.

In the regulated version, each transfer requires two NIZK. For performance reasons, these two proofs $\pi_{transfer}$ and π_{comply} are merged into a single proof with a unique circuit. The proof of non-membership in the merkle tree is implemented as follows: the sanction list has the following form: $SancList = [s_1, s_2, ..., s_\xi]$. $SancList$ is supposed to be published sorted by the regulator. To verify the non-inclusion of an address apk in the list, we prove that there exist s_i and s_j such that:

- s_i and s_j are elements of $SancList$.
- $j = i + 1$ (s_i and s_j are consecutive elements of $SancList$).
- $s_i < apk < s_j$ (s_1 and s_ξ are set to $0x00..00$ and $0xFF..FF$ to make sure this constraint can always be fulfilled).

Hash functions and PRF. To optimize the computational cost of the proving algorithm in SNARK, we rely on MiMC [2]. MiMC is an arithmetization-oriented, collision-resistant and preimage-resistant hash function that is efficiently represented as arithmetic circuits. We use it to derive the the seeds ρ of created coins from the serial numbers sn of spent coins. The pseudorandom function family $\mathsf{PRF}_x()$ is derived from a MiMC function H as $\mathsf{PRF}_x(m) := \mathsf{H}(m||x)$.[4]

Incremental commitment scheme. The incremental commitment scheme is also derived from the MiMC function H. Given a commitment com^{old} and a message m to be added to the commitment with a randomness r, $\mathsf{INCR}(com^{old}, m, r)$ is defined as follows: $\mathsf{INCR}(com^{old}, m, r) = \mathsf{H}(com^{old}|m|r)$. The commitment function COM is then an iteration of increments INCR over the sequence of messages and randomnesses provided.

(k, N)-threshold blind signature. Our signature scheme is based on a slightly modified version of the Coconut scheme [37], which itself is a variant of the Pointcheval and Sanders signature scheme [34]. Indeed, certain checks originally performed using a sigma protocol in Coconut are instead handled in the Groth16 proof in our implementation, reducing the overall verification complexity. We instantiate this scheme with the BLS12-377 curve so that the verification algorithm, which involves pairings, is efficient in the Groth16 proof. (See Appendix B for more details).

[4]The natural way to implement a PRF from a hash function is to consider $\mathsf{PRF}_x := \mathsf{H}(m||x)$, however, replacing concatenation by an addition in the field does not compromise security in our case while reducing the computational cost.

7.2 Performance analysis

We now report on our comparative performance analysis.

NIZK Benchmark. We benchmarked our NIZK, implemented using a Groth16 SNARK, on an Intel i7 @ 2.6 GHz CPU running Ubuntu 22.04. The results are summarized in Table 2. Each value is based on the average of 10 proving and verification runs. A comparison with related protocols is provided in Table 1. The data in this table was gathered from the respective research papers, as the source codes of these protocols were not always publicly available. However, the CPU used for our benchmark has comparable performance to the machines reported in those works: they all use Intel Core i7 CPUs except Zef. For Zef, a specific AWS instance was used for their benchmark. To ensure a fair comparison, we also ran our benchmark on the same AWS instance, with detailed results provided in Table 3 in Appendix D.

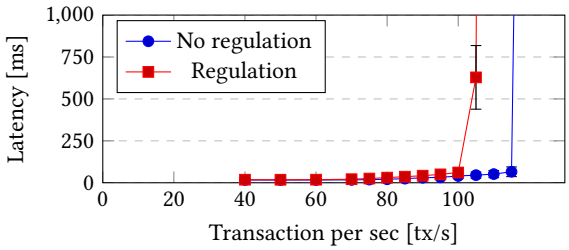
Paxpay has two implementations: with and without the regulatory feature. The benchmark was conducted on both. The regulated version, running on a single core, serves as a reference point. The results indicate that our SNARK verification time is notably short, taking only 5.6 **ms**. As shown in Table 1, this verification time is between 8 and 100 times shorter than the NIZK verification times reported in other works (except for Zcash, which takes approximately the same time but suffers from other major issues discussed in Section 8). The verification time is a key metric since slow verification limits the transaction throughput of the system.

This fast verification comes at the cost of relatively slow proving, requiring nearly 7 **s** to generate a proof for a transfer on a single core. However, with 6 cores, the proving time drops to 2 **s**.

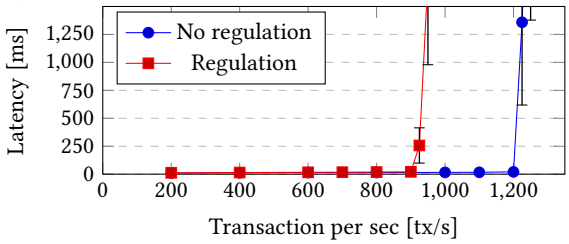
Table 2: SNARK Proving and Verification Times on an Intel i7 2.6 GHz CPU, running Ubuntu 22.04.

	Regulated		Not Regulated	
	1 Core	**6 Cores**	**1 Core**	**6 Cores**
Proving time (ms)	**6959** (± 13)	2001 (± 13)	3080 (± 6)	882 (± 5)
Verification time (ms)	**5.61** (± 0.20)	5.38 (± 0.28)	5.22 (± 0.15)	4.71 (± 0.85)

Latency test and transaction throughput Apart from Zcash, which is already deployed in real-world use, the only payment system in the related work to report on its latency and throughput is Zef [6]. To compare performance with Zef, we run our Paxpay implementation in the same environment by deploying multiple AWS EC2 instances (m5.8xlarge) and assigning one validator per instance. For each version of Paxpay (regulated and non-regulated), we conducted two tests: one with each validator running on a single CPU core, and another with each validator using 16 CPU cores. The results are presented in Figures 2a and 2b. We consider that the system support a given throughput if the corresponding transaction



(a) Average transaction latency with respect to the transaction throughput. $f = 3, N = 10$. Each validator runs on 1 CPU core.



(b) Average transaction latency with respect to the transaction throughput. $f = 3, N = 10$. Each validator runs on 16 CPU cores.

latency is smaller than 500 ms. Our regulated implementation achieves the throughput of 100 **tx/s** (transactions per second) in the single-core setup and 925 **tx/s** in the 16-core setup. Under the same conditions, Zef processes 5 **tx/s** and 88 **tx/s**, respectively.[5] For the non-regulated Paxpay implementation, the throughput reaches 115 **tx/s** in the single-core setup and 1200 **tx/s** in the 16-core setup.

These promising results can be attributed to the low verification time of the SNARK presented in Tables 2 and 3. In the reference setting (regulated implementation with 1-core validators), the verification of the SNARK takes 9 **ms** in our implementation while it takes 142 **ms** in Zef with the same setting (Table 3).

The difference in performance between the regulated and non-regulated versions is due to the additional public parameters in the regulated SNARK. This requires validators to perform more point multiplications on the elliptic curve used in the Groth16 construction.

To improve scalability, an *authority* in Zef (equivalent to our validator) can be *sharded* (run on multiple parallel machines). The same approach could be applied to our system, though it has not been implemented yet. A load balancer could efficiently distribute transfer requests among the validators by assigning each shard of a validator a specific range of serial numbers. Upon receiving a request, the load balancer would determine the appropriate shard to handle it based on these predefined ranges. Our experiments show that Paxpay is scalable. The throughput can be improved even further by allocating more computational power to the validators, either by using more CPU cores per validator or by distributing the workload across additional machines.

[5]As stated in Zef [6], each authority is distributed over multiple shards, each running in a single core. The 16-core result is extrapolated from the linear relationship between throughput and the number of shards per validator, as shown in their paper.

8 DISCUSSION

Comparison with related protocols Table 1 provides a comparison between Paxpay and the related protocols. Lelantus appears to outperform Monero across all metrics in the table and offers larger anonymity sets. Given the space constraints, we therefore decided to omit Monero from the table. Similarly, Platypus was not included in the table, as we focus exclusively on decentralized payment systems. Details and explanations regarding the data presented in the table can be found in Appendix D (Table 4).

Privacy. As shown in Table 1, Paxpay provides the strongest privacy guarantees, comparable to those provided by Zcash, whereas some protocols offer reduced privacy guarantees. This is especially Zef and PRCash.

To ensure full privacy, it is important to "hide" coin details with blind signatures, as well as to verify the signatures within the NIZK. Indeed, suppose that, as in Zef [6], we resort to only using blind signatures. Suppose that a user A creates a coin $\mathbf{c}$ for a user B during a transfer o_1. A requests the validators to sign the blinded coin $\tilde{\mathbf{c}}$, and then sends the aggregated signature to B. B now spends $\mathbf{c}$ during a transfer o_2, and in process reveals $\mathbf{c}$ and the (randomized) signature. Even though validators cannot reveal that the coin spent in o_2 was created in o_1, A knows $\mathbf{c}$ and, thus, can deduce the sender of o_2 and a lower bound on the amount spent during o_2. The use of NIZK obviates this problem, as now B does not reveal $\mathbf{c}$. Using a plain signature scheme (instead of a blind one) and verifying the signature in the NIZK will again infringe FPAT-privacy, as B would be able to identify A, as A would have already revealed $\mathbf{c}$ in clear in o_1.

Model assumptions Paxpay is responsive, meaning it can be implemented in an asynchronous network. Among the related protocols, only Zef and PARScoin also provide responsiveness. However, PARScoin requires correct validators (those who follow the protocol) to be honest, ensuring they do not share any information with the adversary. In contrast, Paxpay tolerates correct validators being semi-honest, meaning they can share any information they receive or transmit during the protocol's execution without compromising the system's privacy guarantees.

Regulation Table 1 includes the following regulation features that are provided by Paxpay or related protocols:

- *Limited held amount per user:* Users cannot hold more than V_{held}^{max}.
- *Limited spendable amount per tx:* Users cannot spend more than V_{sent}^{max} in one transaction.
- *Full asset tracing:* A trusted authority (centralized or decentralized) can reveal the content of transaction to trace back assets and users activities.
- *Limited spendable amount in total:* Users cannot spend more than V_{total}^{max}.
- *Sanction list:* Users cannot send money to another in the sanction list. A user in the sanction list cannot send money anymore.
- *Provable transaction history:* A user can prove arbitrary statements about her transactions. For instance, reveal all of

her transaction and prove that the revealed set is complete (no transactions are missing).

Paxpay does not support the *Limited Held Amount per User* feature. PEReDi [35] implements this feature, requiring synchrony between sender and receiver. PARScoin [36] claims a different approach: the sender initiates a transaction to reduce their balance, and the receiver can asynchronously claim funds later, provided their balance remains within the imposed limit. However, this method merely limits the amount a user can hold at a given time, as receivers can bypass it by spending excess funds before claiming the pending funds they have received. In practice, this mechanism has the same effect as setting a limit on the amount that can be spent in a single transaction. Similarly, limiting the cumulative receivable amount of money is almost the same thing as defining a limit on the total spent amount.

We believe that implementing a *Limited Held Amount per User* feature is particularly challenging in a private and asynchronous payment system. Such a feature would require that transfers atomically change the balances of both the sender and the receiver, to prove that receiver balance does not exceed a certain amount after executing the transfer. However, to affect the balance of a user, privacy typically requires this user to interact with another user or the validators, in order to provide some secret data known only to him (such as coin data or account commitments). Yet, asynchrony precludes live protocols from relying on interaction between senders and receivers. Indeed, in an asynchronous setting, the sender who gets no response from the receiver cannot distinguish whether this absence of response is caused by network delays or by the receiver refusing to respond. Implementing *Limited Held Amount per User* is thus hard without sacrificing synchrony or privacy.

