PriFHEte: Achieving Full-Privacy in Account-based Cryptocurrencies is Possible

Varun Madathil   Alessandra Scafuro *

North Carolina State University
May 20, 2024

Abstract

In cryptocurrencies, all transactions are public. For their adoption, it is important that these transactions, while publicly verifiable, do not leak information about the identity and the balances of the transactors.

For UTXO-based cryptocurrencies, there are well-established approaches (e.g., ZCash) that guarantee full privacy to the transactors. Full privacy in UTXO means that each transaction is anonymous within the set of all private transactions ever posted on the blockchain.

In contrast, for account-based cryptocurrencies (e.g., Ethereum) full privacy, that is, privacy within the set of *all accounts*, seems to be impossible to achieve within the constraints of blockchain transactions (e.g., they have to fit in a block). Indeed, every approach proposed in the literature achieves only a much weaker privacy guarantee called $k-$anonymity where a transactor is private within a set of k account holders. $k-$anonymity is achieved by *adding* k accounts to the transaction, which concretely limits the anonymity guarantee to a very small constant (e.g., 64 for QuisQuis and 256 for anonymous Zether), compared to the set of all possible accounts.

In this paper, we propose a completely new approach that does not achieve anonymity by including more accounts in the transaction, but instead makes the transaction itself "smarter". Our key contribution is to provide a mechanism whereby a compact transaction can be used to *correctly* update *all accounts*. Intuitively, this guarantees that all accounts are equally likely to be the recipients/sender of such a transaction. We, therefore, provide the first protocol that guarantees *full* privacy in account-based cryptocurrencies PriFHEte [1].

The contribution of this paper is theoretical. Our main objective is to demonstrate that achieving full privacy in account-based cryptocurrency is actually possible. We see our work as opening a door to new possibilities for anonymous account-based cryptocurrencies.

Nonetheless, in this paper, we also discuss PriFHEte's potential to be developed in practice by leveraging the power of off-chain scalability solutions such as zk rollups.

*Varun Madathil and Alessandra Scafuro are supported by Protocol Labs

[1] Pronounced like "private" but with an f in place of the v. That is, prifate.

1 Introduction

Account-based cryptocurrencies (e.g., Ethereum[eth], Filecoin[fil], Ripple[rip] etc.) follow the traditional bank model of keeping balances for accounts. In these cryptocurrencies, each public key (account) is associated with a balance, and a payment from public PK_A to PK_B of x coins, results in simply updating the balances of PK_A and PK_B by $-x$ and $+x$ respectively. In contrast, the Unspent Transaction Outputs (UTXO) Model, used in Bitcoin, is organized around transactions, and a payment is created by referencing a public key from the output of an unspent transaction. Account-based cryptocurrencies offer several advantages over UTXO for transactions. For example, the account model has better memory usage. Users only need to store a single account balance as opposed to several UTXOs that together make up the balance. Similarly regarding, miner storage, miners need to maintain an ever-increasing set of UTXOs to verify transactions. On the other hand, the size of the state in account-based cryptocurrencies increases only when new accounts are added.

Privacy in Cryptocurrencies. Privacy in financial transactions has always been deemed important, as people tend to prefer that the amount of money they have and how they use it, remains private. Traditional banking systems inherently provide privacy by keeping account balances confidential, known only to the bank and the account holder. However, in the case of cryptocurrencies, public verifiability is necessary as transactions are only added to the blockchain if they can be verified by the public. As a result, privacy must be added to cryptocurrencies carefully, without compromising public verifiability. De-anonymization attacks on Bitcoin [MPJ+13, RH13] have demonstrated that using randomly generated public keys provides limited privacy, as payments can be traced and, in combination with other metadata, can be used to associate real identities with public keys. This has led to a significant amount of research aimed at adding privacy to cryptocurrencies [MGGR13, SCG+14, NM+16, FMMO19, BAZB20, Dia21], with various trade-offs between privacy and efficiency (as discussed in Section 2).

For the UTXO model, it is possible to achieve full privacy while still maintaining efficiency and public verifiability. For example, ZCash [HBHW] is a fully private, publicly verifiable and practical UTXO payment system. The main idea of such systems is to associate a transaction to a serial number and commit to the serial number and the value of the transaction. This commitment is added to a public pool (also referred to as state), succinctly represented by a Merkle Tree. To spend an unspent transaction, the unique serial number is revealed and a succinct zero-knowledge proof is provided to demonstrate that this serial number represents one of the transactions committed to the pool. Then, a new serial number representing the new unspent transaction is committed to the pool. However, the main disadvantage of this approach is that the pool of private transactions grows infinitely. Additionally, the miners must keep track of all the serial numbers that have been revealed over time.

Privacy in Account-based Cryptocurrencies. In account-based cryptocurrencies, the state is a list of accounts with their respective balances (e.g., PK_A, v_A) and payment is an *update* of two account balances in this state. The sender's balance is decreased by x and the receiver's balance is increased by x (for simplicity, miner fees are ignored). As a result, the state of the blockchain can be viewed as a large table with one row per account, and payments require updating two rows in the table. It can be seen that the serial number-based approach used in Zcash would not work in this model, as payments require updating account balances, rather than just burning an unspent transaction.

To add privacy to account-based cryptocurrencies, current solutions, such as QuisQuis [FMMO19], Zether[BAZB20] and anonymous Zether [Dia21] hide the balance of the users by encrypting or committing to balances using a homomorphic scheme. This achieves confidentiality. To add anonymity they [Dia21, FMMO19] rely on the concept of adding multiple accounts to a payment transaction. This way, an external observer cannot determine the pair of accounts executing the transaction. Instead of creating a transaction with only the public keys of the sender and receiver, the sender will select a set of $k-2$ other public keys to form a "ring". A multi-account transaction is then created, containing k ciphertexts and a zero-knowledge proof that two out of the k ciphertexts correctly encrypt a balance transfer between two account holders, while the remaining ciphertexts are encryptions of 0. The miner processing this transaction, updates the k rows in the state, by homomorphically adding each ciphertext to the correct row. This approach provides k-anonymity to the sender and receiver.

QuisQuis and anonymous Zether suggest using an anonymity set of size 16, while Monero [NM+16, LRR+19] [2] suggests a ring size of 11. These values of k are a very small fraction of the total number of account holders, and provide very fragile guarantees, as shown by the attacks proposed in [MSH+17, KFTS17] on the traceability of the sender of transactions in Monero. Furthermore, the limitation on the anonymity set is inherent with this technique, since the choice of k must be upper-bounded by the maximum size of the transactions that can fit in a block. If we consider the typical size of a blockchain block, the maximum anonymity set that an account holder can obtain is around 64 for QuisQuis and 256 for anonymous Zether [3]. Another significant drawback of this approach is that the choice of the accounts that are included in the anonymity set must be done carefully, since a bad choice of accounts (e.g., accounts that rarely appear in any transaction) can reduce even further the actual anonymity guarantees.

There seems to be a major obstacle to achieving full anonymity in account-based cryptocurrencies. Since anonymity depends on the number of accounts involved in the transaction, full anonymity would require the transaction to be at least as large as the entire table of accounts, which is infeasible to implement.

In this paper we ask the following *feasibility* question:

> *Is it possible to achieve full anonymity in account-based blockchains with transactions that are independent of the anonymity set?*

1.1 Our contribution

In this paper, we answer the above question positively. We provide a novel approach for creating privacy-preserving transactions that are compact and provide *privacy within the set of all account holders* [4]. Our work provides the strongest anonymity degree with the shortest transaction size, and is asymptotically most efficient for account holders. We prove security

[2]Monero is a UTXO-based cryptocurrency that however uses the *ring* approach to achieve anonymity.

[3]We extrapolated these values from the following data from [Dia21]: for an anonymity set of 16, the size of the transaction for QuisQuis is 26KB, and for anonymous Zether is 6KB, and we consider the maximum blocksize to be 100KB (https://bitinfocharts.com/comparison/size-eth.html)

[4]"All account holders" means all account holders that have a private account. If a blockchain does not have privacy by design, then some people could choose to have a public account only. Such accounts, naturally, would not count in the anonymity set.

in the Universally Composable (UC) [Can01] model, using the private-ledger functionality introduced by Kerber et al. [KKKZ19]. To the best of our knowledge, this is also the first account-based privacy-preserving payment protocol that is UC-secure (regardless of the efficiency).

1.2 Our Techniques

Recall that our goal is to create payment transactions that have full anonymity w.r.t. all existing account holders, and confidentiality of the amount transferred. To achieve confidentiality, we first encrypt the balances of each account holder under their public key. The transfer transaction includes ciphertexts that are added to the corresponding encrypted balance. However, for a transaction to have full anonymity, it is essential that every account's ciphertext is updated during the payment transaction processing. If even a single account is not updated, the anonymity set is reduced by one. To solve this challenge, we need to craft two ciphertexts, c_S and c_R, that can be homomorphically evaluated with each ciphertext $c_1, c_2, \ldots, c_N$ in such a way that all ciphertexts are correctly updated with the re-encryption of their current balances, while only the ciphertexts of the sender and receiver's accounts are updated with the new balance.

The main challenge to achieving this is that each every ciphertext is computed under a *different key*. How can two ciphertexts be used to homomorphically update N ciphertexts computed under N different keys?

We solve this conundrum by taking inspiration from the recent elegant work by Liu and Tromer [LT22] that faces a similar challenge for a very different problem (see Section 2). Their work leverages a special property that exists in some LWE-based encryption schemes for plaintexts in $\{0, 1\}$, called wrong key decryption (see Def 3). This property states that, *even when a ciphertext is decrypted with the wrong key, the decryption function always returns a bit* (Regev's encryption scheme [Reg10] and the LWE scheme from PVW [PVW08] satisfy this property).

With this property in hand, our main idea is to use two encryption schemes, a fully homomorphic encryption (FHE) scheme to *encrypt the balance* and an encryption scheme with the property of wrong-key decryption (denoted WKEnc). The encryption scheme WKEnc is used to hide the identities of the sender and receiver. With the wrong-key decryption property, we can (using FHE) homomorphically decrypt [5] a ciphertext, using the encryption of different secret keys. This results in ciphertexts such that only the encryption of the *desired* secret key (of the sender or the receiver) will decrypt the desired result.

This property allows for oblivious selection! When we obliviously decrypt (via FHE evaluations) using the correct keys, i.e., the keys of the sender and the receiver, the oblivious decryption will return encryption of the correct bits; whereas when we attempt to obliviously decrypt using the other keys, the ciphertext resulting from the computation will have at least one wrong bit. We use this observation to compute a flag that can be used to selectively and obliviously update the balance only of the sender and the receiver.

With this intuition in mind, we proceed with a more detailed description of our payment system $\mathsf{PriFHEte}$. We will describe how a user can (1) create a private account, (2) create a private payment, and (3) how the blockchain nodes process a private payment. A visual of

[5] This is reminiscent of Gentry's [Gen09] Recrypt operation associated with FHE schemes for bootstrapping.

our protocol can be found in Fig. 1

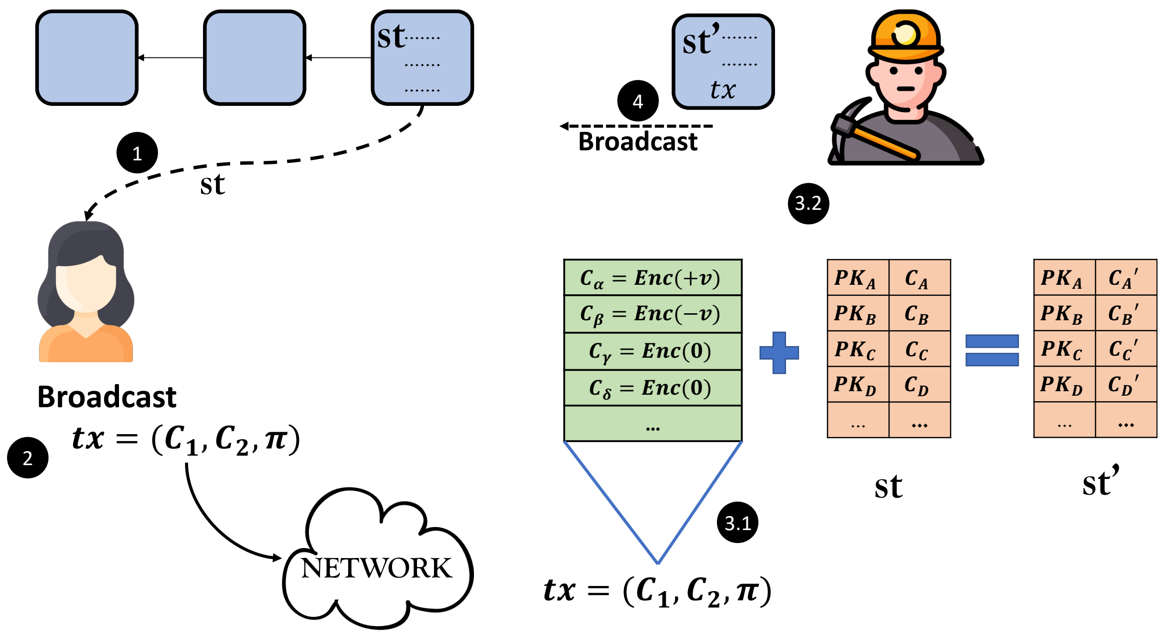

Figure 1: Overview: (1) A user with address PK_B retrieves the latest state commitment - denoted *st*. (2) She computes a transaction C_1, C_2, π, where π is proof that proves that the transaction is valid with respect to the current state. This transaction is broadcast to the network. (3.1) A miner processes this transaction and computes N ciphertexts such that the balance of the sender is decremented by v, the balance of the receiver is incremented by v and an encryption of 0 is added to all the other balances thus re-randomizing them. (3.2) The miner then computes the updated state denoted st' (4) The miner then broadcasts a block with the updated commitment to the state and the transactions.

Creating a Private Account: To create a private account, a user P_i creates two types of keys, a key-pair for an encryption scheme with the wrong-key decryption property (WKEnc): $(\mathsf{WKEnc.pk}_i, \mathsf{WKEnc.sk}_i)$, and a key-pair for a fully homomorphic encryption scheme $(\mathsf{FHE.pk}_i, \mathsf{FHE.sk}_i)$. Furthermore, P_i computes a bit-wise FHE encryption of its WKEnc secret key $\mathsf{WKEnc.sk}_i$ to obtain a vector of $|\mathsf{WKEnc.sk}|$ FHE ciphertexts, that we denote by $\mathsf{k\text{-}ct}_i$. Looking ahead, the FHE encryption of its secret key will be used by the miners to *obliviously decrypt* WKEnc ciphertexts, inside the FHE, in order to decide if this public key is the sender or receiver of the payment. P_i publishes $\mathsf{PK}_i = (\mathsf{WKEnc.pk}_i, \mathsf{FHE.pk}_i, \mathsf{k\text{-}ct}_i)$. The private balance v associated to a public key PK_i is represented as a bit-wise encryption of v, using FHE public key $\mathsf{FHE.pk}_i$. Namely, if $v = (v_1, \ldots, v_\mu)$, the private version is $\mathbf{C}_i = [\mathsf{FHE.Enc}(\mathsf{FHE.pk}_i, v_1), \ldots, \mathsf{FHE.Enc}(\mathsf{FHE.pk}_i, v_\mu)]$. We refer the reader to Figure 2 for the full details of creating an account and registering with the system.

The list of accounts: The table of all accounts consists of N rows, one for each account holder, where N can increase dynamically over time as more accounts are created. Each row consists in the tuple: $[\mathsf{PK}_i \colon \mathbf{C}_i]$

Private Payment. Now, suppose that account holder PK_S (the sender) wants to send the amount x to a receiver PK_R. First PK_S will commit to the public keys of the sender PK_S and the receiver PK_R using a statistically binding commitment scheme. We denote these commitments as C_S and C_R. This cryptographic primitive gives the guarantee that even an unbounded adversary cannot open this commitment to a different public key. Furthermore, PK_S will encrypt the randomness used in the above two commitments with a bit-wise WKEnc encryption, obtaining vectors of ciphertexts $\mathbf{C_{r_S}}$, $\mathbf{C_{r_R}}$. Committing the public-keys involved in the transaction is necessary to bind the transaction to a unique sender and a unique receiver. This rules out the possibility of crafting ciphertexts that can be correctly decrypted with two secret keys (i.e., two wallets) [6]. Next, the sender PK_S creates an WKEnc encryption of the bit-wise representation of the amount x it wishes to transfer to PK_R's account, and another WKEnc encryption of the bit-wise representation of the value $-x$ that should be deducted from PK_S's account. We denote by $\mathbf{C}_C$ (credit) and $\mathbf{C}_D$ (debit) the two vectors of ciphertexts. Finally, the sender computes a succinct zero-knowledge proof of knowledge of the secret key associated with the public key PK_S used to encrypt the debit $\mathbf{C}_D$, that the balance for the account was greater than x before this transaction, and that all ciphertexts were computed correctly. For the exact relation, we refer the reader to Figure 6. A private payment thus consists of the tuple: $(\mathsf{C}_S, \mathsf{C}_R, \mathbf{C_{r_S}}, \mathbf{C_{r_R}}, \mathbf{C_D}, \mathbf{C_C}, \pi)$.

It is important to note at this point that the proof of correctness π must hold with respect to the *latest version* of the "table of private accounts" (e.g., the most updated version of the blockchain), which is succinctly represented by the root of a Merkle tree. Using the latest version can raise subtle concurrency issues like *front-running transactions* and *double-spending*[7]. We will discuss those issues, and what needs to be added to fix them, after we present an overview of how a miner processes transactions.

Processing a Private Payment. Upon receiving the payment transaction $(\mathsf{C}_S, \mathsf{C}_R, \mathbf{C_{r_S}}, \mathbf{C_{r_R}}, \mathbf{C_D}, \mathbf{C_C}, \pi)$ a miner will obliviously update the ciphertext of all account holders as follows. First, recall that each account-holder publishes the FHE encryption of its WKEnc secret key $\mathsf{k\text{-}ct}_i$. For each account holder $\mathsf{PK}_i = (\mathsf{FHE.pk}_i, \mathsf{WKEnc.pk}_i)$ the miner will perform the following five steps. For details we refer the reader to Figure 7 and Figure 8 to see how the state is updated by the miner.

1. First, it tries to obliviously decrypt the randomness encrypted in $\mathbf{C_{r_S}}$, using the secret key encrypted in $\mathsf{k\text{-}ct}_i$. This is done via FHE, namely by evaluating the circuit of the WKEnc decryption function. The result of this operation is FHE ciphertexts of a sequence of λ bits, which is either r_S or is a sequence of λ random bits, let us call it r^* that differs from r_S for at least one bit [8]. We call this ciphertext $\mathbf{C_i^r}$, to denote that this ciphertext could potentially be the randomness used in the commitment C_S.

[6]This attack (and its remedy) was noticed by Biçer and Tschudin in[BT23].

[7]These issues were already outlined Zether [BAZB20].

[8]If it was not the case, this means that we can correctly decrypt using the wrong key, which would break the CPA-security of the encryption scheme.

2. Next, the miner will obliviously compute a commitment to PK_i using the same randomness by evaluating the Com circuit on the PK_i and the above computed $\mathbf{C_i^r}$. We call this ciphertext $\mathbf{C_i}^{com}$. To see why we do this, notice that if the randomness encrypted in $\mathbf{C_i^r}$ matches the randomness used in the commitment of C_S, then $\mathbf{C_i}^{com}$ encrypts C_S.
3. Next, we obliviously xor the bit-string encrypted in $\mathbf{C_i}^{com}$ with the *negation* of the commitment $\overline{\mathsf{C}_S}$. To see why we do this, notice that if the string from $\mathbf{C_i}^{com}$ matches the C_S, then the xor with the negation $\overline{\mathsf{C}_S}$ will result in a string of λ 1s. On the other hand, if the strings don't match (for all remaining public keys performing this xor) the result will be a string that has both zeros and ones. This vector of ciphertexts is denoted $\mathbf{C_i}^{preflag}$.
4. To nullify the noise, and make sure that even one zero disqualifies this public key, we multiply all the ciphertexts in $\mathbf{C_i}^{preflag}$ together. This gives us a flag ciphertext, which we call C_i^{flag}, which is an FHE encryption of the bit 1 if PK_i is the sender of the payment, and of 0 otherwise.
5. Before the miner can use the flag ciphertext C_i^{flag} to update the balance of PK_i, i.e., to update the FHE ciphertext $\mathbf{C_i}$, the miner must transform the WKEnc ciphertext of the amount $-x$, $\mathbf{C_D}$, into an FHE ciphertext encrypted under the same key. This is easily done as above where the circuit of WKEnc decryption function is evaluated on $\mathbf{C_D}$ to get a new ciphertext $\mathbf{C_i^x}$.
6. Now we can finally leverage our flag ciphertext C_i^{flag} and perform a bit-wise multiplication with $\mathbf{C_i^x}$ to obtain an encryption of the value we need to add to the balance $\mathbf{C_i}$ or simply an encryption of 0.
7. The final step is to add $\mathbf{C_i^x}$ to the current balance ciphertext for P_i, and this will complete the update for the debit ciphertext. The same process is then repeated for the credit ciphertext $\mathbf{C_C}$.

For the convenience of the reader, in Appendix B we additionally provide a pictorial representation of all the operations involved in updating the state.

Dealing with Concurrency Issues. As explained above, a payment consists of the tuple $(\mathsf{C}_S, \mathsf{C}_R, \mathbf{C_{r_S}}, \mathbf{C_{r_R}}, \mathbf{C_D}, \mathbf{C_C}, \pi)$, where the proof π asserts that the payer is the owner of the sender's account, and the encrypted balance associated to their account has enough funds to perform the transfer and that the ciphertexts were computed correctly. This proof is computed over the newest version of the table of accounts $\mathcal{T}$ succinctly described by the Merkle Root $\mathsf{rt}_\mathcal{T}$. Now, say that $\mathsf{rt}_\mathcal{T}$ is the most updated version of the table at time t. All payments that are crafted from time t to $t+1$ will have $\mathsf{rt}_\mathcal{T}$, as a reference of the table of accounts, and as a theorem for the proof π. We will now describe the two concurrency-related issues - double spending and front-running and the mechanisms we use to fix them.

Avoiding Double spending : A malicious account holder could craft multiple payments from its account in the interval $[t, t+1]$ referring to the same root $\mathsf{rt}_\mathcal{T}$. In other words, say that an account holder holds a public key pk_i which has a balance of 5 Eth in $\mathsf{rt}_\mathcal{T}$. In the interval $[t, t+1]$, pk_i can create multiple payments for up to 5 Eth and correctly compute the ZK proof π since it is connected to a state $\mathsf{rt}_\mathcal{T}$ where pk_i still owns 5 Eth. All these transactions will be considered valid, and since our transactions are fully anonymous, a miner cannot determine if two transactions originated from the same sender. To address this issue of double-spending, we enforce that each account holder can speak at most once per epoch (η consecutive slots) We achieve this by employing a pseudorandom function (PRF) (a similar approach is used in the anonymous version of Zether [BAZB20], where each transaction includes g^{sk}, where g is a public random nonce that is announced at the beginning of each epoch). In our construction, the account holder commits to the PRF key during setup. The account holder then must attach the deterministic output of the PRF evaluated on the epoch number to each transaction, we denote this value by PRFOut. Since the output of the PRF is deterministic, this prevents the account holder to generate two distinct payments for the same epoch. In the proof, we use the zk property of the NIZK to prove that it knows the opening to a commitment of the PRF key and thus maintains anonymity.

Defenses Against Front-running: Suppose Alice creates an honest transaction in the interval $[t, t+1]$, but the transaction is picked up by a miner at time $t+2$, by which time the state of the blockchain, and thus Alice's ciphertext and the root would have been updated. This would trivially invalidate Alice's proof without any malicious behavior from other parties. We mitigate this front-running problem by allowing an account holder to create a transaction with respect to any of the states in an epoch. This ensures that even if the state has changed, the transaction would be considered valid with respect to one of the previous states in the epoch.

Compactness of our transactions. In our private transactions $\mathsf{tx} = (\mathsf{C}_S, \mathsf{C}_R, \mathbf{C}_{\mathbf{r_S}}, \mathbf{C}_{\mathbf{r_R}}, \mathbf{C_D}, \mathbf{C_C}, \pi)$. The number of ciphertexts in $\mathbf{C}_{\mathbf{r_S}}, \mathbf{C}_{\mathbf{r_R}}$ is λ each, and the number of ciphertexts in $\mathbf{C_D}, \mathbf{C_C}$ is μ, where μ is the number of bits in the maximum possible value that can be transferred. Thus a transaction consists of $2\lambda + 2\mu$ ciphertexts and two commitments, which are independent of the number of users, and a proof π. One can instantiate π, with constant size SNARKs [Gro16, GWC19]. Since we use the Universal Composability framework to prove security, we require that the SNARKs be UC-secure. Work by Kosba et. al. [KZM+] describes how to construct SNARKs with proof size proportional to the size of the witness. This would make our proof size $O(\log N)$ since the witness includes a Merkle path. More recently, succinct UC-secure NIZKs were proposed in the Global Random Oracle Model[GKO+22], where the proof size is constant.

Security of Our Protocol. We prove the security of our protocol in the UC model. There exist multiple definitions of a private ledger functionality [KKKZ19, GRR+21]. We prove security by instantiating the Private Ledger functionality introduced in [KKKZ19]. Note that the Private Ledger functionality captures both the ledger properties (i.e., the underlying consensus protocol) as well as the privacy properties required of the transactions. Since, in this paper, we are providing an account-based payment system on top of any existing ledger, our proof of security will cover only part of the functionality concerning the submission and handling of private transactions.

Furthermore, since the UC specification is often more complex (as it has additional language that is part of the model), for ease of reading, we have provided two descriptions of our protocol. In Section 4, we provide a bare-bones description of the procedures of our protocol (e.g., key registration, mint, transfer, etc), and their implementation. Then in Section 5 we show how the procedures described in Section 4 can be used to instantiate the Private Ledger functionality.

1.3 The significance of this work

The goal of this work is to demonstrate *feasibility* of achieving full anonymity in account-based cryptocurrency. Our key idea has been to enable a global state update (i.e., the entire state is updated per transaction) with a constant number of ciphertexts, by leveraging the power of Fully Homomorphic Encryption.

Our reliance on FHE, however, makes our protocol unlikely to be deployable in practice soon. This naturally raises the question: is a heavy tool such as FHE necessary for the account-based setting and if so, what is the significance of this work for account-based cryptocurrencies?

We do not have a definite answer to this question, but in this paragraph, we will discuss several ideas to contribute to the answer from different angles.

We start with observing that the problem of achieving full privacy in account-based cryptocurrencies in a blockchain environment where a miner works independently, can be abstracted as the problem of anonymously updating an encrypted database containing data from multiple clients (accounts) that is stored on an untrusted server. The miner is the untrusted server, the account-balance table is the database, and the transaction is the message that a party must send to the server in order to anonymously update their entry. Under the assumption that clients do not talk to each other, and there is a single, untrusted server, this problem resembles the problem of server-aided MPC with a single server. For such a problem, the only known solutions are based on Multi-key FHE[LATV12, AJJM20] (such works actually require that clients (account holders) interact with each other even if FHE is used).

Hence, if we stick with the standard blockchain setting where miners work independently and are mutually distrustful it seems that using a powerful tool such as FHE is necessary.

On the other hand, if we allow interactions between miners, or participating of external servers or clients, we could hope for more efficient solutions based on garbled circuits [BHR12]. This direction however does not seem too promising in the blockchain world, where public verifiability is a necessary requirement.

However, we would like to conclude on a positive note, suggesting a completely different approach that could solve the problem at the root – by reducing the state. Indeed, in our previous argument, we were making the assumption that the miners keep the entire state and each party only holds their own secret.

However, in recent years, a different approach has been developed that shifts the work from the miner to the clients, called *stateless cryptocurrencies* (Agrawal et al. [AR20] and Tomescu et al. [TAB$^+$20].) Instead of having the miner update the state upon each transaction, it has the clients update their secrets upon each transaction that is uploaded on the blockchain (that is, even transactions that do not involve the client's balances). The key advantage of such a shift is that it allows a miner to correctly verify the soundness of a state update without having to know the entire state, making the miners' computation very fast and the storage minimal, at the price of having clients making continuous updates to their local state. This is an interesting approach that can potentially allow anonymous updates without heavy machinery. We leave exploring this direction to future work.

1.4 Potential for Deployment

While we acknowledge that PriFHEte requires miners to perform heavy computation and is unlikely to be practical soon, we also would like to discuss avenues for practical deployment that stem from leveraging the power of smart contracts.

