OCash: Fully Anonymous Payments between Blockchain Light Clients

Adam Blatchley Hansen[1][0009−0003−2090−9553][*], Jesper Buus Nielsen[1][0000−0002−7074−0683][**], and Mark Simkin[2][0000−0002−7325−5261]

[1] Aarhus University
[2] Flashbots

Abstract. We study blockchain-based provably anonymous payment systems between *light clients*. Such clients interact with the blockchain through full nodes, which can see what the light clients read and write. The goal of our work is to enable light clients to perform anonymous payments, while maintaining privacy even against the full nodes through which they interact with the blockchain. We formalize the problem in the UC model and present a provably secure solution. We show that a variation of tree ORAM gives obliviousness even when an adversary can follow how its own data elements move in the tree. We use this for anonymity via shuffling of payments on the blockchain, while at the same time allowing the light client to know a few positions among which to find its payment without knowing the current state of the blockchain. In comparison to existing works, we are the first ones that simultaneously provide strong anonymity guarantees, provable security, and anonymity with respect to full nodes. Along the way, we make several contributions that may be of independent interest. We define and construct anonymous-coin friendly encryption schemes and show how they can be used within anonymous payment systems. We define and construct efficient compressible randomness beacons, which produce unpredictable values in regular intervals and allow for storing all published values in a short digest.

[*] Funded by the Concordium Blockchain Research Center, Aarhus University.
[**] Founded by the Danish Independent Research Council under Grant-ID DFF-3103-00077B (CryptoDigi).

Table of Contents

1 Introduction

Blockchains are structured decentralized databases, which are stored redundantly by network participants, known as *full nodes*. In the context of cryptocurrencies, the databases are ledgers, which keep track of all transactions of digital currency among all digital identities. When new transactions are submitted, the full nodes check the validity of those transactions, e.g., whether the payer has enough digital money, and if this is the case, the transaction is added to the ledger. Being a full node in real-world systems such as Bitcoin or Ethereum is a daunting task, as they are required to continuously store hundreds of gigabytes of data. Since regular users cannot be expected to run full nodes, they can alternatively join these distributed systems as *light clients*, who do not need to store all of the data, but can still perform transactions. Without direct access to the full ledger, however, light clients communicate via full nodes.

Both Bitcoin and Ethereum are pseudonymous systems, meaning that the real identities of participants are hidden behind pseudonymous aliases. Importantly though, since all transactions are public on the ledger, any observer may see which pseudonyms interacted and what transactions were sent between them. It may seem that pseudonymity can provide a “reasonable amount of anonymity”, but it has been shown time and time again that this is not the case and that significant amounts of information about the real identities can be learned by carefully inspecting the transactions on the ledger [RH13, MPJ$^+$13, HF16, JBWD18].

Various cryptocurrencies, such as ZCash [BCG$^+$14], Mina [BMRS20], Dash, and Monero have been designed to provide some forms of anonymity for their users. The concrete guarantees that are provided by these schemes differ in their details, but they all have the same overarching goal of hiding which users have how much money and who interacts with whom. Unfortunately, these solutions are all currently somewhat unsatisfactory in one way or another, because they either require users to run full nodes, provide users with weak anonymity guarantees, or require users to give up anonymity towards the full nodes. As both usability and privacy are important for real-world cryptocurrencies, we ask the natural question of whether we can have the best of both worlds:

Can we support light clients and at the same time provide strong anonymity guarantees?

Answering this question is a challenging task, since the two goals may seem to be at odds with each other at first sight. Purely intuitively speaking, anonymity seems to require that users access large portions of the ledger to hide which data is relevant to them specifically, but light clients require the opposite, namely that accesses are highly localized. Nonetheless we will show in this work that, making strong—but plausible—assumptions, the question can be answered positively.

1.1 Our Contribution.

We introduce OCash (<u>O</u>blivious RAM based <u>Cash</u>), a cryptocurrency that provides strong anonymity guarantees and supports light clients. Towards the goal of constructing OCash, we develop multiple tools that may be of independent interest. In more detail, we make the following contributions.

Formal Model of Fully Anonymous Light Payments. To set the stage for a rigorous formal analysis of claims about anonymity, we first propose a conceptually simple model, in the universal composability (UC) framework of Canetti [Can01], of what fully anonymous payment schemes are. The UC framework ensures that any protocol proven secure, will remain secure when run in a larger context, e.g., alongside other protocol executions.

We model the ledger as an append only list that can be accessed by full nodes. Light clients can read from and write to the ledger through full nodes, but any position they access and any message they post will be leaked to the adversary. Light clients will have accounts on the ledger that store encrypted amounts of currency and they can perform anonymous payments between each other. Security will mean that an adversary observing a payment on the blockchain, cannot connect the sending and the receiving accounts, nor can they see the amount that is being transferred. A bit more precisely, we will model payments as payers placing coins on the ledger, which can then be collected by the payees. We distinguish between weak and strong anonymity. In weak anonymity the payer can see when the payee claims a coin. In strong anonymity they cannot. Security notions related to strong anonymity have been proposed before, see [CHK23], but no formalization was provided.

OCash. We show that efficient payment schemes supporting light clients and satisfying strong anonymity can be constructed by using ideas from the oblivious RAM (ORAM) literature [Gol87, Ost90] in combination with several other tricks that we introduce in this work. Placing or collecting a payment, even as a light client, only requires writing and reading a polylogarithmic (in the total number of performed payments in the whole system) amount of data from the ledger. We stress that our work is the first solution that allows for strong anonymity with sublinear read and write overhead to the ledger.

Our construction will make two assumptions that are worth discussing. The first one is the existence of a private off-chain communication channel between the payer and the payee. As an example, the payer should be able to obtain the recipient's account address without needing to ask a full node and without needing to read the whole ledger. For instance, a pizza shop accepting cryptocurrency payments could provide a QR code with its address to hungry customers.

The second assumption is the existence of an anonymizer service associated with the ledger, which can hold a private state and which regularly posts messages on the ledger. In the case of proof-of-stake blockchains, one already often assumes the existence of committees that post messages on the ledger regularly. Given such committees, one can realize the anonymizer service, which holds a private state, via secure multiparty computation [BGG^{+}20, GHK^{+}21]. While our second assumption is stronger than our first one, we still think that it can be reasonable in the context of proof-of-stake blockchains.

Compressible Randomness Beacons. Part of our construction will be a randomness beacon that regularly publishes independent, unpredictable samples. Looking ahead, payees will need to access certain beacon outputs for collecting coins that were paid to them. For anonymity reasons, payees will not want to reveal, which outputs are relevant to them, but if they are light clients, they can also not afford reading all of the outputs. We show how to construct compressible randomness beacons, which allow the payee to read the latest beacon output and from there they can derive all previous outputs. A very similar notion was recently introduced by Beaver *et al.* [BCK^{+}23], but in their construction the computational overhead for computing a beacon value from the past is linear in the time between the latest and the desired beacon output. In our construction, the computational overhead for computing any value from the past is fixed.

Anonymous Coin Friendly Encryption. The anonymizer service will move around encrypted coins inside of some data structure. To be able to both prove efficiently that each individual movement was performed correctly and to also allow for an efficient and anonymous collection, the encryption scheme needs to satisfy certain additional properties. Additionally, it needs to be proof friendly in the sense that associated zero-knowledge (ZK) proofs should ideally be simple and be concretely efficient. We abstract out the properties we need from the encryption scheme into a new notion of an anonymous coin friendly encryption (ANCO) scheme. Our actual construction is similar to the one used in Quisquis [FMMO19], but in contrast to their work, we formalize and prove the properties that we require from the encryption scheme.

1.2 Related Work.

We are far from the first to consider anonymous cryptocurrencies or light clients. To better understand the challenges we need to overcome in our work, let us review existing anonymous cryptocurrencies and see what challenges they face, when trying to support light clients.

Cryptocurrencies Based on Accumulators. Anonymous electronic payments saw their birth in 1982 with David Chaum's eCash system [Cha82]. His construction involves a bank, a user, and a (pizza) shop. The user picks a random transaction identifier tid and asks the bank to sign tid blindly, i.e., without actually seeing the signed message, in exchange for one pound. The user receives signature $\sigma = \mathsf{Sign}(\mathsf{sk}_\mathsf{B}, \mathsf{tid})$, where sk_B is the signing key of the bank. To pay the shop one pound, the user provides it with (tid, σ). The shop can verify the validity of the signature σ and then collect back one pound from the bank by providing it with (tid, σ). If tid was previously collected, the bank rejects the collection. Since the bank has signed tid blindly, the original owner of this coin is anonymous among all parties that have received a coin from the bank.

Fischlin [Fis06] then showed how to construct blind signatures from general assumptions. Combined with Chaum's eCash idea above, Fischlin's approach would proceed as follows. The user would pick tid and send a commitment $c = \mathsf{Commit}(\mathsf{tid}; s)$ to the bank, which returns $\sigma = \mathsf{Sign}_{\mathsf{sk}_\mathsf{B}}(c)$. Here tid is the committed message and s is the commitment randomizer. The user would then locally construct a non-interactive ZK proof of knowledge π for statement tid and witness (σ, c, s), such that σ is a signature on c and $c = \mathsf{Commit}(\mathsf{tid}; s)$. They could then use (tid, π) to pay the shop.

Zerocoin [BCG$^+$14], later deployed under the name ZCash, further generalizes Chaum's original eCash idea by replacing the centralized bank with a decentralized ledger. The user posts $c = \mathsf{Commit}(\mathsf{tid}; s)$ on the ledger and pays, with the currency of the ledger, for having minted a new coin. The presence of c on the ledger is the authentication of c being a coin. To collect a coin the shop computes a ZK proof of knowledge π for the statement "I know (c, w, s) such that w is a witness that c is on the ledger and $c = \mathsf{Commit}(\mathsf{tid}; s)$" and post (tid, π) to the ledger. If tid was used before, the collection is rejected.