Paxpay also avoids *Full Asset Tracing* due to privacy risks. Instead, it allows users to generate privacy-preserving proofs about their transactions. For instance, a user can prove that her account received funds from multiple addresses, each exceeding a certain threshold, demonstrating legitimate income patterns. For example, restaurants would use this proof to show compliance with AML regulations while protecting their privacy and their customers' privacy. This proof could demonstrate that the income pattern is legitimate and consistent with regular business operations, precluding money laundering (which might involve receiving a large sum from a single address).

Additionally, Paxpay introduces a *Sanction List* feature, enabling asset freezing for sanctioned users. This mechanism aligns with CBDC requirements [4] and supports AML and CFT regulations by preventing sanctioned individuals from transferring or receiving funds. Such a mechanism could be hard and costly (in term of performance) to implement in other protocols that use sigma protocols [6, 35, 36, 40].

Performance. As mentioned in Section 7.2, Paxpay outperforms all other protocols in terms of transaction throughput. This performance advantage is directly attributed to its fast NIZK verification, which surpasses all existing protocols. The only exception is Zcash, which also benefits from a fast NIZK verification but is constrained by the limitations of its underlying consensus protocol.

Additionally, the system's scalability, that is achieved by increasing the computational power allocated to each validator, further strengthens its practical applicability. The Visa payment system processed an average of 7,388 transactions per second in 2023 [39], which could be supported by Paxpay by distributing each validator across 8 m5.8xlarge EC2 instances[6].

As mentioned in Section 7.2, this high throughput comes at the cost of a relatively slow transfer proving, requiring between 2 and 7 seconds depending on the user computational power. We believe that this delay is acceptable from the user's perspective, especially since multiple payments can be aggregated into a single transfer. As a result, even if a user needs to make numerous payments, she can prove them all within a single transfer, requiring only one proof. It is worth noting that the statement being proven in such cases would be larger, which would still increase the proving time, though less significantly compared to generating separate proofs for each transfer.

Use cases. Our system is versatile and adaptable to various use cases. It can function as a standalone payment system, a CBDC, or a scaling solution for an existing blockchain (a so-called Layer 2). It can be enriched with a Mint operation that user would call to create new coins. Depending on the use case, Mint can be equipped with a proof showing that the user has the right to mint a new coin. For example, if Paxpay were used as a scaling solution for Bitcoin, the Mint operation would take some SPV (Simplified payment verification) [32] as input to prove the validators that some value has been locked on Bitcoin. The implementation of the Mint and Redeem operations would vary depending on whether the system is deployed as a CBDC or as a Layer 2 solution for a blockchain.

The impressive performance of Paxpay (Section 7.2) and its potential to scale are key factors that enable our system to effectively address these usem cases.

Future work. Our protocol can be reused with some slight modifications to implement the same functionalities as in Zexe [9], hence not only offering a payment system but *private computation*. We can also considerably improve performance of our system by introducing sharding in the implementation, or improve the usability by moving to a different type of NIZK with universal trusted setup.

9 CONCLUSION

In this paper, we focus on fully-private asset transfer (FPAT). We propose Paxpay, an asynchronous FPAT protocol, prove its correctness, make an implementation in Golang and analyzed its performance. Paxpay leverages succinct non-interactive zero-knowledge proofs and threshold blind signatures. Our system demonstrates significant improvements over existing systems, processing transactions at a higher rate while maintaining full anonymity for the users and responsiveness. Paxpay also ensures regulatory enforcement, including some features that no other private payement system has managed to provide so far. The flexibility offered by succinct NIZKs opens avenues for incorporating other regulatory compliance features with minimal

[6]This estimation assumes a direct linear relationship between throughput and the computational power of the validators, as shown in Zef.

impact on the system's performance. These elements make Paxpay suitable for a wide range of applications, from a scaling solution of blockchain to a standalone private payment system that can be used as a technical layer for CBDCs.

REFERENCES

[1] Timothé Albouy, Emmanuelle Anceaume, Davide Frey, Mathieu Gestin, Arthur Rauch, Michel Raynal, and François Taïani. 2024. Asynchronous BFT Asset Transfer: Quasi-Anonymous, Light, and Consensus-Free. arXiv:2405.18072 [cs.DC]

[2] Martin R. Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and Tyge Tiessen. 2016. MiMC: Efficient Encryption and Cryptographic Hashing with Minimal Multiplicative Complexity. In *Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10031)*, Jung Hee Cheon and Tsuyoshi Takagi (Eds.). 191–219.

[3] Karim Baghery, Markulf Kohlweiss, Janno Siim, and Mikhail Volkhov. 2021. Another look at extraction and randomization of Groth's zk-SNARK. In *Financial Cryptography and Data Security: 25th International Conference, FC 2021, Virtual Event, March 1–5, 2021, Revised Selected Papers, Part I 25*. Springer, 457–475.

[4] European Central Bank. 2020. Report on a digital euro. https://www.ecb.europa.eu/pub/pdf/other/Report_on_a_digital_euro~4d7268b458.en.pdf

[5] Mathieu Baudet, George Danezis, and Alberto Sonnino. 2020. FastPay: High-Performance Byzantine Fault Tolerant Settlement. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*. 163–177.

[6] Mathieu Baudet, Alberto Sonnino, Mahimna Kelkar, and George Danezis. 2023. Zef: Low-latency, Scalable, Private Payments. In *Proceedings of the 22nd Workshop on Privacy in the Electronic Society, WPES 2023, Copenhagen, Denmark, 26 November 2023*, Bart P. Knijnenburg and Panos Papadimitratos (Eds.). ACM, 1–16.

[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 *2014 IEEE Symposium on Security and Privacy*. 459–474. https://doi.org/10.1109/SP.2014.36

[8] Gautam Botrel, Thomas Piellard, Youssef El Housni, Ivo Kubjas, and Arya Tabaie. 2024. ConsenSys/gnark: v0.11.0. https://doi.org/10.5281/zenodo.5819104

[9] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. 2018. Zexe: Enabling Decentralized Private Computation. *IACR Cryptol. ePrint Arch.* (2018), 962.

[10] Gabriel Bracha. 1987. Asynchronous Byzantine agreement protocols. *Information and Computation* 75, 2 (1987), 130–143.

[11] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. 1996. The weakest failure detector for solving consensus. *Journal of the ACM (JACM)* 43, 4 (1996), 685–722.

[12] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Psi Vesely, and Nicholas P. Ward. 2020. Marlin: Preprocessing zkSNARKs with Universal and Updatable SRS. In *Advances in Cryptology - EUROCRYPT 2020 - 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10-14, 2020, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 12105)*, Anne Canteaut and Yuval Ishai (Eds.). Springer, 738–768.

[13] Daniel Collins, Rachid Guerraoui, Jovan Komatovic, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, Yvonne-Anne Pignolet, Dragos-Adrian Seredinschi, Andrei Tonkikh, and Athanasios Xygkis. 2020. Online Payments by Merely Broadcasting Messages. In *50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2020, Valencia, Spain, June 29 - July 2, 2020*. IEEE, 26–38.

[14] Danny Dolev and Rüdiger Reischuk. 1985. Bounds on information exchange for Byzantine agreement. *Journal of the ACM (JACM)* 32, 1 (1985), 191–204.

[15] Danny Dolev and H. Raymond Strong. 1983. Authenticated algorithms for Byzantine agreement. *SIAM J. Comput.* 12, 4 (1983), 656–666.

[16] Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. 1984. Consensus in the Presence of Partial Synchrony (Preliminary Version). In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing, Vancouver, B. C., Canada, August 27-29, 1984*, Tiko Kameda, Jayadev Misra, Joseph G. Peters, and Nicola Santoro (Eds.). ACM, 103–118.

[17] Electric Coin Company and Zcash Contributors. 2023. Zcash: Protocol and Reference Implementation. https://github.com/zcash/zcash. Accessed: 01/09/2024.

[18] David Evans, Vladimir Kolesnikov, and Mike Rosulek. 2018. *A Pragmatic Introduction to Secure Multi-Party Computation*. Now Publishers Inc. https://doi.org/10.1561/3300000019

[19] Prastudy Fauzi, Sarah Meiklejohn, Rebekah Mercer, and Claudio Orlandi. 2018. Quisquis: A New Design for Anonymous Cryptocurrencies. Cryptology ePrint Archive, Paper 2018/990. https://eprint.iacr.org/2018/990

[20] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. 1983. Impossibility of Distributed Consensus with One Faulty Process. In *Proceedings of the Second ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, March 21-23, 1983, Colony Square Hotel, Atlanta, Georgia, USA*, Ronald Fagin and Philip A. Bernstein (Eds.). ACM, 1–7.

[21] Oded Goldreich. 2001. *Foundations of Cryptography*. Vol. 1. Cambridge University Press. https://doi.org/10.1017/CBO9780511546891

[22] Jens Groth. 2016. On the size of pairing-based non-interactive arguments. In *Advances in Cryptology–EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II 35*. Springer, 305–326.

[23] Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovič, and Dragos-Adrian Seredinschi. 2019. The Consensus Number of a Cryptocurrency. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing (PODC '19)*. ACM. https://doi.org/10.1145/3293611.3331589

[24] Saurabh Gupta. 2016. *A Non-Consensus Based Decentralized Financial Transaction Processing Model with Support for Efficient Auditing*. Master's thesis. Arizona State University, USA.

[25] Youssef El Housni and Aurore Guillevic. 2020. Optimized and secure pairing-friendly elliptic curves suitable for one layer proof composition. Cryptology ePrint Archive, Paper 2020/351. https://eprint.iacr.org/2020/351

[26] Aram Jivanyan. 2019. Lelantus: A New Design for Anonymous and Confidential Cryptocurrencies. Cryptology ePrint Archive, Paper 2019/373. https://eprint.iacr.org/2019/373

[27] Aram Jivanyan and Aaron Feickert. 2021. Lelantus Spark: Secure and Flexible Private Transactions. Cryptology ePrint Archive, Paper 2021/1173. https://doi.org/10.1007/978-3-031-32415-4_28

[28] A. B. Kahn. 1962. Topological sorting of large networks. *Commun. ACM* 5, 11 (Nov. 1962), 558–562. https://doi.org/10.1145/368996.369025

[29] Chelsea Komlo and Ian Goldberg. 2020. FROST: Flexible Round-Optimized Schnorr Threshold Signatures. In *Selected Areas in Cryptography - SAC 2020 - 27th International Conference, Halifax, NS, Canada (Virtual Event), October 21-23, 2020, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 12804)*, Orr Dunkelman, Michael J. Jacobson Jr., and Colin O'Flynn (Eds.). Springer, 34–65.

[30] Petr Kuznetsov, Yvonne-Anne Pignolet, Pavel Ponomarev, and Andrei Tonkikh. 2023. Permissionless and asynchronous asset transfer. *Distributed Comput.* 36, 3 (2023), 349–371.

[31] Dahlia Malkhi and Michael K. Reiter. 1997. A High-Throughput Secure Reliable Multicast Protocol. *Journal of Computer Security* 5, 2 (1997), 113–128.