Generality of our PriFHEte. An important feature of our protocol is that it is not tied to any blockchain. Our functions can be executed as a smart contract on top of any account-based blockchain that supports smart contracts and does not require any change to the underlying rules of the blockchain. Users create and submit transactions as described above and miners simply execute the smart contract which runs the function to process transactions and update an internally maintained state. We discuss this in more detail in Appendix H. (This is in contrast with solutions for UTXO-based cryptocurrencies that require a significant change in design and resulted in the creation of separate cryptocurrencies, such as Zcash [SCG+14], Monero).

Delegating Miner's Computation. Since PriFHEte can be described as smart contracts that run on the blockchain, the The heavy computations that miners must do when dealing with a PriFHEte's smart contract could be delegated to external servers, by leveraging an emerging technology called zk-rollups [Fou21] (currently available in the Ethereum ecosystem). A zk-rollup is an external server, called rollup operator that maintains the state and executes smart contracts on behalf of the miners. Miners only maintain a succinct representation of this state, typically a Merkle tree root. This aids the storage costs borne by the miner. Users submit their transactions to the rollup operator instead of the blockchain miners. The operator updates the state and broadcasts an updated succinct state, the transactions, and a validity proof proving that the state was updated correctly. A miner now only has to verify this proof and accepts the new succinct state. This aids the heavy computation that needs to be undertaken by the miner. We discuss deployment with rollups in more detail in Appendix H.

Finally, we conclude with a discussion about user's efficiency. Recall that, to craft a payment transaction, the account holder must hold the newest version of her own ciphertext, as well as the most updated version of the Merkle Tree of the entire table of accounts. Since all payments are made public, any user can perform the same computation of the miners for staying updated with the latest state. All previous work [FMMO19, BAZB20] implicitly make the assumption that users will stay updated. In practice, however, it is important to reduce the burden of the computation on the client. Light clients [CBC21] can be used to have the account holder to reliably obtain the information it needs from a blockchain node instead. The question on how/when to ask for this information is as important for guaranteeing anonymity w.r.t. the miners, since asking for the root only when preparing a payment can reveal the network identity of the asker. This question is orthogonal to our work and several existing approaches can be used to address it [XZW+19, WMS+19] hence we do not discuss it further.

1.5 Roadmap

The rest of the paper is organized as follows. Section 2 presents other works that achieve privacy-preserving payments on public ledgers. We present our main cryptographic building blocks in Section 3. We then present our main algorithms that are run by the parties in Section 4. Section 5 presents a UC specification of the protocol Π_{PriFHEte} that makes use of the algorithms presented in Section 4. We sketch a proof overview in Section 6 that realize the $\mathcal{G}_{\text{PL}}$ functionality (presented in Appendix C) and the full proofs are presented in Appendix E.

2 Related Work

Privacy-preserving payments in the account-based model. Fauzi et al. [FMMO19] present QuisQuis where the account is represented as a tuple of public key and a commitment to the balance. To create a transaction, the sender selects a list of valid accounts (that contribute to the anonymity set) and updates these decoy accounts (by re-randomizing them) and the accounts involved in the transaction (by transferring value). Since an adversary cannot learn which accounts were updated with some value, their protocol achieves k-anonymity. Note that the size of the transaction increases with the anonymity guarantee provided. Also, each transaction updates the accounts of the users, and these users are expected to post DestroyAcct to keep the size of the state constant. As observed in [Dia21], since the parties are not incentivized to destroy their old accounts it is unclear if the state of the system is constant. Bünz et al. [BAZB20] present Zether that builds on the same idea as above, except that the balances are stored using ElGamal encryptions. They only achieve confidentiality and not anonymity, therefore each transaction only includes two ciphertexts. In their appendix, they sketch an approach to achieve k-anonymity and this idea was formally analyzed and made more efficient by Diamond [Dia21]. More recently, Guo et. al.[GKP23] present PriDe CT which presents a simplified version of anonymous Zether and enables batching of transactions. But all these approaches still only achieve k-anonymity. In contrast, in our work, we can achieve full privacy in the same setting with transaction size independent of the anonymity set.

Privacy-preserving payments in UTXO-model. Techniques for privacy-enhancing payments in the UTXO require miners to maintain commitments of values as well as the serial numbers of spent coins. Since every transaction in the UTXO model leads to the creation of new coins, the state of the system (consisting of the commitments and the serial numbers) is always increasing. There are mainly two approaches: 1) ones that achieve full privacy - Zerocoin[MGGR13], Zerocash[SCG+14], [Net21] which use zk-snarks[GWC19] and 2) ones that achieve a weaker form of anonymity (k-anonymity) - Monero [NM+16] which use ring signatures.

Our solution does not increase the state of the system with every transaction. The state increases only when a new account joins the system.

Privacy-preserving payments in the account-based model that use UTXOs. Another popular approach to achieve anonymity in the account-based setting is by having users convert funds in their account to private coins and spend these coins in a privacy-preserving way similar to the UTXO setting. This may be deployed as a smart contract as is the case in Zeth [RZ19], AZTEC [Wil18] or standalone - Veksel [CHA22], BlockMaze [GWY+20]. These protocols provide varying guarantees of anonymity. Zeth [RZ19] achieves only receiver anonymity. The sender is not anonymous, since they need to pay gas fees from a public account to execute the smart contract. In Section 4.3 we show how we get around this issue by converting private funds to public gas fees. BlockMaze [GWY+20] and Veksel [CHA22], do not achieve any anonymity. They only guarantee that a sender of a transaction cannot be linked to the recipient of the transaction. In Aztec [Wil18], only the recipient of a transaction is anonymous. Finally, Espresso systems [Tec09] achieve anonymity for the sender (except to a trusted relayer) and the receiver. Our work on the other hand achieves full anonymity for the recipient and sender.

Besides these weaker anonymity guarantees, as noted in Zether [BAZB20], this hybrid approach has several disadvantages based on committed coins. First, storage costs are very expensive in account-based blockchains such as Ethereum, and since the state is always increasing the coin-based solution will be very expensive. Second, using coins creates friction when trying to operate with smart contracts. Finally, in this hybrid approach users now need to keep track of all their unspent coins, instead of maintaining just the secret key of their account. Our work achieves full anonymity and retains many of the benefits of the account-based approach (e.g., the state does not grow, the user does not need to remember all the private coins she possesses, but only needs to remember her secret key).

In Table 1 we compare our work with existing efforts to achieve anonymity in cryptocurrencies.

System	Anonymity	Transaction size	Client Overhead	Miner Overhead
QuisQuis [FMMO19]	k-anonymity	$\mathcal{O}(k)$	$\mathcal{O}(k)$	$\mathcal{O}(k)$
Basic Zether [BAZB20]	None	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Anonymous Zether [Dia21]	k-anonymity	$\mathcal{O}(k)$	$\mathcal{O}(k)$	$\mathcal{O}(k)$
PriDE CT [GKP23]	k-anonymity	$\mathcal{O}(k)$	$\mathcal{O}(k)$	$\mathcal{O}(k)$
Our Work	**Full**	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(N)$

Table 1: Comparison of anonymous account-based cryptocurrency designs with our work.

3 Preliminaries

Fully Homomorphic Encryption. We follow the definition of FHE presented in [BV14]. We use λ as the security parameter and all schemes in this paper encrypt bit-by-bit. A fully homomorphic encryption scheme is a triple of PPT algorithms $\mathsf{FHE} = (\mathsf{FHE.KeyGen}, \mathsf{FHE.Enc}, \mathsf{FHE.Eval})$ which provides CPA security. We present a more formal definition in Appendix A.

Key-Private Public Key encryption with wrong-key decryption. Our constructions use a CPA secure encryption scheme with certain special properties namely that of key-privacy (Def 2) and wrong-key decryption (Def 3). We denote this encryption scheme in the description of our protocols as $(\mathsf{WKEnc.KeyGen}, \mathsf{WKEnc.Enc}, \mathsf{WKEnc.Dec})$. Wrong-key decryption informally states that a ciphertext decrypted with the wrong secret key will always return a valid plaintext and return the correct plaintext only with some negligible advantage ($\frac{1}{2} + \mathsf{negl}$). As noted in [LT22] all the above properties are satisfied by LWE encryption scheme of Regev [Reg10] and Peikert et al[PVW08]. We present notions of key-privacy under chosen plain text attack and the wrong key decryption property in Appendix A. We also present Regev's scheme and a sketch of why Regev's scheme gives a wrong-key decryption property in Appendix F.

Pseudorandom functions with unpredictability under malicious key generation. In our construction, we use a psuedorandom function PRF with an additional property of unpredictability under malicious key generation. The definition for a PRF is that for all PPT distinguishers D, there exists a negligible function negl such that $Pr[D^{\mathsf{PRF}(k,\cdot)}(1^\lambda) = 1] - Pr[D^{f(\cdot)}(1^\lambda) = 1] \leq \mathsf{negl}(\lambda)$, where f is a truly random function. We present more details in Appendix A

Non-interactive zero knowledge. We use the $\mathcal{F}_{\mathsf{nizk}}$ functionality to compute and verify zero-knowledge proofs. We present the ideal functionality in Appendix D (Fig 25). The functionality provides an interface for parties to create proofs π that a statement x is in a given NP language $\mathcal{L}$ with a witness w. Moreover, as proven in [KKKZ19] the $\mathcal{F}_{\mathsf{nizk}}$ functionality can be realized by the SNARK proving system described in [KZM$^+$].

Blockchain A blockchain is an ever-growing hashchain of blocks. We use the following notations in the context of blockchains. Each party P_i may have different version of the blockchain and we use $\mathcal{C}^i_{\mathsf{loc}}$ for user P_i.

There are kinds of parties *miners* and *users* in a system that uses blockchains. The users compute and submit transactions to the network. The miners collect these transactions, validate them and create a block including the valid transactions. A miner then broadcasts the newly created block, thus extending the blockchain. To set some notation, each block is associated with a slot number sl_j, where a slot is unit of time. A set of adjacent η slots is called an epoch ep.

In account-based cryptocurrencies (the setting we consider in this work), a transaction consists of three values: the sender's identity, the receiver's identity and the value to be spent. The miners maintain a list (referred to as the *state*) of (account, balance) pairs. To validate the transaction, the miner checks that the sender of the transaction is not trying to spend more than their balance. If the transaction is valid, the miner then updates the state by deducting the value of the transaction from the sender's balance and adding the same value to the receiver's balance. We denote the state of the cryptocurrency as $\mathcal{T}$. The miners compute a Merkle tree with the elements of $\mathcal{T}$ as the leaves. The root of this Merkle tree(denoted as $\mathsf{rt}_\mathcal{T}$) is also added to every block along with the valid transactions that caused the update to the state. In a privacy-preserving cryptocurrency, we aim to hide the following information included in a payment: the sender's and receiver's identities and the value to be transferred.

The universal composable (UC) framework [Can01] is a model used to define the security properties of complex protocols in a modular way. A definition for an ideal ledger functionality was presented by Badertscher et. al. [BMTZ17] denoted $\mathcal{G}_{\mathrm{LEDGER}}$. Kerber et. al. [KKKZ19] presented a private version of the ledger functionality denoted $\mathcal{G}_{\mathrm{PL}}$ (PL stands for private ledger). We give an overview of this functionality in Section 5 and present the complete functionality in Figure 16.

To ease the presentation, we will denote the state of the blockchain $\mathcal{T}$ as $\mathcal{T}_{\mathsf{privAccounts}} \| \mathcal{T}_{\mathsf{pubAccounts}}$ and $\mathsf{rt}_\mathcal{T} = H(\mathsf{rt}_{\mathcal{T}_{\mathsf{privAccounts}}} \| \mathsf{rt}_{\mathcal{T}_{\mathsf{pubAccounts}}})$. Row i in $\mathcal{T}_{\mathsf{pubAccounts}}$ is of the form $(\mathsf{PK}^{\mathsf{pub}}_i, v_i)$, where $\mathsf{PK}^{\mathsf{pub}}_i$ is the account-holder's public key that is associated with their non-anonymous balance. The account-holder uses the corresponding secret key $\mathsf{SK}^{\mathsf{pub}}_i$ to spend their public balance. Moreover $\mathcal{T}_{\mathsf{privAccounts}}$ includes elements of the form (PK_i, C_i) where C_i is the encrypted balance and PK_i is the public key associated with the account. As above, the account-holder uses the corresponding SK_i to spend an their private balance.

4 The PriFHEte payment system

In this section we present algorithms for the PriFHEte payment system. We first present the interface in Section 4.1 and then instantiate the algorithms in Section 4.2. In Section 5 we will describe how these algorithms will be used to construct an anonymous account-based cryptocurrency protocol.

4.1 Interface for the PriFHEte payment system

Notation: We denote by P_i an account holder. We denote the total number of accounts in the system by NumAccounts. Miners (denoted Q_j) are account holders that additionally update the state. The state maintained by Q_j will be denoted as $\mathcal{T}^j = (\mathcal{T}^j_{\mathsf{privAccounts}} \| \mathcal{T}^j_{\mathsf{pubAccounts}})$. We assume that the parties already have public accounts in the system. Our privacy-preserving payment scheme Π_{PriFHEte} is a tuple of polynomial-time algorithms: (KeyGeneration, Registration, Mint, Transfer, ProcessTransaction).

Key Generation. The algorithm KeyGeneration creates public key and secret key pairs for an account holder.

KeyGeneration$(\lambda) \rightarrow (\mathsf{PK}, \mathsf{SK})$: A user P_i runs KeyGeneration and publishes the public key PK_i, while the SK_i is used to spend the funds that are sent to the account represented by PK_i.

Account registration. The algorithm Registration is run by the miner to register the public key for an account. This algorithm updates the state of the blockchain after initializing the account.

Registration$(\mathsf{PK}, \mathcal{T}_{\mathsf{privAccounts}}) \rightarrow \mathcal{T}'_{\mathsf{privAccounts}}$: A miner Q runs Registration by adding an entry for user with public key PK to the state $\mathcal{T}_{\mathsf{privAccounts}}$.

Minting private funds. The algorithm Mint lets an account-holder transfer funds from a public account to a private account.

Mint$(\mathsf{PK}_i, \mathsf{PK}^{\mathsf{pub}}_i, \mathsf{SK}^{\mathsf{pub}}_i, x, \mathsf{rt}_{\mathcal{T}_{\mathsf{pubAccounts}}}) \rightarrow (\mathsf{tx}_{\mathtt{MINT}}, \sigma)$: A user P_i executes the Mint algorithm to produce a transaction that transfers a value x from the public state to the private state. This algorithm takes as inputs the public key associated to the private account PK_i, the public and secret keys associated to the public account $\mathsf{PK}^{\mathsf{pub}}_i, \mathsf{SK}^{\mathsf{pub}}_i$, the public value x to be transferred to the private account and the root of the public state of the blockchain $\mathsf{rt}_{\mathcal{T}_{\mathsf{pubAccounts}}}$. The algorithm outputs a transaction $\mathsf{tx}_{\mathtt{MINT}}$ and a signature σ on this transaction.

Transferring private funds. The algorithm Transfer allows an account-holder PK_S to transfer private funds to an account associated with PK_R.

Transfer$(\mathsf{PK}_S, \mathsf{SK}_S, \mathsf{PK}_R, x, ep, R, \mathcal{C}^S_{\mathsf{loc}}, \mathsf{path}_i, \mathbf{C_i}) \rightarrow \mathsf{tx}_{\mathtt{TRANSFER}}$: The Transfer algorithm takes as input the sender's account PK_S, the secret key SK_S, the receiver's account PK_R and the value to be transferred x. The algorithm also takes as input the sender's local version of the blockchain $\mathcal{C}^S_{\mathsf{loc}}$, the current epoch number ep, the epoch size η and the entry associated with

PK_S in $\mathcal{T}_{\mathsf{privAccounts}}$, denoted $\mathbf{C_i}$ and the Merkle path from $\mathbf{C_i}$ to $\mathsf{rt}_{\mathcal{T}_{\mathsf{privAccounts}}}$ - denoted path_i. The algorithm outputs a transfer transaction $\mathsf{tx}_{\mathtt{TRANSFER}}$

Verifying transactions and updating state. The algorithm PROCESSTRANSACTION run by a miner Q_j first verifies transactions and then updates the state of the blockchain with valid transactions.

PROCESSTRANSACTION$(\mathsf{tx}, \mathcal{T}^j) \to (\mathcal{T}^j)$: A miner Q_j updates the state $\mathcal{T}^j = \mathcal{T}^j_{\mathsf{pubAccounts}} \| \mathcal{T}^j_{\mathsf{privAccounts}}$ of the blockchain, by taking as input the current state $\mathcal{T}^j$ and a transaction tx.

4.2 Instantiating PriFHEte

We use the following cryptographic building blocks to implement the above-described algorithms: A fully homomorphic encryption scheme - $(\mathsf{FHE.Enc}, \mathsf{FHE.Dec}, \mathsf{FHE.Eval})$. This may be implemented by existing FHE schemes such as the BGV scheme [BGV14]. A key-private encryption scheme for bits with the additional property of wrong key decryption (see Def. 3) which means that even when the ciphertext is decrypted with a wrong key the resultant plaintext is a random valid bit. Such an encryption scheme can be instantiated with PVW LWE-based encryption scheme [PVW08]. A pseudorandom function PRF that is unpredictable under malicious key generation [KKKZ19] with key k. A perfectly binding commitment scheme $(\mathsf{Com}, \mathsf{Verify})$. An ideal functionality $\mathcal{F}_{\mathsf{nizk}}$ that allows users to prove statements. A digital signature scheme $(\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ and collision-resistant hash function $\mathcal{H}$.

Public Parameters. A list of public parameters is available to all users in the system. These are generated at the "start of time". The parameters are: η which denotes the size of each epoch *ep*, a trusted set up (such as CRS) for the non-interactive zero knowledge proofs. Each block corresponds to a slot number denoted *sl*. After every η number of slots, the epoch number is incremented. We now give an overview of the algorithms that we described earlier.

Joining the system (Fig 2) To join the system a party P_i first runs the KEYGENERATION algorithm which generates keys for the fully homomorphic scheme FHE, the encryption scheme WKEnc, a signature scheme and a pseudorandom function. P_i then encrypts each bit of the $\mathsf{WKEnc.sk}_i$ using the FHE public key $\mathsf{FHE.pk}_i$ to obtain a vector of ciphertexts $\mathsf{k\text{-}ct}_\mathbf{i}$ and computes a commitment to this key denoted as $\mathsf{C_{PRF}}$.

P_i then announces its public keys : $(\mathsf{FHE.pk}_i, \mathsf{WKEnc.pk}_i, \mathsf{k\text{-}ct}_i, \mathsf{C_{PRF}})$ and a zero-knowledge proof that the keys were generated correctly: $\pi_{\mathtt{KEYGEN}}$. A miner Q_j registers the party by running the REGISTRATION algorithm where they create an entry for P_i in table $\mathcal{T}^j_{\mathsf{privAccounts}}$. The entry is indexed by the public key PK_i and is initialized with a vector of ciphertexts - that encrypts to 0 under $\mathsf{FHE.pk}_i$. These ciphertexts represent the binary decomposition of the *private* balance of P_i.

$\underline{\text{KEYGENERATION}(\lambda)}$: User P_i does:

1. Generating keys:
   - $(\mathsf{FHE.pk}_i, \mathsf{FHE.sk}_i) \leftarrow \mathsf{FHE.KeyGen}(1^\lambda)$
   - $(\mathsf{WKEnc.pk}_i, \mathsf{WKEnc.sk}_i) \leftarrow \mathsf{WKEnc.KeyGen}(1^\lambda)$
   - $(\mathsf{sk}_i, \mathsf{vk}_i) \leftarrow \mathsf{Sign.KeyGen}(1^\lambda)$
   - $k \leftarrow \mathsf{PRF.KeyGen}(1^\lambda)$
2. Encrypting WKEnc keys:
   - $\mathsf{k\text{-}ct}_\mathbf{i} \leftarrow \{\mathsf{FHE.Enc}(\mathsf{FHE.pk}_i, \mathsf{WKEnc.sk}_i[j])\}_{j=1}^{|\mathsf{WKEnc.sk}_i|}$
3. Committing to the PRF key:
   - $\mathsf{C}_{\mathsf{PRF}} \leftarrow \mathsf{Com}(k; r)$ where $r \leftarrow \{0,1\}^\lambda$
4. Compute a zero knowledge proof that the keys were generated correctly:
   - Let $x := \mathsf{FHE.pk}_i, \mathsf{WKEnc.pk}_i, \mathsf{k\text{-}ct}_i$
   - Let $w := \mathsf{FHE.sk}_i, \mathsf{WKEnc.sk}_i$
   - Send $(\mathsf{Prove}, \mathsf{sid}, x, w)$ to $\mathcal{F}_{\mathsf{nizk}}$ to prove that (x, w) satisfies relation $\mathcal{R}_{\texttt{TRANSFER}}$ (Fig 3) and receive $\pi_{\texttt{KEYGEN}}$.
5. Return $\mathsf{PK}_i := (\mathsf{k\text{-}ct}_i, \mathsf{FHE.pk}_i, \mathsf{WKEnc.pk}_i, \mathsf{vk}_i, \mathsf{C}_{\mathsf{PRF}})$, $\mathsf{SK}_i = (\mathsf{FHE.sk}_i, \mathsf{WKEnc.sk}_i, \mathsf{sk}_i, k)$ and $\pi_{\texttt{KEYGEN}}$

$\underline{\text{REGISTRATION}(\mathsf{PK}_i, \mathcal{T}^j_{\mathsf{privAccounts}})}$ The miner Q_j upon receiving PK_i:

1. Parse PK_i as $(\mathsf{k\text{-}ct}_i, \mathsf{FHE.pk}_i, \mathsf{vk}_i, \mathsf{WKEnc.pk}_i, \mathsf{C}_{\mathsf{PRF}})$
2. For $j \in [\lambda]$ compute $C_{i,j} \leftarrow \mathsf{FHE.Enc}(\mathsf{FHE.pk}_i, 0)$.
3. Set $\mathbf{C}_i := (C_{i,1}, \ldots, C_{i,\lambda})$
4. Update $\mathcal{T}^j_{\mathsf{privAccounts}} := \mathcal{T}^j_{\mathsf{privAccounts}} \cup \{(\mathsf{PK}_i, \mathbf{C}_i)\}$
5. Output $\mathcal{T}^j_{\mathsf{privAccounts}}$.

Figure 2: Joining the system

Statement: $x := \mathsf{FHE.pk}_i, \mathsf{WKEnc.pk}_i, \mathsf{k\text{-}ct}_i$, Witness: $w := \mathsf{FHE.sk}_i, \mathsf{WKEnc.sk}_i$, Relation $\mathcal{R}_{\texttt{KEYGEN}}$:

1. $\mathsf{k\text{-}ct}_i$ is the encryption of the bit-representation of the secret key $\mathsf{WKEnc.sk}_i$ under the FHE public key. $\mathsf{k\text{-}ct}_i = \{\mathsf{FHE.Enc}(\mathsf{FHE.pk}_i, b_j)\}$ such that $\sum_{j=0}^{\lambda} b_j \times 2^j = \mathsf{WKEnc.sk}_i$
2. $\mathsf{WKEnc.sk}_i$ is the secret key that corresponds to $\mathsf{WKEnc.pk}_i$ $(\mathsf{WKEnc.sk}_i, \mathsf{WKEnc}, \mathsf{pk}_i) \in \mathsf{SUPP}(\mathsf{KeyGen}(1^\lambda))$

Figure 3: The relation $\mathcal{R}_{\texttt{KEYGEN}}$

Public transfers (Fig 4) To add funds (say an amount x) to their private balance, a party P_i runs the MINT algorithm, which transfers funds from the public account to the private account. A miner Q_j upon receiving this transaction verifies that the transaction is valid (by running $\mathsf{ValidTx}$) and that the public account $\mathsf{PK}_i^{\mathsf{pub}}$ indeed has public funds greater than the minted value x by running the PROCESSTRANSACTION algorithm. (See Fig 7). If the transaction is valid, Q_j computes encryptions of a binary decomposition of x using $\mathsf{FHE.pk}_i$ and homomorphically adds these ciphertexts to $\mathcal{T}_{\mathsf{privAccounts}}[\mathsf{PK}_i]$.

$\underline{\textsc{Mint}(x, \mathsf{PK}_i, \mathsf{PK}_i^{\mathsf{pub}}, \mathsf{SK}_i^{\mathsf{pub}}, \mathsf{rt}_{\mathcal{T}_{\mathsf{pubAccounts}}})}$ User P_i does:

1. Set $\mathsf{tx}_{\textsc{Mint}} = (x, \mathsf{PK}_i^{\mathsf{pub}}, \mathsf{rt}_{\mathcal{T}_{\mathsf{pubAccounts}}})$
2. Compute $\sigma = \mathsf{Sign}(\mathsf{sk}_i, \mathsf{tx}_{\textsc{Mint}})$ and broadcast $(\mathsf{tx}_{\textsc{Mint}}, \sigma)$

Figure 4: Transferring funds from public to private account

Private Transfers (Fig 5) User P_S executes the TRANSFER algorithm to privately transfer funds to P_R. P_S first receives the latest blockchain $\mathcal{C}$ and $\mathcal{T}_{\mathsf{privAccounts}}[\mathsf{PK}_S] = \mathbf{C}_S$ and a path_S (from the root $\mathsf{rt}_{\mathcal{T}_{\mathsf{privAccounts}}}$ to the leaf $\mathbf{C}_S$) from a full node. We note that there exist works[XZW+19, WMS+19] that use PIR/ORAM-like techniques to retrieve account state in a privacy-preserving way. P_S transfers funds to P_R as follows:

P_S first commits to the sender's public key (pk_S) and receiver's public key (pk_R) using randomness r_S and r_R respectively. P_S then encrypts r_S and r_R (binary decomposed) under $\mathsf{WKEnc.pk}_S$ and $\mathsf{WKEnc.pk}_R$ respectively. P_S then encrypts the value to be credited (denoted x) under the receiver's public key and the value to be debited under the sender's public key. The value x is upper-bounded by MAX (the maximum possible value that can be transferred) which is μ bits long. The user P_S then proves that the transaction is computed correctly using a zero-knowledge proof, which we describe in more detail below.

TRANSFER$(\mathsf{PK}_S, \mathsf{SK}_S, \mathsf{PK}_R, x, ep, R, \mathcal{C}_{\mathsf{loc}}, \mathsf{path}_i, \mathbf{C_i})$ User P_i does:

1. Let $\mathsf{rt} = \mathsf{rt}_{\mathcal{T}_{\mathsf{pubAccounts}}} \| \mathsf{rt}_{\mathcal{T}_{\mathsf{privAccounts}}}$ be the root at current slot sl in $\mathcal{C}_{\mathsf{loc}}$
2. Let locally stored $\mathbf{C_i} = \mathcal{T}_{\mathsf{privAccounts}}[\mathsf{PK}_S]$ at slot number sl
3. Commit sender's identity: compute $\mathsf{C}_S = \mathsf{Com}(\mathsf{pk}_S, r_S)$
4. Commit receiver's identity: compute $\mathsf{C}_R = \mathsf{Com}(\mathsf{pk}_R, r_R)$
5. Encrypt receiver randomness: for $i \in [\lambda]$, compute $C_{r_R,i} = \mathsf{WKEnc.Enc}(r_R[i])$. Let $\mathbf{C}_{r_R} := (C_{r_R,1}, \ldots, C_{r_R,\lambda})$
6. Encrypt sender randomness: for $i \in [\lambda]$, compute $C_{r_S,i} = \mathsf{WKEnc.Enc}(r_S[i])$. Let $\mathbf{C}_{r_S} := (C_{r_S,1}, \ldots, C_{r_S,\lambda})$
7. *Encrypt debited value*
   For $i \in [\mu]$, compute $C_{D,i} = \mathsf{WKEnc.Enc}(\mathsf{WKEnc.pk}_S, b_i)$, where $b_i = x[i]$. Let $\mathbf{C}_D := (C_{D,1}, \ldots, C_{D,\mu})$
8. Encrypt credited value
   For $i \in [\mu]$, compute $C_{C,i} = \mathsf{WKEnc.Enc}(\mathsf{WKEnc.pk}_R, b_i)$, where $b_i = x[i]$. Let $\mathbf{C}_C := (C_{C,1}, \ldots, C_{C,\mu})$
9. Compute PRF output: Compute $\mathsf{PRFOut} = \mathsf{PRF}(k, ep)$
10. Compute a zero-knowledge proof for transaction validity:
    Let $x = \{\mathsf{C}_S, \mathsf{C}_R, \mathbf{C_{r_S}}, \mathbf{C_{r_R}}, \mathbf{C_D}, \mathbf{C_C}, \mathsf{PRFOut}, \mathsf{rt}, ep\}$. Let $w = \{\mathsf{PK}_S, \mathsf{SK}_S, \mathsf{PK}_R, r_R, s_R, x, k, \mathsf{C}_{\mathsf{PRF}}, v_S, \mathbf{C_i}, \mathsf{path}\}$. Send $(\mathsf{Prove}, \mathsf{sid}, x, w)$ to $\mathcal{F}_{\mathsf{nizk}}$ to prove that (x, w) satisfies relation $\mathcal{R}_{\mathtt{TRANSFER}}$ (Fig 6) and receive π.
11. Return $\mathsf{tx} = (\mathsf{C}_S, \mathsf{C}_R, \mathbf{C_{r_S}}, \mathbf{C_{r_R}}, \mathbf{C_D}, \mathbf{C_C}, \mathsf{PRFOut}, \pi)$

Figure 5: Transfer algorithm

As discussed in the introduction (c.f. *Concurrency Issues*), we must ensure that a malicious sender cannot double-spend from their account. We resolve this issue by ensuring that a party can submit only up to one transaction per epoch. We achieve this by including the output of a pseudorandom function PRF with every transaction.