To compute the proof efficiently, the user first aggregates all coins on the ledger into an accumulator value ac via a public aggregation procedure and then provides a proof π for the statement "I know (c, w, s) where w is a proof that c is in ac and where $c = \mathsf{Commit}(\mathsf{tid}; s)$". The accumulator could, for example, be the root of a Merkle tree and w a path in it. The size of the statement and proof would then be poly-logarithmic in size in the number of coins. How to make such proofs concretely efficient was recently shown in a series of works [CH22, CFH$^+$22, ZBK$^+$22, STW23].

Even though cryptocurrencies based on accumulators provide good anonymity guarantees, it is unfortunately not clear how to make them compatible with light clients. To collect a coin, a light client needs to aggregate all coins into the accumulated value ac and prove that c is among them. The light client could outsource this task to a full node, but this would reveal c and thus the coin the client aims to collect towards the full node. Alternatively, the client could also ask the full node to not only compute the accumulator ac, but also an individual proof of membership for every single coin aggregated into ac. The client could then use private information retrieval [CGKS95] to obtain the one membership proof that is relevant to them. This solution would in principle work, but it would incur a prohibitive amount of computation on the full node, which renders this approach practically infeasible.

Cryptocurrencies Based on Mixers and Tumblers. A conceptually different approach was suggested by cryptocurrencies like Dash and Monero, where coins get repeatedly anonymized in small batches. In older systems like Dash this was done through the use of tumblers [Max13] that take n coins as input, produce n coins as output, and ensure that no external observer can link the owner of any input coin to any specific output coin. More recent systems, like Monero, rely on linkable ring signatures [RST01, LWW04], which enable signers to generate signatures for arbitrarily selected groups of n verification keys, concealing the specific secret key used, while ensuring that the signing key belongs to one of the public keys. As all these approaches require linear in n work for placing and collecting coins, their efficiency crucially relies on the value n being not too large. In Monero[3], for instance, the value n is chosen to be just 16 and even though each individual mixing step only provides a small amount of anonymity, the hope behind Monero and its kind is that eventually all coins get mixed enough to be untraceable.

While we have a reasonably good understanding of the relevant cryptographic primitives in isolation, we are currently lacking a solid understanding of the precise anonymity guarantees that cryptocurrencies like Dash and Monero provide, which is evidenced by the various attacks on these systems that have been found over the past years [OMJ$^+$13, KFTS17, YAY$^+$19, DS21, Vij23].

Intermezzo on Consensus and Long-Range Attacks. Before discussing the next approach for constructing cryptocurrencies, we need to look a bit closer at what a decentralized ledger really is and how it works from a consensus perspective. A ledger is, simply speaking, a growing chain of blocks. When new transactions appear on the network, they are placed into a new block, which is then appended to the end of the current chain. What is considered to be the true state of the ledger is decided by a consensus mechanism among the nodes, who are identified by their public verification keys, in the network[4]. A popular method for incentivizing the nodes on the network to behave honestly, is to use financial rewards and punishments. In Ethereum, for instance, nodes that want to play a role in reaching consensus, need to deposit a fixed amount of money that may be partially or fully slashed if they misbehave.

At some point in time, nodes that participate in the consensus may choose to get back their deposit and leave the system. When this happens, their public and private keys become worthless to them as they cannot be used to participate in any future consensus decision making and they are not

[3] `https://www.getmonero.org/resources/moneropedia/ring-size.html`

[4] In our discussion we focus on proof-of-stake blockchains as those are the focus of our work.

tied to any financial stake any longer either. To an attacker, however, these keys may still be very valuable as they allow for key-buying long-range attacks. To perform such an attack, an adversary attempts to buy as many of those "worthless" keys as possible from nodes that used to participate in the consensus. Having acquired a sufficient amount of keys, the adversary may change previous consensus decisions, thereby forking the chain of blocks that used to be the true state of the ledger and creating an alternate chain, which may then be falsely accepted by users. This attack is a serious real-world problem as selling keys that are otherwise worthless is the financially rational thing to do. A technique that aims to prevent this type of attack is checkpointing. Whereas classically a blockchain grows from its first block, the genesis block, by following some fixed hardcoded rules, the idea of checkpointing is to regularly accept intermediate blocks in the chain as unequivocal truths that cannot be changed. Once a block becomes a checkpoint, no adversarial behavior can produce a fork prior to this block, thus severely limiting the scope of possible key-buying long-range attacks.

Cryptocurrencies Based on Recursive Proofs. Armed with the above insights, the last type of cryptocurrency we want to discuss is based on the idea of always proving the validity of the latest block with respect to the genesis block using succinct proof systems. In other words, if γ is the genesis block, then each new block comes along with a succinct proof, attesting that the latest block is on a valid chain starting in γ. Computing such proofs can be done efficiently by extending the proof of the previous block using incrementally verifiable computation [Val08]. A bit more precisely, there exists an efficiently computable predicate Ver and efficiently computable proof τ for message m and position p, such that $\mathsf{Ver}(\gamma, p, m, \tau) = \top$, if and only if m is in position p on the ledger in a finalized block in a chain starting from γ. A prominent example that relies on such an approach is the Coda blockchain [BMRS20], later launched as Mina.

Cryptocurrencies based on recursive proofs as described above have some advantages, when it comes to realizing anonymous payments. To place a coin on the ledger, we again pick a uniformly random tid, randomizer s, and compute $c = \mathsf{Commit}(\mathsf{tid}; s)$. Once the coin is on the ledger in position p, the user obtains a succinct proof τ and they can then pay the shop by providing it with the tuple (p, c, τ, s). To claim the coin, the shop simply computes a ZK proof of knowledge π for the statement "I know (p, c, τ, s) such that $\mathsf{Ver}(\gamma, p, c, \tau) = \top$ and $c = \mathsf{Commit}(\mathsf{tid}; s)$" and posts (tid, π) on the ledger. Since the proof π hides both p and τ, the claimed coin is anonymous among all existing unclaimed coins.

There are some challenges that would need to be overcome, if one wants to make the above approach work for light clients. First of all, light clients need a way of efficiently making sure that tid was not previously used. As part of our construction in this work, we show how this problem can be overcome and we think that our solution would also be applicable here. There is, however, a larger problem that does not have a clear solution. The approach towards anonymous payments we are currently discussing, inherently relies on the fact that we prove statements with respect to the *genesis block* γ, but if we want security against key-buying long-range attacks then we need to prove statements with respect to the latest *checkpoint*. Alas, if the proof τ that the user receives after minting a coin is relative to the latest checkpoint, then claiming the coin as above would reveal the checkpoint and thus reveal temporal information about which coin was claimed. If the shop wants to compute a proof π that does not reveal the checkpoint, or which is relative to the latest checkpoint, then it is not clear how to compute the proof. The shop would as a minimum have to download all checkpoints, which is again a linear amount of data. The shop may again try to outsource the proof computation to a full node, but similarly to what we have already discussed in the context of accumulator-based cryptocurrencies, this would incur a prohibitively high computational overhead on that full node.

1.3 Technical Overview

We will now give an overview of the architecture of OCash and an overview of how we model and prove the protocol secure in the UC framework. Later we give details on the tools used in the construction.

Before discussing our construction, let us recall what we aim to construct. In our setting, we consider a ledger, modelled as an append-only list, which can be accessed by full nodes. Light clients can access the ledger through (possibly untrusted) full nodes and may want to either perform payments by placing coins intended for some recipient or they may want to collect coins that have been payed to them. In terms of anonymity, we would like to ensure that no external observer, not even a collusion of all full nodes, can determine who is paying whom. We say that a construction satisfies strong anonymity if in addition the payer cannot see when a payee collects a coin. We assume that light clients have an off-chain communication channel between them and we also assume the existence of a trusted anonymizer service, which can hold a private state and can repeatedly post messages on the ledger. As mentioned before, the anonymizer service can be implemented via secure computation protocols, but for the sake of this work, we just assume a trusted party performing these actions. The service never needs to interact with any of the parties directly and merely operates on the values that are posted on the blockchain.

In terms of efficiency, we would like all parties, i.e., anonymizer service, full nodes, and light clients, to do as little work as possible. More concretely, we would like them all to only perform a poly-logarithmic (in the total number of coins in the system) amount of work for placing or collecting a coin.

How to Place a Coin. Before discussing how to place a coin in OCash, we first make two simplifying assumptions. We assume that all coins have a unit value and we only aim for weak anonymity, where the payer can see when the payee collects the coin. We discuss how we allow for placing arbitrary monetary values and achieve strong anonymity at the end of this technical overview.

To pay a coin with identifier tid to a shop S, which is identified by a public encryption key ek_S, we use an encryption scheme that is both rerandomizable and key-indistinguishable. By rerandomizable we mean that ciphertexts can be rerandomized without knowing the public key under which the encryption was performed. By key-indistinguishability, which was originally introduced by Bellare *et al.* [BBDP01], we mean that a ciphertext cannot be linked to the public encryption key with which it was generated or, in other words, the ciphertext does not reveal who is the intended recipient. Concretely, the shop will have an ElGamal [ElG85] public key $(g, h = g^x) \in \mathbb{G}^2$ for a group $\mathbb{G}$, where the Diffie-Hellmann problem [DH76] is hard, and the encryption of $\mathsf{tid} \in \mathbb{G}$ will be

$$\mathsf{Enc}_{\mathsf{ek}_\mathsf{S}}(\mathsf{tid}) = (g^\rho, h^\rho, g^\sigma, h^\sigma \cdot \mathsf{tid})$$

for uniformly random $\rho, \sigma \in \mathbb{Z}_q$. The shop can use their secret key x to determine whether a ciphertext is for them, by checking whether $(g^\rho)^x \overset{?}{=} h^\rho$. The ciphertext can be rerandomized with values ρ' and σ' by computing

$$\left((g^\rho)^{\rho'}, (h^\rho)^{\rho'}, (g^\rho)^{\sigma'} g^\sigma, (h^\rho)^{\sigma'}(h^\sigma \cdot \mathsf{tid})\right) = \left(g^{\rho\rho'}, h^{\rho\rho'}, g^{\rho\sigma'+\sigma}, h^{\rho\sigma'+\sigma} \cdot \mathsf{tid}\right).$$