[32] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. http://bitcoin.org/bitcoin.pdf

[33] Shen Noether, Adam Mackenzie, and The Monero Research Lab. 2016. Ring Confidential Transactions. *Ledger* 1 (Dec. 2016), 1–18. https://doi.org/10.5195/ledger.2016.34

[34] David Pointcheval and Olivier Sanders. 2015. Short Randomizable Signatures. Cryptology ePrint Archive, Paper 2015/525. https://eprint.iacr.org/2015/525 https://eprint.iacr.org/2015/525.

[35] Amirreza Sarencheh, Aggelos Kiayias, and Markulf Kohlweiss. 2022. PEReDi: Privacy-Enhanced, Regulated and Distributed Central Bank Digital Currencies. Cryptology ePrint Archive, Paper 2022/974. https://eprint.iacr.org/2022/974

[36] Amirreza Sarencheh, Aggelos Kiayias, and Markulf Kohlweiss. 2024. PARScoin: A Privacy-preserving, Auditable, and Regulation-friendly Stablecoin. In *Cryptology and Network Security: 23rd International Conference, CANS 2024, Cambridge, UK, September 24–27, 2024, Proceedings, Part I* (Cambridge, United Kingdom). Springer-Verlag, Berlin, Heidelberg, 289–313. https://doi.org/10.1007/978-981-97-8013-6_13

[37] Alberto Sonnino, Mustafa Al-Bassam, Shehar Bano, Sarah Meiklejohn, and George Danezis. 2018. Coconut: Threshold issuance selective disclosure credentials with applications to distributed ledgers. *arXiv preprint arXiv:1802.07344* (2018).

[38] Paul F. Syverson, David M. Goldschlag, and Michael G. Reed. 1997. Anonymous Connections and Onion Routing. In *1997 IEEE Symposium on Security and Privacy, May 4-7, 1997, Oakland, CA, USA*. IEEE Computer Society, 44–54. https://doi.org/10.1109/SECPRI.1997.601314

[39] Visa. 2024. Visa Annual Report 2024. https://s29.q4cdn.com/385744025/files/doc_downloads/2024/Visa-Fiscal-2024-Annual-Report.pdf

[40] Karl Wüst, Kari Kostiainen, Vedran Capkun, and Srdjan Capkun. 2018. PRCash: Fast, Private and Regulated Transactions for Digital Currencies. Cryptology ePrint Archive, Paper 2018/412. https://eprint.iacr.org/2018/412

[41] Karl Wüst, Kari Kostiainen, Noah Delius, and Srdjan Capkun. 2021. Platypus: A Central Bank Digital Currency with Unlinkable Transactions and Privacy Preserving Regulation. Cryptology ePrint Archive, Paper 2021/1443. https://doi.org/10.1145/3548606.3560617

A PROTOCOL

Network channel As explained in the model description in Section 3, participants can use private channels. These private channels are formalized as follows. Send(type, data, apk_{dst}) allows to send data without revealing the sender identity. Send triggers a callback Receive(type, data, $\widetilde{apk}_{src}$) on the receiver side. The receiver can call Send(type, data, $\widetilde{apk}_{src}$) to send data back. To show that the receiver can respond but without knowing the identity of the sender, we denote by $\widetilde{apk}_{src}$ an anonymous version of apk_{src} received by the receiver, that allows him to answer without knowing apk_{src}. data is the payload of the message and type its type. These notations will apply in the protocol below.

Setup Phase: During the setup phase, a trusted third party $\mathcal{T}$ first runs the algorithm Setup (Algorithm 1), generating public parameters over the NIZK and keys for the threshold signature scheme. Depending on whether the protocol is run with or without regulation, the public parameters are either $\mathsf{pp}_{\mathsf{NIZK}}^{\mathsf{tx\_no\_regul}}$ (for the transfer operation) in the non-regulated case or $\mathsf{pp}_{\mathsf{NIZK}}^{\mathsf{tx\_regul}}$ and $\mathsf{pp}_{\mathsf{NIZK}}^{\mathsf{register}}$ (for the transfer and register operations) in the regulated case.

The user $u \in \mathcal{U}$ generates her public/secret address pair running AddrGen algorithm (Algorithm 2) and stores it. She also initializes her list coinList (valid unused coins) and rhoList (seeds of the received coins).

The trusted third party distributes initial values to each user via the Initialisation algorithm (Algorithm 3).

Running Phase: The *transfer* operation for users is implemented either as $\mathtt{Transfer}_{no\_regul}$ (Algorithm 6) or $\mathtt{Transfer}_{regul}$ (Algorithm 9), depending on whether regulation enforcement is applied. Algorithms 8 and 7 are used by $\mathtt{Transfer}_{no\_regul}$. Algorithms 10 and 7 are used by $\mathtt{Transfer}_{regul}$. Algorithms 8 and 10 generate the NIZK in the non-regulated and regulated case. The proof in the regulated case is the same as the one in the regulated case with some additional requirements explained in Section 6 as π_{comply}. π_{regul} in Algorithms 10 is thus the implemented merge of $\pi_{transfer}$ introduced in Section 5 and π_{comply}. If regulation enforcement is applied, user register via the $\mathtt{Register}_{regul}$ algorithm (Algorithm 11).

The implementation of the *balance* operation is described in Algorithm 4. When a user receives a transfer, Algorithm 5 is called. If regulation enforcement is not applied, validators handle transfer requests with Algorithm 12. If regulation enforcement is applied, validators handle transfer and register requests with Algorithm 13. Note that the algorithm for handling transfer operations is exactly the same, regardless of whether regulation is enforced.

Algorithm 1: Setup()

$([sk_i]_{i=1}^{N}, pk_{agg}) \leftarrow \mathsf{Setup}_{\mathsf{Sig}}(2f+1, N)$
$\mathsf{pp}_{\mathsf{NIZK}} \leftarrow \mathsf{Setup}_{\mathsf{NIZK}}(\mathcal{R})$
for $p \in (\mathcal{V} \cup \mathcal{U})$ **do**
  Send(type : $\mathsf{setup}_{\mathsf{nizk}}$, $(\mathsf{pp}_{\mathsf{NIZK}}^{\mathsf{tx\_no\_regul}} \mid \mathsf{pp}_{\mathsf{NIZK}}^{\mathsf{tx\_regul}} \,\&\, \mathsf{pp}_{\mathsf{NIZK}}^{\mathsf{register}}), p$)
  Send(type : $\mathsf{setup}_{\mathsf{sig}}$, pk_{agg}, p)
end
for $p \in \mathcal{V}$ **do**
  Send(type : sk, sk_p, p)
end

Algorithm 2: AddrGen()

$ask \xleftarrow{\$} \{0,1\}^{\lambda}$ // Sample ask randomly
$apk \leftarrow \mathsf{PRF}_{ask}(0)$
return (ask, apk)

Algorithm 3: Initialisation($[apk_u]_{u \in \mathcal{U}}$)

for $u \in \mathcal{U}$ **do**
  Choose an initial value v_u^{init}
  $\rho_u^{init} \xleftarrow{\$} \{0,1\}^{\lambda}$ // Sample ρ_u randomly
  $\mathbf{c}_u^{init} \leftarrow (v_u^{init}, apk_u, \rho_u^{init})$
  for $i \in [1, 2f+1]$ **do**
    $\sigma_{u,i} \leftarrow \mathsf{SIGN}(\mathbf{c}_u^{init}, sk_i)$
  end
  $\sigma_u \leftarrow \mathsf{AGGREGATE}([\sigma_{u,i}]_{i=1}^{n})$
  Send(type : coin, $(\mathbf{c}_u^{init}, \sigma_u), u$)
end

Algorithm 4: Balance()

Parse coinList as $[\mathbf{c}_i]_{i=1}^{l}$ // l is the total number or coins in coinList
Parse $[\mathbf{c}_i]_{i=1}^{l}$ as $[(v_i, apk_i, \rho_i)]_{i=1}^{l}$
return $\sum_{i=1}^{l} v_i$

Algorithm 5: Receive()

Upon Receive(type : coin, $(\mathbf{c}^{new}, \sigma), \widetilde{apk}_{src}$):
  Parse $\mathbf{c}^{new}$ as $(v^{new}, apk^{new}, \rho^{new})$
  if $(apk^{new} = \mathsf{PRF}_{ask}(0)) \wedge (\mathsf{VERIFY}_{\mathsf{Sig}}(\sigma, cm, pk_{agg}) == 1) \wedge (\rho^{new} \notin \mathsf{rhoList})$ **then**
    Append $\mathbf{c}$ to coinList

Algorithm 6: $\mathsf{Transfer}_{no\_regul}([v_j^{new}]_{j=1}^m, [apk_j^{new}]_{j=1}^m)$

Function $\mathsf{Transfer}([v_j^{new}]_{j=1}^m, [apk_j^{new}]_{j=1}^m)$:
- **Data:** List of new coin owner $[apk_j^{new}]_{j=1}^m$ and value $[v_j^{new}]_{j=1}^m$.
- **Result:** Values v_j^{new} are transferred to the users of address apk_j^{new}.
- **if** $\exists\ [\mathbf{c}_i]_{i=1}^n \subseteq \mathsf{coinList}\ ,\ \sum_{i=1}^n \mathbf{c}_i.v \geq x$ **then**
  - $[\mathbf{c}_i^{old}]_{i=1}^n \leftarrow [\mathbf{c}_i]_{i=1}^n$
- **else**
  - **return** fail
- $v_{m+1}^{new} \leftarrow \sum_{i=1}^n v_i^{old} - \sum_{j=1}^m v_j^{new}$ `// Redeemed to u`
- $apk_{m+1}^{new} \leftarrow apk$
- Remove $[\mathbf{c}_i^{old}]_{i=1}^n$ from coinList
- $([sn_i^{old}]_{i=1}^n, [\tilde{\mathbf{c}}_j^{new}]_{j=1}^{m+1}, [\mathbf{c}_j^{new}]_{j=1}^{m+1}, [b_j]_{j=1}^{m+1}, \rho_{seed}) \leftarrow \mathsf{Tx}([(\mathbf{c}_i^{old}, sk_i^{old})]_{i=1}^n, [(apk_j^{new}, v_j^{new})]_{j=1}^{m+1})$
- $\mathsf{public\_input} \leftarrow ([sn_i^{old}]_{i=1}^n, [\tilde{\mathbf{c}}_j^{new}]_{j=1}^{m+1})$
- $\mathsf{private\_input} \leftarrow ([\mathbf{c}_i^{old}]_{i=1}^n, [\sigma_i^{old}]_{i=1}^n, [sk_i^{old}]_{i=1}^n, [\mathbf{c}_j^{new}]_{j=1}^{m+1}, [b_j]_{j=1}^{m+1}, \rho_{seed})$
- $\pi \leftarrow \mathsf{Prove}_{no\_regul}(\mathsf{public\_input}, \mathsf{private\_input}, \mathsf{pp}_{\mathsf{NIZK}}^{\mathsf{tx\_no\_regul}})$
- Create $m+1$ new empty lists: $[\mathsf{sigList}_j]_{j=1}^{m+1} \leftarrow [[\ ]]_{j=1}^{m+1}$
- **for** $p \in \mathcal{V}$ **do**
  - `// Send the request to each validator in` $\mathcal{V}$
  - **async** $\mathsf{Send}(\mathsf{type} : \mathsf{tx}, (([sn_i^{old}]_{i=1}^n, [\tilde{\mathbf{c}}_j^{new}]_{j=1}^{m+1}), \pi),\ p)$

`//` $\mathsf{Receive}\ \tilde{\sigma}_j^p$ `send by validator` p:

Upon $\mathsf{Receive}(\mathsf{type} : \mathsf{sig}, [\tilde{\sigma}_j^p]_{j=1}^{m+1}, p)$: **for** $j = 1, 2, \ldots m+1$ **do**
- $\sigma_j^p \leftarrow \mathsf{UNBLIND}(\tilde{\sigma}_j^p, b_j)$
- **if** $(\sigma_j^p \notin \mathsf{sigList_j}) \wedge (\mathsf{VERIFY}_{\mathsf{Sig}}(\sigma_j^p, \mathbf{c}_j^{new}, pk_p) == 1)$ **then**
  - Append σ_j^p to $\mathsf{sigList_j}$
  - **if** $\mathsf{length(sigList)} == 2f+1$ **then**
    - $\sigma_j \leftarrow \mathsf{AGGREGATE}(\mathsf{sigList_j})$
    - $\mathsf{Send}(\mathsf{type} : \mathsf{coin}, (\mathbf{c}_j^{new}, \sigma_j), apk_j^{new})$
    - **return** confirm

Algorithm 7: $\mathsf{Tx}([(\mathbf{c}_i^{old}, ask_i^{old})]_{i=1}^n, [(apk_j^{new}, v_j^{new})]_{j=1}^m)$

Function $\mathsf{Tx}([(\mathbf{c}_i^{old}, ask_i^{old})]_{i=1}^n, [(apk_j^{new}, v_j^{new})]_{j=1}^m)$:
- **Data:** List of old coins to spend with the corresponding secret address. Public address and values for the new coins
- **Result:** Compute the needed data to sign the new coins
- Parse $[\mathbf{c}_i^{old}]_{i=1}^n$ as $[(v_i^{old}, apk_i^{old}, \rho_i^{old})]_{i=1}^n$
- $[sn_i^{old}]_{i=1}^n \leftarrow [\ \mathsf{PRF}_{ask_i^{old}}(\rho_i^{old})\ ]_{i=1}^n$
- $\rho_{seed} \xleftarrow{\$} \{0,1\}^\lambda$
- $[\rho_j^{new}]_{j=1}^{m+1} \leftarrow [\ \mathsf{PRF}_{\rho_{seed}}(sn_1^{old}||...||sn_n^{old}||j)\ ]_{j=1}^{m+1}$
- $[\mathbf{c}_j^{new}]_{j=1}^{m+1} \leftarrow [(v_j^{new}, apk_j^{new}, \rho_j^{new})]_{j=1}^{m+1}$
- $[b_j \xleftarrow{\$} \{0,1\}^\lambda]_{j=1}^{m+1}$
- $[\tilde{\mathbf{c}}_j^{new}]_{j=1}^{m+1} \leftarrow [\mathsf{BLIND}(\mathbf{c}_j^{new}, b_j)]_{j=1}^{m+1}$
- **return** $([sn_i^{old}]_{i=1}^n, [c\tilde{m}_j^{new}]_{j=1}^{m+1}, [\mathbf{c}_j^{new}]_{j=1}^{m+1}, [b_j^{new}]_{j=1}^{m+1})$

Algorithm 8: $\mathsf{Prove}_{no\_regul}(\mathsf{public\_input}, \mathsf{private\_input}, \mathsf{pp}_{\mathsf{NIZK}})$

Function $\mathsf{Prove}_{no\_regul}(\mathsf{public\_input}, \mathsf{private\_input}, \mathsf{pp}_{\mathsf{NIZK}})$:
- **Result:** π_{no_regul} the proof for the transfer without regulation
- Compute the proof π_{no_regul} using the NIZK function PROVE with inputs (public_input, private_input) and public parameters $\mathsf{pp}_{\mathsf{NIZK}}$ representing the following relation:
  - `// Created notes are well-formed:`
  - $(([sn_i^{old}]_{i=1}^n, [\tilde{\mathbf{c}}_j^{new}]_{j=1}^{m+1}), [\mathbf{c}_j^{new}]_{j=1}^{m+1}) == \mathsf{Tx}([(\mathbf{c}_i^{old}, sk_i^{old})]_{i=1}^n, (pk_j^{new}, v_j^{new}))$
  - `// Balance is conserved:`
  - $\sum_{i=1}^n v_i^{old} == \sum_{j=1}^{m+1} v_j^{new}$
  - `// Secret addresses corresponding to each consumed coin are known:`
  - $[apk_i]_{i=1}^n == [\ \mathsf{PRF}_{ask_i}(0)\ ]_{i=1}^n$
  - `// Spent coins are signed by 2f+1 validators:`
  - **for** $i \in [1, n]$ **do**
    - $\mathsf{VERIFY}_{\mathsf{Sig}}(\sigma_i, \mathbf{c}_i^{old}, pk) == 1$
  - `// The blinding is correct`
  - $[\tilde{\mathbf{c}}_j^{new}]_{j=1}^{m+1} == [\mathsf{BLIND}(\mathbf{c}_j^{new}, b_j)]_{j=1}^{m+1}$
- **return** π_{no_regul}

Algorithm 9: $\mathtt{Transfer}_{regul}([v_j^{new}]_{j=1}^m, [apk_j^{new}]_{j=1}^m)$

Function $\mathtt{Transfer}([v_j^{new}]_{j=1}^m, [apk_j^{new}]_{j=1}^m)$:
- **Data:** List of new coin owner $[apk_j^{new}]_{j=1}^m$ and value $[v_j^{new}]_{j=1}^m$.
- **Result:** Values v_j^{new} are transferred to the users of address apk_j^{new}.
- **if** $\exists\ [\mathbf{c}_i]_{i=1}^n \subseteq \text{coinList}\ ,\ \sum_{i=1}^n \mathbf{c}_i.v \geq x$ **then**
  - $[\mathbf{c}_i^{old}]_{i=1}^n \leftarrow [\mathbf{c}_i]_{i=1}^n$
- **else**
  - **return** fail
- $v_{m+1}^{new} \leftarrow \sum_{i=1}^n v_i^{old} - \sum_{j=1}^m v_j^{new}$ `// Redeemed to u`
- $apk_{m+1}^{new} \leftarrow apk$
- Remove $[\mathbf{c}_i^{old}]_{i=1}^n$ from coinList
- $([sn_i^{old}]_{i=1}^n, [\tilde{\mathbf{c}}_j^{new}]_{j=1}^{m+1}, [\mathbf{c}_j^{new}]_{j=1}^{m+1}, [b_j]_{j=1}^{m+1}, \rho_{seed}) \leftarrow \mathsf{Tx}([(\mathbf{c}_i^{old}, sk_i^{old})]_{i=1}^n, [(apk_j^{new}, v_j^{new})]_{j=1}^{m+1})$
- Parse cc^{old} as $(apk_{cc}^{old}, \rho_{cc}^{old}, v_{cc}^{old}, com^{old})$
- $apk_{cc}^{new} \leftarrow apk_{cc}^{old}$
- $\rho \leftarrow \mathsf{PRF}_{\rho_{seed}}(sn_1^{old}||...||sn_n^{old}||m+2)$
- $v_{cc}^{new} \leftarrow v_{cc}^{old} + \sum_{j=1}^m v_j^{new}$
- $[r_j \xleftarrow{\$} \{0,1\}^\lambda]_{j=1}^m$
- $com_0 \leftarrow com^{old}$
- $\forall j \in [1,m], com_j = \mathsf{INCR}(com_{j-1}, apk_j^{new}, v_j^{new}, r_j)$
- $com^{new} \leftarrow com^m$
- $b_{m+2} \xleftarrow{\$} \{0,1\}^\lambda$
- $cc^{new} \leftarrow (apk_{cc}^{new}, \rho_{cc}^{new}, v_{cc}^{new}, com^{new})$
- $\tilde{cc} \leftarrow \mathsf{BLIND}(cc^{new}, b_{m+2})$
- $sn_{n+1}^{old} = \mathsf{PRF}_{ask_{cc}^{old}}(\rho_{cc}^{old})$
- public_input $\leftarrow ([sn_i^{old}]_{i=1}^n, [\tilde{\mathbf{c}}_j^{new}]_{j=1}^{m+1}, V_{sent}^{max}, V_{toal}^{max}, SancList, \tilde{cc^{new}})$
- private_input $\leftarrow ([\mathbf{c}_i^{old}]_{i=1}^n, [\sigma_i^{old}]_{i=1}^n, sk, [\mathbf{c}_j^{new}]_{j=1}^{m+1}, [b_j]_{j=1}^{m+2}, \rho_{seed}, \sigma_{cc}^{old}, cc^{old}, cc^{new})$
- $\pi \leftarrow \mathsf{Prove}_{regul}(\text{public\_input}, \text{private\_input}, \mathsf{pp}_{\mathsf{NIZK}}^{\text{tx\_regul}})$
- Append $[(apk_j^{new}, v_j^{new}, r_j)]_{j=1}^m$ to comList
- Create $m+1$ new empty lists: $[\text{sigList}_j]_{j=1}^{m+1} \leftarrow [[\ ]]_{j=1}^{m+1}$
- **for** $p \in \mathcal{V}$ **do**
  - `// Send the request to each validator in` $\mathcal{V}$
  - **async** $\mathsf{Send}(\text{type} : \text{tx}, (([sn_i^{old}]_{i=1}^n, [\tilde{\mathbf{c}}_j^{new}]_{j=1}^{m+1}), \pi),\ p)$

`// Receive` $\tilde{\sigma}_j^p$ `send by validator` p:

Upon $\mathtt{Receive}(\text{type} : \text{sig}, [\tilde{\sigma}_j^p]_{j=1}^{m+1}, p)$: **for** $j = 1, 2, \ldots m+1$ **do**
- $\sigma_j^p \leftarrow \mathsf{UNBLIND}(\tilde{\sigma}_j^p, b_j)$
- **if** $(\sigma_j^p \notin \text{sigList}_\text{j}) \wedge (\mathsf{VERIFY}_{\mathsf{Sig}}(\sigma_j^p, \mathbf{c}_j^{new}, pk_p) == 1)$ **then**
  - Append σ_j^p to sigList_j
  - **if** $\text{length}(\text{sigList}) == 2f+1$ **then**
    - $\sigma_\text{j} \leftarrow \mathsf{AGGREGATE}(\text{sigList}_\text{j})$
    - $\mathsf{Send}(\text{type} : \text{coin}, (\mathbf{c}_j^{new}, \sigma_j), apk_j^{new})$
    - **return** confirm

Algorithm 10: $\mathsf{Prove}_{regul}(\text{public\_input}, \text{private\_input}, \mathsf{pp}_{\mathsf{NIZK}})$