The PRF takes as input the current epoch ep and therefore if a user attempts to speak twice within the same epoch, a miner would see the same PRF output (since PRFs are deterministic) and rejects the second transaction. We prevent denial-of-service attacks where an adversary front-runs a user's transaction by submitting an adversarial transaction with the same PRFOut as the target by using PRFs that are secure under malicious key generation.

Zero Knowledge Proofs for Transfer Transactions Our construction invokes the $\mathcal{F}_{\mathsf{nizk}}$ functionality for a specific relation (see Fig. 6). A transfer transaction is of the form: $(\mathsf{C}_S, \mathsf{C}_R, \mathbf{C_{r_S}}, \mathbf{C_{r_R}}, \mathbf{C_D}, \mathbf{C_C}, \mathsf{PRFOut}, \pi)$. In this transaction, the proof π needs to prove that certain conditions are satisfied by the transfer transaction. The conditions are: (a) The sender has a balance greater than the value to be transferred at epoch ep (b) the value debited is equal to the value credited (c) the sender speaks only once in the current epoch (d) the credited value is positive.

The relation

- Statement: $x = (\mathsf{C}_S, \mathsf{C}_R, \mathbf{C_{r_S}}, \mathbf{C_{r_R}}, \mathbf{C_C}, \mathbf{C_D}, \mathsf{PRFOut}, \mathsf{rt}_{\mathcal{T}_{\mathsf{privAccounts}}}, ep)$. The statement specifies the commitments of the sender and receiver identities – $\mathsf{C}_S, \mathsf{C}_R$ and encryption to the randomness of the commitments – $\mathbf{C_{r_S}}, \mathbf{C_{r_R}}$, encryptions of the values to be credited and debited – $\mathbf{C_D}, \mathbf{C_C}$, the output of the PRF – (PRFOut), the root of a Merkle tree over private state – ($\mathcal{T}_{\mathsf{privAccounts}}$) in the epoch ep denoted $\mathsf{rt}_{\mathcal{T}_{\mathsf{privAccounts}}}$.

- Witness: $w = (\mathsf{PK}_S, \mathsf{SK}_S, \mathsf{PK}_R, r_S, r_R, x, v_S, \mathbf{C}, \mathsf{path}, r_{\mathsf{PRF}})$ where $\mathsf{PK}_S = (\mathsf{k\text{-}ct}_S, \mathsf{FHE.pk}_S, \mathsf{WKEnc.pk}_S, \mathsf{vk}_S, \mathsf{C}_{\mathsf{PRF}})$. The witness specifies the public keys of the sender and the receiver, the value to be transferred, the balance and the entry in the private state corresponding to PK_S and an authentication path from the sender's entry in $\mathcal{T}_{\mathsf{privAccounts}}$ to the root of the Merkle tree on $\mathcal{T}_{\mathsf{privAccounts}}$.

Given an instance x, a witness w is valid for x if the relation specified in Figure 6 holds.

Relation $\mathcal{R}_{\mathtt{TRANSFER}}$:

- C_S is the commitment to pk_S with randomness r_S , i.e. $\mathsf{C}_S = \mathsf{Com}(\mathsf{pk}_S, r_S)$
- C_R is the commitment to pk_R with randomness r_R , i.e $\mathsf{C}_R = \mathsf{Com}(\mathsf{pk}_R, r_R)$
- $\mathbf{C_{r_R}}$ is the encryption of the bit-representation of the randomness used for the receiver (r_R) encrypted under the public key $\mathsf{WKEnc.pk}_R$. i.e. $\mathbf{C_{r_R}} = \{\mathsf{WKEnc.Enc}(\mathsf{WKEnc.pk}_R, b_j)\}$ such that $\sum_{j=0}^{\lambda} b_j \times 2^j = r_R$
- $\mathbf{C_{r_S}}$ is the encryption of the bit-representation of the randomness used for the receiver (r_S) encrypted under the public key $\mathsf{WKEnc.pk}_S$. i.e. $\mathbf{C_{r_S}} = \{\mathsf{WKEnc.Enc}(\mathsf{WKEnc.pk}_S, b_j)\}$ such that $\sum_{j=0}^{\lambda} b_j \times 2^j = r_S$
- $\mathbf{C_C}$ is the encryption of the bit-representation of the credited value x to the receiver's account, encrypted under the public key of the receiver $\mathsf{WKEnc.pk}_R$ i.e. $\mathbf{C_C} = \{\mathsf{WKEnc.Enc}(\mathsf{WKEnc.pk}_R, b_j)\}$ such that $\sum_{j=0}^{\mu} b_j \times 2^j = x$
- $\mathbf{C_D}$ is the encryption of the bit-representation of the debited value x from the sender's account, encrypted under the public key of the sender $\mathsf{WKEnc.pk}_S$. i.e. $\mathbf{C_D} = \{\mathsf{WKEnc.Enc}(\mathsf{WKEnc.pk}_S, b_j)\}$ such that $\sum_{j=0}^{\mu} b_j \times 2^j = x$
- The value x is not negative and is less than the max possible value MAX. $x \in [0, \mathsf{MAX}]$
- The sender knows the secret key associated with the account from which the funds are to be debited. $\mathsf{ValidPath}((\mathsf{PK}_S, \mathbf{C}), \mathsf{path}, \mathsf{rt}_{\mathcal{T}_{\mathsf{privAccounts}}}) = 1 \wedge \mathsf{PK}_S = (\mathsf{FHE.pk}_S, \mathsf{WKEnc.pk}_S, \mathsf{k\text{-}ct}_S, \mathsf{C}_{\mathsf{PRF}})$
- The balance associated with the sender's account is greater than the value x i.e. $\mathsf{FHE.Dec}(\mathsf{FHE.sk}_S, C) = v \wedge v - x \in [0, \mathsf{MAX}]$
- The PRF output was computed correctly : $\mathsf{PRFOut} = \mathsf{PRF}(k, ep) \wedge \mathsf{C}^S_{\mathsf{PRF}} = \mathsf{Com}(k; r_{\mathsf{PRF}})$

Figure 6: The relation $\mathcal{R}_{\mathtt{TRANSFER}}$

Private to Public transfer A user P_S can transfer funds of value x from their private account PK_S to a public account vk^*, by referencing their private account and the public account in the following way: P_S will first compute a PRF on the current epoch and will compute a zero-knowledge proof proving that the balance in $\mathbf{C}_S$ is greater than x, that the evaluated the PRF on the current epoch and that they know the secret keys corresponding to the public key associated with their account. The transaction is $\mathsf{tx} = (\mathtt{PUB\text{-}TRANSFER}, \mathsf{PK}_S, \mathsf{vk}^*, x, \mathsf{PRFOut}, \pi_{pub})$.

PROCESSTRANSACTION$(\mathsf{tx}, \mathcal{T}^j)$ Upon receipt of a tx, a miner Q_j does the following:

1. Parse $\mathcal{T}^j$ as $\mathcal{T}^j_{\mathsf{pubAccounts}} \| \mathcal{T}^j_{\mathsf{privAccounts}}$
2. If tx is of type `MINT`:
   (a) Parse tx as $((x, \mathsf{PK}_i, \mathsf{rt}_{\mathcal{T}_{\mathsf{pubAccounts}}}), \sigma)$
   (b) Check that $\mathsf{Verify}(\mathsf{vk}_i, (x, \mathsf{PK}_i, \mathsf{rt}_{\mathcal{T}_{\mathsf{pubAccounts}}}), \sigma) = 1$
   (c) Check that $\mathsf{rt}_{\mathcal{T}_{\mathsf{pubAccounts}}} = \mathsf{MerkleCRH}(\mathcal{T}^j_{\mathsf{pubAccounts}})$
   (d) Check that $\mathcal{T}^j_{\mathsf{pubAccounts}}[\mathsf{PK}_i] > x$
   (e) Let $x_1, \ldots, x_\mu$ be the bit-decomposition of x. Let $\mathcal{T}^j_{\mathsf{privAccounts}}[\mathsf{PK}_i] = \mathbf{C_i}$.
   (f) For $j \in [\mu]$:
      i. Compute $C'_{i,j} = \mathsf{FHE.Enc}(\mathsf{FHE.pk}_i, x_j)$
      ii. Update $\mathbf{C_i}[j] = \mathbf{C_i}[j] + C'_{i,j}$
   (g) Output $\mathcal{T}^j = \mathcal{T}^j_{\mathsf{pubAccounts}} \| \mathcal{T}^j_{\mathsf{privAccounts}}$
   (h) Update $\mathcal{T}^j_{\mathsf{pubAccounts}}[\mathsf{PK}_i] = \mathcal{T}^j_{\mathsf{pubAccounts}}[\mathsf{PK}_i] - x$
3. If tx is of type `TRANSFER`:
   (a) Run $\mathsf{ValidTx}(\mathsf{tx}_i, \mathcal{T}^j)$
   (b) For $i \in [\mathsf{NumAccounts}]$: compute $\mathcal{T}^j_{\mathsf{privAccounts}}[i] = \mathsf{UpdateCiphertext}(\mathcal{T}^j_{\mathsf{privAccounts}}[i], \mathsf{tx}, \mathsf{PK}_i)$
   (c) Output $\mathcal{T}^j = \mathcal{T}^j_{\mathsf{pubAccounts}} \| \mathcal{T}^j_{\mathsf{privAccounts}}$
4. If tx is of type `PUB-TRANSFER`:
   (a) Parse tx as $(\mathsf{PK}_i, \mathsf{vk}^*, x, \mathsf{PRFOut}, \pi_{pub})$. Check that π_{pub} is valid.
   (b) Let $x_1, \ldots, x_\mu$ be the bit-decomposition of x. Let $\mathcal{T}^j_{\mathsf{privAccounts}}[\mathsf{PK}_i] = \mathbf{C_i}$.
   (c) For $j \in [\mu]$:
      i. Compute $C'_{i,j} = \mathsf{FHE.Enc}(\mathsf{FHE.pk}_i, x_j)$
      ii. Update $\mathbf{C_i}[j] = \mathbf{C_i}[j] - C'_{i,j}$
   (d) Output $\mathcal{T}^j = \mathcal{T}^j_{\mathsf{pubAccounts}} \| \mathcal{T}^j_{\mathsf{privAccounts}}$
   (e) Update $\mathcal{T}^j_{\mathsf{pubAccounts}}[\mathsf{PK}^*] = \mathcal{T}^j_{\mathsf{pubAccounts}}[\mathsf{PK}^*] + x$

Figure 7: Verification of transactions and updating the state

Updating the state. A miner Q_j upon receiving a transaction ($\mathsf{tx}_{\text{MINT}}$ or $\mathsf{tx}_{\text{TRANSFER}}$) updates the state by running the PROCESSTRANSACTION algorithm. As the identities and the values are encrypted using a key-private encryption scheme, Q_j does not know which entries to update in table $\mathcal{T}_{\mathsf{privAccounts}}$. Therefore the Q_j must update all the entries in $\mathcal{T}_{\mathsf{privAccounts}}$ as in Figure 8. We present the proof of correctness of this update in Appendix B and also present a simple example in Fig 15 and Fig 14 that may aid the reader in understanding the UpdateCiphertext function.

$\underline{\mathsf{UpdateCiphertext}(\mathbf{C_i}, \mathsf{tx}, \mathsf{PK}_i)}$

1. Parse tx as $(\mathsf{C}_S, \mathsf{C}_R, \mathbf{C_{s_S}}, \mathbf{C_{r_R}}, \mathbf{C}_D, \mathbf{C}_C, \mathsf{PRFOut}, \pi)$
2. Parse PK_i as $(\mathsf{k\text{-}ct}_i, \mathsf{FHE.pk}_i, \mathsf{WKEnc.pk}_i, \mathsf{vk}_i, \mathsf{C_{PRF}})$
3. *Obliviously decrypt randomness* ($\mathbf{C_{r_S}}$) *ciphertext with* $\mathsf{k\text{-}ct}_i$ *(encryption of* $\mathsf{WKEnc.sk}_i$*) to get an encryption of some* r^* *under* $\mathsf{FHE.pk}_i$
   - For $j \in [\lambda]$, compute $C^r_{i,j} = \mathsf{FHE.Eval}(\mathsf{FHE.pk}_i, \mathsf{WKEnc.Dec}, (\mathsf{k\text{-}ct}_i, \mathbf{C}_{r_S}[j]))$
   - Compute $\mathbf{C^r_i} = (C^r_{i,1}, \ldots, C^r_{i,\lambda})$

   `// if` i `corresponds to` P_S`, then` $\mathbf{C^{id}_i}$ `is an encryption of the sender's randomness, i.e` $r^* = r_S$
4. *Obliviously compute a commitment to the* pk_i using encryptions of r_S denoted $\mathbf{C_i}^r$: Compute $\mathbf{C}^{com}_i = \mathsf{FHE.Eval}(\mathsf{FHE.pk}_i, \mathsf{Com}, (\mathsf{pk}_i, (C^r_{i,1}, \ldots, C^r_{i,\lambda}))$. Let $\mathbf{C}_i$ denote the plaintext corresponding to $\mathbf{C}^{com}_i$
5. *Obliviously compute* $\overline{\mathsf{C}_S} \oplus \mathbf{C}_i$ *bitwise (where* $\overline{\mathsf{C}_S}$ *is the negated bitwise decomposition of* C_S *) as follows:*
   - For $j \in [\lambda]$, compute $C^{preflag}_{i,j} = \mathsf{FHE.Eval}(\mathsf{FHE.pk}_i, \oplus, (\overline{\mathsf{C}_S}[j], \mathbf{C^{com}_{i,j}}))$
   - Let $\mathbf{C^{preflag}_i} = (C^{preflag}_{i,1}, \ldots, C^{preflag}_{i,\lambda})$

   `// if` i `corresponds to` P_S`,` $\mathbf{C^{preflag}_i}$ `is an encryption of all ones`
6. *Obliviously multiply the bits of* $preflag$ *to get a* flag *bit*
   - Compute $C^{\mathsf{flag}}_i = \mathsf{FHE.Eval}(\mathsf{FHE.pk}_i, \times, (C^{preflag}_{i,1}, \ldots, C^{preflag}_{i,\lambda}))$

   `// if` i `corresponds to` P_S`,` C^{flag}_i `is an encryption of 1, else encryption of 0`
7. *Obliviously decrypt value to be debited* $(\mathbf{C_D})$ *with* $\mathsf{k\text{-}ct}_i$ *to get an encryption of some* x^* *under* $\mathsf{FHE.pk}_i$
   - For $j \in [\mu]$, compute $C^x_{i,j} = \mathsf{FHE.Eval}(\mathsf{FHE.pk}_i, \mathsf{WKEnc.Dec}, (\mathsf{k\text{-}ct}_i, \mathbf{C}_D[j]))$
   - Set $\mathbf{C^x_i} = (C^x_{i,1}, \ldots, C^x_{i,\mu})$

   `// if` i `corresponds to` P_S`,` $\mathbf{C^x_i}$ `is an encryption of the value` x`, i.e.` $x^* = x$`, else` x^* `is random`
8. *Obliviously multiply the* flag *bit with* x^*
   - For $j \in [\mu]$, compute $C^{upd}_{i,j} = \mathsf{FHE.Eval}(\mathsf{FHE.pk}_i, \times, (\mathbf{C}^x_{i,j}, C_{\mathsf{flag}}))$
   - Set $\mathbf{C^{upd}_i} = (C^{upd}_{i,1}, \ldots, C^{upd}_{i,\mu})$

   `// if` i `corresponds to` P_S`,` $\mathbf{C^{upd}_i}$ `is an encryption of the value` x`, else 0`
9. *Obliviously subtract* x^* *from the balance of* P_i
   - For $j \in [\mu]$, compute $\mathsf{FHE.Eval}(\mathsf{FHE.pk}_i, \mathsf{FullSubtracter}$[a]$, (C_{i,j}, C^{upd}_{i,j}))$
   - Set $\mathbf{C_i} = (C_{i,1}, \ldots, C_{i,\mu})$

   `// if` i `corresponds to` P_S`, the balance of` P_i `is subtracted by` x`, else the balance of` P_i `stays the same (0 is subtracted from the balance)`

   Do the same computations as above with $(\mathsf{C}_R, \mathbf{C_{r_R}}, \mathbf{C_C})$ instead of $(\mathsf{C}_S, \mathbf{C_{s_S}}, \mathbf{C_D})$, except that in Step 8, obliviously add $(x^* \times \mathsf{flag})$ to the balance of P_i, i.e. compute $\mathsf{FHE.Eval}(\mathsf{FHE.pk}_i, \mathsf{FullAdder}, (C_{i,j}, C^{upd}_{i,j}))$

[a] for completeness, we present the logic for full adder and full subtracter in Appendix G

Figure 8: Updating the private state entry $\mathcal{T}_{\mathsf{privAccounts}}[\mathsf{PK}_i]$ with a transaction tx

4.3 Practical Considerations

Transaction size and processing time. We present an estimate on the size of our transactions and the time taken to process a transaction. A transfer transaction is of the form $\mathsf{tx} = (\mathsf{C}_S, \mathsf{C}_R, \mathbf{C}_{\mathbf{r_S}}, \mathbf{C}_{\mathbf{r_R}}, \mathbf{C_D}, \mathbf{C_C}, \mathsf{PRFOut}, \pi)$. The commitments C_S and C_R are two group elements each of size $\lambda = 128$ bits. Thus the commitments are of size 512 bits. The ciphertexts $\mathbf{C}_{\mathbf{r_S}}$ and $\mathbf{C}_{\mathbf{r_R}}$ encrypt vectors of size $\lambda = 128$ and $\mathbf{C_D}$, $\mathbf{C_C}$ encrypt vectors of size $\mu = 20$. PVW [PVW08] ciphertexts ($\in (\mathbb{Z}_q^\lambda, \mathbb{Z}_q^\ell)$) present an encryption scheme where we can efficiently pack these ciphertexts into one ciphertext. Therefore $|\mathbf{C}_{\mathbf{r_S}} + \mathbf{C_D}| : (\lambda + \mu) \times \lambda + (\lambda \times \lambda)$ bits = $(128 + 20) \times 128 + 128 \times 128 = 8832$ bytes and $|\mathbf{C}_{\mathbf{r_R}} + \mathbf{C_D}| = 8832$ bytes. Assuming Groth16[Gro16] proofs and SHA for the PRF, the transaction size is approximately 18KB.

Paying gas fees. In the presentation of our protocol, we don't specify how parties can pay gas fees to the miner as part of the transaction. We can add a public component to the transaction as follows: The sender adds the gas value in the clear, and encrypts $gas + x$ as the debited value. The zero-knowledge proof now proves that sum of the x and the public gas fee is equal to the debited value and that $x + gas$ is less than the balance of the sender.

Processing time: As for the time taken to process a transaction we present a back of an envelope computation. The biggest factor is the oblivious detection step which is the same as in OMR[LT22]. Using their measurements we observe that it takes 0.0099s per message to detect correctly when there are 50 messages pertinent to a receiver from a pool of 500,000 messages and the number of ciphertexts per clue is 4. Thus the time taken to do an oblivious detection per message is approximately $0.0099 \times 50 \times 148/4s = 18.4s$ when the set size is 500,000. Since we need 4 re-encryptions per transaction, we roughly estimate that it will take $75s$ to process each transaction, when $N = 500,000$. Since this is per-recipient, it take $75 * M$ seconds, where M is the number of parties availing this service.

5 UC-secure privacy-preserving payments

In the previous section we presented algorithms for PriFHEte payment system. To show that our algorithms can be used to instantiate a privacy-preserving account-based cryptocurrency, we present a UC protocol that makes use of the algorithms to realize the $\mathcal{G}_{\mathrm{PL}}$ ideal functionality. In this section we first describe the $\mathcal{G}_{\mathrm{PL}}$ ideal functionality, and then describe how the PriFHEte algorithms will be used to construct a protocol that will realize the $\mathcal{G}_{\mathrm{PL}}$ functionality.

5.1 The $\mathcal{G}_{\mathrm{PL}}$ functionality [KKKZ19]

The $\mathcal{G}_{\mathrm{PL}}$ functionality (Figure 16 and 17) captures an ideal private ledger functionality. We describe the different interfaces of the functionality by separately considering the transaction layer and the consensus layer. Before we explain the interface, we describe the variables associated with the functionality: the state is the state of the ledger that includes blocks of transactions and the buffer is a list of unconfirmed transactions that have not yet been added to the state.

In the transaction layer, a party should be able to submit a transaction. The $\mathcal{G}_{\mathrm{PL}}$ functionality therefore includes a SUBMIT command in its interface that allows parties to submit their transactions. The $\mathcal{G}_{\text{PL}}$ functionality on receiving the SUBMIT command, creates a transaction ID, checks if the transaction is valid using the ValidTx predicate. We note that ValidTx is specific to the protocol that realizes the functionality. In our setting this predicate is instantiated as in Figure 18. The predicate returns true only if the value of the transaction is less than the balance of the sending account and that there is no other transaction from this sender in the current epoch. The adversary is informed that a transaction was received and a *blinded* version of the transaction is sent to the adversary. Parties should also be able to join the system at any point in time. Parties join the system by simply registering with the $\mathcal{G}_{\text{PL}}$ functionality.

In the consensus layer, the functionality guarantees that the parties agree on a common state. But this is not possible in the real-world due to network delays or an adversarial influence. Therefore, the functionality guarantees that there is a prefix of the state that is common to all parties. Since different parties may have different local chains, a pointer pt_i denotes length of the local chain of P_i. To read the state of the ledger, the party issues a READ command and is returned a blinded version of the state upto either block number pt_i or $|\mathsf{state}|$ (whichever is smaller). The adversary is given the power to determine the view of all parties as long as they have a common prefix. The adversary uses the SET-SLACK and DESYNC-STATE interfaces to achieve this.

In this description, we have not yet discussed how the ideal functionality extends the state with new blocks of transactions. In the real-world a party may be selected to propose the next block on the chain depending on some lottery protocol that is defined with respect to the consensus protocol. Similarly, in the ideal world a party sends the MAINTAIN-LEDGER command to the $\mathcal{G}_{\text{PL}}$ functionality. The functionality records this command, and informs the ideal-world adversary of this command. A new block is then proposed by the ideal-world adversary using the NEXT-BLOCK command. This new block is a list of transactions along with a flag called hFlag that indicates if the block is proposed on behalf of an honest party or malicious party. The ideal functionality records this block. When the ideal functionality is queried with any command, the functionality updates the state with these blocks. Note that an adversary can of course propose *bad* blocks that have illegal transactions or transactions that are inconsistent with the state. The $\mathcal{G}_{\text{PL}}$ functionality therefore evaluates an ExtendPolicy function on the block. This function checks if the block is valid and if not, proposes a default block that is used to extend the state of the system.

5.2 Protocol Π_{PriFHEte}

Now that we have explained the $\mathcal{G}_{\text{PL}}$ functionality, we are ready to present our main protocol. More specifically, we will present how we integrate the PriFHEte algorithms from Section 4 in the main protocol. We will then prove that this protocol realizes the $\mathcal{G}_{\text{PL}}$ ideal functionality.

Registration/Deregistration: Upon receiving ($\texttt{REGISTER}, \mathcal{R}$) where $\mathcal{R} \in \{\mathcal{G}_{\mathsf{clock}}, \mathcal{G}_{\mathrm{PL}}\}$ from the environment $\mathcal{Z}$, a party P_i does:

- if $\mathcal{R} = \mathcal{G}_{\mathsf{clock}}$, register with the $\mathcal{G}_{\mathsf{clock}}$ functionality.
- if $\mathcal{R} = \mathcal{G}_{\mathrm{PL}}$ and P_i has not registered with $\mathcal{G}_{\mathsf{clock}}$ ignore the command, else register with the $\mathcal{F}_{\mathsf{N\text{-}MC}}, \mathcal{F}_{\mathsf{nizk}}, \mathcal{F}_{\mathsf{anon\text{-}selection}}$ functionalities

(The full specification is presented in Ouroboros Genesis [BGK+18])

P_i then calls Initialization-PrivProtocol returning $(\mathsf{PK}_i, \mathsf{SK}_i, \pi_{\mathsf{KeyGen}})$.

`// Transaction layer`

Submitting a transaction: Upon receiving $I = (\texttt{SUBMIT}, \mathsf{sid}, \mathsf{tx})$ from $\mathcal{Z}$,

- P_i calls $\mathsf{SubmitXfer}(\mathsf{tx}, \mathcal{C}^i_{\mathsf{loc}})$, where $\mathcal{C}^i_{\mathsf{loc}}$ is local chain maintained by P_i.

`// Consensus layer`

Maintaining the ledger: Upon receiving $I = (\texttt{MAINTAIN-LEDGER}, \mathsf{sid})$ from $\mathcal{Z}$,

- the party P_i invokes $\mathsf{LedgerMaintenance}(\mathcal{C}^i_{\mathsf{loc}}, P_i)$

Reading the state: Upon receiving $I = (\texttt{READ})$ from $\mathcal{Z}$,

- the party P_i invoke the protocol $\mathsf{ReadState}(\mathsf{sid}, \mathcal{C}^i_{\mathsf{loc}}, P_i)$.

Handling external (protocol-unrelated calls) to the clock: as in Ouroboros Genesis [BGK+18].

Figure 9: The protocol Π_{PriFHEte}

Protocol Initialization-PrivProtocol$(P_i, \mathsf{sid}) \to (\mathsf{PK}_i, \mathsf{SK}_i, \pi_{\mathsf{KeyGen}})$:

These steps are executed in a $(\texttt{MAINTAIN-LEDGER}, \mathsf{sid})$-interruptible manner:

1. Compute $(\mathsf{PK}_i, \mathsf{SK}_i, \pi_{\mathsf{KeyGen}}) \leftarrow \textsc{KeyGeneration}(\lambda)$
2. Use the clock to update τ, $ep \leftarrow \lceil \tau/R \rceil$ and $sl \leftarrow \tau$
3. If $\tau = 0$ then execute the following steps in a $(\texttt{MAINTAIN-LEDGER}, \mathsf{sid})$-interruptible manner:
   (a) Send $(\mathsf{claim}, \mathsf{sid}, P_i, \mathsf{PK}_i)$ to $\mathcal{F}_{\mathsf{init}}$.
   (b) Send $(\texttt{CLOCK-UPDATE}, \mathsf{sid}_C)$ to $\mathcal{G}_{\mathsf{clock}}$
   (c) Use clock to update τ, $ep \leftarrow \lceil \tau/R \rceil$ and $sl \leftarrow \tau$; give up the activation.
4. Else
   (a) Send $(\mathsf{gen\text{-}block}, \mathsf{sid}, P_i)$ to $\mathcal{F}_{\mathsf{init}}$. If $\mathcal{F}_{\mathsf{init}}$ signals an error then halt. Otherwise, receive from $\mathcal{F}_{\mathsf{init}}$ the response $(\mathsf{gen\text{-}block}, \mathsf{sid}, \mathbb{G} = (\mathbb{C}_1, \eta_1))$
   (b) Set $\mathcal{C}_{\mathsf{loc}} \leftarrow (\mathbb{G})$
   (c) Send $(\texttt{NEW-PARTY}, \mathsf{sid}, P_i, \mathsf{PK}_i, \pi_{\mathsf{KeyGen}})$ to $\mathcal{F}_{\mathsf{N\text{-}MC}}$
5. Return $(\mathsf{PK}_i, \mathsf{SK}_i, \pi_{\mathsf{KeyGen}})$

Figure 10: Protocol Initialization-PrivProtocol

Recall, from Section 4 that the MINT and TRANSFER invoked the $\mathcal{F}_{\mathsf{nizk}}$ ideal functionality. Apart from the $\mathcal{F}_{\mathsf{nizk}}$ functionality, our main protocol will make calls to other functionalities. We give an overview of these functionalities below:

1. $\mathcal{G}_{\mathsf{clock}}$: In both the real world and the ideal world, our protocols require a notion of time.