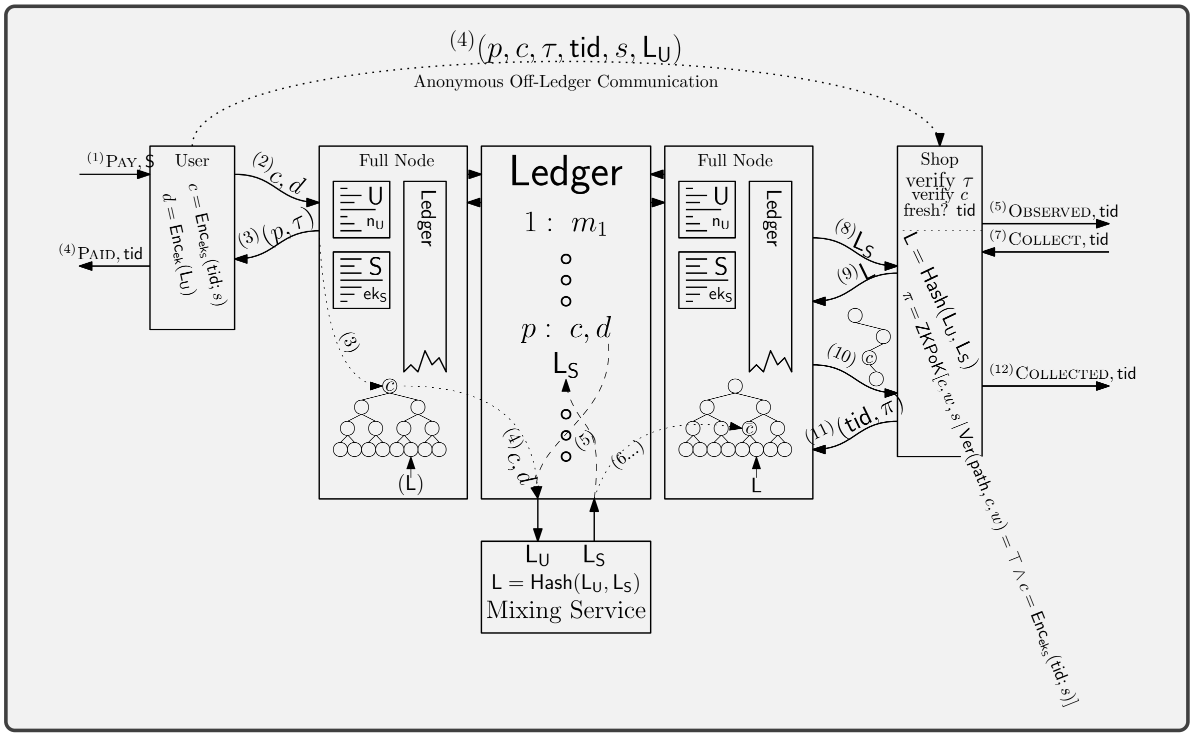

Fig. 1. An overview of Π_{AnonPay}. The user posts an encryption c (for the shop) of the transaction identifier tid along with an encryption d (for the service) of its contribution $\mathsf{L_U}$ to the leaf.$^{(2)}$ Once c and d were posted on the ledger the SOROM smart contract places c in the root of the tree and the user gets a proof τ that (c, d) was posted successfully on the ledger.$^{(3)}$ After that the user sends tid, coin c, opening s of c, τ, and $\mathsf{L_U}$ anonymously to the shop.$^{(4)}$ If τ verifies, then the shop considers the payment observed.$^{(5)}$ In parallel with this the service retrieves c and d.$^{(4)}$ It publishes its contribution $\mathsf{L_S}$ (via a CRaB).$^{(5)}$ The service will continually push a rerandomized version $\overline{c}$ of c towards leaf $\mathsf{L} = \mathsf{Hash}(\mathsf{L_U}, \mathsf{L_S})$.$^{(6\ldots)}$ The shop can via a full node anonymously learn $\mathsf{L_S}$ and publish L.$^{(8,9)}$ It gets back all coins on the path to L.$^{(10)}$ It collects the coin by posting tid and proving that it is in one of the encryptions on the path.$^{(11)}$

High-Level Approach. On a conceptual level, OCash uses the anonymizer service to maintain an ORAM [Gol87, Ost90, SCSL11], which can be accessed by light clients via full nodes for placing and collecting coins. An ORAM can be thought of as an encrypted array stored on an untrusted server, which can be accessed via read and write operations by a data owner holding the corresponding secret key unknown to the server. The main security guarantee of an ORAM, known as obliviousness, dictates that the server cannot see which operations are performed at which locations. To achieve obliviousness the data owner performs dummy accesses along with the real operations and shuffles around elements in the encrypted array. Efficiency of an ORAM is measured in terms of how many dummy accesses are needed for each real operation and it is known that for an array of length n, it is both necessary [LN18] and sufficient [Ost90] to perform $\Theta(\mathsf{polylog}\ n)$ dummy accesses. In our context, the ledger will play the role of the server, whereas the anonymizer service will play the role of the data owner.

There are several important differences between what an ORAM provides and what we need. In terms of functionality, we do not strictly need an array with arbitrary read and write accesses, but just some data structure that allows for inserting coins and then retrieving them at most once. In terms of security, we will require a stronger notion than standard obliviousness, because payees will be able to track coins that have been paid to them in the data structure. This means that the adversary obtains additional leakage about the movements of some elements in the ORAM, which is not part of the regular ORAM obliviousness definition. Intuitively, we need to ensure that the movements of the adversarially tracked coins within the ORAM do not reveal any information about the movements of the honest users' coins.

We call the cryptographic primitive we require a *strongly oblivious read-once*[5] *map* (SOROM) and as we show in Section 5, there are existing ORAM constructions, which almost immediately provide us with our desired primitive.

In particular, we will make use of the tree-based ORAM of Shi *et al.* [SCSL11]. In their construction, an array of length n is represented as a binary tree with n leaves. Each node in the tree is a bucket, which can store a fixed number of data elements. The write operation inserts the new data elements into the root node bucket and assigns each of them a uniformly random leaf index. A maintenance operation is performed regularly and ensures that data elements are pushed down towards their assigned leaves, thereby making sure that no buckets overflow with elements. The data structure obeys the invariant that each element is always in one of the buckets on the path between root node and assigned leaf. Reading an element is done by first magically determining the corresponding leaf index[6] and then retrieving all $\log n$ buckets on the path from root to leaf.

In our context, we observe that no coin is spent twice and thus no coin is read more than once in the ORAM. This means that the coin does not need to be moved back to the root after being read, as is usually done, and will always be on the path that it was assigned to upon its initial insertion in the data structure. This in turn will simplify finding the leaf index belonging to a specific data element. Additionally, we will show that using a variation of the maintenance operation from [SCSL11], all elements' movements are independent of each other and thus the adversary does not learn anything about the positions of the honest coins by observing the movements of the adversarial coins. The variation is that when maintaining a bucket, we push *all* elements in the bucket one level down, whereas in [SCSL11] at most *one* element is pushed down, which would correlate the movement of coins. We prove that this new eviction rule maintains obliviousness.

We assume a ledger with smart contracts. When posting a coin c the users gives a proof of knowledge that it is well-formed. The smart contract for the SOROM checks that the proofs are valid and if so places c in the root of the tree. The mixing server will interact with the smart contract to update the tree. It reads up coins c along with associated encryptions d of L, routes them, rerandomizes the c and writes them back in their new position in the tree via the smart contract, along with a fresh encryption d' of L.

How to Find a Coin. Before we can talk about how the shop can find coins they receive, we first need to talk about how the label L is chosen when a coin is inserted into our SOROM. The payer cannot choose it arbitrarily, as correctness properties of our SOROM rely on it being uniformly random. The label can also not be publicly known during coin placement as the shop will later reveal it during coin collection, which would lead to anonymity issues. Lastly, the payment process should be non-interactive in the sense that the payer can simply post a single message on the ledger, so the label can also not be chosen via a protocol that would require interaction between anonymizer service and payer.

[5] Spending a coin will require reading it and since no coin can be spent twice, we never need to read an element in the map more than once.

[6] We will elaborate on how this works in our context below.

Our idea for choosing the label L is to let the user U and the anonymizer service perform a coin flip into the well [Blu82]. When placing the coin at time t, the user chooses a label L_U and provides it to both the shop and the anonymizer service. Providing L_U to the shop happens off-chain and providing it to the anonymizer can be done by placing an encryption thereof, under the public key of the anonymizer service, on the ledger. The anonymizer service then publishes its label share $\mathsf{L}_{\mathsf{A},t}$ for time t and defines the coin's label as $\mathsf{Hash}(\mathsf{L}_\mathsf{U} \| \mathsf{L}_{\mathsf{A},t})$, where Hash is modeled as a random oracle. This approach already gives us most of what we want. The label is uniformly random and if U was honest, then it is also unpredictable for any outside observer who knows $\mathsf{L}_{\mathsf{A},t}$.

There is, however, still one problem. How does the shop, which runs a light client learn $\mathsf{L}_{\mathsf{A},t}$? Retrieving all labels ever published by the anonymizer is not feasible for a light client and asking a full node for the specific label would reveal the value t, which would in turn pose a problem for anonymity as t would leak information about the time of the coin's placement.

To circumvent this problem, we introduce the notion of a compressible randomness beacon. Rather than choosing the values $\mathsf{L}_{\mathsf{A},1}, \mathsf{L}_{\mathsf{A},2}, \ldots$ fully at random, we will let the anonymizer have a secret key k and let them choose $\mathsf{L}_{\mathsf{A},t} := \mathsf{PRF}(k, t)$ using a special pseudorandom function (PRF), which is inspired by constrained PRFs [BW13, KPTZ13, BGI14]. Now rather than publishing $\mathsf{L}_{\mathsf{A},t}$ at time step t, we let the anonymizer publish a *short* constrained key $k_{\leq t}$, which allows anybody to compute $\mathsf{L}_{\mathsf{A},t'}$ for any $t' \leq t$, but keeps any value larger than t unpredictable. This allows the light client to simply retrieve the latest $k_{\leq t}$ and recompute whatever specific label they need locally.