Function $\mathsf{Prove}_{regul}(\text{public\_input}, \text{private\_input}, \mathsf{pp}_{\mathsf{NIZK}})$:
- **Result:** π_{regul} the proof for the transfer without regulation
- Compute the proof π_{regul} using the NIZK function PROVE with inputs (public_input, private_input) and public parameters $\mathsf{pp}_{\mathsf{NIZK}}$ representing the following relation:
  - `// Created notes are well-formed:`
  - $(([sn_i^{old}]_{i=1}^n, [\tilde{\mathbf{c}}_j^{new}]_{j=1}^{m+1}), [\mathbf{c}_j^{new}]_{j=1}^{m+1}) == \mathsf{Tx}([(\mathbf{c}_i^{old}, sk_i^{old})]_{i=1}^n, (pk_j^{new}, v_j^{new}))$
  - `// Balance is conserved:`
  - $\sum_{i=1}^n v_i^{old} == \sum_{j=1}^{m+1} v_j^{new}$
  - `// Secret addresses corresponding to each consumed coin are known:`
  - $[apk_i]_{i=1}^n == [\ \mathsf{PRF}_{ask_i}(0)\ ]_{i=1}^n$
  - `// Spent coins are signed by 2f+1 validators:`
  - **for** $i \in [1,n]$ **do**
    - $\mathsf{VERIFY}_{\mathsf{Sig}}(\sigma_i, \mathbf{c}_i^{old}, pk) == 1$
  - `// The blinding is correct`
  - $[\tilde{\mathbf{c}}_j^{new}]_{j=1}^{m+1} == [\mathsf{BLIND}(\mathbf{c}_j^{new}, b_j)]_{j=1}^{m+1}$
  - `// Regulation checks`
  - **for** $i \in [1,n]$ **do**
    - $apk_i^{old} == apk_{cc}^{old}$
  - $apk_{cc}^{old} == \mathsf{PRF}_{ask_{cc}^{old}}(0)$
  - $apk_{cc}^{new} == apk_{cc}^{old}$
  - $\rho_{cc}^{new} == \mathsf{PRF}_{\rho_{seed}}(sn_1^{old}||...||sn_n^{old}||m+2)$
  - $v_{cc}^{new} == v_{cc}^{old} + \sum_{j=1}^m v_j^{new}$
  - $com_0 \leftarrow com^{old}$
  - $\forall j \in [1,m], com_j \leftarrow \mathsf{INCR}(com_{j-1}, apk_j^{new}, v_j^{new}, r_j)$
  - $com^{new} == com_m$
  - $\sum_{j=1}^m v_j^{new} \leq V_{total}^{max}$
  - $v_{cc}^{new} \leq V_{total}^{max}$
  - $apk_{cc}^{old} \notin SancList$
  - $\forall j \in [1,m], apk_j^{new} \notin SancList$
- **return** π_{regul}

Algorithm 11: $\texttt{Register}_{regul}()$

Function Register():
  Result: cc_u an initial compliance coin
  if $\exists\ [\boldsymbol{c}_i]_{i=1}^n \subseteq \text{coinList}\ ,\ \sum_{i=1}^n \boldsymbol{c}_i.v \geq x$ **then**
    $[\mathbf{c}_i^{old}]_{i=1}^n \leftarrow [\mathbf{c}_i]_{i=1}^n$
  else
    return fail
  $\rho \xleftarrow{\$} \{0,1\}^\lambda$
  $cc_u \leftarrow (0, \rho, 0, 0)$
  $b \xleftarrow{\$} \{0,1\}^\lambda$
  $\tilde{cc}_u \leftarrow \text{BLIND}((apk_u, \rho, 0, 0), b)$
  $\text{public}_\text{i}\text{nput} \leftarrow (apk_u, \tilde{cc}_u)$
  $\text{private}_\text{i}\text{nput} \leftarrow (ask_u, cc_u, b)$
  Compute $\pi_{register}$ by calling $\text{PROVE}_{\text{NIZK}}$ with inputs (public_input, private_input) and public parameters $\text{pp}_{\text{NIZK}}^{\text{register}}$ that represent the following relations:
    $\tilde{cc}_u == \text{BLIND}((apk_u, \rho, 0, 0), b)$
    $apk_u == \text{PRF}_{ask_u}(0)$
  Create a new empty list: sigList $\leftarrow$ [ ]
  for $p \in \mathcal{V}$ **do**
    // Send the request to each validator in $\mathcal{V}$
    async Send(type : register, $(\tilde{cc}_u, apk_u, u, \pi_{register}),\ p$)
  // Receive $\tilde{\sigma}_{cc}^p$ send by validator p:
  Upon Receive(type : sig, $\tilde{\sigma}_{cc}^p, p$):
    $\sigma_{cc}^p \leftarrow \text{UNBLIND}(\tilde{\sigma}_{cc}^p, b)$
    if $\text{VERIFY}_{\text{Sig}}(\sigma_{cc}^p, \mathbf{cc}_u, pk_p) == 1$ **then**
      if length(sigList) $== 2f + 1$ **then**
        $\sigma_{\text{cc}} \leftarrow \text{AGGREGATE}(\text{sigList})$
        Store σ_{cc}, cc_u
        return confirm

Algorithm 12: $\texttt{Validator}_{no\_regul}()$

Storage:
  snList // Stores all the serial numbers sn seen in valid transactions. Used to avoid double spending.
  signedCoinList // Stores all the signed blinded coins $\tilde{\mathbf{c}}$ seen in valid transactions. Used to avoid malicious replica attack.
Upon Receive(type : tx, $(([sn_i^{old}]_{i=1}^n, [\tilde{\mathbf{c}}_j^{new}]_{j=1}^{m+1}), \pi), \widetilde{apk}_{src}$) :
  if $(\text{VERIFY}_{\text{NIZK}}(([sn_i^{old}]_{i=1}^n, [\tilde{\mathbf{c}}_j^{new}]_{j=1}^{m+1}), pp_{NIZK}^{tx\_no\_regul}, \pi) \wedge (\forall i \in [1,n], sn_i^{old} \notin \text{snList}) \vee ([\tilde{\mathbf{c}}_j^{new}]_{j=1}^{m+1} \subseteq \text{singedCoinList})$ **then**
    $[\tilde{\sigma}_i \leftarrow \text{SIGN}(\tilde{\mathbf{c}}_i^{new}, sk)]_{i=1}^n$// Computes the blinded coin's signatures
    for $sn \in [sn_i^{old}]_{i=1}^n$ **do**
      Append sn to snList
    Append $[\tilde{\mathbf{c}}_j^{new}]_{j=1}^{m+1}$ to signedCoinList
    Send(type : sig, $[\tilde{\sigma}_i]_{i=1}^{m+1}, \widetilde{apk}_{src}$)
  else
    Abort

Algorithm 13: $\texttt{Validator}_{regul}()$

Storage:
  snList // Stores all the serial numbers sn seen in valid transactions. Used to avoid double spending.
  signedCoinList // Stores all the signed blinded coins $\tilde{\mathbf{c}}$ seen in valid transactions. Used to avoid malicious replica attack.
  registeredList // Stores a pair (u,apk) of user ID associated with a public key for all the registered users
Upon Receive(type : tx, $(([sn_i^{old}]_{i=1}^n, [\tilde{\mathbf{c}}_j^{new}]_{j=1}^{m+1}), \pi), \widetilde{apk}_{src}$) :
  if $(\text{VERIFY}_{\text{NIZK}}(([sn_i^{old}]_{i=1}^n, [\tilde{\mathbf{c}}_j^{new}]_{j=1}^{m+1}), pp_{NIZK}^{tx\_regul}, \pi) \wedge (\forall i \in [1,n], sn_i^{old} \notin \text{snList}) \vee ([\tilde{\mathbf{c}}_j^{new}]_{j=1}^{m+1} \subseteq \text{singedCoinList})$ **then**
    $[\tilde{\sigma}_i \leftarrow \text{SIGN}(\tilde{\mathbf{c}}_i^{new}, sk)]_{i=1}^n$// Computes the blinded coin's signatures
    for $sn \in [sn_i^{old}]_{i=1}^n$ **do**
      Append sn to snList
    Append $[\tilde{\mathbf{c}}_j^{new}]_{j=1}^{m+1}$ to signedCoinList
    Send(type : sig, $[\tilde{\sigma}_i]_{i=1}^{m+1}, \widetilde{apk}_{src}$)
  else
    Abort
Upon Receive(type : register, $(\tilde{cc}_u, apk_u, u, \pi), \widetilde{apk}_{src}$) :
  if $(\text{VERIFY}_{\text{NIZK}}((\tilde{cc}_u, apk_u), pp_{NIZK}^{register}, \pi) \wedge ((u, *) \notin \text{registeredList}) \vee (\tilde{cc}_u \in \text{singedCoinList}))$ **then**
    $\tilde{\sigma}_{cc} \leftarrow \text{SIGN}(\tilde{cc}_u^{new}, sk)$// Computes the blinded coin's signature
    Append $\tilde{cc}_u$ to signedCoinList
    Send(type : sig, $\tilde{\sigma}_{cc}, \widetilde{apk}_{src}$)
  else
    Abort

B IMPLEMENTATION DETAILS

B.1 Cryptographic building blocks

Pointcheval-Sanders. As explained in Section 7, the signature scheme is a slightly modified version of the Coconut scheme [37], which itself is a variant of the Pointcheval-Sanders signature scheme [34]. It is based on type-3 bilinear pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ over secure groups $\mathbb{G}_1, \mathbb{G}_2$ and $\mathbb{G}_T$. We concretely instantiated it with BLS12-377. We denote by $\mathbb{F}_q$ its scalar field. First, let:

- $\mathcal{H}$ be a cryptographic collision and pre-image resistant hash function with image in $\mathbb{G}_1$, so called hash-to-curve[7].
- H be a cryptographic collision and pre-image resistant hash function [8] with image in $\mathbb{F}_q$. Since the message m will be a coin in the implementation, we allow H to take a tuple $(m_1, m_2, ..., m_n)$ as input and return $\mathsf{H}(m_1||m_2||...||m_n)$. m is thus always manipulated as $\mathsf{H}(m)$ in the signature scheme.

The signature scheme is as follows:

- $\mathsf{SETUP}_{\mathsf{sig}}() \rightarrow ([\mathsf{sk_i}]_{\mathsf{i=1}}^{\mathsf{N}}, [\mathsf{pk_i}]_{\mathsf{i=1}}^{\mathsf{N}}, pp, pk_{agg})$: Operate two Shamir secret sharing choosing two random polynomials $P_1, P_2 \in \mathbb{F}_q[X]$ of degree $k-1$ evaluating it in the points $0, 1, 2, ..., N \in \mathbb{F}_q$. Let the partial secret keys be the couples $[sk_i = (x_i, y_i) = (P_1(i), P_2(i))]_{i=1}^{N}$. Sample a generator $g_1, \in \mathbb{G}_1$ and a generator $g_2 \in \mathbb{G}_2$ and let the global verification key be $pk_{agg} = (X, Y) = (g_2^{P_1(0)}, g_2^{P_2(0)})$. Compute and publish the list $[\gamma_i]_{i=1}^{N} = [g_1^{y_i}]_{i=1}^{N}$ altogether with g_1 and g_2.
- $\mathsf{BLIND}(m, b) \rightarrow \tilde{m}$: Sample a random $s \in \mathbb{F}_q$, compute $c_m = \mathsf{PRF}_s(m)$ and $h = \mathcal{H}(c_m)$. Let $\tilde{m} = (c_m, h, h^{\mathsf{H}(m)} g_1^b)$ and store b. Compute a proof π_{aux} that $\tilde{m}$ is well computed[9] and send it altogether with $\tilde{m}$ to the signer.
- $\mathsf{SIGN}(\tilde{m}, sk_i) \rightarrow \tilde{\sigma_i}$: Parse $\tilde{m}$ as $(\tilde{m}_1, \tilde{m}_2, \tilde{m}_3)$. Parse sk_i as (x_i, y_i). Check that $\mathcal{H}(\tilde{m}_2) = \tilde{m}_1$ and π_{aux} is valid. Compute $\tilde{\sigma_i} = (\tilde{\sigma_{i,1}}, \tilde{\sigma_{i,2}}) = (\tilde{m}_2, \tilde{m}_2^{x_i} \tilde{m}_3^{y_i})$.
- $\mathsf{UNBLIND}(\tilde{m}, \tilde{\sigma_i}, b) \rightarrow \sigma_i$: Parse $\tilde{\sigma_i}$ as $(\tilde{\sigma_{i,1}}, \tilde{\sigma_{i,2}})$. Compute $\sigma_{i,2} = \tilde{\sigma_{i,2}} \gamma_i^{-b}$. Compute $\sigma_i = (\tilde{\sigma_{i,1}}, \sigma_{i,2})$.
- $\mathsf{AGGREGATE}([\sigma_i]_{i=1}^{n}) \rightarrow \sigma$: Compute $\sigma_2 = \prod_{i=1}^{n} \sigma_{i,2}^{\lambda_i}$ where the $\lambda_i = \prod_{j \in (0,k) \setminus \{i\}} \frac{j}{j-i}$ are the Lagrange coefficients. Sample a random t to randomize the signature. Let $\sigma = (h^t, \sigma_2^t)$.
- $\mathsf{VERIFY}_{\mathsf{Sig}}(m, \sigma, pk) \rightarrow b$: Parse σ as (σ_1, σ_2). Parse pk as (X, Y). Check that $e(\sigma_1, XY^{\mathsf{H}(m)}) == e(\sigma_2, g_2)$[10].