This is achieved using the $\mathcal{G}_{\mathsf{clock}}$ functionality (see Figure 24). The clock maintains a variable τ that denotes the current time. When all registered honest parties (at a given time τ) signal the functionality that they are done with the current round, the functionality advances the time counter τ. Parties can also query the functionality to read the current time.

2. $\mathcal{F}_{\mathsf{N\text{-}MC}}$: Parties in the real-world multicast transactions and blocks to their peers. $\mathcal{F}_{\mathsf{N\text{-}MC}}$ models a network functionality (see App. A2 of [BGK+19]) that captures a multicast network. We stress that this network functionality does not give any anonymity properties.
3. $\mathcal{F}_{\mathsf{anon\text{-}selection}}$: As described above, parties run a lottery to check if they are elected to propose blocks and if elected they broadcast the block with an anonymous proof. The anonymous selection functionality (Figure 26 and defined in [BMSZ20]) allows parties to check if they are eligible to win a lottery. The functionality also provides an interface for parties to receive a proof of winning the lottery and an interface to verify the proofs.
4. $\mathcal{F}_{\mathsf{nizk}}$: As discussed in Section 4 parties are required to attach a zero-knowledge proof that proves that the submitted transactions are well-formed. The parties therefore query a non-interactive zero-knowledge functionality (Figure 25 and defined in [KKKZ19]). This functionality allows generating proofs that a statement x is in a given NP language $\mathcal{L}$, with a witness w.

In Figure 9 we present the overall protocol that realizes the $\mathcal{G}_{\mathrm{PL}}$ ideal functionality. In our protocol, a block is proposed in a slot. Every η slots constitute an epoch ep. We now present an overview of the protocol.

Joining the system Upon receiving a ledger-registration request from the environment, the party registers with each of the functionalities. Once registered with the functionalities, the party is considered online. The party then becomes operational by invoking the Initialization-PrivProtocol protocol.

Upon execution of the Initialization-PrivProtocol protocol, the party P_i first generates keys by running the KeyGeneration(λ) algorithm (as presented in Section 4). The Initialization-PrivProtocol protocol works in two modes depending on the whether or not the current round is the genesis round. In the genesis mode, which is executed when $\tau = 0$, the party interacts with $\mathcal{F}_{\mathsf{init}}$ to register its keys. The $\mathcal{F}_{\mathsf{init}}$ functionality calls the Registration function here to add the party's entry to $\mathcal{T}_{\mathsf{privAccounts}}$. In the non-genesis mode, as in [BGK+18], the protocol Initialization-PrivProtocol queries $\mathcal{F}_{\mathsf{init}}$ to receive the genesis block. If the underlying protocol is a Proof-of-Stake protocol, the parties need to claim stake in the genesis mode, and in the non-genesis mode the $\mathcal{F}_{\mathsf{init}}$ functionality determines the lottery difficulty for the newly joined P_i. We refer to [BGK+18] for details. Finally, the party announces to the network that it is a new party by broadcasting $(\texttt{NEW-PARTY}, \mathsf{sid}, P_i, \mathsf{PK}_i)$. This interaction is presented in more details in Figure 10.

Submitting a transaction. A party P_S receives a SUBMIT command from the environment. Recall that the transaction could be either a TRANSFER transaction or a MINT transaction. If the transaction is of type TRANSFER, then parse the command as $\texttt{TRANSFER}\|(\mathsf{PK}_S, \mathsf{PK}_R, v)$ where PK_S is the public key associated with the account of the sender and PK_R is the public key associated with the account of the receiver and x is the value to be transferred.

The transaction is computed using the Transfer algorithm defined in Figure 5. The transaction is then broadcast to the network by submitting $(\texttt{MULTICAST}, \mathsf{tx})$ to the network functionality ($\mathcal{F}_{\mathsf{N\text{-}MC}}$). Similarly, if the command is of type MINT, then parse the command as $\texttt{MINT}\|(\mathsf{PK}_S, v)$. The real-world transaction is computed using MINT algorithm defined in Figure 4 and is broadcast using the $\mathcal{F}_{\text{N-MC}}$ functionality.

Protocol SubmitXfer$(\mathsf{tx}, \mathcal{C}_{\mathsf{loc}})$

1. Execute **FetchInformation** (as in Ouroboros Genesis (full version) [BGK+19]) to receive the newest messages of the round; denote the output by $(\mathcal{C}_1, \ldots, \mathcal{C}_M), (\mathsf{tx}_1, \ldots, \mathsf{tx}_k)$.
2. Set $\mathcal{N} \leftarrow \{\mathcal{C}_1, \ldots \mathcal{C}_M\}$
3. Invoke protocol $\mathsf{SelectChain}(\mathcal{N}, \mathcal{C}_{\mathsf{loc}}, \ldots)$ (as defined in Ouroboros Genesis [BGK+18]) and receive an updated chain $\mathcal{C}_{\mathsf{loc}}$.
4. If $\mathsf{tx} = (\texttt{TRANSFER}, \mathsf{tx}')$:
   (a) Let $(\mathsf{PK}_S, \mathsf{PK}_R, x) \leftarrow \mathsf{tx}'$
   (b) Use the clock to update $\tau, ep \leftarrow \lceil \tau / R \rceil$ and $sl \leftarrow \tau$
   (c) Let $\mathsf{tx}^* := \textsc{Transfer}(\mathsf{PK}_S, \mathsf{PK}_R, x, ep, R, \mathcal{C}_{\mathsf{loc}}, \mathbf{C}_i)$
   (d) Submit $(\texttt{MULTICAST}, \mathsf{tx}^*)$ to $\mathcal{F}_{\text{N-MC}}$
5. Else if $\mathsf{tx} = (\texttt{MINT}, \mathsf{tx}')$
   (a) Let $(\mathsf{PK}_S, x) \leftarrow \mathsf{tx}'$
   (b) Use the clock to update $\tau, ep \leftarrow \lceil \tau / R \rceil$ and $sl \leftarrow \tau$
   (c) Let $\mathsf{tx}^* := \textsc{Mint}(x, \mathsf{PK}_S, \mathsf{SK}_S, \mathsf{rt}_{\mathcal{T}_{\mathsf{pubAccounts}}})$
   (d) Submit $(\texttt{MULTICAST}, \mathsf{tx}^*)$ to $\mathcal{F}_{\text{N-MC}}$

Figure 11: Protocol SubmitXfer

Maintaining the ledger. Upon receiving a MAINTAIN-LEDGER command from the environment, a miner Q_j invokes the LedgerMaintenance algorithm. The algorithm invokes the **FetchInformation** command as defined in [BGK+19]. This algorithm fetches the recent messages in the round - this includes both the local chains broadcast by other parties $\mathcal{C}_1, \ldots, \mathcal{C}_M$ and the transactions broadcast by other parties $\mathsf{tx}_1, \ldots, \mathsf{tx}_k$. The miner then updates the buffer with these transactions and then selects the longest valid chain using the SelectChain algorithm defined in [BGK+18] to update its local chain $\mathcal{C}_j^{\mathsf{loc}}$. Q_j then invokes the LotteryProcedure algorithm to check if it is selected as a leader to propose the next block on the chain.

Protocol $\mathsf{LedgerMaintenance}(\mathcal{C}_{\mathsf{loc}}, Q_j)$

The following steps are executed in a $(\texttt{MAINTAIN-LEDGER}, \mathsf{sid})$-interruptible manner:

1. Execute **FetchInformation** (as in Ouroboros Genesis [BGK+19]) to receive the newest messages of the round; denote the output by $(\mathcal{C}_1, \ldots, \mathcal{C}_M), (\mathsf{tx}_1, \ldots, \mathsf{tx}_k)$.
2. Use the clock to update $\tau, ep \leftarrow \lceil \tau / R \rceil$ and $sl \leftarrow \tau$
3. Set $\mathsf{buffer} \leftarrow \mathsf{buffer} \| (\mathsf{tx}_1, \ldots, \mathsf{tx}_k), t_{\mathsf{on}} \leftarrow \tau, \mathcal{N} \leftarrow \{\mathcal{C}_1, \ldots \mathcal{C}_M\}$
4. Invoke protocol $\mathsf{SelectChain}(\ldots)$ (as defined in Ouroboros Genesis [BGK+18]) and receive an updated $\mathcal{C}^j_{\mathsf{loc}}$. Let $\mathcal{C}^*_{\mathsf{loc}}$ be the original local chain.
5. Let $\mathcal{U}$ be the set of transactions that are in $\mathcal{C}^j_{\mathsf{loc}}$ but not in $\mathcal{C}^*_{\mathsf{loc}}$.
6. Invoke protocol $\mathsf{LotteryProcedure}(Q_j, ep, sl, \mathsf{buffer}, \mathcal{C}^j_{\mathsf{loc}}, \mathcal{U})$ (in a $(\texttt{MAINTAIN-LEDGER}, \mathsf{sid})$-interruptible manner)
7. Send $(\texttt{CLOCK-UPDATE}, \mathsf{sid}_C)$ to $\mathcal{G}_{\mathsf{clock}}$.

Figure 12: Protocol LedgerMaintenance

The LotteryProcedure (see Figure 13) first sends the ELIGIBILITY-CHECK command to the $\mathcal{F}_{\mathsf{anon\text{-}selection}}$ functionality to check if the miner Q_j is eligible to propose the next block on the chain. If eligible, the miner first computes a local state based on the recently updated local chain $\mathcal{C}^j_{\mathsf{loc}}$. Now to create the next block on the chain, the miner iterates through the buffer and checks if each transaction is valid (using the ValidTx predicate, defined in Figure 22). If the transaction is valid, the miner updates the state with this transaction by running $\textsc{ProcessTransaction}(\mathsf{tx}_i, \mathcal{T}^j)$. The miner then adds this transaction to a block. The miner also adds the root of a Merkle tree computed over the updated state $\mathcal{T}^j$ and broadcasts the block using the $\mathcal{F}_{\mathsf{N\text{-}MC}}$ functionality.

To join the system, a new party must first register with the hybrid functionalities - $\mathcal{G}_{\mathsf{clock}}$, $\mathcal{F}_{\mathsf{init}}$, $\mathcal{F}_{\mathsf{nizk}}$, $\mathcal{F}_{\mathsf{N\text{-}MC}}$ and $\mathcal{F}_{\mathsf{anon\text{-}selection}}$. The party then runs the Initialization-PrivProtocol protocol, which internally runs the algorithm $\textsc{KeyGeneration}$ to return $(\mathsf{PK}_i, \mathsf{SK}_i, \pi_{\mathsf{KeyGen}})$. The party then broadcasts to the network the message $(\texttt{NEW-PARTY}, \mathsf{sid}, P_i, \mathsf{PK}_i, \pi_{\mathsf{KeyGen}})$. A miner upon receiving this message and tasked with maintaining the ledger updates the state by running the $\textsc{Registration}$ algorithm.

Protocol LotteryProcedure$(k, Q_j, ep, sl, \text{buffer}, \mathcal{C}^j_{\text{loc}}, \mathcal{U})$ The following steps are executed in a (MAINTAIN-LEDGER, sid)-interruptible manner:

1. Let $\mathcal{T}^j = (\mathcal{T}^j_{\text{pubAccounts}} \| \mathcal{T}^j_{\text{privAccounts}})$ be the state associated with $\mathcal{C}^j_{\text{loc}}$ maintained by Q_j
2. Send (ELIGIBILITY-CHECK, sid, (sl, ep)) to $\mathcal{F}_{\text{anon-selection}}$ and receive (ELIGIBILITY-CHECK, b). If $b = 0$, exit the protocol.
3. Else update $\mathcal{T}^j \leftarrow$ PROCESSTRANSACTION$(\text{tx}, \mathcal{T}^j)$ for each tx $\in \mathcal{U}$, initialize $\mathbf{N} = \emptyset$ and for each $\text{tx}_i \in$ buffer do (or until $\mathbf{N}$ can not increase any more):
   (a) if ValidTx$(\text{tx}_i, \mathcal{T}), \mathcal{C}^j_{\text{loc}}$ then $\mathbf{N} \leftarrow \mathbf{N} \| \text{tx}_i$
   (b) Remove tx_i from buffer
   (c) If $\text{tx}_i = (\text{NEW-PARTY}, \text{PK}_i)$, then run REGISTRATION$(\text{PK}_i, \mathcal{T}^j)$
   (d) Set $B' \leftarrow \text{blockify}(\mathbf{N})$ and update $\mathcal{T}^j \leftarrow$ PROCESSTRANSACTION$(\text{tx}_i, \mathcal{T}^j)$
4. Set $ptr \leftarrow H(\mathcal{C}_{\text{loc}})$
5. Compute $\text{rt}_{\text{privAccounts}} = \text{MerkleCRH}(\mathcal{T}_{\text{privAccounts}})$ and $\text{rt}_{\text{pubAccounts}} = \text{MerkleCRH}(\mathcal{T}_{\text{pubAccounts}})$ and set $\text{rt} = (\text{rt}_{\mathcal{T}_{\text{privAccounts}}} \| \text{rt}_{\mathcal{T}_{\text{pubAccounts}}})$.
6. Send (CREATE-PROOF, sid, $(ep, sl), \mathcal{T}$) to $\mathcal{F}_{\text{anon-selection}}$ and receive π. Set $\text{tx}_{\text{lead}} = ((ep, sl), ptr, \text{rt}, \pi)$
7. Set $B \leftarrow (\text{tx}_{\text{lead}}, B')$ and $\mathcal{C}_{\text{loc}} = \mathcal{C}_{\text{loc}} \| B$
8. Send (MULTICAST, (sid, tx_{lead})) to $\mathcal{F}^{\text{tx}}_{\text{N-MC}}$ and (MULTICAST, sid, $\mathcal{C}_{\text{loc}}$) to $\mathcal{F}^{\text{bc}}_{\text{N-MC}}$ and proceed from here upon next activation of this procedure.
9. While a (CLOCK-UPDATE, sid_C) has not been received during the current round: give up activation, and upon next activation of this procedure, proceed from here.

Figure 13: Proposing a new block if miner wins the lottery

6 Security Analysis

In this section we informally argue security of our scheme. We present the full security proofs in Appendix E.

Theorem 1. *The protocol Π_{PriFHEte} UC realizes the $\mathcal{G}_{\text{PL}}$ functionality in the $(\mathcal{G}_{\text{clock}}, \mathcal{F}_{\text{anon-selection}}, \mathcal{F}_{\text{init}}, \mathcal{F}_{\text{nizk}}, \mathcal{F}_{\text{N-MC}})$-hybrid world, assuming key-private CPA secure encryption, CPA secure fully homomorphic encryption, secure pseudorandom functions, perfectly binding commitment schemes and unforgeable signature scheme.*

Proof. (*Sketch*) To prove UC-security, we must show that there exists a PPT simulator interacting with $\mathcal{G}_{\text{PL}}$ that generates a transcript that is indistinguishable from the transcript generated by the real world protocol. We first give a high-level overview of the simulator (described in Fig 27, Fig 28 and Fig 29). Our simulator internally simulates the ideal functionalities $\mathcal{F}_{\text{init}}, \mathcal{F}_{\text{anon-selection}}, \mathcal{F}_{\text{nizk}}, \mathcal{F}_{\text{N-MC}}$ towards the adversary and relays any communication between the adversary and the emulated functionality. Since the general framework of the protocol and functionalities are the same as in Ouroboros Crypsinous [KKKZ19] and Genesis [BGK+18], we only focus on the simulation that concerns algorithms that we modify or add to. Upon receiving claim command from a party, on behalf of the simulated $\mathcal{F}_{\text{init}}$

functionality, the simulator first sends a REGISTER command on behalf of the party to the $\mathcal{G}_{\mathrm{PL}}$ functionality. Upon receiving Prove requests on behalf of the simulated $\mathcal{F}_{\mathsf{nizk}}$ functionality, the simulator records any witnesses provided by the adversary. Finally, to simulate $\mathcal{F}_{\mathsf{anon\text{-}selection}}$ the simulator executes the commands as the ideal functionality would. The simulation of $\mathcal{F}_{\mathsf{N\text{-}MC}}$ is indeed more interesting than the other functionalities since the simulator needs to create ideal-world transactions and blocks on behalf of the adversary using these transactions. The main idea to retrieve the private information associated with a transaction is to extract the witness that was recorded by the $\mathcal{F}_{\mathsf{nizk}}$ functionality for the corresponding transaction. Specifically, the simulator retrieves the witness w from the recorded witnesses in Π and extracts $\mathsf{PK}_S, \mathsf{PK}_R, v$ and submits an ideal world transaction to $\mathcal{G}_{\mathrm{PL}}$. Note that if such a witness does not exist, then the simulator aborts with ZKSoundnessFailure. Since we use the $\mathcal{F}_{\mathsf{nizk}}$ functionality, this event occurs with negligible probability. Moreover, if the transaction is of type MINT and the submitted transaction corresponds to that of an honest party, then the simulator aborts with sigFailure. Since we use unforgeable signatures the probability of this event occurring is negligible. The adversary may also send new blocks over the $\mathcal{F}_{\mathsf{N\text{-}MC}}$ functionality, the simulator first simulates the transactions in these blocks as described above in the case that ideal transactions for these transactions do not exist. Then the simulator runs EXTENDLEDGERSTATE function as defined in Ouroboros Genesis[BGK+18], which essentially creates new blocks and submits them to the $\mathcal{G}_{\mathrm{PL}}$ functionality. To simulate honest transactions, the simulator does the following: upon receiving a registration command, the simulator generates FHE, WKEnc and PRF keys as an honest party would. But instead of encrypting the WKEnc.sk the simulator encrypts all 0s. By the CPA security of the underlying FHE scheme, this is indistinguishable from the real world to an adversary. Similarly, the commitment to PRF key is replaced by a commitment to 0. Here we leverage the hiding property of the commitment scheme to argue that the two worlds are indistinguishable. To simulate honest transactions, the simulator generates a new PK, SK and computes a transfer transaction that sends from PK to PK a value of 0. By the key-privacy and CPA security of the underlying WKEnc scheme, the ideal and the real worlds are indistinguishable to a PPT adversary. The output of the PRF is replaced with a random string, and we leverage the pseudorandomness property of the PRF to argue indistinguishability. We argue in Appendix E through a sequence of hybrids that the real world and the ideal world are indistinguishable. □

References

[AJJM20] Prabhanjan Ananth, Abhishek Jain, Zhengzhong Jin, and Giulio Malavolta. Multi-key fully-homomorphic encryption in the plain model. In *Theory of Cryptography: 18th International Conference, TCC 2020, Durham, NC, USA, November 16–19, 2020, Proceedings, Part I 18*, pages 28–57. Springer, 2020.

[AR20] Shashank Agrawal and Srinivasan Raghuraman. Kvac: Key-value commitments for blockchains and beyond. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 839–869. Springer, 2020.

[BAZB20] Benedikt Bünz, Shashank Agrawal, Mahdi Zamani, and Dan Boneh. Zether: Towards privacy in a smart contract world. In *International Conference on Financial Cryptography and Data Security*, pages 423–443. Springer, 2020.

[BBDP01] Mihir Bellare, Alexandra Boldyreva, Anand Desai, and David Pointcheval. Key-privacy in public-key encryption. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 566–582. Springer, 2001.

[BGK+18] Christian Badertscher, Peter Gaži, Aggelos Kiayias, Alexander Russell, and Vassilis Zikas. Ouroboros genesis: Composable proof-of-stake blockchains with dynamic availability. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 913–930, 2018.

[BGK+19] Christian Badertscher, Peter Gazi, Aggelos Kiayias, Alexander Russell, and Vassilis Zikas. Ouroboros genesis: Composable proof-of-stake blockchains with dynamic availability. 2019.

[BGV14] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):1–36, 2014.

[BHR12] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 784–796, 2012.

[BMSZ20] Foteini Baldimtsi, Varun Madathil, Alessandra Scafuro, and Linfeng Zhou. Anonymous lottery in the proof-of-stake setting. In *2020 IEEE 33rd Computer Security Foundations Symposium (CSF)*, pages 318–333. IEEE, 2020.

[BMTZ17] Christian Badertscher, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. Bitcoin as a transaction ledger: A composable treatment. In *Annual international cryptology conference*, pages 324–356. Springer, 2017.

[BT23] Osman Biçer and Christian Tschudin. Oblivious homomorphic encryption. *Cryptology ePrint Archive*, 2023.

[BV14] Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) lwe. *SIAM Journal on computing*, 43(2):831–871, 2014.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*, pages 136–145. IEEE, 2001.

[CBC21] Panagiotis Chatzigiannis, Foteini Baldimtsi, and Konstantinos Chalkias. Sok: Blockchain light clients. *Cryptology ePrint Archive*, 2021.

[CHA22] Matteo Campanelli and Mathias Hall-Andersen. Veksel: simple, efficient, anonymous payments with large anonymity sets from well-studied assumptions. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, pages 652–666, 2022.

[DGKR18] Bernardo David, Peter Gaži, Aggelos Kiayias, and Alexander Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 66–98. Springer, 2018.

[Dia21] Benjamin E Diamond. Many-out-of-many proofs and applications to anonymous zether. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1800–1817. IEEE, 2021.

[eth] Ethereum.

[fil] Filecoin.

[FMMO19] Prastudy Fauzi, Sarah Meiklejohn, Rebekah Mercer, and Claudio Orlandi. Quisquis: A new design for anonymous cryptocurrencies. In *International conference on the theory and application of cryptology and information security*, pages 649–678. Springer, 2019.

[Fou21] Ethereum Foundation. Ethereum zk-rollups, 2021. Accessed on: 2023-02-12.

[Gen09] Craig Gentry. *A fully homomorphic encryption scheme*. Stanford university, 2009.

[GKO+22] Chaya Ganesh, Yashvanth Kondi, Claudio Orlandi, Mahak Pancholi, Akira Takahashi, and Daniel Tschudi. Witness-succinct universally-composable snarks. *Cryptology ePrint Archive*, 2022.

[GKP23] Yue Guo, Harish Karthikeyan, and Antigoni Polychroniadou. Pride ct: Towards public consensus, private transactions, and forward secrecy in decentralized payments. *Cryptology ePrint Archive*, 2023.

[Gro16] Jens Groth. On the size of pairing-based non-interactive arguments. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 305–326. Springer, 2016.

[GRR+21] Mike Graf, Daniel Rausch, Viktoria Ronge, Christoph Egger, Ralf Küsters, and Dominique Schröder. A security framework for distributed ledgers. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 1043–1064, 2021.

[GWC19] Ariel Gabizon, Zachary J Williamson, and Oana Ciobotaru. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. *Cryptology ePrint Archive*, 2019.

[GWY+20] Zhangshuang Guan, Zhiguo Wan, Yang Yang, Yan Zhou, and Butian Huang. Blockmaze: An efficient privacy-preserving account-model blockchain based on zk-snarks. *IEEE Transactions on Dependable and Secure Computing*, 2020.

[HBHW] Daira Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. Zcash protocol specification - zips.z.cash.

[KFTS17] Amrit Kumar, Clément Fischer, Shruti Tople, and Prateek Saxena. A traceability analysis of monero's blockchain. In *European Symposium on Research in Computer Security*, pages 153–173. Springer, 2017.

[KKKZ19] Thomas Kerber, Aggelos Kiayias, Markulf Kohlweiss, and Vassilis Zikas. Ouroboros crypsinous: Privacy-preserving proof-of-stake. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 157–174. IEEE, 2019.

[KL20] Jonathan Katz and Yehuda Lindell. *Introduction to modern cryptography*. CRC press, 2020.

[KZM+] Ahmed E Kosba, Zhichao Zhao, Andrew Miller, Yi Qian, and T-H Hubert Chan. How to use snarks in universally composable protocols.

[LATV12] Adriana López-Alt, Eran Tromer, and Vinod Vaikuntanathan. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*, pages 1219–1234, 2012.

[LRR+19] Russell WF Lai, Viktoria Ronge, Tim Ruffing, Dominique Schröder, Sri Aravinda Krishnan Thyagarajan, and Jiafan Wang. Omniring: Scaling private payments without trusted setup. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 31–48, 2019.

[LT22] Zeyu Liu and Eran Tromer. Oblivious message retrieval. 2022.

[MGGR13] Ian Miers, Christina Garman, Matthew Green, and Aviel D Rubin. Zerocoin: Anonymous distributed e-cash from bitcoin. In *2013 IEEE Symposium on Security and Privacy*, pages 397–411. IEEE, 2013.

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

[MSH+17] Malte Möser, Kyle Soska, Ethan Heilman, Kevin Lee, Henry Heffan, Shashvat Srivastava, Kyle Hogan, Jason Hennessey, Andrew Miller, Arvind Narayanan, et al. An empirical analysis of traceability in the monero blockchain. *arXiv preprint arXiv:1704.04299*, 2017.

[Net21] Anoma Network. Taiga. https://github.com/anoma/taiga, January 21 2021.

[NM+16] Shen Noether, Adam Mackenzie, et al. Ring confidential transactions. *Ledger*, 1:1–18, 2016.

[PVW08] Chris Peikert, Vinod Vaikuntanathan, and Brent Waters. A framework for efficient and composable oblivious transfer. In *Annual international cryptology conference*, pages 554–571. Springer, 2008.

[Reg10] Oded Regev. The learning with errors problem. *Invited survey in CCC*, 7(30):11, 2010.

[RH13] Fergal Reid and Martin Harrigan. An analysis of anonymity in the bitcoin system. In *Security and privacy in social networks*, pages 197–223. Springer, 2013.

[rip] Ripple.

[RZ19] Antoine Rondelet and Michal Zajac. Zeth: On integrating zerocash on ethereum. *arXiv preprint arXiv:1904.00905*, 2019.

[SCG+14] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE symposium on security and privacy*, pages 459–474. IEEE, 2014.

[TAB+20] Alin Tomescu, Ittai Abraham, Vitalik Buterin, Justin Drake, Dankrad Feist, and Dmitry Khovratovich. Aggregatable subvector commitments for stateless cryptocurrencies. In *Security and Cryptography for Networks: 12th International Conference, SCN 2020, Amalfi, Italy, September 14–16, 2020, Proceedings 12*, pages 45–64. Springer, 2020.

[Tec09] CAPE Technologies. Espresso systems documentation. https://docs.cape.tech/espresso-systems/, accessed 2023-03-09.

[Wil18] Zachary J Williamson. The aztec protocol. 2018.

[WMS+19] Karl Wüst, Sinisa Matetic, Moritz Schneider, Ian Miers, Kari Kostiainen, and Srdjan Čapkun. Zlite: Lightweight clients for shielded zcash transactions using trusted execution. In *International Conference on Financial Cryptography and Data Security*, pages 179–198. Springer, 2019.

[XZW+19] Yankai Xie, Chi Zhang, Lingbo Wei, Yukun Niu, Faxing Wang, and Jianqing Liu. A privacy-preserving ethereum lightweight client using pir. In *2019 IEEE/CIC International Conference on Communications in China (ICCC)*, pages 1006–1011. IEEE, 2019.

A Extended Preliminaries

A.1 Fully Homomorphic Encryption

We follow the definition of FHE presented in [BV14]. We use λ as the security parameter and all schemes in this paper encrypt bit-by-bit. A fully homomorphic encryption scheme $\mathsf{FHE} = (\mathsf{FHE.KeyGen}, \mathsf{FHE.Enc}, \mathsf{FHE.Eval})$ is a quadruple of PPT algorithms as follows.

- **Key Generation.** The algorithm $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{FHE.KeyGen}(1^\lambda)$ takes as input the security parameter and outputs a public encryption key pk, and a secret decryption key sk. Unlike [BV14] we treat the evaluation key evk as part of the public key pk.
- **Encryption.** The algorithm $c \leftarrow \mathsf{FHE.Enc}(\mathsf{pk}, m)$ takes the public key pk and a single bit message $m \in \{0, 1\}$ and a secret decryption key sk.
- **Decryption.** The algorithm $m \leftarrow \mathsf{FHE.Dec}(\mathsf{sk}, c)$ takes the secret key sk and a ciphertext c and outputs a message $m \in \{0, 1\}$
- **Homomorphic evaluation.** The algorithm $c_f \leftarrow \mathsf{FHE.Eval}(\mathsf{pk}, f, (c_1, \ldots, c_\ell))$ takes the public key pk, a function $f : \{0, 1\}^\ell \to \{0, 1\}$ and a set of ℓ ciphertexts $c_1, \ldots, c_\ell$ and outputs a ciphertext c_f.