How to Collect a Coin. At this point, we know how the coin is placed and we know how the shop can figure out the path in our SOROM on which the coin will be. Given the properties of our encryption scheme, the shop can also determine which specific ciphertext in the buckets on the path was intended for them. The last question remaining is how the shop can collect the coin. For this, the shop posts tid on the ledger and proves in ZK that one of the ciphertexts on the path. The obliviousness guarantees of the SOROM ensure that revealing the path does not reveal anything about when the coin was inserted.

If tid was already posted on the ledger, then the collection of the coin is rejected. If the tid is chosen arbitrary by the payer, then the shop has no way of verifying that the identifier was not used before. The used tid may already appear among the published identifiers on the ledger or maybe the same tid was used to pay another shop, who also did not yet collect the coin. To prevent these types of double spending attacks, we endow the tid with some more structure. We will assume that each user has a public nonce n_U. Whenever the user puts a new coin on the ledger, the nonce gets incremented. The transaction identifier chosen by U, when paying shop S the amount a is defined as the commitment $\mathsf{tid} = \mathsf{Commit}(\mathsf{U}, \mathsf{S}, n_\mathsf{U}, a)$. Concretely, we will use $\mathsf{tid} = g_0^\xi \cdot g_1^\mathsf{U} \cdot g_2^\mathsf{S} \cdot g_3^{n_\mathsf{U}} \cdot g_4^a$ as our commitment, where ξ are the random coins and the values U and S are assumed to be the involved parties' identifiers interpreted as finite field elements. When placing an encryption of this tid on the ledger, the user U proves in ZK that everything is well formed and that tid contains the correct U and the correct value n_U. We show that computing these ZK proofs can be done very efficiently. All in all, the encryption of this tid has the form

$$c = \mathsf{Enc}_{\mathsf{ek}_\mathsf{S}}(\mathsf{tid}) = \left(g^\rho, h^\rho, g^\sigma, h^\sigma \cdot \underbrace{g_0^\xi \cdot g_1^\mathsf{U} \cdot g_2^\mathsf{S} \cdot g_3^{n_\mathsf{U}} \cdot g_4^a}_{\mathsf{tid}} \right) . \quad (1)$$

If the coin is well formed, the ledger accepts it and provides a proof τ, which attests that the coin is on the ledger.

During payment, the user sends $(c, (\rho, \sigma, \xi, \mathsf{U}, \mathsf{S}, a), \tau)$ to the shop via their off-chain communication channel. The shop checks that the received coin c was correctly constructed using the values $(\rho, \sigma, \xi, \mathsf{U}, \mathsf{S}, a)$ and that τ indeed attests that c is on the ledger. These checks do not require the shop to interact with the ledger or any full node. The binding property of the commitment ensures that two valid coins for different shops cannot have the same identifier tid. Thus the shop only needs to make sure that it itself did not already accept tid.

How to Achieve Strong Anonymity. With the above approach, the shop needs to reveal tid during collection, which allows the payer to see when the coin is collected. To achieve strong anonymity, we need to hide tid during collection. For this purpose, we further extend the encryption that defines a coin by one component. Let $\mathsf{hid} = \mathsf{Hash}(\mathsf{U}, n_{\mathsf{U}})$ be the hashed identifier, where Hash is a collision-resistant hash function and let

$$c = \mathsf{Enc}_{\mathsf{ek_S}}(\mathsf{tid}) = \left(g^{\rho}, h^{\rho}, g^{\sigma}, h^{\sigma} \cdot g_0^{\xi} \cdot g_1^{\mathsf{U}} \cdot g_2^{\mathsf{S}} \cdot g_3^{n_{\mathsf{U}}} \cdot g_4^{a} \cdot g_5^{\mathsf{hid}}\right) .$$

When placing a coin, the user U proves that the ciphertext is well formed by proving in ZK that the components g_1, g_3, g_5 contain the correct U, n_{U}, and hid, where $\mathsf{hid} = \mathsf{Hash}(\mathsf{U}, n_{\mathsf{U}})$. Note that $(\mathsf{U}, n_{\mathsf{U}}, \mathsf{hid})$ is in the instance so $\mathsf{hid} = \mathsf{Hash}(\mathsf{U}, n_{\mathsf{U}})$ can be verified in the clear and we do not have to give a ZK proof over the circuit of Hash. During collection, the shop proves that the g_2-component is S. What remains to be proven is that the coin was not previously collected. For this we note that the value $(\mathsf{U}, n_{\mathsf{U}})$ is unique as n_{U} is incremented for each payment of U. By the collision-resistance of the hash function, this means that hid is also computationally unique.

The idea behind achieving strong anonymity is to extend the shop's public key by including a commitment to a PRF key K and then, upon collection, letting it reveal an oblivious transaction identifier $\mathsf{otid} = \mathsf{PRF}(K, \mathsf{hid})$ along with a proof that otid was correctly computed w.r.t. the claimed coin. If hid is used only once, then otid is pseudorandom and leaks no information, even to the user knowing hid. If the shop attempts to collect the same hid multiple times, then otid will repeat and the collection will be rejected.

To make this proof efficient, we will use a slight modification of the Dodis-Yampolskiy verifiable random function(VRF) [DY05]. In general VRFs can be seen as commitments to random functions, which allow the committing party to open function evaluations at arbitrary points. We extend the shops public key by a component g_5^K, where K is the corresponding secret key and

$$\mathsf{PRF}(K, \mathsf{hid}) = g_5^{1/(K+\mathsf{hid})}.$$

During collection the shop reveals otid and proves knowledge of a vector $(c, x, K, \mathsf{hid}, \xi, a, \mathsf{U}, \mathsf{S}, n_{\mathsf{U}})$, where

$$c = \mathsf{Enc}_{\mathsf{ek_S}}(\mathsf{tid}) = \left(g^{\rho}, h^{\rho}, g^{\sigma}, h^{\sigma} \cdot g_0^{\xi} \cdot g_1^{\mathsf{U}} \cdot g_2^{\mathsf{S}} \cdot g_3^{n_{\mathsf{U}}} \cdot g_4^{a} \cdot g_5^{\mathsf{hid}}\right) =: (c_1, c_2, c_3, c_4),$$

such that

$$c_1^x = c_2 \wedge c_3^x \cdot g_0^{\xi} \cdot g_1^{\mathsf{U}} \cdot g_2^{\mathsf{S}} \cdot g_3^{n_{\mathsf{U}}} \cdot g_4^{a} \cdot g_5^{\mathsf{hid}} = c_4 \wedge \mathsf{otid} = \mathsf{PRF}(K, \mathsf{hid}) \ .$$

This can be done efficiently with off-the-shelf Σ-protocols.

Implementing the Service. In this work the service is assumed to be implemented by an incorruptible $\mathcal{F}_{\textsc{Service}}$, and we leave it as future work to implement $\mathcal{F}_{\textsc{Service}}$ in MPC. However, we want to add a few sentences on how one could proceed. We propose to implement $\mathcal{F}_{\textsc{Service}}$ using an MPC among n servers, where privacy is guaranteed if any $t < n$ servers are corrupted and where correctness is guaranteed even if $t = n$ servers are corrupted. This notion was dubbed *Universally Verifiable Multiparty Computation* (uvMPC) in [SV15]. In [DPSZ12] such a uvMPC was constructed based on somewhat homomorphic encryption, as a modification of the SPDZ protocol[DPSZ12]. The SPDZ protocol has good practical efficiency. The construction in [DPSZ12] only adds a factor 2 in overhead over SPDZ. The protocol is proven secure in the UC model, so we can use the UC theorem to plug it in for $\mathcal{F}_{\textsc{Service}}$. The reason for choosing [DPSZ12] is that the computation being MPC'ed can be done modulo a prime q. Furthermore, values being computed on are committed using Pedersen commitments in a group $\mathbb{G}$ of order q. All that is needed is that the discrete logarithm (DL) problem is hard in $\mathbb{G}$. Concretely we can get a protocol $\Pi_{\textsc{Service}}$ which is UC secure based on somewhat homomorphic encryption and DL being hard in $\mathbb{G}$.[DPSZ12, (Thm.1, Thm. 2)] If we set $\mathbb{G}$ to be the group of our ANCO RPKE, this will likely allow a relatively small circuit for $\mathcal{F}_{\textsc{Service}}$, as randomisation in RPKE can be done using "native" operations in [DPSZ12]. Developing a concretely efficient circuit for $\mathcal{F}_{\textsc{Service}}$ is future work.

Using MPC to implement $\mathcal{F}_{\textsc{Service}}$ asks the question why we do not just let the MPC handle everything. The reason is that then everything would also be broken if the MPC is corrupted. With the current solution only anonymity and conciseness would be broken. Note that if $t = n$ servers are corrupted then the MPC servers will know L and therefore anonymity is completely broken. The protocol will, however, still be correct and live. It is correct because of public verifiability. For liveness, the seeming problem is that with $t = n$ any server can make $\Pi_{\textsc{Service}}$ deadlock. However, the service is not needed for posting c in the root of the tree, only for routing the coin. If $\Pi_{\textsc{Service}}$ deadlocks, the coin c would just keep sitting in the root of the tree, or where it ended up before the deadlock. This means that the path sent to the shop in step$^{(10)}$ in Fig. 1 would contain up to M coins, where M is the total number of coins ever posted. Therefore the communication complexity has a factor $O(\log M)$ replaced by a factor $O(M)$. However, the shop can still collect the coin. Note that if the service deadlocked before posting L_S, then the label L is not defined. In this case the shop asks for a random L. This works as c would be found in the root in this case.