Note that a great advantage of this signature scheme over concurrent threshold signature schemes (as the popular FROST [29]) is that aggregation is completely non-interactive while working on an unlimited number of messages, which makes it suitable for usage in our asynchronous protocol.

C PROOFS

For convenience, our implementations of `Transfer` and `Validator` (Algorithms 6 and 12) allow performing transfers to multiple receivers in one `Transfer` call as explained in Section 5. This could be seen as a batch of successive transfers with only one receiver. The proofs below strictly follows the specification introduced in Section 4 and only consider transfers with one receiver ($m = 1$). Then, the proofs can be extended to transfers with multiple receivers. As a consequence, in this section, `Transfer` will creates two coins: a coin $\mathbf{c}_1^{new}$ for the recipient of the payment and a redeem coin $\mathbf{c}_2^{new}$.

To simplify notations, we also assume that each user has exactly one public/secret address pair. Thus, apk_2^{new} must equal the sole public address of the sender.

The proofs concern the non-regulated protocol but can be extended to the regulated protocol as explained in Appendix C.4.

C.1 FPAT-Safety

Let H be the history of an execution of Paxpay (thus composed of FPAT operations). Let us denote L_p the local history of a participant p. Notice that, as the Byzantine participants are not expected to follow the protocol, the FPAT-safety specification allows us to assign them arbitrary local histories, as long as the safety property for the correct participants is met. There are only three ways for a correct participant to receive a message:

- line 1 of Algorithm 5 - `Receive` (the participant is a user);
- line 16 of Algorithm 6 - `Transfer` (the participant is a user);
- line 4 of Algorithm 12 - `Validator` (the participant is a validator).

In each of these cases, the message content is checked (lines 3 in Algorithm 5, 18 in Algorithm 6 and 5 in Algorithm 12). The messages coming from Byzantine participants that do not pass these checks are ignored, we thus do not consider them in our analysis.

Given the history $H = (L_1, ..., L_U)$ we now construct a legal serialisation S of H. Considering successful *transfer* and *balance* operations in H, we build a directed graph G_H, which accounts of local histories L_p for correct participants, as well as causality relations across the operations. We then show that G_H is acyclic and construct a legal serialization of H from it.

Let us first introduce some terms. We say that a *transfer* operation o *creates* a coin $\mathbf{c}^{new}$ if a correct user u appends $\mathbf{c}^{new}$ to coinList in line 4 of `Receive` (Algorithm 5). We say that a *transfer* operation o invoked by a correct user u *consumes* a coin $\mathbf{c}^{old}$ if the execution of line 2 of `Transfer` (Algorithm 6) selects the coin $\mathbf{c}^{old}$. We say that a *balance* operation invoked by correct user *reads* a coin $\mathbf{c}$ if the execution of line 1 of `Balance` (Algorithm 4) parses $\mathbf{c}$.

Lemma 1. *No more than one transfer operation can consume a coin.*

[7] https://pkg.go.dev/github.com/consensys/gnark-crypto@v0.13.0/ecc/bls12-377#HashToG1

[8] https://pkg.go.dev/github.com/consensys/gnark-crypto/ecc/bls12-377/fr/mimc

[9] i.e π_{aux} has public inputs $\tilde{m} = (\tilde{m}_1, \tilde{m}_2, \tilde{m}_3) = (c_m, h, h^{\mathsf{H}(m)} g_1^b)$, private inputs (m, s, b) and statement $(\tilde{m}_1 == \mathsf{PRF}_s(m) \wedge \tilde{m}_3 == \tilde{m}_2^m g_1^b)$. Computing first $c_m = \mathsf{PRF}_s(m)$ and then $\mathcal{H}$ instead of directly computing $\mathcal{H}(m + k)$ saves us from computing the costly hash-to-curve $\mathcal{H}$ inside of the NIZK. In our protocol, this condition is verified within the proof π in `Transfer` algorithm. The fact that this modification doesn't impact blinding property of the scheme is given proof of *FPAT-privacy. Preimage-Resistance* of the PRF implies the security is neither impacted, because it prevents the requester from reusing several times the group element h.

[10] Let $sk_{agg} = (x, y) = (P_1(0), P_2(0))$ and (σ_1, σ_2) a valid signature. For left-hand side of the equation we have $e(\sigma_1, XY^{H(m)}) = e(h^t, g_2^x g_2^{yH(m)}) = e(h^t, g_2^{x+yH(m)}) = e(h, g_2)^{t(x+yH(m))}$, and for the right-hand side we have $e(\sigma_2, g_2) = e((h^{x+yH(m)})^t, g_2) = e(h^{t(x+yH(m))}, g_2) = e(h, g_2)^{t(x+yH(m))}$, so equality holds.

Proof. Suppose that a *transfer* operation $o_1 \in H$ consumes a coin $\mathbf{c}^{old}$. We show that for any operation $o_2 \in H, o_2 \neq o_1$, o_2 does not consume $\mathbf{c}^{old}$.

o_1 has been completed, hence a set $\mathcal{V}_1$ of at least $2f+1$ validators have added sn^{old} to their snList (line 7 of `Validator`). Consider another successful operation o_2. o_2 consumes coins $[\mathbf{c}_i]_{i=1}^n$. Thus, a set $\mathcal{V}_2$ of $2f+1$ validators have added $[sn_i^{old}]_{i=1}^n$ to their snList. Since their are $N = 3f+1$ validators in total, the intersection of $\mathcal{V}_1$ and $\mathcal{V}_2$ is composed of at least $f+1$ validators. Hence, at least one correct validator has added sn^{old} and $[sn_i^{old}]_{i=1}^n$ to its snList. Because of line 5 of `Validator`, sn^{old} is different from any $[sn_i^{old}]_{i=1}^n$. □

Lemma 2. *If a transfer operation o consumes a coin $\mathbf{c}^{old}$, either there exists another transfer o' that creates $\mathbf{c}^{old}$, or $\mathbf{c}^{old}$ has been created during the initialization.*

Proof. Suppose that a *transfer* o consumes a coin $\mathbf{c}^{old}$. By the Algorithm 6 (line 2), this coin has been previously appended to coinList. The only way to append a coin in coinList is either at initialisation, or through the algorithm `Receive` (Algorithm 5) line 4. The second and third clause of the check in line 3 of `Receive` (Algorithm 5), combined with the unforgeability of the signature scheme, imply that this coin has been signed by $2f+1$ validators. Indeed, the unforgeability of the signature scheme directly comes from the one of the variant of the Pointcheval-Sanders signature scheme which is randomizable and aggregatable. This holds under the SXDH assumption, as stated in [34]. There are at $f+1$ correct validators amongst them. Those correct validators only produce these signatures if this coin has been created in the algorithm `Transfer` or in the algorithm `Initialisation`. □

We can associate H with a directed graph G_H defined as follows: the set of vertices of G_H is the set of all successful *transfer* and *balance* operations in H. There is a directed edge from an operation o_1 to an operation o_2 if and only if there exists a coin $\mathbf{c}$ such that one of the following conditions holds:

- o_1 and o_2 are both *transfer*, o_1 *creates* $\mathbf{c}$ and o_2 *consumes* $\mathbf{c}$
- o_1 is a *transfer*, o_2 is a *balance*, o_1 *creates* $\mathbf{c}$ and o_2 *reads* $\mathbf{c}$
- o_1 is a *balance*, o_2 is a *transfer*, o_1 *reads* $\mathbf{c}$ and o_2 *consumes* $\mathbf{c}$

The *preimage-resistance* of the PRF used to derive the seeds ρ^{new} of the created coins from the seed ρ^{old} of the consumed coins, together with the *lemma* 1, guarantee that the graph G_H is acyclic. We use the Kahn algorithm [28] to run a topological sort on the graph. The following version of Kahn algorithm applied to G_H terminates and produces a sequence S containing all successful operations of H.

At each iteration of the main loop (line 3), for a certain local history L_i, the smallest operation o with respect to $\prec_{L_i}$ amongst all the operations with no incoming edge is selected (line 5) and appended to the sequence S, so S is by construction a serialization of H. We only need to prove the following lemma:

Lemma 3. *Every sequence produced by the Kahn algorithm (Algorithm 14) is legal.*

Proof. Let us consider the couple formed by the sequence S and the set of vertices V defined line 1 (resp. line 2) of Kahn Algorithm 14,

Algorithm 14: Kahn Algorithm(G_H)

```
1  Initialise sequence S
2  Initialise V as the set of vertices of G_H with no incoming
     edge
3  while V ≠ ∅ do
4      Choose o ∈ V
5      while ∃L_i ∈ H, ∃o' ∈ V such that (o ≺_{L_i} o') do
6          (o, o') ← (o', o)
7      Remove o from V
8      Append o to S
9      for each node o' with an incoming edge e from o do
10         Remove e
11         if o' has no other incoming edge then
12             Append o' to V
13 return S
```

which is updated at each loop iteration. We denote by (S_i, V_i) the state of this couple at the i^{th} iteration. (S_i, V_i) is said to be valid if S_i is legal and remains legal by adding any operation o from V_i to it. We will show by induction that the couple (S_i, V_i) remains valid at each iteration of the loop line 3, which implies that the sequence S returned by Algorithm 14 is legal.

At initialisation, S_0 is void so trivially legal. By lemma 2, the only *transfer* in the set V_0 are the ones consuming coins $\mathbf{c}^{init}$ with value v^{init} created by the trusted party $\mathcal{T}$ during the initialisation phase. Similarly, the only *balance* operations in V_0 are the ones reading the coins $\mathbf{c}^{init}$. Thus, any operation picked from V_0 at line 4 will lead to a legal sequence S_1.