The security notion we consider is IND-CPA security defined as follows.

Definition 1. *(CPA security). A scheme* FHE *is IND-CPA secure if for any polynomial time adversary* $\mathcal{A}$ *it holds that*

$$\mathrm{Adv}_{\mathrm{CPA}}[\mathcal{A}] = |Pr[\mathcal{A}(\mathsf{pk}, \mathsf{FHE.Enc}(\mathsf{pk}, 0)) = 1] - Pr[\mathcal{A}(\mathsf{pk}, \mathsf{FHE.Enc}(\mathsf{pk}, 1)) = 1]| = \mathsf{negl}(\lambda)$$

A.2 Key-Private Public Key encryption with wrong-key decryption

We denote this encryption scheme in the description of our protocols as $(\mathsf{WKEnc.KeyGen}, \mathsf{WKEnc.Enc}, \mathsf{WKEnc.Dec})$. Wrong-key decryption informally states that a ciphertext decrypted with the wrong secret key will always return a valid plaintext and return the correct plaintext only with some negligible advantage ($\frac{1}{2} + \mathsf{negl}$).
We present the notion of key-privacy under chosen plaintext attacks as defined in [BBDP01] and the wrong key decryption defined in [LT22]:

Definition 2. *(Key privacy) A scheme* WKEnc *is IK-CPA secure if for any polynomial time adversary* $\mathcal{A}$ *it holds that*

$$\mathrm{Adv}_{\text{IK-CPA}}[\mathcal{A}] = |Pr[\mathcal{A}(\mathsf{pk}_0, \mathsf{pk}_1, x, \mathsf{WKEnc.Enc}(\mathsf{pk}_0, x)) = 1]$$
$$-Pr[\mathcal{A}(\mathsf{pk}_0, \mathsf{pk}_1, \mathsf{WKEnc.Enc}(\mathsf{pk}_1, x)) = 1]| = \mathsf{negl}(\lambda)$$

Definition 3. *(Wrong-key Decryption) For an encryption scheme with plaintext space* $\mathbb{Z}_2$ *letting* $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{WKEnc.KeyGen}(1^\lambda)$ *and* $(\mathsf{sk}', \mathsf{pk}') \leftarrow \mathsf{WKEnc.KeyGen}(1^\lambda), \mathsf{ct} \leftarrow \mathsf{WKEnc.Enc}(\mathsf{pk}, 1)$, *and* $m' \leftarrow \mathsf{WKEnc.Dec}(\mathsf{sk}', \mathsf{ct})$, *it holds that*

$$Pr[m' = 1] \leq 1/2 + \mathsf{negl}(\lambda)$$

A.3 Pseudorandom functions with unpredictability under malicious key generation

Informally, unpredictability under malicious key generation (introduced in [DGKR18]) requires that the function PRF does not have any "bad" keys that an adversary can use to manipulate the output of the PRF.

In the random oracle model, the property can be expressed as follows: For any PPT adversary $\mathcal{A}$ and $x \in X, T \in \mathbb{N}$, the probability of the event $Pr[\mathsf{PRF}(k, x) = T | x \notin Q_H] = \frac{1}{2^\lambda}$, where the adversary outputs k and Q_H is the set of queries of $\mathcal{A}$ to the hash function H. The construction presented in Crypsinous[KKKZ19] is $H(m)^k$. By the DDH assumption, this is a secure PRF. Regarding unpredictability, observe that $Pr[H(x)^k = T] = Pr[H(x) = T^{1/k} = 1/2^\lambda]$ in the conditional space that $x \notin Q_H$.

A.4 Non-interactive zero knowledge

We use the $\mathcal{F}_{\mathsf{nizk}}$ functionality to compute and verify zero-knowledge proofs. We present the ideal functionality in Appendix D (Fig 25). The functionality provides an interface for parties to create proofs π that a statement x is in a given NP language $\mathcal{L}$ with a witness w. Moreover, as proven in [KKKZ19] the $\mathcal{F}_{\mathsf{nizk}}$ functionality can be realized by the SNARK proving system described in [KZM+].

A.5 Blockchain

A blockchain is an ever-growing hashchain of blocks. Each block consists of transactions and this hashchain is agreed upon by a dynamic set of nodes, often referred to as miners. Each user in the network may have a different version of the blockchain (denoted $\mathcal{C}^i_{\mathsf{loc}}$ for user P_i), constrained by the fact that each $\mathcal{C}^i_{\mathsf{loc}}$ has a common prefix.

Blockchains generally consist of two kinds of parties *miners* and *users*. The users compute and submit transactions to the network. The miners collect these transactions, validate them and create a block including the valid transactions. A miner then broadcasts the newly created block, thus extending the blockchain. The algorithms used to create and submit transactions are referred to as transaction layer algorithms and the ones used to create and broadcast blocks are referred to as consensus layer algorithms. To set some notation, each block is associated with a slot number sl_j, where a slot is unit of time. A set of adjacent η slots is called an epoch ep.

In account-based cryptocurrencies (the setting we consider in this work), a transaction consists of three values: the sender's identity, the receiver's identity and the value to be spent. The miners maintain a list of accounts where each element in the list is a (public key, balance) pair. This list is referred to as the *state* of the blockchain. To validate the transaction, the miner checks that the sender of the transaction is not trying to spend more than their balance. If the transaction is valid, the miner then updates the state by deducting the value of the transaction from the sender's balance and adds the same value to the receiver's balance. We denote the state of the cryptocurrency as $\mathcal{T}$. The miners compute a Merkle tree with the elements of $\mathcal{T}$ as the leaf. The root of this Merkle tree(denoted as $\mathsf{rt}_{\mathcal{T}}$) is also added to every block along with the valid transactions that caused the update to the state.

In a privacy-preserving cryptocurrency, we aim to hide the following information included in a payment: the sender's and receiver's identities and the value to be transferred. The universal composable (UC) framework [Can01] is a model used to define security properties of complex protocols in a modular way. A definition for an ideal ledger functionality was presented by Badertscher et. al. [BMTZ17] denoted $\mathcal{G}_{\text{LEDGER}}$. Kerber et. al. [KKKZ19] presented a private version of the ledger functionality denoted $\mathcal{G}_{\text{PL}}$ (PL stands for private ledger). The properties of hiding the information in a payment transaction as well as other security properties required of a blockchain is captured by the $\mathcal{G}_{\text{PL}}$ functionality. We give an overview of this functionality in Section 5 and present the complete functionality in Figure 16.

To ease the presentation, we will denote the state of the blockchain $\mathcal{T}$ as $\mathcal{T}_{\mathsf{privAccounts}} \| \mathcal{T}_{\mathsf{pubAccounts}}$ and $\mathsf{rt}_{\mathcal{T}} = H(\mathsf{rt}_{\mathcal{T}_{\mathsf{privAccounts}}} \| \mathsf{rt}_{\mathcal{T}_{\mathsf{pubAccounts}}})$. Row i in $\mathcal{T}_{\mathsf{pubAccounts}}$ is of the form $(\mathsf{PK}_i^{\mathsf{pub}}, v_i)$, where $\mathsf{PK}_i^{\mathsf{pub}}$ is the account-holder's public key that is associated with their non-anonymous balance. The account-holder uses the corresponding secret key $\mathsf{SK}_i^{\mathsf{pub}}$ to spend their public balance. Moreover $\mathcal{T}_{\mathsf{privAccounts}}$ includes elements of the form (PK_i, C_i) where C_i is the encrypted balance and PK_i is the public key associated with the account. As above, the account-holder uses the corresponding SK_i to spend an their private balance.

B Proof of correctness and Example

Proof of correctness To prove the correctness of PROCESSTRANSACTION we need to show that the state of the blockchain is updated correctly, i.e. when the entry in $\mathcal{T}_{\mathsf{privAccounts}}$ does not correspond to that of P_S (or P_R w.l.o.g.) the balance remains the same and when the entry does correspond to that of P_S, the balance is updated with the value in the transaction.

Case 1: $\mathsf{UpdateCiphertext}(\mathcal{T}_{\mathsf{privAccounts}}[\mathsf{PK}_i], (\mathsf{C}_S, \mathsf{C}_R, \mathbf{C}_{\mathbf{r_S}}, \mathbf{C}_{\mathbf{r_R}}, \mathbf{C}_{\mathbf{D}}, \mathbf{C}_{\mathbf{C}}, \mathsf{PRFOut}, \pi), \mathsf{PK}_i)$ is executed when C_S does not correspond to an commitment of $\mathsf{WKEnc.pk}_i$. Let $\mathsf{C}_S = Com(\mathsf{WKEnc.pk}_S, r_S)$ and $\mathbf{C}_{\mathbf{r_R}}[j] = \mathsf{WKEnc.Enc}(\mathsf{WKEnc.pk}_S, b_j)$ where $b_j = \mathsf{WKEnc.pk}_S[j]$ for $j \in [\lambda]$.

UpdateCiphertext works as follows:

1. Compute $\mathbf{C_i^r} = (C_{i,1}^r, \ldots, C_{i,\lambda}^r)$, where $C_{i,j}^r = \mathsf{FHE.Eval}(\mathsf{FHE.pk}_i, \mathsf{WKEnc.Dec}, (\mathsf{k\text{-}ct}_i, \mathbf{C}_{r_S}[j]))$.
2. Compute $\mathbf{C}_i^{com}$ as an FHE encryption of commitment to pk_i using the randomness encrypted in $\mathbf{C_i^r}$. By the wrong-key decryption (Def 3) and the binding property of the commitment scheme, $C_{i,j}^{com}$ encrypts a random bit $\in \{0,1\}$ and therefore $\mathbf{C_i^{com}}$ is an encryption of a random bit vector.
3. Compute $\mathbf{C_i^{preflag}} = (C_{i,1}^{preflag}, \ldots, C_{i,\lambda}^{preflag})$ where $C_{i,j}^{preflag} = \mathsf{FHE.Eval}(\mathsf{FHE.pk}_i, \oplus, (\overline{\mathsf{C}_S}[j], \mathbf{C_{i,j}^{com}}))$

for $j \in [\lambda]$. Since $C^{com}_{i,j}$ encrypts a random bit b, with high probability the bit encrypted in $C^{com}_{i,j} \neq \mathsf{C}_S[j]$ for all $j \in [\lambda]$. Therefore $\mathbf{C}^{\mathbf{preflag}}_{\mathbf{i}}$ is an encryption of a random bit vector, except with negligible probability.

4. Compute $C^{\mathsf{flag}}_i = \mathsf{FHE.Eval}(\mathsf{FHE.pk}_i, \times, (C^{preflag}_{i,1}, \ldots, C^{preflag}_{i,\lambda}))$. Since $\mathbf{C}^{\mathbf{preflag}}_{\mathbf{i}}$ is a random vector, with high probability there is atleast j s.t. $C^{preflag}_{i,j}$ encrypts 0. Therefore, C^{flag}_i is an encryption of 0 except with negligible probability.
5. Compute $\mathbf{C}^{\mathbf{x}}_{\mathbf{i}} = (C^x_{i,1}, \ldots, C^x_{i,\mu})$ where $C^x_{i,j} = \mathsf{FHE.Eval}(\mathsf{FHE.pk}_i, \mathsf{WKEnc.Dec}, (\mathsf{k\text{-}ct}_i, \mathbf{C}_D[j]))$. Since the $\mathsf{k\text{-}ct}_i$ encrypts $\mathsf{WKEnc.sk}_i \neq \mathsf{WKEnc.sk}_S$, by the wrong-key decryption (Def 3), $C^x_{i,j}$ encrypts a random bit $\in \{0,1\}$ and therefore $\mathbf{C}^{\mathbf{x}}_{\mathbf{i}}$ is an encryption of a random bit vector.
6. Compute $\mathbf{C}^{\mathbf{upd}}_{\mathbf{i}} = (C^{upd}_{i,1}, \ldots, C^{upd}_{i,\mu})$, where $C^{upd}_{i,j} = \mathsf{FHE.Eval}(\mathsf{FHE.pk}_i, \times, (\mathbf{C}^x_{i,j}, C_{\mathsf{flag}}))$. Since C_{flag} is an encryption of 0, $C^{upd}_{i,j}$ is an encryption of 0.
7. Update $\mathbf{C}_{\mathbf{i}} = (C_{i,1}, \ldots, C_{i,\mu}$ as $\mathsf{FHE.Eval}(\mathsf{FHE.pk}_i, -, (C_{i,j}, C^{upd}_{i,j}))$ for $j \in [\mu]$. Since $C^{upd}_{i,j}$ is an encryption of 0, the value encrypted in $C_{i,j}$ does not change.

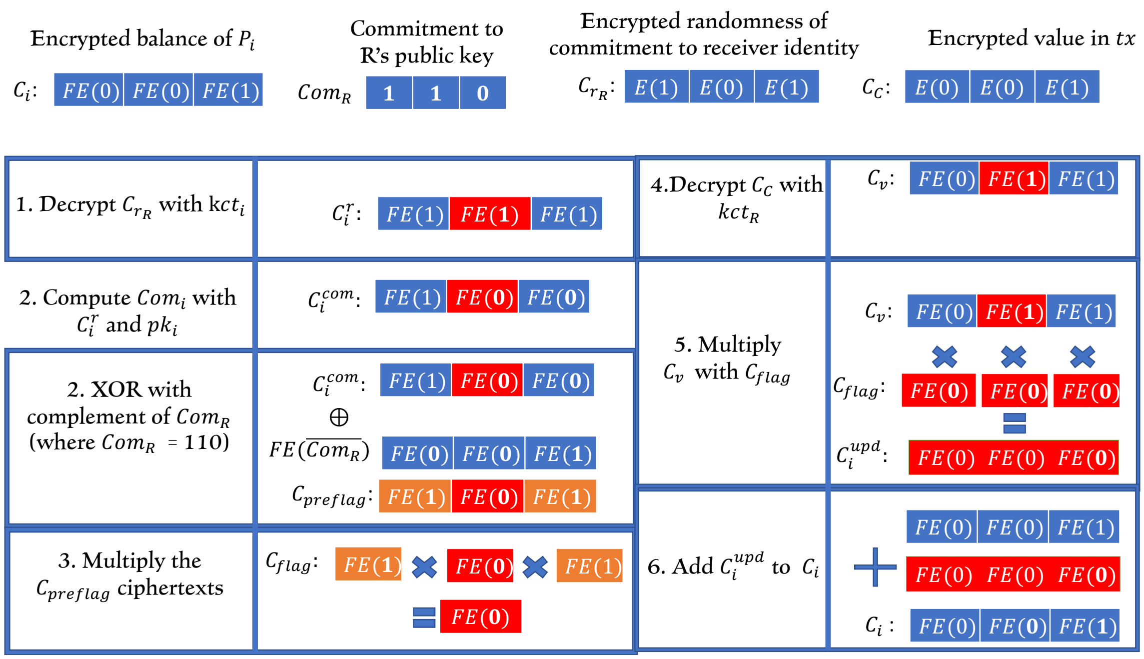

Figure 14: Case 2: When the receiver of the payment does not correspond to the entry updated in the state. Let the randomness used in committing to receiver's public key $r_R = 101$, the value of the transaction be $x = 001$ and the balance of the receiver be $v = 001$. (1) The first step is to decrypt $\mathsf{Enc}(101)$ with $\mathsf{k\text{-}ct}_i$, and since $i \neq R$, $\mathbf{C_i^r}$ encrypts a random bit string $\mathsf{FHE.Enc}(111)$. (2) Next C_i^{com} computes the encryption of the commitment of pk_i using the randomness encrypted in $\mathbf{C_i^r}$ (3) Next, $\mathbf{C_i^{com}}$ is homomorphically XORed with the complement of Com_R,which is 001,and this gives an encryption of 101. (3) These ciphertexts are then multiplied together to give a single encryption of 0 (4) Next we homomorphically decrypt the value $\mathbf{C_C}$ with $\mathsf{k\text{-}ct}_i$ to get an encryption of x under the FHE key, denoted $\mathbf{C_v}$, which is $\mathsf{FHE.Enc}(011)$ in our example. (5) Each of these ciphertexts are then multiplied with the $\mathbf{C_{flag}}$ ciphertext. Since the flag is 0, the value encrypted $\mathbf{C_v}$ now encrypts 0. (6) Finally these ciphertexts are added to the encryption of the balance in the state which does not change the value encrypted.

Case 2: $\mathsf{UpdateCiphertext}(\mathcal{T}_{\mathsf{privAccounts}}[\mathsf{PK}_i], (\mathbf{C_S}, \mathbf{C_R}, \mathbf{C}_D, \mathbf{C}_C, \mathsf{PRFOut}, \pi), \mathsf{PK}_i)$ is executed when $\mathbf{C_S}$ corresponds to an encryption of $\mathsf{WKEnc.pk}_i$. Let $\mathbf{C_S} = \mathsf{WKEnc.Enc}(\mathsf{WKEnc.pk}_S, b_j)$ where $b_j = \mathsf{WKEnc.pk}_S[j]$ for $j \in [\lambda]$. UpdateCiphertext works as follows:

1. Compute $\mathbf{C_i^r} = (C_{i,1}^r, \ldots, C_{i,\lambda}^r)$, where $C_{i,j}^r = \mathsf{FHE.Eval}(\mathsf{FHE.pk}_i, \mathsf{WKEnc.Dec}, (\mathsf{k\text{-}ct}_i, \mathbf{C}_{r_S}[j]))$.
2. Compute $\mathbf{C}_i^{com}$ as an FHE encryption of commitment to pk_i using the randomness encrypted in $\mathbf{C_i^r}$. Since the randomness and the pk_i are the same as in C_S, this ciphertext encrypts C_S
3. Compute $\mathbf{C_i^{preflag}} = (C_{i,1}^{preflag}, \ldots, C_{i,\lambda}^{preflag})$ where $C_{i,j}^{preflag} = \mathsf{FHE.Eval}(\mathsf{FHE.pk}_i, \oplus, (\overline{\mathsf{C}_S}[j], \mathbf{C_{i,j}^{com}}))$ for $j \in [\lambda]$. The $\mathbf{C_i^{preflag}}$ is an encryption of all 1s vector.
4. Compute $C_i^{\mathsf{flag}} = \mathsf{FHE.Eval}(\mathsf{FHE.pk}_i, \times, (C_{i,1}^{preflag}, \ldots, C_{i,\lambda}^{preflag}))$. Since $\mathbf{C_i^{preflag}}$ is a vector of all 1s, C_i^{flag} is an encryption of 1

5. Compute $\mathbf{C_i^x} = (C_{i,1}^x, \ldots, C_{i,\mu}^x)$ where $C_{i,j}^x = \mathsf{FHE.Eval}(\mathsf{FHE.pk}_i, \mathsf{WKEnc.Dec}, (\mathsf{k\text{-}ct}_i, \mathbf{C}_D[j]))$. Since the $\mathsf{k\text{-}ct}_i$ encrypts $\mathsf{WKEnc.sk}_i = \mathsf{WKEnc.sk}_S$, $C_{i,j}^x$ encrypts a bit $b_j \in \{0,1\}$ such that $\sum_{j=1}^{\mu} b_j \times 2^j = x$.
6. Compute $\mathbf{C_i^{upd}} = (C_{i,1}^{upd}, \ldots, C_{i,\mu}^{upd})$, where $C_{i,j}^{upd} = \mathsf{FHE.Eval}(\mathsf{FHE.pk}_i, \times, (\mathbf{C}_{i,j}^x, C_{\mathsf{flag}}))$. Since C_{flag} is an encryption of 1, $C_{i,j}^{upd}$ is an encryption of a bit $b_j \in \{0,1\}$ such that $\sum_{j=1}^{\mu} b_j \times 2^j = x$.
7. Update $\mathbf{C_i} = (C_{i,1}, \ldots, C_{i,\mu}$ as $\mathsf{FHE.Eval}(\mathsf{FHE.pk}_i, -, (C_{i,j}, C_{i,j}^{upd}))$ for $j \in [\mu]$. Since $C_{i,j}^{upd}$ is an encryption of $b_j \in \{0,1\}$ such that $\sum_{j=1}^{\mu} b_j \times 2^j = x$, the $C_{i,1}$ is updated with x added to the balance.

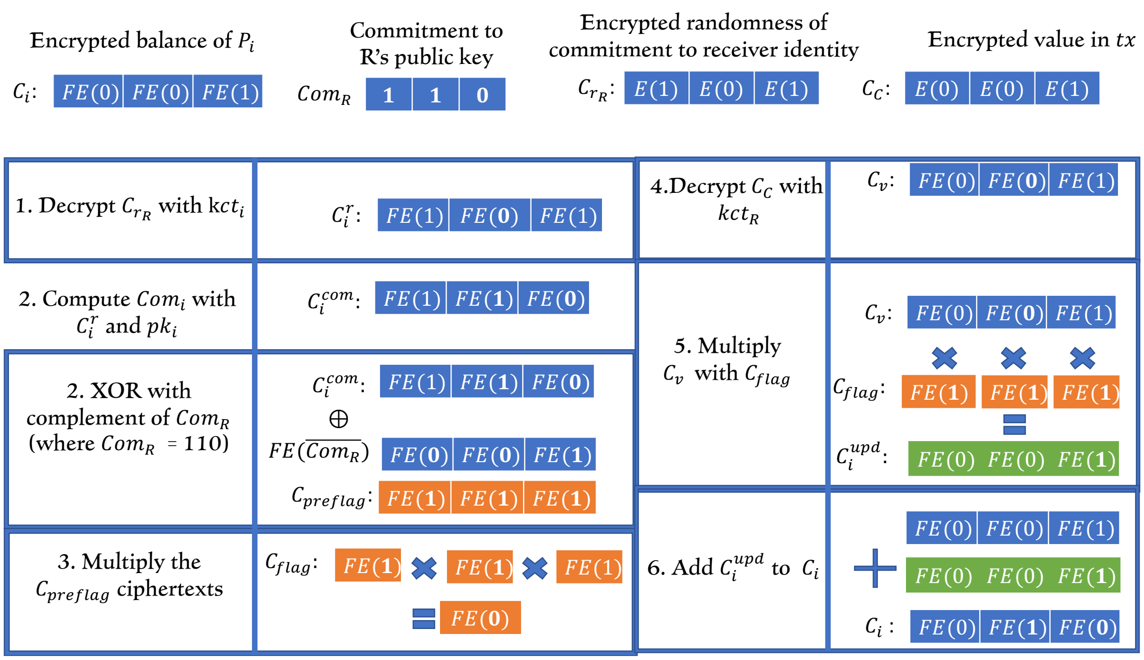

Figure 15: Case 1: When the receiver of the payment corresponds to the entry updated in the state. Let the randomness used in committing to receiver's public key $r_R = 101$, the commitment be the bit string 110 the value of the transaction be $x = 001$ and the balance of the receiver be $v = 001$. (1) The first step is to decrypt $\mathsf{Enc}(101)$ with $\mathsf{k\text{-}ct}_i$, and since $i = R$, $\mathbf{C_i^r} = \mathsf{FHE.Enc}(101)$. (2) Next $\mathbf{C_i^{com}}$ is computed using $\mathbf{C_i^r}$. Since the randomness and the public key are the same, the commitment that is encrypted $\mathbf{C_i^{com}}$ is the same as Com_R. (3) Next, $\mathbf{C_i^{com}}$ is homomorphically XORed with the complement of Com_R, which is 001,and this gives an encryption of 111. (4) These ciphertexts are then multiplied together to give a single encryption of 1, this encryption is called C_{flag}. (5) Next we homomorphically decrypt the value $\mathbf{C_C}$ with $\mathsf{k\text{-}ct}_i$ to get an encryption of x under the FHE key, denoted $\mathbf{C_v}$. (6) Each of these ciphertexts are then multiplied with the $\mathbf{C_{flag}}$ ciphertext. Since the flag is 1, the value encrypted $\mathbf{C_v}$ does not change. (7) Finally these ciphertexts are added to the encryption of the balance in the state

C The private ledger functionality - $\mathcal{G}_{\text{PL}}$

$\mathcal{G}_{\text{PL}}$ is parameterized by seven algorithms, Validate, ExtendPolicy, blockify, Lkg, BlindTx, Blind and predict-time, along with three parameters: $\texttt{windowSize}, \texttt{Delay} \in \mathbb{N}$ and $\mathcal{T}_1 = \{(P_1, v_1), \ldots, (P_n, v_n)\}$. These parameters are publicly known. The functionality manages variables state, NxtBC, buffer, τ_L and $\boldsymbol{ø}_{\text{state}}$. The variables are initialized as follows: state $:= \boldsymbol{ø}_{\text{state}} :=$ NxtBC $:=$ ids $:= \varepsilon$, buffer $:= \emptyset$, $\tau_L = 0$.
The functionality maintains the set of registered parties $\mathcal{P}$, the (sub-)set of honest parties $\mathcal{H} \subseteq \mathcal{P}$, and the subset of de-synchronized honest parties $\mathcal{P}_{DS} \subset \mathcal{H}$. The sets $\mathcal{P}, \mathcal{H}, \mathcal{P}_{DS}$ are all initially set to $\emptyset$. When a new honest party is registered at the ledger, if it is registered with the clock and the global RO already, then it is added to the party sets $\mathcal{H}$ and $\mathcal{P}$ and the current time of the registration is also recorded; if the current time is $\tau_L > 0$ it is also added to $\mathcal{P}_{DS}$. Similarly, when a party is de-registered, it is removed from $\mathcal{P}$. The ledger maintains the invariant that it is registered (as a functionality) to the clock whenever $\mathcal{H} \neq \emptyset$. Finally, during registration, the adversary is informed that a registration has occurred. The adversary responds with an ID and P_i is replaced with the resulting ID in $\mathcal{T}_1$. Further, the registration procedure returns ID. For each party $P_i \in \mathcal{P}$ the functionality maintains a pointer pt_p (initially set to 1) and a current-state view $\mathsf{state}_p := \varepsilon$ (initially set to empty). We refer to the vector $\mathsf{pt}_1, \ldots, \mathsf{pt}_n$ as pt.

Handling initial parties: If during the round $\tau = 0$, the ledger did not receive a registration from each initial party, $(P_i, v_i) \in \mathcal{T}_1$, the functionality halts.

Upon receiving any input I from any party or from the adversary, send $(\texttt{CLOCK-READ}, \mathsf{sid}_C)$ to $\mathcal{G}_{\mathsf{clock}}$; upon receiving response $(\texttt{CLOCK-READ}, \mathsf{sid}_C, \tau)$ set $\tau_L := \tau$ and do the following if $\tau > 0$ (otherwise, ignore input):

1. Let $\hat{\mathcal{P}} \subseteq \mathcal{P}_{DS}$ denote the set of de-synchronized honest parties that have been registered (continuously) since time $\tau' < \tau_L - \texttt{Delay}$. Set $\mathcal{P}_{DS} := \mathcal{P}_{DS} \setminus \hat{\mathcal{P}}$.
2. If I was received from an honest party $P_i \in \mathcal{P}$:
   (a) If $I = (\texttt{SUBMIT}, \mathsf{sid}, \mathsf{tx})$, set $\mathcal{I}_H^T := \mathcal{I}_H^T \| ((\texttt{SUBMIT}, \mathsf{sid}, \mathsf{BlindTx}_{\mathcal{A}}(\mathsf{state}, \mathcal{P} \setminus \mathcal{H}, \mathsf{ids}, \mathsf{tx})), P_i, \tau_L)$; else set $\mathcal{I}_H^T := \mathcal{I}_H^T \| (I, P_i, \tau_L)$
3. Compute $\mathbf{N} = (\mathbf{N_1}, \ldots, \mathbf{N}_\ell) := \mathsf{ExtendPolicy}(\mathcal{I}_H^T, \mathsf{state}, \mathsf{NxtBC}, \mathsf{buffer}, \boldsymbol{ø}_{\mathsf{state}})$ and if $\mathbf{N} \neq \varepsilon$ set state $:= \mathsf{state} \| \mathsf{blockify}(\mathbf{N}_1) \| \ldots \mathsf{blockify}(\mathbf{N}_\ell)$ and $\boldsymbol{ø}_{\mathsf{state}} := \boldsymbol{ø}_{\mathsf{state}} \| \tau_L^\ell$ where $\tau_L^\ell := \tau_L \| \ldots \| \tau_L$.
4. For each BTX $\in$ buffer: if $\mathsf{Validate}(\mathsf{BTX}, \mathsf{state}, \mathsf{buffer}, \mathsf{pt}, \mathcal{H}, \mathsf{ids}) = 0$ then delete tx from buffer. Also reset NxtBC $:= \varepsilon$.
5. If there exists $P_i \in \mathcal{H} \setminus \mathcal{P}_{DS}$ such that $|\mathsf{state}| - \mathsf{pt}_j > \texttt{windowSize}$ or $\mathsf{pt}_j < |\mathsf{state}_j|$, then set $\mathsf{pt}_k := |\mathsf{state}|$ for all $P_k \in \mathcal{H} \setminus \mathcal{P}_{DS}$.