With a centralized service ($n = 1$), an even simpler approach can be taken. Whenever the service replaces coins $c_1, \ldots, c_\ell$ by rerandomized coins $d_1, \ldots, d_\ell$ during operation of the SOROM it gives a ZK proof that there *exist* a permutation π and $\rho_1, \ldots, \rho_\ell$ such that

$$(d_1, \ldots, d_\ell) = (\mathsf{Ran}(c_{\pi(1)}, \rho_{\pi(1)}), \ldots, \mathsf{Ran}(c_{\pi(\ell)}, \rho_{\pi(\ell)})) ,$$

where Ran is our rerandomization of ciphertexts: $\mathsf{Ran} : \mathcal{C} \times \mathcal{R} \to \mathcal{C}$, where $\mathcal{C} = \mathbb{G}^4$ and $\mathcal{R} = \mathbb{Z}_q^2$ and $\mathsf{Ran}((A, B, C, D), (\rho', \sigma')) = (A^{\rho'}, B^{\rho'}, A^{\sigma'}C, B^{\sigma'}D)$. If there exist ρ' and σ' such that $(A', B', C', D') = (A^{\rho'}, B^{\rho'}, A^{\sigma'}C, B^{\sigma'}D)$, then (A', B', C', D') encrypts the same tid as (A, B, C, D) and decrypts under the same secret key as $\mathrm{DL}_{A'} B' = \mathrm{DL}_A B$. So, if the ZK proof verifies then the set of tid's are preserved and can be collected by the same shops, even if the service otherwise completely deviates from the protocol. Thus, each shop can still find its coin by worst-case downloading M coins when it does not find its coin on the path. To prove existence of π and $\rho_1, \ldots, \rho_\ell$ we can directly apply the ZK proof of a shuffle in [Wik09]. As discussed on page 409 in [Wik09] the proof works for any homomorphic rerandomization map $\phi : \mathcal{C} \times \mathcal{R} \to \mathcal{C}$ between groups $\mathcal{C}$ and $\mathcal{R}$, and clearly $\phi = \mathsf{Ran}$ is a group homomorphism.

Modelling and Analysis. We now discuss how we define and prove security. By anonymity we mean that after a sequence of payments and collections, an attacker gets no knowledge beyond some unavoidable leakage. As an example of unavoidable knowledge, imagine S collects a payment at time t. Then the coin was necessarily created at some earlier time $t' \leq t$. As another example, consider a shop that keeps querying the ledger to check whether it received a payment. Necessarily the shop will learn when a payment was initiated. Besides this kind of unavoidable knowledge, the adversary should learn nothing. This should hold, even if they are given the tid's of all payments in the system, to ensure that we do not rely on the secrecy of the tid's in the security or anonymity of the system.

Defining anonymity using game-based definitions can be subtle and error prone, so we have opted for a simulation-based definition. We require that the view of a run of the system can be simulated given only the unavoidable knowledge. Since payment systems are designed to be potentially used in other contexts and may possibly interact with many other systems, it is natural to require general and concurrent composability. We thus define security in the UC framework [Can01] by giving an ideal functionality $\mathcal{F}_{\mathsf{AnonPay}}$, modelling an ideal payment system only leaking unavoidable information to the simulator. Then we say that Π_{AnonPay} is a secure payment system, if it UC-securely realizes $\mathcal{F}_{\mathsf{AnonPay}}$.

Consider the ideal functionality $\mathcal{F}_{\mathsf{AnonPay}}$ in Fig. 2. It can interact with several users and several shops. Parties can have both roles, but for our explanation it is easier to think of users and shops as separate entities. We illustrate the ideal functionality with one user and one shop, but there can be any number of them. When $\mathcal{F}_{\mathsf{AnonPay}}$ gets a command from U to pay S, then the ideal functionality will inform the adversary/simulator[7] that U initiated a payment but not to whom. This models the fact that we allow for leaking, say via traffic analysis, that U is doing some payment, but we do not allow leaking any information about who is being payed or what the amount is. We let the adversary/simulator decide when the payment is completed. When this happens U is informed and its account is deducted a monetary units. We let the adversary decide when events like payments happened to avoid explicitly modeling time. We do not explicitly consider any liveness guarantees and we believe it is better to analyze them separately for a concrete implementation of an anonymous payment system. All we require from our protocols at the present level of abstraction, is that they are non-trivial in the sense of [Can01], i.e., if all messages are delivered then the protocol produces outputs. Implementing $\mathcal{F}_{\mathsf{AnonPay}}$ then guarantees that these are the right outputs and that privacy is maintained.

When a payment is created then $\mathcal{F}_{\mathsf{AnonPay}}$ samples a random tid with a distribution independent of U and S, i.e., the tids of all payments are sampled from the same distribution and hence they leak nothing about the payment they are associated to. Then tid is output to U, but *not* the simulator/adversary. This means that when proving security, the simulator needs to simulate without knowing tid towards an environment which does know tid. This might look draconian, but is needed to ensure that tid can be used as desired in any context without hurting security of the system. Later, the simulator/adversary can inform $\mathcal{F}_{\mathsf{AnonPay}}$ that the payment has become observable, at which time S is informed about the transaction identifier and the amount. In some implementations, it might be the case that the shop can observe that a payment happened, before it can actually collect it. We have therefore introduce another state called *collectable*. Again the adversary decides when a payment becomes collectable. The simulator/adversary is not informed about the identity

[7] When proving security of a protocol Π_{AnonPay} relative to $\mathcal{F}_{\mathsf{AnonPay}}$, then $\mathcal{F}_{\mathsf{AnonPay}}$ interacts with the simulator. When $\mathcal{F}_{\mathsf{AnonPay}}$ is being used as an ideal functionality in a hybrid world it interacts with the adversary.

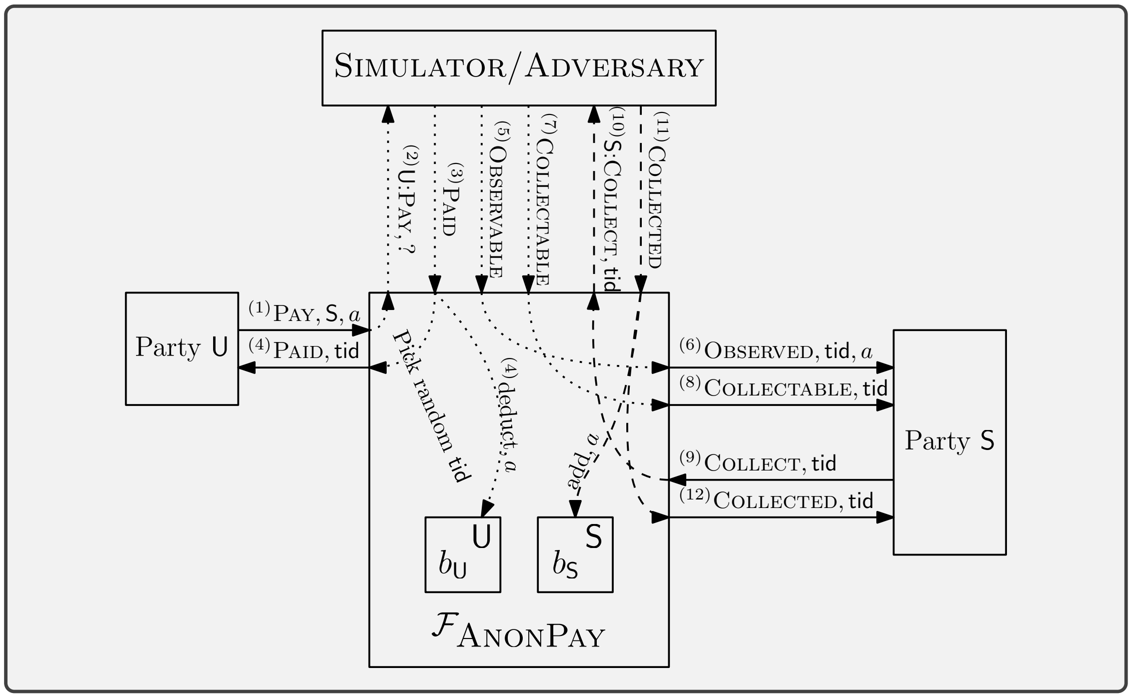

Fig. 2. A sketch of the ideal functionality $\mathcal{F}_{\mathsf{AnonPay}}$ for anonymous payment. When strong anonymity is modelled then the ideal functionality does not leak tid during collection but only "S: COLLECT, ?".

of S when making these decision. They are given a handle on the payment (U : PAY, ?) and can use this handle to say when the payment should become observable and collectable.[8] Once a payment is collectable the shop might collect it. This will leak tid to the simulator/adversary to model the fact that the protocol is allowed to leak tid at this point. Note that this does not violate anonymity as tid is random and independent of U and tid has so far not been leaked. From the point of view of the simulator/adversary tid can originate from any previous payment.[9,10] Again, the simulator/adversary decides when the collection completes and at this point the account of the shop is incremented. The non-triviality of $\mathit{\Pi}_{\text{ANONPAY}}$ will guarantee that once a payment is observable, it will always become collectable and once it is collectable any attempt to collect it will succeed.

Note that our notion of anonymity is a relative one. We prove that the system leaks no more information than the times at which payments are created and collected. To get a feeling for the anonymity provided by this relative notion, consider a setting with n honest users $\mathsf{U}_1, \ldots, \mathsf{U}_n$ and n honest shops $\mathsf{S}_1, \ldots, \mathsf{S}_n$. First, all users pay one unique shop in some arbitrary order. Then each

[8] This captures the fact that the timing of when a payment comes observable and collectable cannot depend on the identity of the shop, which could otherwise have been a covert leakage channel. This is an example of the simulation-based definition automatically capturing aspects one might not have explicitly thought about in game-based definitions.

[9] Again, it might look odd that we gave tid to the environment, but the crucial point is that we did not give it to the simulator. Giving it to the environment only gives stronger security: the simulator must simulate without tid and must fool even an environment who knows tid.

[10] Leaking tid models weak anonymity. We can model strong anonymity by not leaking tid. Then even the user cannot see when the shop collects its coin.