Now suppose that (S_i, V_i) is valid. Let us prove that (S_{i+1}, V_{i+1}) is also valid. S_{i+1} is built by appending elements of V_i to S_i during the $(i)^{th}$ iteration. Hence, S_i at the end of the $(i)^{th}$ iteration constitutes a legal sequence by hypothesis (since the couple (S_i, V_i) is valid). At the beginning of the $(i+1)^{th}$ iteration, S_{i+1} equals S_i at the end of the $(i)^{th}$ iteration, and is thus legal. It remains to show that (V_{i+1}, S_{i+1}) is valid, hence that S_{i+1} is a legal sequence at the end of the $(i+1)^{th}$ iteration. Let o be any operation in V_{i+1}:

- *Case 1. o is a balance*: By construction, the vertex representing this operation has no incoming edge. Hence for each coin $\mathbf{c}$ read by o, the matching *transfer* operation that creates $\mathbf{c}$ have already been appended to the sequence S_{i+1}. Also by construction, these coins have not been spent yet by any *transfer* operation since the transfers that spend them have o as input. Hence, the concatenation of S_{i+1} with o is legal.
- *Case 2. o is a transfer*: By construction, the vertex representing o has no incoming edge, hence all the coins that consumes o have already been created. By lemma 1 and balance between consumed and created value made line 4 of Algorithm 8, o is the only $transfer$ operation that consumes these coins. Hence, the concatenation of S_{i+1} with o is legal.

Then, at the end of the $(i+1)^{th}$ iteration, S_{i+1} is legal and thus (V_{i+1}, S_{i+1}) is valid. By induction, the sequence S resulting of our Khan algorithm is legal. □

Lemma 4. *If a correct user u invokes an operation $o = balance_u()$ returning value v_1 then, if the next operation $transfer_u(v_2, w)$ invoked is such that $v_2 \leq v_1$, it does not return* fail.

Proof. This condition is immediately verified because the only event that can make a *transfer* operation invoked by a correct user fail is non-existence of a list of coins $[\mathbf{c}_i^{old}]_{i=1}^n \subseteq$ coinList such that $\sum_{i=1}^n \mathbf{c}_i^{old}.v \geq x$ (line 2 of `Transfer`). But no elements are removed from coinList during protocol execution except in a later step of `Transfer` algorithm. This guarantees that all coins read (line 1) in the `Balance` algorithm called by the preceding *balance* operation are still in coinList. By hypothesis they pass the check, so the *transfer* does not return fail. □

Lemma 3 and 4 imply:

Theorem 1. *Paxpay ensures FPAT-Safety.*

Proof. By lemma 3, we can obtain a legal sequence S^C of all the operations that confirmed in H. We can then insert in S^C the operations of H that has failed (respecting the order of the local histories L) to form a sequence S.

With lemma 4, the sequence S remain legal, and contain all operations of H. □

As said in Section 5, our `Transfer` algorithm allows to transfer money to several users in one call. Each call to `Transfer` can be translated as successive $transfer$ operations to a single user, thus the Theorem 1 still holds for our algorithm `Transfer`.

C.2 FPAT-Liveness

We remind that *FPAT*-Liveness guarantees that (1) every operation invoked by a correct user eventually terminates and (2) every transfer operation tx performed by a correct user eventually takes effect, i.e., there is a time after which *balance* operations at correct users (sender or receiver) return values accounting for tx.

Theorem 2. *Paxpay ensures FPAT-Liveness.*

Proof. **(1)** Operation *balance* is performed completely locally and so trivially terminates. Concerning the $transfer$ operation, the only blocking instruction in the `Transfer` algorithm is the request of coins' signatures (line 16). The first clause of the check performed line 3 of the `Receive` algorithm (Algorithm 5) ensures that the calling user u knows the secret key corresponding to all the coins of her coinList. The second clause guarantees that all these coins have valid signatures. Then, the *completeness* property of the NIZK guarantees that the proof π computed at line 12 of Algorithm 6 is valid. The removal of spent coins from coinList at line 8 of the `Transfer` algorithm, along with the last clause of the check line 3 of the `Receive` algorithm guarantees that u will not try to spend the same coin twice. Moreover, the *preimage-resistance* of the PRF guarantees that secret addresses ask of u have not been computed by any Byzantine participant. The same property of the PRF implies that no Byzantine participant have computed the serial numbers sn computed at line 3, so it does not already appears in the snList of validators. Thus, since the serial number do not already appear in their snList and the proof π is valid, the correct validators that receive the transfer request will sign the new coins and answer. Then termination of *transfer* operation directly follows from the network assumption (communication channels are reliable) and from the model assumption (at least $2f + 1$ validators are correct).

(2) There are only a finite number of $transfer_*(v+, u)$ operation completed before time t and the channels are reliable. Thus, there exists a time t' after which all the coins that were sent to a correct user u before time t are received by u. Because of the line 4 of the `Receive` algorithm, these coins have been appended to u's coinList. Let o be a *balance* operation invoked by u at time $t'' \geq t'$. The only coins removed from her coinList have been removed during completed *transfers* operation she has invoked before t''. The coins in her coinList at time t'' are thus the coins that have been received before t' but not removed by the transfer operation she has called before t''. Hence:

$$\sum_{\substack{transfer_*(v_+,u) \\ \text{invoked by correct users} \\ \text{completed before time } t}} v_+ \;-\; \sum_{\substack{transfer_u(v_-,*) \\ \text{completed before time} t''}} v_-$$

Thus, FPAT-Liveness holds. □

C.3 FPAT-privacy

In this section we describe more precisely the game $\mathcal{G}^{priv}$ defining *FPAT-privacy*.

Let $Pr(\mathcal{G}^{priv})$ be the probability that the adversary wins the game $\mathcal{G}^{priv}$. By definition, *FPAT-privacy* means that it exists a negligible function Adv such that, for a security parameter λ:

$$|Pr(\mathcal{G}^{priv}) - \frac{1}{2}| \leq \mathsf{Adv}(\lambda)$$

First we give more details about the functioning of the Paxpay oracle O^{Paxpay}. Its provides the three following interfaces:

- An interface allowing the caller to create addresses. Upon receiving such a *createAddress* query, it runs Algorithm 2, stores the generated private address ask and returns to the caller the corresponding public address apk. It also initialises an empty coinList attached to this address.
- An interface allowing the caller to append coins to the coinList of the different users.
- An interface allowing the caller to submit abstract transactions. Abstract transactions are tuples $tx = (apk_S, [c_i]_{i=1}^n, v, apk_R)$ specifying a transaction sender's public key apk_S, a list consumed coins $[c_i]_{i=1}^n$ belonging to the coinList attached to apk_S, a transferred value v, and a receiver public address apk_R. Upon receiving such a query tx, O^{Paxpay} runs Algorithm 6 with inputs (v, apk_R) (with knowledge of the local storage of user apk_S, including her coinList and apk_S) as well as Algorithm 12, internally simulating the interaction between the sender and validators. Finally, it outputs the execution traces of all parties.

Remark: Algorithm 4, corresponding to *balance* operation, as Algorithm 5, by which users receive funds, are performed completely locally. So they trivially do not impact the privacy guarantees of our protocol and we exclude them from our analysis.

We suppose the NIZK scheme is perfect zero-knowledge, with a simulator taking as input a trapdoor td_{NIZK} generated with the common reference string pp_{NIZK} during execution of $\mathsf{SETUP}_{\mathsf{NIZK}}^{\mathsf{trapdoor}}$, as it is the case for Groth16.

The distinguishing game $\mathcal{G}^{priv}$ is defined by following interactions between the adversary $\mathcal{A}$, and a challenger C:

Initialisation Phase:

- C chooses the security parameter λ. He generates public parameters pp_{NIZK} of the NIZK scheme running $\mathsf{SETUP}_{\mathsf{NIZK}}^{\mathsf{trapdoor}}$, obtaining at the same time the corresponding trapdoor td_{NIZK} taken as input by the NIZK simulator. C also runs $\mathsf{SETUP}_{\mathsf{sig}}$ obtaining the public parameters pp_{Sig} of the signature scheme. C stores the secret key $sk_{agg} = (P_1(0), P_2(0))$ corresponding to the threshold public key $pk_{agg} = (g_2^{P_1(0)}, g_2^{P_2(0)})$.
  C sends those public parameters $(pp_{\mathsf{NIZK}}, pp_{\mathsf{sig}})$ to the adversary $\mathcal{A}$ and uses them to initialise two oracles O_1^{Paxpay} and O_2^{Paxpay}.
- For each user, $\mathcal{A}$ sends a *createAddress* queries to C who transmit it to the oracles. They outputs public keys to C, who transmits them to $\mathcal{A}$.
- $\mathcal{A}$ initialises the balances of the users generating coins appended to the coinList of each participant in both oracles O_1^{Paxpay} and O_2^{Paxpay}.
- C samples a random bit b.

Challenge Phase: The challenge phase consists in several iterations of the following steps:

- $\mathcal{A}$ submits pairs of consistent abstract transactions (tx_0, tx_1) to the challenger C. Two transactions are consistent if and only if the number n of coins they consume are identical, and, if the receiver apk_R is controlled by $\mathcal{A}$ in one of the two transactions, then the receiver of the other transaction is the same apk_R in the other transaction and the transferred value v is also equal in both transactions.
- Receiving couple of abstract transactions (tx_0, tx_1), the challenger C checks its consistency. If the two abstract transactions are consistent, he provides the oracles O_1^{Paxpay} and O_2^{Paxpay} respectively with tx_0 and tx_1, receiving as output execution traces Tr_0 and Tr_1.
- Finally C provides $\mathcal{A}$ with the couple $(Tr_b^{|\mathcal{A}}, Tr_{1-b}^{|\mathcal{A}})$ of the restrictions of execution traces to the parties controlled by $\mathcal{A}$, in a permuted order depending on the bit b sampled in initialisation phase.

Decision Phase: At the end of the interactions with the challenger C, the adversary $\mathcal{A}$ outputs a guess b' about the bit b sampled by C at initialisation. $\mathcal{A}$ wins the game if $b' = b$.

Proof: We give a sketch of proof of *FPAT-privacy* by an hybrid argument. We describe a sequence $[\mathcal{G}_i]_{i=0}^5$ of games such that $\mathcal{G}_0$ is the game $\mathcal{G}^{priv}$ defining our security property, and $\mathcal{G}_5$ is a game where the challenger's responses to the adversary queries does not depend on the bit b. We prove that for all i in $\{0, ..., 5\}$ the adversary's advantage in distinguishing game $\mathcal{G}_i$ from game $\mathcal{G}_{i+1}$ is negligible. Because the winning advantage of $\mathcal{A}$ in $\mathcal{G}_5$ is trivially null, this proof the adversary's advantage is negligible in $\mathcal{G}^{priv}$ by triangular inequality.
In Paxpay protocol, correct users only interact with the validators by broadcasting a same message to all of them (message of type tx line 15 of `Transfer`). Thus, the views of each validator are identical up to the order of received requests. Moreover, network assumptions states that, for each user u, communication channels between u and any validator have similar latency distributions. It is thus sufficient to restrict our analysis to the view of a generic validator. Indeed, the combined knowledge of all the Byzantine validators leaks no more information to $\mathcal{A}$ than this generic validator's knowledge.
We note $\mathsf{Adv}^{\mathsf{PRF}}$ the advantage the adversary has in distinguishing a random function from an element of the PRF family sampled with an uniformly random seed. For the threshold blind signature scheme, rather than with an abstraction, we reason directly on the modified Coconut described in Appendix B.1. We note $\mathsf{Adv}^{\mathsf{DDH}}$ the advantage in winning the Decisional Diffie-Hellman (DDH) game in $\mathbb{G}_1$.