Figure 16: The $\mathcal{G}_{\text{PL}}$ functionality - Part 1

3. If the calling party P_i is stalled (according to the definition above), then no further actions are taken. Otherwise, depending on the above input I and its sender's ID, $\mathcal{G}_{\text{PL}}$ executes the corresponding code from the following list:
   - *Submitting a transaction:*
     If $I = (\texttt{SUBMIT}, \mathsf{sid}, \mathsf{tx})$ and is received from a party $P_i \in \mathcal{P}$ or from $\mathcal{A}$ (on behalf of a corrupted party P_i) do the following
     (a) Choose a unique transaction ID txid and set $\mathsf{BTX} := (\mathsf{tx}, \mathsf{txid}, \tau_L, P_i)$
     (b) If $\mathsf{Validate}(\mathsf{BTX}, \mathsf{state}, \mathsf{buffer}, \mathsf{pt}, \mathcal{H}, \mathsf{ids}) = 1$, then $\mathsf{buffer} := \mathsf{buffer} \cup \mathsf{BTX}$
     (c) Send $(\texttt{SUBMIT}, \mathsf{BlindTx}_{\mathcal{A}}(\mathsf{state}, \mathcal{P} \setminus \mathcal{H}, \mathsf{ids}, \mathsf{BTX}))$ to $\mathcal{A}$
   - *Generating IDs*
   - *Reading the state*
     If $I = (\texttt{READ}, \mathsf{sid})$ is received from a party $P_i \in \mathcal{P}$, then set $\mathsf{state}_i := \mathsf{state}_i|_{\min\{\mathsf{pt}_i, |\mathsf{state}|\}}$ and return $(\texttt{READ}, \mathsf{sid}, \mathsf{Blind}(\{P_i\}, \mathsf{ids}, \mathsf{state}_i))$ to the requestor. If the requestor is $\mathcal{A}$ then send $(\mathsf{Blind}_{\mathcal{A}}(\mathcal{P} \setminus \mathcal{H}, \mathsf{ids}, \mathsf{state}), \mathsf{map}(\mathsf{BlindTx}_{\mathcal{A}}(\mathsf{state}, \mathcal{P} \setminus \mathcal{H}), \mathsf{ids}, \mathsf{buffer}), \mathsf{Lkg}(\mathsf{state}, \mathsf{buffer}, \tau_L), \mathcal{I}_H^T)$ to $\mathcal{A}$
   - *Maintiaining the ledger state*:
     If $I = (\texttt{MAINTAIN-LEDGER}, \mathsf{sid})$ is received by an honest party $P_i \in \mathcal{P}$ and (after updating $\mathcal{I}_H^T$ as above) $\mathsf{predict\text{-}time}(\mathcal{I}_H^T) = \hat{\tau} > \tau_L$ then send $\texttt{CLOCK-UPDATE}, \mathsf{sid}_C)$ to $\mathcal{G}_{\mathsf{clock}}$. Else send I to $\mathcal{A}$.
   - *The adversary proposing the next block*:
     If $I = (\texttt{NEXT-BLOCK}, \mathsf{hFlag}, (\mathsf{txid}_1, \ldots, \mathsf{txid}_\ell))$ is sent from the adversary update NxtBC as follows:
     (a) Set listOfTxid $\leftarrow \varepsilon$
     (b) For $i = 1, \ldots, \ell$ do: if there exists $\mathsf{BTX} := (x, \mathsf{txid}, \tau_L, P_i) \in \mathsf{buffer}$ with ID $\mathsf{txid} = \mathsf{txid}_i$ then set listOfTxid := listOfTxid$\|\mathsf{txid}_i$.
     (c) Finally set $\mathsf{NxtBC} := \mathsf{NxtBC}\|(\mathsf{hFlag}, \text{listOfTxid})$ and output $(\texttt{NEXT-BLOCK}, \texttt{ok})$ to $\mathcal{A}$
   - The adversary setting state-slackness:
     If $I = (\texttt{SET-SLACK}, (P_i, \hat{\mathsf{pt}}_i), \ldots, (P_\ell, \hat{\mathsf{pt}}_\ell))$ with $\{P_i, \ldots P_\ell\} \subseteq \mathcal{H} \setminus \mathcal{P}_{DS}$ is received from the adversary $\mathcal{A}$ do the following:
     (a) If for all $j \in [\ell] : |\mathsf{state}| - \hat{\mathsf{pt}}_j \leq \texttt{windowSize}$ and $\hat{\mathsf{pt}}_j \geq |\mathsf{state}_j|$, set $\mathsf{pt}_i = \hat{\mathsf{pt}}_i$ for every $j \in [i, \ell]$ and return $(\texttt{SET-SLACK}, \texttt{ok})$ to $\mathcal{A}$.
     (b) Otherwise set $\mathsf{pt}_j := |\mathsf{state}|$ for all $j \in [i, \ell]$.
   - The adversary setting the state for desynchronized parties:
     If $I = (\texttt{DESYNC-STATE}, (P_i, \mathsf{state}'_i), \ldots, (P_\ell, \mathsf{state}'_\ell))$ with $(P_i, \ldots, P_\ell) \subseteq \mathcal{P}_{DS}$ is received from the adversary $\mathcal{A}$, set $\mathsf{state}_j := \mathsf{state}'_j$ for each $j \in [i, \ell]$ and return $(\texttt{DESYNC-STATE}, \texttt{ok})$ to $\mathcal{A}$.

Figure 17: The $\mathcal{G}_{\text{PL}}$ functionality - Part 2

Function Validate(BTX, state, buffer, $\mathcal{H}$, ids):
1. Parse BTX as (tx, txid, τ_L, P_i)
2. Let ep^* be the epoch corresponding to τ_L, and the current epoch be ep. Check that $ep^* = ep$
3. Parse tx as (P_i, P_j, v) where $P_i, P_j \in$ ids
4. Check that $v_i > v$
5. Let τ_{ep} be the time when the current epoch starts.
6. Check that there exists no BTX$'$ $\in$ {state, buffer} after time τ_{ep} from party P_i.
7. If any of the above checks return false, return 0, else return 1.

Figure 18: Ideal Validation Predicate

Let tx = $(\mathsf{stx}_1, \ldots, \mathsf{stx}_\ell)$, where stx = (pk_r, x)
Function BlindSTx(state, $\mathcal{P}$, ids, (pk, stx))
1. Let $b \leftarrow 0$
2. If stx = (pk_r, x) and $\mathsf{pk}_r \in \mathcal{P} \setminus \mathcal{H}$, set $b \leftarrow 1$
3. If pk $\neq$ MINT $\vee$ pk not owned by $P_i \in \mathcal{P}$, set $b \leftarrow 0$
4. If b, return (pk, stx), else return $(\perp, |\mathsf{stx}|)$

Now,

$\mathsf{BlindTx}(\mathsf{state}, \mathcal{P}, \mathsf{ids}, (\mathsf{tx}, \mathsf{txid}, \cdot, \cdot)) = (\mathsf{map}(\mathsf{BlindSTx}(\mathsf{state}, \mathcal{P}, \mathsf{ids}), \mathsf{tx}), \mathsf{txid})$

$\mathsf{BlindTx}_{\mathcal{A}}(\mathsf{state}, \mathcal{P}, \mathsf{ids}, (\mathsf{tx}, \mathsf{txid}, \tau_L, P_s)) = (\mathsf{map}(\mathsf{BlindSTx}(\mathsf{state}, \mathcal{P}, \mathsf{ids}), \mathsf{tx}), \mathsf{txid}, \tau_L, P_s)$

Figure 19: Blinding function

This function is the same as the one defined in Ouroboros Genesis [BGK+18]. We present an overview below:
1. Create an honest client block as an alternative $\mathbf{N}_{\mathsf{df}}$.
2. Parse the block proposed by the adversary:
   - Check if upon adding a transaction from the block invalidates the rest of the transactions in the block.
   - If yes, return $\mathbf{N}_{\mathsf{df}}$.
   - If the proposed block is proposed on behalf of an honest party and there exist *old enough* valid transactions in the buffer of any other honest party set a flag oldValidTxMissing $\leftarrow$ true and return $\mathbf{N}_{\mathsf{df}}$
   - If there are too many adversarially generated blocks return $\mathbf{N}_{\mathsf{df}}$
   - If a sequence of blocks takes too much time to be proposed, return $\mathbf{N}_{\mathsf{df}}$
   - Else update the state with the newly proposed block.

Figure 20: Extend Policy function

C.1 Additional UC protocols

Protocol $\mathsf{ReadState}(\mathsf{sid}, \mathcal{C}_{\mathsf{loc}}, P_i)$

1. Execute **FetchInformation** to receive the newest messages for this round; denote the output chains by $\mathcal{C}_1, \ldots \mathcal{C}_M$
2. Use the clock to update $\tau, ep \leftarrow \lceil \tau/R \rceil$ and $sl \leftarrow \tau$
3. Let $\mathcal{N} \leftarrow \{\mathcal{C}_1, \ldots, \mathcal{C}_M\}$
4. Invoke protocol $\mathsf{SelectChain}(\mathcal{N}, \ldots)$ (as defined in Ouroboros Genesis [BGK+18]) and receive an updated $\mathcal{C}_{\mathsf{loc}}$
5. Extract the list of transactions st from the current local chain $\mathcal{C}_{\mathsf{loc}}$.
6. Set $\mathsf{st}_{ideal} = \emptyset$
7. For each tx in $\mathsf{st}^{\lceil k}$
   - If tx $=$ TRANSFER and $\mathsf{Dec}(\mathsf{WKEnc.sk}_i, \mathbf{C_R}) = \mathsf{WKEnc.pk}_i$ then decrypt $\mathbf{C_R} = v$ and record $(\texttt{TRANSFER}, v)$ as $\mathsf{st}_{ideal} = \mathsf{st}_{ideal} || (\texttt{TRANSFER}, v)$
   - If tx $=$ MINT and is equal to $(\mathsf{PK}_i, v, \mathsf{rt}, \sigma)$, record $(\texttt{MINT}, v)$ as $\mathsf{st}_{ideal} = \mathsf{st}_{ideal} || (\texttt{MINT}, v)$
8. Return st_{ideal}

Figure 21: Read State

Function $\mathsf{ValidTx}(\mathsf{tx}_i, \{\mathcal{T}\}_{ep}, \mathcal{C}_{\mathsf{loc}})$

If tx $=$ TRANSFER

1. Let ep be the current epoch and $\{\mathcal{T}\}_{ep}$ be the set of states in the current epoch.
2. Parse tx as $(\mathsf{C}_S, \mathsf{C}_R, \mathbf{C_{r_S}}, \mathbf{C_{r_R}}, \mathbf{C_D}, \mathbf{C_C}, \mathsf{PRFOut}, \pi)$
3. Verify that $\mathsf{rt}_\mathcal{T} = \mathsf{MerkleCRH}(\mathcal{T})$ for atleast one of $\mathcal{T}$ in $\{\mathcal{T}\}_{ep}$. Else abort.
4. Run $\mathsf{Verify}(\mathsf{zk.vk}, x, \mathsf{Proof})$ where $x = (\mathsf{rt}_\mathcal{T}, (\mathsf{C}_S, \mathsf{C}_R, \mathbf{C_{r_S}}, \mathbf{C_{r_R}}, \mathbf{C_D}, \mathbf{C_C}, \mathsf{PRFOut})$ and $\mathsf{Proof} = \pi$
5. Verify that PRFOut does not already appear in the buffer and $\mathcal{C}_{\mathsf{loc}}$ after slot $ep * R$.
6. If any of the checks above fail return 0, else return 1.

If tx $=$ MINT

1. Parse tx as $(\mathsf{PK}_i, x, \mathsf{rt}_{\mathcal{T}_{\mathsf{pubAccounts}}}, \sigma)$
2. Check that $\mathcal{T}_{\mathsf{pubAccounts}}[\mathsf{PK}_i] > x$
3. Check that $\mathsf{Verify}(\mathsf{vk}_i, (\mathsf{PK}_i, x, \mathsf{rt}_{\mathcal{T}_{\mathsf{pubAccounts}}}), \sigma) = 1$
4. If any of these checks fail, output 0, else output 1.

If tx $=$ PUB-TRANSFER:

1. Parse tx as $(\mathsf{PK}_S, \mathsf{vk}^*, x, \mathsf{PRFOut}, \pi_{pub})$
2. Run $\mathsf{Verify}(\mathsf{zk.vk}, x, \mathsf{Proof})$ where $x = (\mathsf{rt}_\mathcal{T}, \mathsf{PK}_S, \mathsf{vk}^*, x, \mathsf{PRFOut},)$ and $\mathsf{Proof} = \pi$
3. Verify that PRFOut does not already appear in the buffer and $\mathcal{C}_{\mathsf{loc}}$ after slot $ep * R$.
4. If any of the checks above fail return 0, else return 1.

Figure 22: Real world validation

D Hybrid functionalities

The functionality $\mathcal{F}_{\mathsf{init}}$ is parameterized by the number of initial account-holders n and their respective balances $b_1, \ldots, b_n$. $\mathcal{F}_{\mathsf{init}}$ interacts with $P_i, \ldots, P_n$ as follows:

- In the first round, upon a request from some account-holder P_i of the form $(\mathsf{claim}, \mathsf{sid}, P_i, \mathsf{PK}_i)$, the functionality computes $\textsc{Registration}(\mathsf{PK}_i)$.
- Once all parties have registered, it samples and stores a random value $\eta_1 \leftarrow_\$ \{0,1\}^\lambda$. it then constructs a genesis block $(\mathbb{C}, \eta_1)$, where $\mathbb{C} = (\mathbf{C_1}, \mathsf{PK}_n), \ldots, (\mathbf{C_n}, \mathsf{PK}_n)$

If this is not the first round then do the following:

- If any of the account-holders did not send a request of the above form in the genesis round, then $\mathcal{F}_{\mathsf{init}}$ outputs an error and halts.
- Otherwise, if the currently received input is a request of the form $(\mathsf{gen\text{-}req}, \mathsf{sid}, P_i)$ from any account-holder P_i, then $\mathcal{F}_{\mathsf{init}}$ sends $(\mathsf{gen\text{-}block}, \mathsf{sid}, (\mathbb{C}, \eta_1))$ to P_i.

Figure 23: $\mathcal{F}_{\mathsf{init}}$ functionality

The functionality maintains the set $\mathcal{P}$ of registered identities that is parties $P_i = (\mathsf{sid}, \mathsf{pid})$. It also manages the set F of functionalities. Initially $\mathcal{P} = \emptyset$ and $F = \emptyset$.
For each session sid the clock maintains a variable τ_{sid}. For each identity $P_i = (\mathsf{pid}, \mathsf{sid}) \in \mathcal{P}$ it manages variable d_{P_i}. For each pair $(\mathcal{F}, \mathsf{sid}) \in F$ it manages a variable $d_{(\mathcal{F},\mathsf{sid})}$ all initialized to 0.

Synchronization :

- Upon receiving $(\texttt{CLOCK-UPDATE}, \mathsf{sid}_C)$ from some party $P_i \in \mathcal{P}$ set $d_{P_i} := 1$; execute **Round-Update** and forward $(\texttt{CLOCK-UPDATE}, \mathsf{sid}, P_i)$ to $\mathcal{A}$
- Upon receiving $(\texttt{CLOCK-UPDATE}, \mathsf{sid}_C)$ from some functionality $\mathcal{F} \in \mathcal{P}$ set $d_{(\mathcal{F},\mathsf{sid})} := 1$; execute **Round-Update** and forward $(\texttt{CLOCK-UPDATE}, \mathsf{sid}, \mathcal{F})$ to this instance of $\mathcal{F}$.
- Upon receiving $(\texttt{CLOCK-READ}, \mathsf{sid}_C)$ from any participant return $(\texttt{CLOCK-READ}, \mathsf{sid}_C, \tau)$ to the requester.

Procedure Round-Update : For each session sid do: If $d_{(\mathcal{F},\mathsf{sid})} := 1$ for all $\mathcal{F} \in F$ and $d_{P_i} = 1$ for all honest parties $P_i = (\cdot, \mathsf{sid}) \in \mathcal{P}$ then set $\tau_{\mathsf{sid}} := \tau_{\mathsf{sid}} + 1$ and reset $d_{(\mathcal{F},\mathsf{sid})} := 0$ and $d_{P_i} := 0$ for all parties $P_i = (\cdot, \mathsf{sid}) \in \mathcal{P}$.

Figure 24: $\mathcal{G}_{\mathsf{clock}}$ functionality

The non-interactive zero-knowledge functionality $\mathcal{F}^{\mathcal{L}}_{\mathsf{nizk}}$ allows proving of statements in an NP language $\mathcal{L}$. It maintains a set of statement/proof pairs Π, initialized to $\emptyset$.

Proving Upon receiving a message $(\mathsf{Prove}, \mathsf{sid}, x, w)$:

1. If $(x, w) \notin \mathcal{L}$ then return $(\mathsf{proof}, \mathsf{sid}, x, \perp)$
2. Else send $(\mathsf{Prove}, \mathsf{sid}, x)$ to $\mathcal{A}$ and receive the reply $(\mathsf{proof}, \mathsf{sid}, x, \pi)$. Do $\Pi = \Pi \cup \{(x, \pi)\}$ and return $(\mathsf{proof}, \mathsf{sid}, x, \pi)$

Proof Verification When receiving a message $(\mathsf{verify}, \mathsf{sid}, x, \pi)$:

1. If $(x, \pi) \notin \Pi$ then send $(\mathsf{verify}, \mathsf{sid}, x, \pi)$ to $\mathcal{A}$ and then receive the reply R.
2. If $R = (\mathsf{witness}, \mathsf{sid}, x, \pi, w) \wedge (x, w) \in \mathcal{L}$ then let $\Pi = \Pi \cup (x, \pi)$.
3. Return $(\mathsf{verify}, \mathsf{sid}, x, \pi, (x, \pi) \in \Pi)$

Figure 25: $\mathcal{F}_{\mathsf{nizk}}$ functionality

The ideal functionality is parameterized by an Eligible predicate and maintains the following elements: (1) a global set of participants $\mathcal{P} = (P_1, b_1), \ldots (P_n, b_n)$ (2) A table T which has one row per party and column for each tag $\in [\mathbb{N}]$ given by parties when checking eligibility. The table stores the eligibility information of each party in each tag. (3) A list $\mathcal{L}$ to store a proof π corresponding to a message msg in some tag

Check Eligibility Upon receiving $(\texttt{ELIGIBILITY-CHECK}, \mathsf{sid}, \mathsf{tag})$ from a party P_i do the following:

1. If $P_i \in \mathcal{P}$ and $T(P_i, \mathsf{tag})$ is undefined, sample $r \in \{0,1\}^\ell$, run $\mathsf{Eligible}(P_i, r, \mathsf{tag})$ to get $b \in \{0,1\}$. Set $T(P_i, \mathsf{tag}) = b$
2. Output $(\texttt{ELIGIBILITY-CHECK}, \mathsf{sid}, T(P_i, \mathsf{tag}))$ to P_i.

Proof of eligibility Upon receiving $(\texttt{CREATE-PROOF}, \mathsf{sid}, \mathsf{tag}, msg)$ from some party P_i:

1. If $T(P_i, \mathsf{tag}) = 1$, send $(\texttt{PROVE}, \mathsf{tag}, msg)$ to $\mathcal{A}$. Else send $(\texttt{DECLINED}, \mathsf{tag}, msg)$ to P_i.
2. Upon receiving $(\texttt{DONE}, \psi, \mathsf{tag}, msg)$ from $\mathcal{A}$, set $\pi := \psi$ and record (π, tag, msg) in $\mathcal{L}$. Send $(\texttt{CREATE-PROOF}, \pi, \mathsf{tag}, msg)$ to P_i.

Verifying proofs Upon receiving $(\texttt{VERIFY}, \mathsf{sid}, \pi, \mathsf{tag}, msg)$ from some party P':

1. If $(\pi, \mathsf{tag}, msg) \in \mathcal{L}$ output $(\texttt{VERIFIED}, \mathsf{sid}, (\pi, \mathsf{tag}, msg), 1)$ to P'
2. If $(\pi, \mathsf{tag}, msg) \notin \mathcal{L}$ send $(\texttt{VERIFY}, \mathsf{sid}, (\pi, \mathsf{tag}, msg))$ to $\mathcal{A}$ and wait for a witness w from the adversary $\mathcal{A}$. Check if w is valid as follows:
   - Parse $w = (P_i, \mathsf{tag}, msg)$ and check that $T(P_i, \mathsf{tag}) = 1$
   - If yes, store (π, tag, msg) in the list $\mathcal{L}$ and send $(\texttt{VERIFIED}, \mathsf{sid}, (\pi, \mathsf{tag}, msg), 1)$ to P'

   If either of these checks are false, output $(\texttt{VERIFIED}, \mathsf{sid}, (\pi, \mathsf{tag}, msg), 0)$ to P'.

Figure 26: $\mathcal{F}_{\mathsf{anon\text{-}selection}}$ functionality of [BMSZ20]

E Security Proof

The simulator internally emulates the hybrid functionalities $\mathcal{F}_{\mathsf{init}}, \mathcal{F}_{\mathsf{nizk}}, \mathcal{F}_{\mathsf{anon\text{-}selection}}$ and relays any communication between $\mathcal{A}$ (on behalf of corrupted party) and the emulated functionality.

Simulation of $\mathcal{F}_{\mathsf{init}}$ towards $\mathcal{A}$

1. Upon receiving $(\mathsf{claim}, \mathsf{sid}, P_i, \mathsf{PK}_i)$ from $\mathcal{A}$, send $(\mathtt{REGISTER}, \mathsf{sid})$ on behalf of P_i to $\mathcal{G}_{\mathrm{PL}}$.
2. The functionality updates the state of the blockchain by running $\textsc{Registration}(\mathcal{T} \| \mathsf{PK}_i)$.

Simulation of $\mathcal{F}_{\mathsf{nizk}}$ towards $\mathcal{A}$

1. The simulator maintains a set of statement, witness and proof pairs for the relation $\mathcal{R}_{\mathtt{TRANSFER}}$ in $\Pi_{\mathtt{TRANSFER}}$ and for the relation $\mathcal{R}_{\mathtt{KEYGEN}}$ in $\Pi_{\mathtt{KEYGEN}}$.
2. Upon receiving a message $(\mathsf{Prove}, \mathsf{sid}, x, w)$ from some corrupted P_i, check if $(x, w) \in \mathcal{L}$. If not respond with $\perp$, else send $(\mathsf{Prove}, \mathsf{sid}, x)$ to $\mathcal{A}$ and receive back $(\mathsf{proof}, \mathsf{sid}, x, \pi)$. Record $(\pi, x, w) \in \Pi_{\mathtt{TRANSFER}}$ (or $\Pi_{\mathtt{KEYGEN}}$) and return $(\mathsf{proof}, \mathsf{sid}, x, \pi)$ to the corrupted party.
3. Upon receiving a message $(\mathsf{verify}, \mathsf{sid}, x, \pi)$ from a corrupt party, check if $(x, *, \pi) \in \Pi_{\mathtt{TRANSFER}}$ (or $\Pi_{\mathtt{KEYGEN}}$). If yes, return $(\mathsf{verify}, \mathsf{sid}, x, \pi, 1)$ to the corrupted party. If $(x, *, \pi) \notin \Pi_{\mathtt{TRANSFER}}$ or $\Pi_{\mathtt{KEYGEN}}$, send $(\mathsf{verify}, \mathsf{sid}, x, \pi)$ to $\mathcal{A}$ and receive back a reply R. If $R = (\mathsf{witness}, \mathsf{sid}, x, \pi, w)$ and $(x, w) \in \mathcal{L}$, then update $\Pi_{\mathtt{TRANSFER}} = \Pi_{\mathtt{TRANSFER}} \cup (x, w, \pi)$ or $\Pi_{\mathtt{KEYGEN}} = \Pi_{\mathtt{KEYGEN}} \cup (x, w, \pi)$ depending on the relation of the proof, and return $(\mathsf{verify}, \mathsf{sid}, x, \pi, 1)$, else respond with $(\mathsf{verify}, \mathsf{sid}, x, \pi, 0)$ to the corrupted party.

Simulation of $\mathcal{F}_{\mathsf{anon\text{-}selection}}$ towards $\mathcal{A}$

1. Upon receiving $(\mathtt{ELIGIBILITY\text{-}CHECK}, (sl, ep))$ from a corrupt party, sample a random $r \in \{0,1\}^{\ell}$ and run $\mathsf{Eligible}(P_i, r, (sl, ep))$ to get $b \in \{0,1\}$. Return $(\mathtt{ELIGIBILITY\text{-}CHECK}, (sl, ep), b)$ to the corrupt party. And store $T(P_i, (sl, ep)) = 1$
2. Upon receiving $(\mathtt{CREATE\text{-}PROOF}, \mathsf{sid}, (sl, ep), msg)$, from a corrupt party P_i, check that $T(P_i, (sl, ep)) = 1$ and if yes, forward the request to the adversary and receive Ψ. Record $(\Psi, (ep, sl), msg)$ and return $(\mathtt{CREATE\text{-}PROOF}, \pi, (ep, sl), msg)$ to the corrupt party.
3. Upon receiving $(\mathtt{VERIFY}, \mathsf{sid}, \pi, (ep, sl), msg)$ from a corrupt party P_i, check if $(\pi, (ep, sl), msg)$ has been recorded, if yes return $(\mathtt{VERIFIED}, \mathsf{sid}, (\pi, \mathsf{tag}, msg), 1)$ to the party. Else send $(\mathtt{VERIFY}, \mathsf{sid}, \pi, (ep, sl), msg)$ to the adversary and receive back a witness w. Parse $w = (P_i, (sl, ep), msg)$ and check if $T(P_i, (sl, ep)) = 1$, If yes, record $(\pi, (ep, sl), msg)$ and send $(\mathtt{VERIFIED}, \mathsf{sid}, (\pi, \mathsf{tag}, msg), 1)$ to P_i, else send $(\mathtt{VERIFIED}, \mathsf{sid}, (\pi, \mathsf{tag}, msg), 0)$ to P_i.

Figure 27: Simulation of hybrid functionalities towards the adversary

Simulation of $\mathcal{F}_{\text{N-MC}}$ The simulation is similar to that of Ouroboros Genesis [BGK+18]. We present below the additional changes to the simulation.

1. Upon receiving $(\texttt{MULTICAST}, (\mathsf{tx}_{i_1}, P_{i_1}), \ldots, (\mathsf{tx}_{i_\ell}, P_{i_\ell}))$ with list of transactions from $\mathcal{A}$ on behalf of some corrupted P_i do the following:
   <u>$\mathsf{SimulateAdvTransaction}(\mathsf{tx})$</u>
   If tx is a $\texttt{TRANSFER}$ transaction:
   (a) Parse tx as $(\mathsf{C}_S, \mathsf{C}_R, \mathbf{C}_{\mathbf{r_S}}, \mathbf{C}_{\mathbf{r_R}}, \mathbf{C}_D, \mathbf{C}_C, \mathsf{PRFOut}, \pi)$
   (b) Check that $((\mathsf{C}_S, \mathsf{C}_R, \mathbf{C}_{\mathbf{r_S}}, \mathbf{C}_{\mathbf{r_R}}, \mathbf{C}_D, \mathbf{C}_C, \mathsf{PRFOut}, \mathsf{rt}_{\mathcal{T}_{\mathsf{privAccounts}}}, ep), w)$ exists in $\Pi_{\texttt{TRANSFER}}$ as was recorded by the simulation of $\mathcal{F}_{\mathsf{nizk}}$ towards the adversary. If such an entry does not exist, abort with ZKSoundnessFailure
   (c) If there exists an honestly simulated transaction with PRFOut equal to the one parsed from tx, abort with error PRFFailure
   (d) Else parse w as $(\mathsf{PK}_S, \mathsf{SK}_S, \mathsf{PK}_R, v, v_S, \mathbf{C}, r_S, r_R\mathsf{path})$. If $(\mathsf{PK}_S, \mathbf{C}) \notin \mathcal{T}_{\mathsf{privAccounts}}$ but $\mathsf{VerifyPath}(\mathsf{rt}_{\mathcal{T}_{\mathsf{privAccounts}}}, \mathsf{path}) = 1$, abort with CRHFailure.
   (e) Let $\mathsf{SK}_S = (\mathsf{FHE.sk}_S, \mathsf{WKEnc.sk}_S, \mathsf{sk}_S, k_S)$ and from $\Pi_{\texttt{KEYGEN}}$, find the record $(\pi, \mathsf{PK}_S, w*)$ and let $w = (\mathsf{FHE.sk}_S, \mathsf{WKEnc.sk}_S, \mathsf{sk}_S, k_S^*)$. If $k_S^* \neq k_S$, abort with error CommFailure.
   (f) Set $\mathsf{tx} = (\texttt{TRANSFER}, (\mathsf{PK}_S, \mathsf{PK}_R, x))$ and send $(\texttt{SUBMIT}, \mathsf{sid}, \mathsf{tx})$ to $\mathcal{G}_{\mathrm{PL}}$ and receive back $(\texttt{SUBMIT}, (\mathsf{tx}, \mathsf{txid}, \tau_L, P_S))$ from $\mathcal{G}_{\mathrm{PL}}$. Record txid.