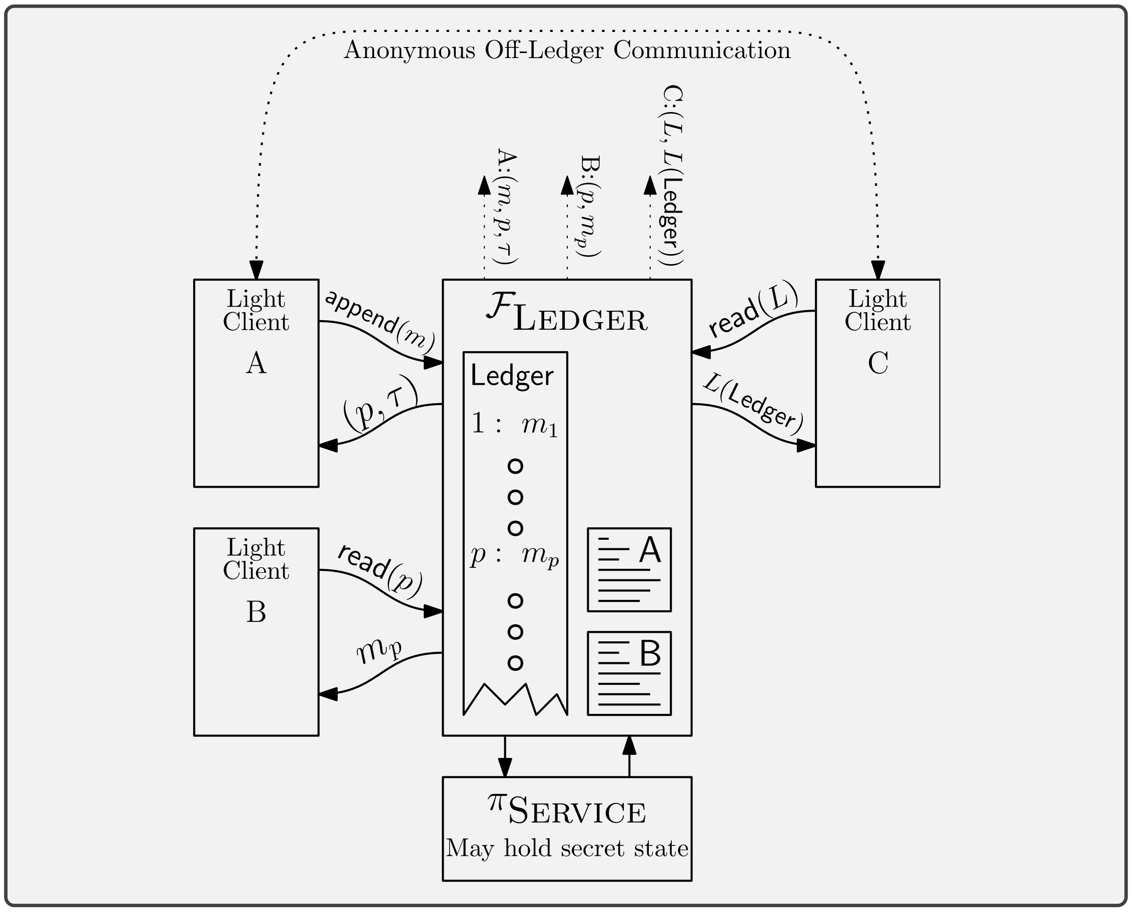

Fig. 3. Sketch of our UC model of a ledger accessed via full nodes, anonymous off-ledger communication, and a service protocol updating the ledger. When writing m the client learns the position p and gets a proof τ that m is on the ledger. The same is leaked to the adversary (upward arrows). When reading a position p or using a general read function L it is leaked who read what to the adversary.

shop collects their payment in some order. What the simulator/adversary will see is

$$(\mathsf{U}_1\colon \text{PAY}, ?), \ldots, (\mathsf{U}_n\colon \text{PAY}, ?), (\mathsf{S}_1\colon \text{COLLECT}, \mathsf{tid}_1), \ldots, (\mathsf{S}_n\colon \text{COLLECT}, \mathsf{tid}_n) \ ,$$

where the tid_i are uniform and independently distributed. This leaks nothing about who paid whom. If, however, it is known that a shop always collects payments right after being paid, then we know they U_i paid S_i. This means that, externally to the system, some measures need to be taken to mitigate traffic analysis. The payment system itself, however, is compatible with any way to mitigate the traffic analysis, thereby breaking up the system into very different problems, which can be handled using different tools.

We then address how we model the protocol. We will model the ledger using an ideal functionality, see Fig. 3. Instead of explicitly modeling both full nodes and light nodes, we have absorbed the full nodes into the ideal functionality. All parties using the ideal functionality are consider light clients. When a party performs a given operation, $\mathcal{F}_{\text{LEDGER}}$ leaks the identity of the light client and the information the full node would have learned in the real-world setting to the adversary. This models the worst case, where there is only a single full node, which is used by all light clients in the anonymous payment scheme. The anonymous off-ledger communication is modelled using a separate ideal functionality called $\mathcal{F}_{\mathrm{AAT}}$. Finally there might be a "service" protocol which helps update the ledger via the same interface as light clients. We then prove our protocol secure by proving that it implements $\mathcal{F}_{\mathsf{AnonPay}}$ in the hybrid model with $\mathcal{F}_{\textsc{Ledger}}$ and $\mathcal{F}_{\mathrm{AAT}}$. We will give a sketch of the proof after presenting details of our tools.

Paper Outline The details follow the above technical overview closely. In Section 2 we give technical preliminaries. We define and construct an anonymous coin friendly encryption scheme in Section 3, we define and construct compressible randomness beacons in Section 4, and we define and construct SOROMs in Section 5. In Sections 6 and 7 we then give the UC model of $\mathcal{F}_{\mathsf{AnonPay}}$ and $\mathcal{F}_{\textsc{Ledger}}$. In Section 8 we give pseudocode of the OCash protocol with weak anonymity, and in Section 9 we give the UC proof of security. In Section 10 we discuss how to add strong anonymity. In Section 11 we give the details of some of the proof systems that we use. Finally in Section 12 we give a slightly generalised proof of the security of the Dodis-Yampolskiy VRF which we use for strong anonymity.

2 Preliminaries

We use λ to denote the security parameter. When we work with lists, for instance a list Ledger to represent the ledger, we index from 1. We use $\mathsf{Ledger}[k]$ to denote position k and use $\mathsf{Ledger}[k] = \perp$ to denote that it is not the case that $1 \leq k \leq |\mathsf{Ledger}|$.

We prove security in the UC framework [Can20]. We assume the reader is familiar with the UC framework. When we specify ideal functionalities all inputs will start with a command name, $\textsc{CmdName}$. A canonical implementation of a command on an ideal functionality will be off the form. "On input $(\textsc{CmdName}, x)$ from P do the following ...." Such a command later gives an output y by outputting $(\textsc{CmdName}, x, y)$ to P. We output the command name and input x again to link $(\textsc{CmdName}, x, y)$ uniquely to $(\textsc{CmdName}, x)$. This allows us to use the following short hand notion for parties interacting with ideal functionalities: $\mathcal{F}.\textsc{CmdName}(x) \to y$. It expands to mean "Input $(\textsc{CmdName}, x)$ to $\mathcal{F}$, wait for $\mathcal{F}$ to return the first value of the form $(\textsc{CmdName}, x, z)$, assign z to y and then proceed."

We use $\Pr[F|E]$ to denote conditional probability, i.e., the probability of event F given event E occured. When A is an algorithm or process and E an event defined in that process we use $\Pr[A : E]$ and $\Pr[A\|E]$ to denote the probability that E occurs when executing experiment A.

2.1 Commitment Scheme

Definition 1 (Commitment Scheme). *A commitment scheme is a tuple of PPT algorithms* $\mathsf{Com} = (\mathsf{Gen}, \mathsf{Commit})$*, which are defined as follows:*

$\mathsf{ck} \leftarrow \mathsf{Gen}(1^\lambda)$**:** *The key generation algorithm takes security parameter λ as input and outputs commitment key* ck*.*

$\mathsf{com} \leftarrow \mathsf{Commit}(\mathsf{ck}, m)$**:** *The randomized commitment algorithm takes commitment key* ck *and message m as input and outputs commitment* com*.*

Definition 2 (Perfect Hiding). *We say that* $\mathsf{Com} = (\mathsf{Gen}, \mathsf{Commit})$ *is a perfectly hiding, if for all $\lambda \in \mathbb{N}$, all correctly generated* $\mathsf{ck} \leftarrow \mathsf{Gen}(1^\lambda)$ *and all messages m_0 and m_1 of the same length, it holds that* $\mathsf{Commit}(\mathsf{ck}, m_0)$ *and* $\mathsf{Commit}(\mathsf{ck}, m_1)$ *have the same distribution.*

Definition 3 (Computational Binding). *We say that* $\mathsf{Com} = (\mathsf{Gen}, \mathsf{Commit})$ *is computationally hiding, if for all* $\lambda \in \mathbb{N}$ *and all PPT adversaries* $\mathcal{A}$*, it holds that*

$$\Pr\left[\begin{array}{r} \mathsf{ck} \leftarrow \mathsf{Gen}(1^\lambda) \\ (m_0, m_1, \rho_0, \rho_1) \leftarrow \mathcal{A}(\mathsf{ck}) \end{array} \middle\| \begin{array}{r} \mathsf{Commit}(\mathsf{ck}, m_0; \rho_0) = \mathsf{Commit}(\mathsf{ck}, m_1; \rho_1) \\ \wedge\ m_0 \neq m_1 \end{array}\right] \leq \mathsf{negl}(\lambda).$$

Construction. Our commitment scheme for constructing transaction identifiers and accounts is a Pedersen commitment [Ped92] in a group $\mathbb{G}$ of prime order q where the discrete logarithm problem is hard. Specifically we assume that five uniformly random, independent generators $\mathsf{ck} = (g_0, g_1, g_2, g_3, g_4)$ have been chosen. We assume that q has been chosen large enough that account names, nonces and amounts can be represented bijectively in $\mathbb{Z}_q$. To commit to $(\mathsf{A}, \mathsf{B}, \mathsf{nonce}, a) \in \mathbb{Z}_q$ using randomness $s \in \mathbb{Z}_q$ we compute

$$\mathsf{tid} = \mathsf{Commit}_{\mathsf{ck}}(\mathsf{A}, \mathsf{B}, \mathsf{nonce}, a; s) = g_0^s g_1^{\mathsf{A}} g_2^{\mathsf{B}} g_3^{\mathsf{nonce}} g_4^a \ .$$

This scheme is perfectly hiding and is computationally binding is the DL problem is hard in $\mathbb{G}$ [Ped92]. When using the commitment scheme to commit to account balances we let $\mathsf{Commit}_{\mathsf{ck}}(a) = \mathsf{Commit}_{\mathsf{ck}}(0, 0, 0, a; s) = g_0^s g_4^a$.