Game $\mathcal{G}_1$ is the same game as $\mathcal{G}_0$ excepts the NIZK in validator's trace is replaced by a simulated proof π_{sim} generated by the challenger using the trapdoor td_{NIZK} corresponding to public parameters pp_{NIZK} computed at initialisation of the game. We have:

$$Pr(\mathcal{G}_0) = Pr(\mathcal{G}_1)$$

Proof. Straightforwardly follows from definition of perfect zero-knowledge. □

Game $\mathcal{G}_2$ is the same as $\mathcal{G}_1$ with the blinded coins $[\tilde{\mathbf{c}}_j]_{j=1}^m$ in validator's trace replaced by random values of corresponding form. More precisely, each blinded coin $\tilde{\mathbf{c}}_j = (c_{\mathbf{c}_j}, h, h^{\mathsf{H}(\mathbf{c}_j)} g^b)$ with $c_{\mathbf{c}_j} = \mathsf{PRF}_s(\mathbf{c}_j)$ and $h = \mathcal{H}(c_{\mathbf{c}_j})$, is replaced by a tuple $(r_j, \mathcal{H}(r_j), g_j)$ where r_j is a random element of the field $\mathbb{F}_q$, and g_j a random element of $\mathbb{G}_1$. C also replaces the proof π_{aux} of the algorithm BLIND by a simulated one. We have:

$$|Pr(\mathcal{G}_1) - Pr(\mathcal{G}_2)| \leq 1 - (1 - \mathsf{Adv}^{\mathsf{PRF}})^2$$

Proof. The seed s used in the PRF to compute $c_{\mathbf{c}_j}$ being sampled at random in Algorithm BLIND, it is indistinguishable from the random element r_j. Moreover, element b being chosen at random in algorithm BLIND, g^b is perfectly indistinguishable from a random element of $\mathbb{G}_1$ and so is $h^{\mathsf{H}(\mathbf{c}_j)} g^b$. Finally, π_{aux} is also perfectly indistinguishable from a simulated proof by hypothesis and thus, the above upper bound follows. The right-hand side is quadratic in $\mathsf{Adv}^{\mathsf{PRF}}$ since two coins are created (the payment coin and the redeem coin).

□

Game $\mathcal{G}_3$ is the same game as $\mathcal{G}_2$ with the serial numbers $[sn_i]_{i=1}^n$ in validator's trace replaced by truly random values of corresponding size (concretely, an elements uniformly sampled in $\mathbb{F}_q$). We have:

$$|Pr(\mathcal{G}_2) - Pr(\mathcal{G}_3)| \leq 1 - (1 - \mathsf{Adv}^{\mathsf{PRF}})^n$$

Proof. The seed ask is chosen at random by the oracles (line 1 of Algorithm 2), and evaluated on the values ρ_i which are unique with overwhelming probability, so the upper bound directly follows from the pseudorandomness property of the used PRF. □

Game $\mathcal{G}_4$ is the same as $\mathcal{G}_3$ except that the challenger C modifies not only the generic validator's trace, but also the receiver's trace in case it is controlled by $\mathcal{A}$. C replaces the threshold signature σ received from the oracle by a signature σ_{sim} generated by C with $sk_{agg} = (x, y)$. C generates σ_{sim} sampling a random $\sigma_{sim,1} \in \mathbb{G}_1$, computing $\sigma_{sim,2} = \sigma_{sim,1}^{x+y\cdot \mathrm{H}(m)}$ and letting $\sigma_{sim} = (\sigma_{sim,1}, \sigma_{sim,2})$.

$$|Pr(\mathcal{G}_3) - Pr(\mathcal{G}_4)| \leq \mathsf{Adv}^{\mathsf{DDH}}$$

Proof. The randomisation step in algorithm AGGREGATE, takes (σ_1, σ_2) and compute (σ_1^t, σ_2^t) with t uniformly sampled in $\{0, |\mathbb{G}_1| - 1\}$. The not randomized signature is thus $(h, h^{x+y\mathrm{H}(m)})$, the randomized signature is $(h^t, h^{(x+y\cdot \mathrm{H}(m))\cdot t})$ and the simulated signature is $(z, z^{x+y\cdot \mathrm{H}(m)})$, with z a random element of $\mathbb{G}_1$.

With the knowledge of h obtained from the blinded coin in the validator's trace, distinguish the randomized and the simulated signature is equivalent to distinguish $(h, h^t, h^{t(x+yH(m))})$ from $(h, z, z^{x+yH(m)})$. The DDH problem reduces to it, as it is argued in [34].

□

Game $\mathcal{G}_5$ is the same as $\mathcal{G}_4$ with a modification of the coin sent to the receiver in case it is controlled by $\mathcal{A}$. The coin $\mathbf{c}_{sim}^{new} = (v, apk_{\mathcal{A}}, \rho_{sim}^{new})$ has the same value v and public key $apk_{\mathcal{A}}$ as in the game $\mathcal{G}_4$ (C extract it directly from the adversary's queries without calling any oracle), but ρ_{sim}^{new} is a field element chosen at random in $\mathcal{G}_5$. We have:

$$|Pr(\mathcal{G}_4) - Pr(\mathcal{G}_5)| \leq \mathsf{Adv}^{\mathsf{PRF}}$$

Proof. ρ^{new} is computed from a PRF with a seed ρ_{seed} chosen uniformly at random (line 4 Algorithm 6) and evaluated on an input $sn_1||sn_2||...||sn_n||j$ which is unique with overwhelming probability, so upper bound directly follows from definition of a PRF. □

C.4 Regulation impact

The various proofs provided above still apply to the regulated version, as the *transfer* operation is handled in exactly the same way by the validators. The main problem that could arise would concern the privacy. While the NIZK proof differs between the regulated and non-regulated versions, its public inputs remain the same. The compliance coin is processed like any other coin from an external perspective. Therefore, the revealed data includes the blinding of the new compliance coin and the serial number of the old compliance coin, just as it would for a standard coin in the non-regulated NIZK.

D COMPARISON TABLE

Table 3: SNARK Proving and Verification Time on a m5.8xlarge EC2 instance (AWS) with Ubuntu 22.

	Regulated		Not Regulated	
	1 Core	16 Cores	1 Core	16 Cores
Proving time (ms)	12079 (± 19)	1624 (± 40)	5471 (± 15)	783 (± 22)
Verification time (ms)	9.60 (± 0.01)	5.98 (± 0.04)	8.97 (± 0.02)	5.69 (± 0.04)

Table 4: Detailed comparison of Paxpay vs. Zcash [17], Lelantus [26], Quisquis [19], Zef [6], PRCash [40], PEReDi [35] and PARScoin [36].

	Zcash	Lelantus	Quisquis	Zef	PRCash	PEReDi	PARScoin	Paxpay
PRIVACY PROPERTIES								
Confidential transfers: Yes ‖ No ‖ Partial	●	●	●	◑[11]	◑[12]	●	●	●
Sender-anonymous transfers: Yes ‖ No ‖ Partial	●	●	●	❍	◑[13]	◑[14]	◑[15]	●
Receiver-anonymous transfers: Yes ‖ No	●	●	●	●	●	●	●	●
Unlikable transfers: Yes ‖ No	●	●	●	❍	◑[16]	●	●	●
Anonymity strategy: Full ‖ AS (Anonymity Set)	Full	AS	AS	Full	Full	Full	Full	Full
MODEL ASSUMPTIONS								
Asynchronous network: Yes ‖ No	❍	❍	❍	●	❍	❍	●	●
Correct validators model: H (Honest) ‖ SH (Semi-Honest)	SH	SH	SH	SH	SH	H	H	SH
REGULATION FEATURES								
Limited held amount per user	❍	❍	❍	❍	❍	●	❍[17]	❍
Limited spendable amount per tx	❍	❍	❍	❍	◑[18]	●	●	●
Limited spendable amount in total	❍	❍	❍	❍	❍	●	●	●
Full asset tracing	❍	❍	❍	❍	❍	●	●	❍
Sanction list	❍	❍	❍	❍	❍	❍	❍	●
Provable transaction history	❍	❍	❍	❍	❍	❍	❍	●
PERFORMANCE								
Transaction throughput (tx/s)	25	N/A	N/A	88[19]	N/A[20]	N/A	N/A	925[21]
Transaction latency (s)	1000[22]	N/A	N/A	< 1	N/A	N/A	N/A	< 1
NIZK Proving time (ms)	21K	2378	2110	438	100	3100	392	6959
NIZK Verification time (ms)	9[23]	40[24]	251[25]	142[26]	96[27]	518[28]	159[29]	5[30]

Table 3 details Paxpay NIZK benchmark on an AWS m5.8xlarge EC2 instance running Ubuntu 22.

Table 4 gives additional details. Note that the NIZK proving and verification time have been tested in the same conditions for all the protocols.

[11]The sender of the output coin will know a lower bound (or the exact value) on the amount of the future payment during which the receiver will spend the coin.
[12]The sender of the output coin will know a lower bound (or the exact value) on the amount of the future payment during which the receiver will spend the coin.
[13]The receiver knows the sender.
[14]The receiver knows the sender.
[15]The receiver of the transaction is the only one to know the sender's identity. The protocol could be sender-anonymous but it would weaken the regulatory enforcement. In our construction, we circumvent this problem with the $SanctionList$, proving that the sender is legit within the NIZK.
[16]Validators can link transactions emitted during the same epoch by the same user.
[17]See Section 8, in the **Regulation** paragraph
[18]PRCash as a limit defined per epoch, not per transaction.
[19]Estimate from Zef [6] paper on the throughput for each validator running on 16 one-core shard, each shard running on a m5.8xlarge EC2 (AWS) machine equiped with an Intel Xeon Platinum 8175 CPU.
[20]PRCash paper provide a theoretical throughput that is only computed based on the NIZK verification time, and provide no more information this metric.
[21]Tests running on 16 CPU core of a m5.8xlarge EC2 instance equipped with an Intel Xeon Platinum 8175 CPU, equivalent to Zef.
[22]Considering immediate inclusion in a block and 15 block validation with 1min15 average block production time
[23]Benchmark done on a computer equipped with an Intel Core i7 3.5GHz CPU, 32 GB of RAM and Linux.
[24]Over an anonymity set of 2^{16} and a proof batch size of 50. Computed on an Intel I7-4870HQ CPU.
[25]Anonymity set of size 64. Computed on an Intel Core i7 2.8GHz CPU.
[26]Tests running on one CPU core of a m5.8xlarge EC2 instance equipped with an Intel Xeon Platinum 8175 CPU. The verification time has only a relative impact on the scalability of the system since a validator can be efficiently distributed over several machines, increasing the transaction throughput.
[27]On one core of an Intel Core i7-4770 CPU. Range proof on 2^{20}
[28]The benchmark was made on a computer with an Intel Core i7-9850H CPU at 2.60 GHz with 16 GB of RAM using Ubuntu 20.04.2 LTS. This value is computed with the formula provided in the paper for 8 byzantine validators and all compliance parameters set to $2^{20} \approx 1,000,000$ (Considering a maximum value of 1000 coins with precision of 10^{-3}).
[29]The benchmark was made on a computer with an Intel Core i7-9850H CPU at 2.60 GHz with 16 GB of RAM using Ubuntu 20.04.2 LTS. This value is computed with the formula provided in the paper for 8 byzantine validators and the maximum receivable amount set to $2^{20} \approx 1,000,000$ (Considering a maximum value of 1000 coins with precision of 10^{-3}).
[30]Benchmark on one CPU core of an Intel i7 2.6 GHz CPU. The verification time has a relative impact on the scalability of the system since a validator can be efficiently distributed over several machines, increasing the transaction throughput.