   If tx is a $\texttt{MINT}$ transaction:
   (a) Parse tx as (tx', σ) where $\mathsf{tx}' = (v, \mathsf{PK}_i, \mathsf{rt}_{\mathcal{T}_{\mathsf{pubAccounts}}})$
   (b) If σ corresponds to that of an honest party abort with sigFailure.
   (c) Else send $(\texttt{SUBMIT}, \mathsf{sid}, \mathsf{tx} = (\texttt{MINT}, (\mathsf{PK}_i, x)))$ to $\mathcal{G}_{\mathrm{PL}}$ and receive back $(\texttt{SUBMIT}, (\mathsf{tx}, \mathsf{txid}, \tau_L, P_S))$ from $\mathcal{G}_{\mathrm{PL}}$. Record txid.
2. Upon receiving $(\texttt{MULTICAST}, \mathsf{sid}, (\mathcal{C}_{i_1}, P_{i_1}), \ldots, (\mathcal{C}_{i_\ell}, P_{i_\ell}))$
   (a) Let $\mathcal{C}_l$ be the longest chain out of $\mathcal{C}_{i_1}, \ldots, \mathcal{C}_{i_\ell}$
   (b) Let $\mathsf{tx}_1, \ldots, \mathsf{tx}_n$ be transactions $\notin \mathcal{C}_l^{\lceil} k$.
   (c) For each $\mathsf{tx}_i \in \{\mathsf{tx}_1, \ldots, \mathsf{tx}_n\}$
      i. Find recorded txid that corresponds to tx_i.
      ii. If txid does not exist, run $\mathsf{SimulateAdvTransaction}(\mathsf{tx})$
   (d) Run EXTENDLEDGERSTATE(τ) as defined in [BGK+19]: which sends $(\texttt{NEXT-BLOCK}, \mathsf{hFlag}_j, \mathsf{list}_j)$ to $\mathcal{G}_{\mathrm{PL}}$ and receive $(\texttt{NEXT-BLOCK}, ok)$ as an immediate response. Here list_j is a list of txids that are not in the state but in $\mathcal{C}_l^{\lceil k}$ and hFlag_j denotes if the corresponding blocks were proposed by honest parties.
3. Upon receiving $(\texttt{NEW-PARTY}, \mathsf{sid}, P_i, \mathsf{PK}_i, \pi_{\texttt{KEYGEN}})$ from a corrupt party,
   (a) Check if (π, PK_i, w) exists in $\Pi_{\texttt{KEYGEN}}$ as was recorded by the simulation of $\mathcal{F}_{\mathsf{nizk}}$ towards the adversary. If such an entry does not exist, abort with ZKSoundnessFailure.
   (b) Else register with $\mathcal{G}_{\mathrm{PL}}$ on behalf of P_i and upon receiving a notification that a new party has registered, send PK_i.

Figure 28: Simulation of network functionality towards $\mathcal{A}$

Generating keys : Upon receiving registration request from the environment:
1. Generating keys:
   - $(\mathsf{FHE.pk}_i, \mathsf{FHE.sk}_i) \leftarrow \mathsf{FHE.KeyGen}(1^\lambda)$
   - $(\mathsf{WKEnc.pk}_i, \mathsf{WKEnc.sk}_i) \leftarrow \mathsf{FHE.KeyGen}(1^\lambda)$
   - $(\mathsf{sk}_i, \mathsf{vk}_i) \leftarrow \mathsf{Sign.KeyGen}(1^\lambda)$
   - $k \leftarrow \mathsf{PRF.KeyGen}(1^\lambda)$
2. Encrypting WKEnc keys:
   - $\mathsf{k\text{-}ct}_\mathbf{i} \leftarrow \{\mathsf{FHE.Enc}(\mathsf{FHE.pk}_i, 0)\}_{i=1}^{\lambda}$
3. Committing to the PRF key:
   - $\mathsf{C}_{\mathsf{PRF}} \leftarrow \mathsf{Com}(0; r)$ where $r \leftarrow \{0,1\}^\lambda$
4. Return $\mathsf{PK}_i := (\mathsf{k\text{-}ct}_i, \mathsf{FHE.pk}_i, \mathsf{WKEnc.pk}_i, \mathsf{vk}_i, \mathsf{C}_{\mathsf{PRF}})$ and $\mathsf{SK}_i = (\mathsf{FHE.sk}_i, \mathsf{WKEnc.sk}_i, \mathsf{sk}_i, k)$

Submitting honest transactions : Upon receiving $(\texttt{SUBMIT}, \mathsf{tx})$ from the environment for honest transactions:
1. If tx is of the form $(\texttt{TRANSFER} \| \mathsf{tx}')$
   (a) Let $(\mathsf{PK}^*, \mathsf{SK}^*) \leftarrow \textsc{KeyGeneration}(\lambda)$
   (b) Set $x = 0$
   (c) Use the clock to update $\tau, ep \leftarrow \lceil \tau / R \rceil$ and $sl \leftarrow \tau$
   (d) Let $\mathcal{C}_{\mathsf{loc}}$ be the chain upto the beginning of the epoch ep.
   (e) Let $(\mathsf{C}_S, \mathsf{C}_R, \mathbf{C_{r_S}}, \mathbf{C_{r_R}}, \mathbf{C_D}, \mathbf{C_C}, \pi, \mathsf{PRFOut}) := \textsc{Transfer}(\mathsf{PK}^*, \mathsf{PK}^*, x, ep, R, \mathcal{C}_{\mathsf{loc}})$
   (f) Sample $y \leftarrow \{0,1\}^\lambda$ and replace PRFOut with y.
   (g) Replace C_S and C_R with $\mathsf{Com}(0; r_S)$ and $\mathsf{Com}(0; r_R)$
   (h) Replace C_{r_S} and C_{r_R} with encryptions of 0.
   (i) Replace π with simulated proofs.
   (j) Submit $(\texttt{MULTICAST}, \mathsf{tx})$ to $\mathcal{F}_{\mathsf{N\text{-}MC}}$
2. Else if tx is of the form $(\texttt{MINT}, \mathsf{tx}')$
   (a) Parse tx′ as (pk_S, x)
   (b) Compute $\mathsf{tx} = \textsc{Mint}(v, \mathsf{PK}_i, \mathsf{SK}_i, \mathsf{rt}_{\mathcal{T}_{\mathsf{pubAccounts}}})$
   (c) Submit $(\texttt{MULTICAST}, \mathsf{tx})$ to simulated $\mathcal{F}_{\mathsf{N\text{-}MC}}$.

Simulating leader election : Upon receiving $(\texttt{MAINTAIN-LEDGER}, \mathsf{sid})$, extract from $\mathcal{I}_H^T$, the party P_i that issued this query. If P_i has already completed the round task then ignore the request. Otherwise:
1. Let (ep, sl, ptr, h, B') be as defined in LotteryProcedure executed by P_i.
2. Send $(\texttt{CREATE-PROOF}, (ep, sl), B)$ to $\mathcal{A}$ and receive back π.
3. Set $\mathsf{tx}_{\mathsf{lead}} = ((ep, sl), \mathsf{pt}, h, \pi)$ and broadcast $(\mathsf{tx}_{\mathsf{lead}}, B')$ to $\mathcal{F}_{\mathsf{N\text{-}MC}}^{\mathsf{bc}}$.

Figure 29: Simulating honest parties

Theorem 1.(restated)*The protocol Π_{PriFHEte} UC realizes the $\mathcal{G}_{\mathrm{PL}}$ functionality in the ($\mathcal{G}_{\mathsf{clock}}$, $\mathcal{F}_{\mathsf{anon\text{-}selection}}$, $\mathcal{F}_{\mathsf{init}}$, $\mathcal{F}_{\mathsf{nizk}}$, $\mathcal{F}_{\mathsf{N\text{-}MC}}$)-hybrid world, assuming key-private CPA secure encryption, CPA secure fully homomorphic encryption, secure pseudorandom functions, secure commitment schemes and unforgeable signature scheme.*

Proof. Proof by hybrids We prove security via a sequence of hybrids where we start from real world and move to the ideal world. The properties of the blockchain such as consistency, chain quality, liveness are handled by the ExtendPolicy algorithm. Since we do not modify this algorithm from the one defined in Ouroboros Genesis [BGK+18], these properties are achieved by our protocols as well. We therefore only consider the hybrids that correspond to the protocols on the transactional layer below:

- **Hybrid**$_0$: The real world protocol.
- **Hybrid**$_1$: This hybrid is the same as **Hybrid**$_0$, except upon receiving a `SUBMIT` command, the zero knowledge proofs π by simulated zero knowledge proofs in the TRANSFER algorithm. By the zero knowledge property of the underlying NIZK scheme we have that the two hybrids are indistinguishable.

  $\textsc{Transfer}(\mathsf{PK}_S, \mathsf{SK}_S, \mathsf{PK}_R, x, ep, R, \mathcal{C}_{\mathsf{loc}})$ User P_i does:

  1. ...
  9. Let $x = \{(\mathsf{C}_S, \mathsf{C}_R, \mathbf{C}_{\mathbf{r_S}}, \mathbf{C}_{\mathbf{r_R}}, \mathbf{C}_D, \mathbf{C}_C, \mathsf{PRFOut}, \mathsf{rt}_{\mathcal{T}}\}$. Send $(\mathsf{Prove}, \mathsf{sid}, x)$ to the $\mathcal{A}$ and receive π (just as in $\mathcal{F}_{\mathsf{nizk}}$ functionality).
  10. Return $\mathsf{tx} = (\mathsf{C}_S, \mathsf{C}_R, \mathbf{C}_{\mathbf{r_S}}, \mathbf{C}_{\mathbf{r_R}}, \mathbf{C}_D, \mathbf{C}_C, \mathsf{PRFOut}, \pi)$

  3
- **Hybrid**$_2$: This hybrid is the same as **Hybrid**$_1$, except that upon receiving a `SUBMIT` command and PK_R is honest, the ciphertexts $\mathbf{C}_{\mathbf{r_S}}, \mathbf{C}_{\mathbf{r_R}}, \mathbf{C}_{\mathbf{D}}, \mathbf{C}_{\mathbf{C}}$ are replaced by encryptions to 0. By the CPA security of the underlying encryption scheme, the two hybrids are indistinguishable.

  $\textsc{Transfer}(\mathsf{PK}_S, \mathsf{PK}_R, x, ep, R, \mathcal{C}_{\mathsf{loc}})$ User P_i does:

  1. ...
  4. Encrypt sender's randomness
     - For $i \in [\lambda]$, compute $C_{r_S,i} = \mathsf{WKEnc.Enc}(\mathsf{WKEnc.pk}_S, b_i)$, where $b_i = 0$
     - $\mathbf{C}_{\mathbf{r_S}} := (C_{r_S,1}, \ldots, C_{r_S,\lambda})$
  5. Encrypt receiver's randomness
     - For $i \in [\lambda]$, compute $C_{r_R,i} = \mathsf{WKEnc.Enc}(\mathsf{WKEnc.pk}_R, b_i)$, where $b_i = 0$
     - $\mathbf{C}_{r_R} := (C_{r_R,1}, \ldots, C_{r_R,\lambda})$
  6. Encrypt credited value
     - For $i \in [\lambda]$, compute $C_{D,i} = \mathsf{WKEnc.Enc}(\mathsf{WKEnc.pk}_R, b_i)$, where $b_i = 0$
     - $\mathbf{C}_D := (C_{D,1}, \ldots, C_{D,\lambda})$
  7. Encrypt debited value
     - For $i \in [\lambda]$, compute $C_{C,i} = \mathsf{WKEnc.Enc}(\mathsf{WKEnc.pk}_S, b_i)$, where $b_i = 0$
     - $\mathbf{C}_C := (C_{C,1}, \ldots, C_{C,\lambda})$
  8. Compute PRF output:
     - Compute $(\mathsf{PRFOut}) \leftarrow \mathsf{PRF}(k, ep)$
  9. Let $x = \{\mathsf{C}_S, \mathsf{C}_R, \mathbf{C}_{\mathbf{r_S}}, \mathbf{C}_{\mathbf{r_R}}, \mathbf{C}_{\mathbf{D}}, \mathbf{C}_{\mathbf{C}}, \mathsf{PRFOut}, \mathsf{rt}_{\mathcal{T}}\}$. Send $(\mathsf{Prove}, \mathsf{sid}, x)$ to the $\mathcal{A}$ and receive π (just as in $\mathcal{F}_{\mathsf{nizk}}$ functionality).
  10. Return $\mathsf{tx} = (\mathsf{C}_S, \mathsf{C}_R, \mathbf{C}_{\mathbf{r_S}}, \mathbf{C}_{\mathbf{r_R}}, \mathbf{C}_{\mathbf{D}}, \mathbf{C}_{\mathbf{C}}, \mathsf{PRFOut}, \pi)$

  We prove in Lemma 1 that the two hybrids are indistinguishable.
- **Hybrid**$_3$: This hybrid is the same as **Hybrid**$_2$, except that upon receiving a `SUBMIT` command, run $(\mathsf{WKEnc.pk}^*, \mathsf{WKEnc.sk}^*) \leftarrow \mathsf{WKEnc.KeyGen}(1^\lambda)$ and replace ciphertexts $\mathbf{C}_{\mathbf{r_S}}, \mathbf{C}_{\mathbf{r_R}}, \mathbf{C}_{\mathbf{D}}, \mathbf{C}_{\mathbf{C}}$

with encryptions under pk^*. By the key-privacy property of the underlying encryption scheme the two hybrids are indistinguishable.

$\underline{\textsc{Transfer}(\mathsf{PK}_S, \mathsf{PK}_R, x, ep, R, \mathcal{C}_{\mathsf{loc}})}$ User P_i does:

1. Run $(\mathsf{PK}^*, \mathsf{SK}^*) \leftarrow \textsc{KeyGeneration}(\lambda)$
2. Let $\mathsf{PK}^* = (\mathsf{k\text{-}ct}^*, \mathsf{FHE.pk}^*, \mathsf{WKEnc.pk}^*, \mathsf{vk}^*, \mathsf{C}^*_{\mathsf{PRF}})$
3. Set $\mathsf{WKEnc.pk}_S = \mathsf{WKEnc.pk}^*$ and $\mathsf{WKEnc.pk}_R = \mathsf{WKEnc.pk}^*$
4. ...

10. Return $\mathsf{tx} = (\mathsf{C}_S, \mathsf{C}_R, \mathbf{C_{r_S}}, \mathbf{C_{r_R}}, \mathbf{C_D}, \mathbf{C_C}, \mathsf{PRFOut}, \pi)$

We prove in Lemma 2 that the two hybrids are indistinguishable.

- **Hybrid**$_3$: This hybrid is the same as the previous hybrid except that the commitments to identities of the sender and the receiver are replaced by commitments to 0. By the computational hiding property of the underlying commitment scheme, these two hybrids are indistinguishable.
- **Hybrid**$_4$: This hybrid is the same as **Hybrid**$_3$, except that commitment to the PRF key k is replaced by a commitment to 0. By the commitment property of the underlying commitment scheme, the two hybrids are indistinguishable.

  $\underline{\textsc{KeyGeneration}(\lambda)}$: User P_i does:

  1. Key Generation: ...
     - $(\mathsf{FHE.pk}_i, \mathsf{FHE.sk}_i) \leftarrow \mathsf{FHE.KeyGen}(1^\lambda)$
     - $(\mathsf{WKEnc.pk}_i, \mathsf{WKEnc.sk}_i) \leftarrow \mathsf{FHE.KeyGen}(1^\lambda)$
     - $(\mathsf{sk}_i, \mathsf{vk}_i) \leftarrow \mathsf{Sign.KeyGen}(1^\lambda)$
     - $k \leftarrow \mathsf{PRF.KeyGen}(1^\lambda)$
  2. Encrypting WKEnc keys:
     - $\mathsf{k\text{-}ct}_i \leftarrow \mathsf{FHE.Enc}(\mathsf{FHE.pk}_i, \mathsf{WKEnc}, \mathsf{sk}_i[1]), \ldots, \mathsf{FHE.Enc}(\mathsf{FHE.pk}_i, \mathsf{WKEnc}, \mathsf{sk}_i[\lambda])$
  3. Commiting to the PRF key:
     - $\mathsf{C}_{\mathsf{PRF}} \leftarrow \mathsf{Com}(0; r)$ where $r \leftarrow \{0,1\}^\lambda$.
  4. Return $\mathsf{PK}_i := (\mathsf{k\text{-}ct}_i, \mathsf{FHE.pk}_i, \mathsf{WKEnc.pk}_i, \mathsf{vk}_i, \mathsf{C}_{\mathsf{PRF}})$ and $\mathsf{SK}_i = (\mathsf{FHE.sk}_i, \mathsf{WKEnc.sk}_i, \mathsf{sk}_i, k)$

  We prove in Lemma 3 that the two hybrids are indistinguishable.
- **Hybrid**$_5$: This hybrid is the same as **Hybrid**$_4$, except that the upon receiving a `SUBMIT` command, the PRFOut is replaced by a random value. By the psuedorandomness property of the underlying PRF scheme, the two hybrids are indistinguishable.

  $\underline{\text{Protocol } \mathsf{SubmitXfer}(\mathsf{tx}, \mathcal{C}_{\mathsf{loc}})}$

  1. If $\mathsf{tx} = (\texttt{TRANSFER}, \mathsf{tx}')$
     (a) Let $(\mathsf{PK}^*, \mathsf{SK}^*) \leftarrow \textsc{KeyGeneration}(\lambda)$
     (b) Set $x = 0$
     (c) Use the clock to update $\tau, ep \leftarrow \lceil \tau / R \rceil$ and $sl \leftarrow \tau$
     (d) Let $\mathcal{C}_{\mathsf{loc}}$ be the chain upto the beginning of the epoch ep.
     (e) Let $(\mathsf{C}_S, \mathsf{C}_R, \mathbf{C_{r_S}}, \mathbf{C_{r_R}}, \mathbf{C_D}, \mathbf{C_C}, \mathsf{PRFOut}, \pi) := \textsc{Transfer}(\mathsf{PK}^*, \mathsf{PK}^*, x, ep, R, \mathcal{C}_{\mathsf{loc}})$
     (f) Sample $y \leftarrow \{0,1\}^\lambda$ and replace PRFOut with y.
     (g) Submit $(\texttt{MULTICAST}, \mathsf{tx})$ to $\mathcal{F}_{\mathsf{N\text{-}MC}}$
  2. Else if $\mathsf{tx} = (\texttt{MINT}, \mathsf{tx}')$ ...

  We prove in Lemma 4 that the two hybrids are indistinguishable.
- **Hybrid**$_6$: This hybrid is the same as **Hybrid**$_5$ except that the upon receiving a registration request, replace $\mathsf{k\text{-}ct}_i$ with $\mathsf{FHE.Enc}(\mathsf{FHE.pk}_i, 0)$ instead of encrypting $\mathsf{WKEnc.sk}_i$. By the

CPA security of the underlying FHE scheme, the two hybrids are indistinguishable.
KEYGENERATION(λ): User P_i does:
1. Key Generation: ...
2. Encrypting WKEnc keys:
   – $\text{k-ct}_\mathbf{i} \leftarrow \text{FHE.Enc}(\text{FHE.pk}_i, 0), \ldots, \text{FHE.Enc}(\text{FHE.pk}_i, 0)$
3. Committing to PRF key: ...
4. Return $\text{PK}_i := (\text{k-ct}_i, \text{FHE.pk}_i, \text{WKEnc.pk}_i, \text{vk}_i, \text{C}_{\text{PRF}})$ and $\text{SK}_i = (\text{FHE.sk}_i, \text{WKEnc.sk}_i, \text{sk}_i, k)$

We prove in Lemma 5 that the two hybrids are indistinguishable.

- **Hybrid**$_7$: This hybrid is the same as **Hybrid**$_6$ except that the simulator may now abort with sigFailure. Since we use unforgeable signatures the simulator aborts with negligible probability and therefore the two hybrids are indistinguishable.
  If tx is a `MINT` transaction:
  1. Parse tx as (tx', σ) where $\text{tx}' = (v, \text{PK}_i, \text{rt}_{\mathcal{T}_{\text{pubAccounts}}})$
  2. If σ corresponds to that of an honest party abort with sigFailure.
  3. ...

  We prove in Lemma 6 that the two hybrids are indistinguishable.
- **Hybrid**$_8$: This hybrid is the same as **Hybrid**$_7$ except that the simulator may now abort with ZKSoundnessFailure. By the soundness property of the underlying zero knowledge scheme, this occurs with negligible probability.
  If tx is a `TRANSFER` transaction:
  1. Parse $\text{tx} = (\text{C}_S, \text{C}_R, \mathbf{C}_{\mathbf{r_S}}, \mathbf{C}_{\mathbf{r_R}}, \mathbf{C_D}, \mathbf{C_C}, \text{PRFOut}, \pi)$
  2. Check that $(\pi, (\mathbf{C_S}, \mathbf{C_R}, \mathbf{C_C}, \mathbf{C_D}, \text{C}_{\text{PRF}}, \text{rt}_{\mathcal{T}_{\text{privAccounts}}}, ep), w)$ exists in Π as was recorded by the simulation of $\mathcal{F}_{\text{nizk}}$ towards the adversary. If such an entry does not exist, abort with ZKSoundnessFailure
  3. ...

  This event occurs with negligible probability since we use the $\mathcal{F}_{\text{nizk}}$ ideal functionality to compute zero knowledge proofs.
- **Hybrid**$_9$: This hybrid is the same as **Hybrid**$_8$ except that the simulator may now abort with CRHFailure. Since we use collision-resistant hash functions, this event occurs with negligible probability.
  If tx is a `TRANSFER` transaction:
  1. Parse $\text{tx} = (\text{C}_S, \text{C}_R, \mathbf{C}_{\mathbf{r_S}}, \mathbf{C}_{\mathbf{r_R}}, \mathbf{C_D}, \mathbf{C_C}, \text{PRFOut}, \pi)$
  2. ...
  3. Else parse w as $(\text{PK}_S, \text{SK}_S, \text{PK}_R, v, v_S, \mathbf{C}, \text{path})$. If $(\text{PK}_S, \mathbf{C}) \notin \mathcal{T}_{\text{privAccounts}}$ but VerifyPath $(\text{rt}_{\mathcal{T}_{\text{privAccounts}}}, \text{path}) = 1$, abort with CRHFailure.

  We prove in Lemma 7 that the two hybrids are indistinguishable.
- **Hybrid**$_{10}$: This hybrid is the same as **Hybrid**$_9$, except that the simulator may now abort with CommFailure. Since we use statistically-binding commitments, this event occurs with negligible probability.
  If tx is a `TRANSFER` transaction:
  1. Parse tx as $\text{tx} = (\text{C}_S, \text{C}_R, \mathbf{C}_{\mathbf{r_S}}, \mathbf{C}_{\mathbf{r_R}}, \mathbf{C_D}, \mathbf{C_C}, \text{PRFOut}, \pi)$
  2. ...
  3. Let $\text{SK}_S = (\text{FHE.sk}_S, \text{WKEnc.sk}_S, \text{sk}_S, k_S)$ and from $\Pi_{\texttt{KEYGEN}}$, find the record $(\pi, \text{PK}_S, w*)$ and let $w = (\text{FHE.sk}_S, \text{WKEnc.sk}_S, \text{sk}_S, k_S^*)$. If $k_S^* \neq k_S$, abort with error CommFailure.

  We prove in Lemma 8 that the two hybrids are indistinguishable.

- $\mathbf{Hybrid}_{11}$: This hybrid is the same as $\mathbf{Hybrid}_{10}$ except that the simulator may now abort with $\mathsf{PRFFailure}$. Since we use PRF with the property of unpredictability malicious key generation, this occurs with negligible probability.
- $\mathbf{Hybrid}_{12}$: This hybrid is the same as $\mathbf{Hybrid}_{11}$ except that the commitments to pk_S and pk_R are replaced by commitments to 0. By the computational hiding property of the commitment scheme these two hybrids are indistinguishable.

Finally this hybrid is the same as the ideal world, and therefore the real world and the ideal world are indistinguishable. □

Lemma 1. *By the CPA security over multiple encryptions[KL20] of the underlying encryption scheme* WKEnc, $\mathbf{Hybrid}_1$ *and* $\mathbf{Hybrid}_2$ *are indistinguishable to a PPT adversary.*

Proof. The difference between $\mathbf{Hybrid}_1$ and $\mathbf{Hybrid}_2$ is that the simulator replaces the encryptions $\mathbf{C_{r_S}}, \mathbf{C_{r_R}}, \mathbf{C_D}, \mathbf{C_C}$ with encryptions of 0.

Assume a distinguisher D can distinguish between $\mathbf{Hybrid}_1$ and $\mathbf{Hybrid}_2$, i,e. $Pr[D(\mathbf{Hybrid}_1) = 1] - Pr[D(\mathbf{Hybrid}_2) = 1] > \mathsf{negl}$

Using this distinguisher D we construct a reduction B that can break the CPA security of encryption scheme.

Reduction B:

1. Activate the distinguisher D
2. The reduction simulates the protocol Π_{PriFHEte} as in $\mathbf{Hybrid}_1$.
3. Send $\mathbf{m_0} = (\mathsf{PK}_1, \mathsf{PK}_2, x, x)$ and $\mathbf{m_1} = (0, 0, 0, 0)$ to the challenger and receive $\mathbf{C_1}, \mathbf{C_2}, \mathbf{C_3}, \mathbf{C_4}$
4. Instruct the environment to submit a transaction $(\mathsf{PK}_1, \mathsf{PK}_2, x)$, and replace the ciphertexts in the transfer transaction with $\mathbf{C_1}, \mathbf{C_2}, \mathbf{C_3}, \mathbf{C_4}$.
5. Submit tx to $\mathcal{F}_{\mathsf{N\text{-}MC}}$.
6. Output whatever D outputs.

Note that in the case $\mathbf{C_{r_S}}, \mathbf{C_{r_R}}, \mathbf{C_D}, \mathbf{C_C}$ was the encryption of m_0 the distinguisher sees the hybrid world - $\mathbf{Hybrid}_1$ and on the other hand when encryption of m_1 is returned the distinguisher sees the hybrid world $\mathbf{Hybrid}_2$.

Now since $Pr[D(\mathbf{Hybrid}_1) = 1] - Pr[D(\mathbf{Hybrid}_2) = 1] > \mathsf{negl}$, we have that $\mathrm{Adv}_{\mathrm{CPA}} > \mathsf{negl}$ which is a contradiction since we assume CPA secure encryption over multiple encryptions. This implies $Pr[D(\mathbf{Hybrid}_1) = 1] - Pr[D(\mathbf{Hybrid}_2) = 1] = \mathsf{negl}$.

□

Lemma 2. *By the key-privacy property (Def 2) of the underlying encryption scheme, the hybrids* $\mathbf{Hybrid}_2$ *and* $\mathbf{Hybrid}_3$ *are indistinguishable.*

Proof. The difference between $\mathbf{Hybrid}_2$ and $\mathbf{Hybrid}_3$ is that the simulator replaces the encryptions $\mathbf{C_{r_S}}, \mathbf{C_{r_R}}, \mathbf{C_D}, \mathbf{C_C}$ with encryptions under a freshly generated key $\mathsf{WKEnc.pk}^*$ where $(\mathsf{WKEnc.pk}^*, \mathsf{WKEnc.sk}^*) = \mathsf{WKEnc.KeyGen}(\lambda)$.