2.2 Symmetric and Public Key Encryption

A symmetric-key encryption scheme is a tuple of PPT algorithms $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$. We use the notion of IND-P2-C2 from [KY00] and will just denote it as IND-CCA. We also use an INC-CCA secure public key encryption scheme $\mathsf{PKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ see, e.g., [BDPR98].

2.3 Σ-Protocols

We will make use of several zero-knowledge proofs of knowledge and membership in our protocols. They will all be based on Σ-protocols [Cra97], which are three-move proof systems.

Definition 4 (Σ-Protocols). *A Σ-protocol with challenge space $\mathcal{E}$ for a relation $\mathcal{R} \subseteq \{0,1\}^* \times \{0,1\}^*$ is a tuple of PPT algorithms (A, Z, V), which are defined as follows:*

$a \leftarrow A(x, w; \rho)$**:** *The algorithm takes statement x, witness w with $(x, w) \in \mathcal{R}$ and random coins ρ as input and generates the prover's first round message a.*

$z \leftarrow Z(x, w, e, \rho)$**:** *The algorithm takes statement x, witness w with $(x, w) \in \mathcal{R}$, challenge $e \in \mathcal{E}$, and auxiliary input ρ as input and generates the prover's third round message z.*

$b \leftarrow V(x, a, e, z)$**:** *The verification algorithm takes statement x, prover's messages (a, z), and challenge $e \in \mathcal{E}$ as input and outputs a bit b.*

that are defined by tuples $(\mathcal{R}, A, \mathcal{E}, Z, V, \mathsf{Ext}, \mathsf{Sim}, \mathcal{T})$, where $\mathcal{R} \subset \{0,1\}^* \times \{0,1\}^*$ is a poly-time binary relation, $A, Z, V, \mathsf{Ext}, \mathsf{Sim}$ are poly-time algorithms, $\mathcal{E}$ is the finite challenge space, and $\mathcal{T}$ is a unary predicate used to recognize trapdoors when defining strong special soundness as explained below.

Σ-protocols are expected to be complete, special sound, and honest-verifier zero-knowledge as defined below.

Definition 5 (Completeness). *A Σ-protocol (A, Z, V) with challenge space $\mathcal{E}$ for relation $\mathcal{R}$ is said to be complete, if for any $(x, w) \in \mathcal{R}$, it holds that*

$$\Pr\left[\begin{array}{r} a \leftarrow \mathsf{A}(x, w; \rho) \\ e \leftarrow \mathcal{E} \\ z \leftarrow Z(x, w, e, \rho) \end{array} \middle\| V(x, a, e, z) = \top \right] = 1,$$

where the probability is taken over the random coins of all involved algorithms.

Definition 6 (Special Soundness). *A Σ-protocol (A, Z, V) with challenge space $\mathcal{E}$ for relation $\mathcal{R}$ is said to be special sound, if there exists a PPT algorithm* Ext*, such that for any (x, a, e, z, e', z') with e $\neq$ e$'$, it holds that*

$$V(x, a, e, z) = \top \wedge V(x, a, e', z') = \top \implies (x, \mathsf{Ext}(x, a, e, z, e', z')) \in \mathcal{R}.$$

Definition 7 (Honest-Verifier Zero-Knowledge). *A Σ-protocol (A, Z, V) with challenge space $\mathcal{E}$ for relation $\mathcal{R}$ is said to be honest-verifier zero-knowledge, if there exists a PPT algorithm* Sim*, such that for any $(x, w) \in \mathcal{R}$, it holds that*

$$\left\{\begin{array}{r} a \leftarrow \mathsf{A}(x, w; \rho) \\ e \leftarrow \mathcal{E} \\ z \leftarrow Z(x, w, e, \rho) \\ \mathsf{return}\ (x, a, e, z) \end{array}\right\} \equiv \left\{\begin{array}{r} e \leftarrow \mathcal{E} \\ (a, z) \leftarrow \mathsf{Sim}(x, e) \\ \mathsf{return}\ (x, a, e, z) \end{array}\right\}.$$

We will also require a notion of strong special soundness, as introduced by Kondi and Shelat [KS22], which asks for extraction to work even in the case where $e' = e$, but $z' \neq z$. Most Σ-protocols have this stronger property or can be massaged into having it. If extraction fails, the extractor is allowed to instead recover a "system parameter trapdoor". We define PPT algorithm $\mathcal{T}$, which recognizes system parameter trapdoors, i.e., we call t a system parameter trapdoor if and only if $\mathcal{T}(t) = \top$.

Definition 8 (Strong Special Soundness). *A Σ-protocol (A, Z, V) with challenge space $\mathcal{E}$ for relation $\mathcal{R}$ is said to be strong special sound with respect to trapdoor predicate $\mathcal{T}$, if there exists a PPT algorithm* Ext*, such that for any (x, a, e, z, e', z') with $z \neq z'$, it holds that*

$$\begin{aligned} &V(x, a, e, z) = \top \wedge V(x, a, e', z') = \top \\ \implies &(x, \mathsf{Ext}(x, a, e, z, e', z')) \in \mathcal{R} \vee \mathcal{T}(\mathsf{Ext}(x, a, e, z, e', z')) = 1. \end{aligned}$$

The idea is of course that when used in a protocol, it should be computationally hard to obtain a system parameter trapdoor. In that case, the extractor will return a witness for x except with negligible probability.

2.4 Simulation-Sound NIZK Arguments

In our constructions we will need universally composable zero-knowledge proofs of membership (UC ZKM). For a PPT relation $\mathcal{R}$, the corresponding language is defined as

$$\mathcal{L}_{\mathcal{R}} = \{x \mid \exists w\, (x, w) \in \mathcal{R}\}.$$

A proof of membership for x is a proof that $x \in \mathcal{L}_\mathcal{R}$.[11] Both Camenisch, Krenn, and Shoup [CKS11] as well as Nielsen [Nie17] present (equivalent) definitions of UC ZKMs. An ideal functionality $\mathcal{F}_{\text{ZKM}}$, which is parameterized by a PPT relation $\mathcal{R}$, for ZKMs is specified. An honest prover P can prove statements x in front of a verifier V by providing (x, w) as input to the functionality. If $(x, w) \in \mathcal{R}$, then the functionality outputs $(x, \top)$ to V, signaling that x is in the language. When a *corrupt* prover P wants to prove a statement x in front of verifier V, then the simulator/adversary only provides x to the ideal functionality, which will just output $(x, \top)$ to V. Importantly, the corrupt prover does not have to present a witness w. At this point the two models of Camenisch, Krenn, and Shoup [CKS11] and Nielsen [Nie17] differ slightly.

In [CKS11] the ideal functionality is called "gullibly" as it accepts x without a witness, so it could be the case that $x \notin \mathcal{L}_\mathcal{R}$. However, UC ZKM of a protocol is then defined as implementing $\mathcal{F}_{\text{ZKM}}$ using a simulator which only inputs $x \notin \mathcal{L}_\mathcal{R}$ with negligible probability. This means that if the real protocol accepts $x \notin \mathcal{L}_\mathcal{R}$ with non-negligible probability then the simulator cannot simulate, which guarantees soundness.

In [Nie17], when the simulator/adversary inputs x, the ideal functionality will check that $x \in \mathcal{L}_\mathcal{R}$. If this is not the case, it outputs Fail to the environment. Call this ideal functionality $\mathcal{F}^{\text{Fail}}_{\text{ZKM}}$. A protocol is called a UC ZKM, if it implements $\mathcal{F}^{\text{Fail}}_{\text{ZKM}}$, and now there is no explicit restriction on the simulator.[12] However, since Fail is never output in the real protocol, it follows that the simulation fails, if it happens that $x \notin \mathcal{L}_\mathcal{R}$ with non-negligible probability. This means that the simulator only inputs $x \notin \mathcal{L}_\mathcal{R}$ with negligible probability. Therefore the simulator is an admissible simulator for the definition in [CKS11]. The definitions are therefore equivalent.

Nielsen [Nie17] furthermore shows that a protocol is UC ZKM, if it is complete, zero-knowledge and weak simulation-sound against a PPT adversary running multiple sessions as defined below. Below we give the definitions specifically for NIZK proofs.

Definition 9 (Non-Interactive Zero-Knowledge). *A NIZK is a tuple of PPT algorithms* $\mathsf{NIZK} = (\mathsf{Gen}, \mathsf{Prv}, \mathsf{Ver})$*, which are defined as follow:*

$\mathsf{crs} \leftarrow \mathsf{Gen}(1^\lambda)$**:** *The public parameter generation algorithm takes security parameter λ as input and returns common reference string* crs*.*

$\pi \leftarrow \mathsf{Prv}(\mathsf{crs}, x, w)$**:** *The prover algorithm takes common reference string* crs*, statement x, and witness w as input and returns a proof π.*

$b \leftarrow \mathsf{Ver}(\mathsf{crs}, x, \pi)$**:** *The verification algorithm takes common reference string* crs*, statement x, and proof π as input and returns bit b.*

Definition 10 (Completeness). *A non-interactive zero-knowledge proof* $\mathsf{NIZK} = (\mathsf{Gen}, \mathsf{Prv}, \mathsf{Ver})$ *for relation $\mathcal{R}$ is said to be complete, if for any PPT adversary $\mathcal{A}$, it holds that*

$$\Pr\left[\begin{array}{r} \mathsf{crs} \leftarrow \mathsf{Gen}(1^\lambda) \\ (x, w) \leftarrow \mathcal{A}(\mathsf{crs}) \\ \pi \leftarrow \mathsf{Prv}(\mathsf{crs}, x, w) \end{array} \middle\| \begin{array}{l} (x, w) \in \mathcal{R} \land \mathsf{Ver}(\mathsf{crs}, x, \pi) \neq \top \\ \\ \\ \end{array}\right] \leq \mathsf{negl}(\lambda),$$

where the probability is taken over the random coins of the adversary and all involved algorithms.