Assume a distinguisher D can distinguish between $\mathbf{Hybrid}_2$ and $\mathbf{Hybrid}_3$, i,e. $Pr[D(\mathbf{Hybrid}_2) = 1] - Pr[D(\mathbf{Hybrid}_3) = 1] > \mathsf{negl}$

Using this distinguisher D we construct a reduction B that can break the IK-CPA security of encryption scheme.

Reduction B:

1. Activate the distinguisher D
2. Receive two public keys $\mathsf{pk}_0, \mathsf{pk}_1$ from the challenger.

3. The reduction simulates the protocol Π_{PriFHEte} as in $\mathbf{Hybrid}_2$, such that $\mathsf{WKEnc.pk}_i$ of a party P_i is replaced with pk_0
4. Send $\mathsf{WKEnc.pk}_R, \mathsf{WKEnc.pk}_S, v, v$ to the challenger and receive $\mathbf{C_1}, \mathbf{C_2}, \mathbf{C_3}, \mathbf{C_4}$.
5. Instruct the environment to submit a transaction $(\mathsf{PK}_1, \mathsf{PK}_2, x)$, and replace the ciphertexts in the transfer transaction with $\mathbf{C_1}, \mathbf{C_2}, \mathbf{C_3}, \mathbf{C_4}$.
6. Submit tx to $\mathcal{F}_{\mathsf{N\text{-}MC}}$.
7. Output whatever D outputs.

Note that in the case $\mathbf{C_S}, \mathbf{C_R}, \mathbf{C_D}, \mathbf{C_C}$ was encrypted under pk_0 the distinguisher sees the hybrid world - $\mathbf{Hybrid}_2$ and on the other hand when encryptions are under pk_0 the distinguisher sees the hybrid world $\mathbf{Hybrid}_3$.

Now since $Pr[D(\mathbf{Hybrid}_2) = 1] - Pr[D(\mathbf{Hybrid}_3) = 1] > \mathsf{negl}$, we have that $\mathrm{Adv}_{\text{IK-CPA}} > \mathsf{negl}$ which is a contradiction since we assume IK-CPA secure encryption over multiple encryptions. This implies $Pr[D(\mathbf{Hybrid}_2) = 1] - Pr[D(\mathbf{Hybrid}_3) = 1] = \mathsf{negl}$.

□

Lemma 3. *By the hiding property of the underlying commitment scheme,* $\mathbf{Hybrid}_3$ *and* $\mathbf{Hybrid}_4$ *are indistinguishable to a PPT adversary.*

Proof. The difference between $\mathbf{Hybrid}_3$ and $\mathbf{Hybrid}_4$ is that the simulator replaces the commitment to the PRF key with a commitment to 0.

Assume a distinguisher D can distinguish between $\mathbf{Hybrid}_3$ and $\mathbf{Hybrid}_4$, i,e. $Pr[D(\mathbf{Hybrid}_3) = 1] - Pr[D(\mathbf{Hybrid}_4) = 1] > \mathsf{negl}$

Using this distinguisher D we construct a reduction B that can break the hiding property of the commitment scheme.

<u>Reduction B</u>:
1. Activate the distinguisher D
2. The reduction simulates the protocol Π_{PriFHEte} as in $\mathbf{Hybrid}_3$.
3. Let PK_i be the public key of an honest party P_i
4. Send $m_0 = k$ and $m_1 = 0$ to the challenger and receive C
5. Replace the $\mathsf{C_{PRF}}$ in PK_i with C for party P_i
6. Instruct the environment to submit a transaction $(\texttt{NEW-PARTY}, \mathsf{PK}_i)$
7. Submit tx to $\mathcal{F}_{\mathsf{N\text{-}MC}}$.
8. Output whatever D outputs.

Note that in the case C was the encryption of m_0 the distinguisher sees the hybrid world - $\mathbf{Hybrid}_3$ and on the other hand when encryption of m_1 is returned the distinguisher sees the hybrid world $\mathbf{Hybrid}_4$.

Now since $Pr[D(\mathbf{Hybrid}_3) = 1] - Pr[D(\mathbf{Hybrid}_4) = 1] > \mathsf{negl}$, we have that $\mathrm{Adv}_{\text{CommHiding}} > \mathsf{negl}$ which is a contradiction since we assume a secure commitment scheme. This implies $Pr[D(\mathbf{Hybrid}_3) = 1] - Pr[D(\mathbf{Hybrid}_4) = 1] = \mathsf{negl}$.

□

Lemma 4. *By the pseudorandomness property of the underying* PRF *scheme, the hybrids* $\mathbf{Hybrid}_4$ *and* $\mathbf{Hybrid}_5$ *are indistinguishable.*

Proof. The difference between $\mathbf{Hybrid}_4$ and $\mathbf{Hybrid}_5$ is that the simulator replaces the PRFOut with a randomly sampled $y \leftarrow \{0,1\}^{\ell}$

Assume a distinguisher D can distinguish between $\mathbf{Hybrid}_4$ and $\mathbf{Hybrid}_5$, i,e. $Pr[D(\mathbf{Hybrid}_4) = 1] - Pr[D(\mathbf{Hybrid}_5) = 1] > \mathsf{negl}$

Using this distinguisher D we construct a reduction B that can break the pseudorandomness property of the underlying PRF scheme.

Reduction B:

1. Activate the distinguisher D
2. The reduction simulates the protocol Π_{PriFHEte} as in $\mathbf{Hybrid}_4$
3. Send ep the current epoch number to the challenger and receive y.
4. Instruct the environment to submit a transaction $(\mathsf{PK}_1, \mathsf{PK}_2, x)$, and replace the PRFOut in the transfer transaction with y.
5. Submit tx to $\mathcal{F}_{\text{N-MC}}$.
6. Output whatever D outputs.

Note that in the case PRFOut was computed using $\mathsf{PRF}(k, \cdot)$ the distinguisher sees the hybrid world - $\mathbf{Hybrid}_4$ and on the other hand when PRF output is a random $y \leftarrow \{0,1\}^\ell$ the distinguisher sees the hybrid world $\mathbf{Hybrid}_5$.

Now since $Pr[D(\mathbf{Hybrid}_4) = 1] - Pr[D(\mathbf{Hybrid}_5) = 1] > \mathsf{negl}$, we have that advantage of the adversary winning the PRF pseudorandomness game which is a contradiction since we assume secure PRFs. This implies $Pr[D(\mathbf{Hybrid}_4) = 1] - Pr[D(\mathbf{Hybrid}_5) = 1] = \mathsf{negl}$.

□

Lemma 5. *By the CPA security over multiple encryptions[KL20] of the underlying encryption scheme* FHE, $\mathbf{Hybrid}_5$ *and* $\mathbf{Hybrid}_6$ *are indistinguishable to a PPT adversary.*

Proof. The difference between $\mathbf{Hybrid}_5$ and $\mathbf{Hybrid}_6$ is that the simulator replaces the encryptions $\mathsf{k\text{-}ct}$ with encryptions of 0.

Assume a distinguisher D can distinguish between $\mathbf{Hybrid}_5$ and $\mathbf{Hybrid}_6$, i,e. $Pr[D(\mathbf{Hybrid}_5) = 1] - Pr[D(\mathbf{Hybrid}_6) = 1] > \mathsf{negl}$

Using this distinguisher D we construct a reduction B that can break the CPA security of encryption scheme.

Reduction B:

1. Activate the distinguisher D
2. The reduction simulates the protocol Π_{PriFHEte} as in $\mathbf{Hybrid}_5$.
3. Let PK_i be the public key of an honest party P_i
4. Send $\mathbf{m_0} = \mathsf{WKEnc.sk}_i$ and $\mathbf{m_1} = \mathbf{0}$ to the challenger and receive c
5. Replace the $\mathsf{k\text{-}ct}_i$ with c for party P_i
6. Instruct the environment to submit a transaction $(\mathsf{PK}_i, \mathsf{PK}_j, x)$ where P_j is another party.
7. Submit tx to $\mathcal{F}_{\text{N-MC}}$.
8. Output whatever D outputs.

Note that in the case $\mathsf{k\text{-}ct}_\mathbf{i}$ was the encryption of m_0 the distinguisher sees the hybrid world - $\mathbf{Hybrid}_5$ and on the other hand when encryption of m_1 is returned the distinguisher sees the hybrid world $\mathbf{Hybrid}_6$.

Now since $Pr[D(\mathbf{Hybrid}_5) = 1] - Pr[D(\mathbf{Hybrid}_6) = 1] > \mathsf{negl}$, we have that $\mathrm{Adv}_{\mathrm{CPA}} > \mathsf{negl}$ which is a contradiction since we assume CPA secure encryption over multiple encryptions. This implies $Pr[D(\mathbf{Hybrid}_5) = 1] - Pr[D(\mathbf{Hybrid}_6) = 1] = \mathsf{negl}$.

□

Lemma 6. *Assuming existential unforgeable signatures that are secure against chosen message attacks,* $\mathbf{Hybrid}_6$ *and* $\mathbf{Hybrid}_7$ *are indistinguishable.*

Proof. Note that the difference between $\mathbf{Hybrid}_6$ and $\mathbf{Hybrid}_7$ is that in $\mathbf{Hybrid}_6$ the event $\mathsf{sigFailure}_1$ can occur. We prove in this section that the probability of this event occurring is negligible.

First we observe that $\mathsf{sigFailure}$ occurs when the simulator receives a `MINT` transaction from the adversary that contains a signature that corresponds to that of an honest party.

Assume a distinguisher D can distinguish between $\mathbf{Hybrid}_6$ and $\mathbf{Hybrid}_7$, i,e. $Pr[D(\mathbf{Hybrid}_6) = 1] - Pr[D(\mathbf{Hybrid}_7) = 1] > \mathsf{negl}$

This implies that $Pr[\mathsf{sigFailure}] > \mathsf{negl}$.

Using this adversary we present a reduction B that breaks the EUF-CMA property of signature schemes.

Reduction B

1. Receive vk from the challenger. Update PK_i of an honest party P_i with vk.
2. Simulate the world as in $\mathbf{Hybrid}_6$.
3. Upon receiving a `MINT` transaction via the $\mathcal{F}_{\mathsf{N\text{-}MC}}$ functionality, check if the signature σ' corresponds to that of vk. If not, ignore.
4. If yes, output $m = (\mathsf{PK}_i, x, \mathsf{rt}_{\mathcal{T}_{\mathsf{pubAccounts}}})$ and $\sigma = \sigma'$

Observe that

$$\begin{aligned}\mathrm{Adv}^{\mathsf{euf-cma}}_{\Sigma,\mathcal{A}} &= Pr[\mathbf{Exp}^{\mathsf{euf-cma}}_{\Sigma,\mathcal{A}}(\lambda) = 1] \\ &= Pr[\Sigma.\mathsf{Verify}(\mathsf{vk}, m\sigma) = 1] > \mathsf{negl}\end{aligned}$$

But this is a contradiction since we assume EUF-CMA signatures and therefore $\mathrm{Adv}^{\mathsf{euf-cma}}_{\Sigma,\mathcal{A}} < \mathsf{negl}$

Hence $Pr[\mathsf{sigFailure}] < \mathsf{negl}$ and therefore $Pr[D(\mathbf{Hybrid}_6) = 1] - Pr[D(\mathbf{Hybrid}_7) = 1] < \mathsf{negl}$

□

Lemma 7. *Assuming collision-resistant hash functions,* $\mathbf{Hybrid}_8$ *and* $\mathbf{Hybrid}_9$ *are indistinguishable to a PPT adversary*

Proof. Note that the difference between $\mathbf{Hybrid}_8$ and $\mathbf{Hybrid}_9$ is that in $\mathbf{Hybrid}_8$ the event $\mathsf{CRHFailure}_1$ can occur. We prove in this section that the probability of this event occurring is negligible.

First we observe that $\mathsf{CRHFailure}$ occurs when the simulator receives a `TRANSFER` transaction from the adversary that contains a path that does not correspond to a path from an account and balance in $\mathcal{T}_{\mathsf{privAccounts}}$ owned by the sender to the root of the Merkle tree computed over $\mathcal{T}_{\mathsf{privAccounts}}$

Assume a distinguisher D can distinguish between $\mathbf{Hybrid}_8$ and $\mathbf{Hybrid}_9$, i,e. $Pr[D(\mathbf{Hybrid}_8) = 1] - Pr[D(\mathbf{Hybrid}_9) = 1] > \mathsf{negl}$

This implies that $Pr[\mathsf{CRHFailure}] > \mathsf{negl}$.

Using this adversary we present a reduction B that breaks the collision resistance property of the underlying hash scheme.

Reduction B

1. Simulate the world as in $\mathbf{Hybrid}_8$.

2. Upon receiving a TRANSFER transaction via the $\mathcal{F}_{\mathsf{N\text{-}MC}}$ functionality, get the witness w that corresponds to the proof π in the transaction.
3. Let path^* be the path in the Merkle tree (computed over $\mathcal{T}_{\mathsf{privAccounts}}$) from $(\mathsf{PK}_{\mathcal{A}}, v_{\mathcal{A}})$ to the root of the Merkle root $\mathsf{rt}_{\mathsf{privAccounts}}$.
4. Let $w = \mathsf{PK}_S, \mathsf{SK}_S, \mathsf{PK}_R, v, v_S, \mathbf{C}, \mathsf{path})$ and $\mathsf{VerifyPath}(\mathsf{rt}_{\mathcal{T}_{\mathsf{privAccounts}}}, \mathsf{path}) = 1$
5. If (PK_S, v_S) does not correspond to the adversary's entry in $\mathcal{T}_{\mathsf{privAccounts}}$, output $(m_0 = \mathsf{path}, m_1 = \mathsf{path}^*)$

Observe that

$$\mathrm{Adv}^{\mathsf{CRHF}}_{\mathcal{H},\mathcal{A}} = Pr[\exists m_0, m_1 \text{ s.t. } \mathcal{H}(m_0) = \mathcal{H}(m_)1))] > \mathsf{negl}$$

But this is a contradiction since we assume collision-resistant hash functions and therefore $\mathrm{Adv}^{\mathsf{CRHF}}_{\mathcal{H},\mathcal{A}} < \mathsf{negl}$

Hence $Pr[\mathsf{CRHFailure}] < \mathsf{negl}$ and therefore $Pr[D(\mathbf{Hybrid}_8) = 1] - Pr[D(\mathbf{Hybrid}_9) = 1] < \mathsf{negl}$

□

Lemma 8. *Assuming statistically binding commitments, the hybrids* $\mathbf{Hybrid}_9$ *and* $\mathbf{Hybrid}_{10}$ *are indistinguishable.*

Proof. Note that the difference between $\mathbf{Hybrid}_9$ and $\mathbf{Hybrid}_{10}$ is that in $\mathbf{Hybrid}_9$ the event $\mathsf{CommFailure}$ can occur. We prove in this section that the probability of this event occurring is negligible.

First we observe that $\mathsf{CommFailure}$ occurs when the simulator receives a TRANSFER transaction from the adversary and the PRF key in the extracted witness from this transaction is not the same as the PRF key that was committed to.

Assume a distinguisher D can distinguish between $\mathbf{Hybrid}_9$ and $\mathbf{Hybrid}_{10}$, i,e. $Pr[D(\mathbf{Hybrid}_9) = 1] - Pr[D(\mathbf{Hybrid}_{10}) = 1] > \mathsf{negl}$

This implies that $Pr[\mathsf{CommFailure}] > \mathsf{negl}$.

Using this adversary we present a reduction B that breaks the binding property of the underlying commitment scheme.

<u>Reduction B</u>

1. Simulate the world as in $\mathbf{Hybrid}_9$.
2. Upon receiving a TRANSFER transaction via the $\mathcal{F}_{\mathsf{N\text{-}MC}}$ functionality, get the witness w that corresponds to the proof π in the transaction.
3. Retrieve k_S from w.
4. From Π_{KeyGen}, retrieve the record for sk_S read k_S*
5. If $k_S \neq k_S^*$ output $(m_0 = \mathsf{path}, m_1 = \mathsf{path}^*)$ and $\mathsf{C}_{\mathsf{PRF}}$.

Observe that

$$\mathrm{Adv}^{\mathsf{Com}}_{\mathcal{A}} = Pr[\exists m_0, m_1 \text{ s.t. } Open(\mathsf{C}_{\mathsf{PRF}}) = k_S = k_S^*] > \mathsf{negl}$$

But this is a contradiction since we assume collision-resistant hash functions and therefore $\mathrm{Adv}^{\mathsf{Com}}_{\mathcal{A}} < \mathsf{negl}$

Hence $Pr[\mathsf{CommFailure}] < \mathsf{negl}$ and therefore $Pr[D(\mathbf{Hybrid}_9) = 1] - Pr[D(\mathbf{Hybrid}_{10}) = 1] < \mathsf{negl}$

□

F Regev Cryptosystem and Wrong-Key Decryption

The Regev cryptosystem is parameterized by integers n (the security parameter), m (number of equations), and a real $\alpha > 0$ (noise parameter). All operations are done modulo q (a prime)

- **Setup**: Choose a prime $q \leftarrow_{\$} [n^2, 2n^2]$, $m = 1.1 \cdot n \log q$ and $\alpha = 1/(\sqrt{n} \log^2 n)$
- **Key Generation**: Private key is a vector $\mathbf{s} \leftarrow \mathbb{Z}_q^n$ and the public key consists of m samples $(\mathbf{a}_i, b_i)_{i=1}^m$ from the LWE distribution with secret $\mathbf{s}$, modulus q and error parameter α. That is, $\mathbf{a}_i \leftarrow_{\$} \mathbb{Z}_q^n$ and $b_i = \mathbf{s}^T \mathbf{a}_i + e_i$ where e_i is error sampled from error distribution χ for $i \in [m]$
- **Encryption**: For each bit of the message do the following. Choose a random set S uniformly among all 2^m subsets of $[m]$. The encryption is $(\sum_{i \in S} \mathbf{a_i}, \sum_{i \in S} b_i)$ if the bit is 0 and $(\sum_{i \in S} \mathbf{a}_i, \lfloor \frac{q}{2} \rfloor + \sum_{i \in S} b_i)$ if the bit is 1.
- **Decryption**: The decryption of a pair $(\mathbf{a}, b)$ is 0 if $b - \langle \mathbf{a}, \mathbf{s} \rangle$ is closer to 0 than to $\lfloor \frac{q}{2} \rfloor$ mod q, and 1 otherwise.

Lemma 9. *Regev's encryption scheme described above satisfies the property of wrong-key decryption (Def 3).*

Proof. Let $(\mathsf{sk}, \mathsf{pk})$ be $(\mathbf{s}^*, (\mathbf{a}_i^*, b_i^*)_{i=1}^m)$ and let $(\mathsf{sk}', \mathsf{pk}')$ be $(\mathbf{s}^\dagger, (\mathbf{a}_i^\dagger, b_i^\dagger)_{i=1}^m)$

Now,

$$\mathsf{ct} \leftarrow \mathsf{WKEnc.Enc}(\mathsf{pk}, 1)$$
$$\implies \mathsf{ct} = (\sum_{i \in S} \mathbf{a}_i^*, \left\lfloor \frac{q}{2} \right\rfloor + \sum_{i \in S} b_i^*)$$

Furthermore let $\mathsf{ct} = (\mathbf{a}, b)$ then,

$$m' = \mathsf{WKEnc.Dec}(\mathsf{sk}', \mathsf{ct})$$
$$\implies m' = b - \langle \mathbf{a}, \mathbf{s}^\dagger \rangle \mod q$$
$$\implies m' = \left(\left\lfloor \frac{q}{2} \right\rfloor + \sum_{i \in S} b_i^* \right) - \left(\sum_{i \in S} a_{i,1}^* s_1^\dagger + \ldots + \sum_{i \in S} a_{i,n}^* s_n^\dagger \right) \mod q$$
$$\implies m' = \left(\left\lfloor \frac{q}{2} \right\rfloor + \sum_{i \in S} a_{i,1}^* s_1^* + e_1 \ldots + \sum_{i \in S} a_{i,n}^* s_n^* + e_n - \sum_{i \in S} a_{i,1}^* s_1^\dagger + \ldots + \sum_{i \in S} a_{i,n}^* s_n^\dagger \right) \mod q$$
$$\implies m' = \left(\left\lfloor \frac{q}{2} \right\rfloor + (s_1^* - s_1^\dagger) \sum_{i \in S} a_{i,1}^* \ldots + (s_n^* - s_n^\dagger) \sum_{i \in S} a_{i,n}^* + (e_1 + \ldots + e_n) \right) \mod q$$

Since $\mathbf{a}_i^*, e_i$ and $\mathbf{s}$ are all sampled randomly:

$$m' = \text{uniformly random element in } \mathbb{Z}_q$$

Thus,

$$m' > \frac{q}{2} \text{ with probability } \frac{1}{2}$$

Therefore

$$Pr[m' = 1] \leq 1/2 + \mathsf{negl}(\lambda)$$

□

G Full adder and subtracter

Let $\mathbf{a} = \{a_1, \ldots, a_\mu\}$ and $\mathbf{b} = \{b_1, \ldots, b_\mu\}$ be two vectors where each $a_i, b_i \in \{0, 1\}$. We present a full adder below that computes $\mathbf{c} = \mathbf{a} + \mathbf{b}$.

FullAdder$(\mathbf{a}, \mathbf{b})$
1. Set $\mathsf{cin} = 0$
2. For $i \in [\mu]$:
   (a) Compute $c_i = \mathsf{cin} \oplus a_i \oplus b_i$
   (b) Compute $\mathsf{cin} = a_i b_i \oplus b_i \mathsf{cin} \oplus a_i \mathsf{cin}$
3. Return $(\mathsf{cin}, c_1, \ldots, c_\mu)$

Figure 30: Full Adder

Below we describe a full subtracter that computes $\mathbf{c} = \mathbf{a} - \mathbf{b}$

FullSubtracter$(\mathbf{a}, \mathbf{b})$
1. Set $\mathsf{cin} = 0$
2. For $i \in [\mu]$:
   (a) Compute $c_i = \mathsf{cin} \oplus a_i \oplus b_i$
   (b) Compute $\mathsf{cin} = (\neg a_i) b_i \oplus b_i \mathsf{cin} \oplus (\neg a_i) \mathsf{cin}$
3. Return $(\mathsf{cin}, c_1, \ldots, c_\mu)$

Figure 31: Full Adder

H Potential for Deployment

As discussed in the introduction, the PriFHEte algorithms can be deployed as smart contracts. In this section, we discuss how our algorithms can be deployed on Ethereum. Next we discuss how we can alleviate the computation of miners by deploying PriFHEte as zk-rollups[Fou21] which are a new scalability solution for Ethereum.

H.1 Background on Ethereum and Smart Contracts

Accounts. Ethereum is an account-based cryptocurrency. There are two types of accounts in Ethereum - Externally Owned Accounts (EOA) and Contract Accounts. An EOA is associated with signature public key/private key pair and anyone who knows the private key can control the account. On the other hand, a Contract Account is controlled by the code of the smart contract. Now what is a smart contract? It's a collection of code (its functions) and data (its state) that resides at a specific address on the Ethereum blockchain. This address is simply the hash of the public key of the creator of the smart contract.

State of the blockchain. In Ethereum the state is a data structure called a modified Merkle Patricia tree, where the leaves of this tree are the accounts (both EOA and contract accounts). Each leaf is an address, data pair. The data for EOA accounts includes the balance associated to the address, and the data field of contract accounts include the code and the state of the contract.

Transactions. In Ethereum, there are three types of transactions.

1. A *regular transaction*, that transfers funds from one EOA to another EOA.
2. A *contract deployment transaction*, which deploys a smart contract on Ethereum. This transaction includes the code of the smart contract, and the address of the smart contract.
3. A *contract execution transaction*, which is addressed to one of the deployed smart contracts. This transaction may include inputs to the functions of the smart contract that are to be executed. Upon receiving this transaction, a miner executes the smart contract and updates the state of the smart contract and therefore the state of Ethereum.

H.2 PriFHEte as a smart contract

To describe the deployment of PriFHEte, we need to specify three different aspects: the setup, description of the smart contract, and the user algorithms. We will show how the algorithms described in Section 4 can be cast as smart contract functions and user algorithms.

The setup In the setup phase, public parameters such as the CRS are generated. Some entity, will also submit a Contract Deployment Transaction with the code for the PriFHEte smart contract. A miner updates the state by adding a smart contract account. The state of this account includes an empty table that will maintain account/encrypted balance pairs.

The smart contract. The smart contract has two functions: $\textsc{Registration}(\mathsf{PK}, \mathcal{T}) \rightarrow \mathcal{T}'$ and $\textsc{ProcessTransaction}(\mathsf{tx}, \mathcal{T}) \rightarrow \mathcal{T}'$

Observe that the two functions take as input the state $\mathcal{T}$ and output an updated state $\mathcal{T}'$. This is the internal state of the smart contract which is simply a table of account-encrypted balance pairs. The REGISTRATION function appends a new row with PK and an encryption to 0 under FHE.pk to the $\mathcal{T}$. Similarly PROCESSTRANSACTION processes a transaction as discussed in Figure 7 and updates *all rows* of the state.

User Algorithms A user runs one of the following algorithms to interact with PriFHEte smart contract.

The output of these algorithms are contract execution transactions, which are addressed to the PriFHEte smart contract.

1. $\textsc{KeyGeneration}(\lambda) \rightarrow (\mathsf{PK}, \mathsf{SK})$. With this transaction the user registers with the smart contract and joins the PriFHEte system. This transaction invokes the REGISTRATION function of the smart contract.
2. $\textsc{Mint}(\mathsf{PK}_i, \mathsf{PK}_i^{\mathsf{pub}}, \mathsf{SK}_i^{\mathsf{pub}}, x, \mathsf{rt}_{\mathcal{T}_{\mathsf{pubAccounts}}}) \rightarrow (\mathsf{tx}_{\mathtt{MINT}}, \sigma)$. With this transaction the user invokes a function that transfers funds from the main chain to the the PriFHEte smart contract. This transaction invokes the PROCESSTRANSACTION function of the smart contract.
3. $\textsc{Transfer}(\mathsf{PK}_S, \mathsf{SK}_S, \mathsf{PK}_R, x, ep, R, \mathcal{C}_{\mathsf{loc}}^S, \mathsf{path}_i, \mathbf{C_i}) \rightarrow \mathsf{tx}_{\mathtt{TRANSFER}}$. With this transaction the user invokes a function that transfers funds from one account to another maintained by the PriFHEte smart contract. This transaction invokes the PROCESSTRANSACTION function of the smart contract.

H.3 Alleviating storage and computation costs for the miners

As discussed in the introduction, we envision zk-rollups[Fou21] to aid the storage and computation costs of the miners. Below we first describe how rollups work and then briefly describe how the PriFHEte algorithms could be deployed as a rollup. We observe that this is the same approach taken by AZTEC[Wil18] to achieve privacy. The main difference between their work and ours is that they do a UTXO-style transactions on top of Ethereum, whereas we dont depart from the account-based paradigm. They use stealth address to achieve anonymity. This doesnt give full anonymity, since the sender of a transaction can always trace how the receiver is going to spend the coin.

Zero Knowledge (ZK) Rollups: There are three entities in a rollup protocol. The users, the miners and rollup operators. Rollup operators execute transactions off-chain. This reduces the amount of computation and the storage that miners need to do. These operators submit a summary of changes as well as validity proofs to prove correctness of the summary of changes. A miner can verify this validity proof to be convinced that the received state is a result of the execution of all the transactions in a batch.

The rollup architecture is comprised of two components:

- On-chain contract: this contract includes code to keep track of the updated state (which is a succinct representaion of the total state) and also code to verify validity proofs
- Off-chain computation: this is done by rollup operators that maintain the entire state, execute the transactions, compute validity proofs and compute succinct representations of the updated state.

PriFHEte as a rollup Users submit their transactions to the rollup operators. The rollups execute these transactions in batches and update the maintained state. The operators then submit a succinct representation of the state, along with validity proofs and the transactions to the main chain network. A miner verifies the validity proofs by executing the verifier function of the rollup smart contract. They then update the smart contract state with the received succinct representation of state.

We note that while we do not need to trust a rollup operator for privacy, we trust that they will include transactions of users.

Finally, we note using rollups help with gas fees because there is a fixed cost to writing to Ethereum's state. Without rollups each transaction would update $O(N)$ data entries of the state whereas with rollups one needs to update only a single data entry for multiple transactions. Therefore a rollup reduces this fixed cost by spreading the it across many users.