[11] We stress, however, that the verifier is not guaranteed that the prover *knows* a witness w, such that $(x, w) \in \mathcal{R}$.

[12] Note that $\mathcal{F}^{\text{Fail}}_{\text{ZKM}}$, is not PPT as the check $x \in \mathcal{L}_\mathcal{R}$ is not necessarily poly-time. However, it is shown in [Nie17] that composition still works: in any PPT protocol using $\mathcal{F}^{\text{Fail}}_{\text{ZKM}}$ as a blackbox we can securely replace $\mathcal{F}^{\text{Fail}}_{\text{ZKM}}$ with a UC ZKM. After this the overall protocol is PPT.

Definition 11 (Zero-Knowledge). *A non-interactive zero-knowledge proof* $\mathsf{NIZK} = (\mathsf{Gen}, \mathsf{Prv}, \mathsf{Ver})$ *for relation $\mathcal{R}$ is said to be zero-knowledge, if there exists a pair of PPT algorithms* $(\mathsf{SimGen}, \mathsf{Sim})$*, such that for any PPT adversary $\mathcal{A}$, it holds that*

$$\Pr\left[\begin{array}{r}(\mathsf{crs}, \mathsf{tSim}) \leftarrow \mathsf{SimGen}(1^\lambda)\\ b \leftarrow \{0,1\}\\ g \leftarrow \mathcal{A}^{\mathcal{O}^{\textsc{SimOrReal}}_{\mathsf{tSim},b}(\cdot,\cdot)}(\mathsf{crs})\end{array}\middle\|\, g = b\right] \leq \frac{1}{2} + \mathsf{negl}(\lambda),$$

where the probability is taken over the random coins of the adversary and all involved algorithms and where the oracle $\mathcal{O}^{\textsc{SimOrReal}}_{\mathsf{tSim},b}(\cdot,\cdot)$ takes as input pairs (x, w). If $(x, w) \notin \mathcal{R}$, then the oracle returns $\perp$. Otherwise, if $b = 0$, it returns $\pi \leftarrow \mathsf{Sim}(\mathsf{tSim}, x)$ and if $b = 1$, it returns $\pi \leftarrow \mathsf{Prv}(\mathsf{crs}, x, w)$.

Definition 12 (Weak Simulation-Soundness). *A non-interactive zero-knowledge proof* $\mathsf{NIZK} = (\mathsf{Gen}, \mathsf{Prv}, \mathsf{Ver})$ *for relation $\mathcal{R}$ is said to be weak simulation-sound, if there exists a pair of PPT algorithms* $(\mathsf{SimGen}, \mathsf{Sim})$*, such that for any PPT adversary $\mathcal{A}$, it holds that*

$$\Pr\left[\begin{array}{r}(\mathsf{crs}, \mathsf{tSim}) \leftarrow \mathsf{SimGen}(1^\lambda)\\ (x, \pi) \leftarrow \mathcal{A}^{\mathcal{O}^{\textsc{Sim}}_{\mathsf{tSim}}(\cdot,\cdot)}(\mathsf{crs})\end{array}\middle\|\, x \notin \mathcal{L}_{\mathcal{R}} \wedge \mathsf{Ver}(\mathsf{crs}, x, \pi) = \top\right] \leq \mathsf{negl}(\lambda),$$

where the probability is taken over the random coins of the adversary and all involved algorithms and where the oracle $\mathcal{O}^{\textsc{Sim}}_{\mathsf{tSim}}(\cdot,\cdot)$ takes as input pairs (x, w). If $(x, w) \notin \mathcal{R}$, then the oracle outputs $\perp$ and otherwise it computes $\pi \leftarrow \mathsf{Sim}(\mathsf{tSim}, x)$ and returns π.

Definition 13 (UC NIZK). *A non-interactive zero-knowledge proof* $\mathsf{NIZK} = (\mathsf{Gen}, \mathsf{Prv}, \mathsf{Ver})$ *for relation $\mathcal{R}$ with an associated pair of simulation algorithms* $(\mathsf{SimGen}, \mathsf{Sim})$ *is said to be UC NIZK proof of membership, if it simultaneously satisfies completeness, zero-knowledge, and weak simulation-soundness as defined in Definitions 10, 11, and 12 respectively.*

Lindell [Lin15] presents a generic compiler that transforms Σ-protocols into NIZK proofs in the CRS model in the presence of a non-programmable random oracle. The transformation is proven to have the three properties defined above and therefore it actually produces a UC NIZK. Since the random oracle is only used for soundness, we can apply the Fiat-Shamir heuristic and replace it by a real life hash function and get a UC NIZK in the random oracle devoid model. We use this to compile Σ-protocols into UC NIZKs in the CRS model without a random oracle.

We will not go into the details of how Lindell's transformation works, but we want to highlight one of its main ideas, which will be relevant to our work. To show that their UC NIZK is sound, Lindell argues that one could hypothetically extract a witness from an adversarially generated, but accepting, proof by rewinding an entire UC execution, including the environment itself. Now clearly one is usually not allowed to rewind the entire UC execution, but the fact that one can in principle extract a witness means that it must *exist*, which is enough to show soundness. We refer to this proof strategy as *global extraction*.

Attema, Fehr, Klooß [AFK22] show that the Fiat-Shamir transformation applied to multi-round Σ-protocols results in proofs of knowledge. Since a witness can be extracted from a proof of knowledge, it must logically exist. This means that using Lindell's approach in combination with global extraction, one can show that multi-round Σ-protocols can be transformed into UC NIZKs of *membership* as well. Note, however, that since extracting the witness can only happen using extraction, the resulting proof system is not a UC proof of *knowledge*, as the UC simulator cannot rewind its environment.

2.5 Simulation-Extractable NIZK Arguments

In some cases we will not just need non-interactive proofs of membership, but rather proofs of *knowledge.* Turning Σ-protocols into non-interactive proofs of knowledge via the Fiat-Shamir transformation [FS87] requires rewinding for extracting the witness and for this reason this transformation does not give us proofs of knowledge in the UC setting. Fischlin [Fis05] presents an alternative transformation, which has an online extractor, thus also works in the UC setting, but requires the Σ-protocols to have unique responses z. Unfortunately, this property is not satisfied by some of the protocols we would like to use in our work.

Kondi and Shelat [KS22] present a randomized version of Fischlin's transformation, which allows for online extraction and only requires the starting Σ-protocol to satisfy strong special soundness. Lysyanskaya and Rosenbloom [LR22] show that Kondi and Shelat's transformation can transform Σ-protocols into non-interactive proofs of knowledge in the UC model with a so-called restricted, programmable, observable, random oracle $\mathcal{G}_{\text{RORO}}$. We point to the work of Lysyanskaya and Rosenbloom for a formalization of $\mathcal{G}_{\text{RORO}}$. For our purposes it suffices to know that $\mathcal{G}_{\text{RORO}}$ is one of the possible formalizations of the usual random oracle lifted to the UC model. The only technicality needed, is that the simulator is allowed to observe the queries made by the environment, but we do not allow the environment to observe queries made by the simulator, which would make it trivial for it to know that it is in the simulation. For more details on this, we refer the reader to the work of Lysyanskaya and Rosenbloom. In the descriptions below we will denote the random oracle by $\mathcal{O}$.

In our work, we will use a definition of GUC NIZK PoKs that is equivalent to the definition given by Lysyanskaya and Rosenbloom [LR22, Definition 11], but is more convenient to work with. We will state our definition here and defer the discussion of why the two definitions are equivalent to supplementary material 11.1.

Definition 14 (Non-Interactive Zero-Knowledge Proof of Knowledge). *A NIZK is a tuple of PPT algorithms* $\mathsf{NIZKPoK} = (\mathsf{Gen}, \mathsf{Prv}, \mathsf{Ver})$*, all with access to oracle* $\mathcal{O}$*, which are defined as follow:*

$\mathsf{crs} \leftarrow \mathsf{Gen}(1^\lambda)$**:** *The public parameter generation algorithm takes security parameter* λ *as input and returns common reference string* crs*.*

$\pi \leftarrow \mathsf{Prv}(\mathsf{crs}, x, w)$**:** *The prover algorithm takes common reference string* crs*, statement* x*, and witness* w *as input and returns a proof* π*.*

$b \leftarrow \mathsf{Ver}(\mathsf{crs}, x, \pi)$**:** *The verification algorithm takes common reference string* crs*, statement* x*, and proof* π *as input and returns bit* b*.*

In all of the following security definitions related to $\mathsf{NIZKPoK}$, we will assume that the extractor Ext and simulator Sim can observe all queries made to the $\mathcal{O}$ by the adversary but not vice versa.

Definition 15 (Completeness). *A non-interactive zero-knowledge proof of knowledge* $\mathsf{NIZKPoK} = (\mathsf{Gen}, \mathsf{Prv}, \mathsf{Ver})$ *for relation* $\mathcal{R}$ *is said to be complete, if for any PPT adversary* $\mathcal{A}$*, it holds that*

$$\Pr\left[\begin{array}{r} \mathsf{crs} \leftarrow \mathsf{Gen}(1^\lambda) \\ (x, w) \leftarrow \mathcal{A}^{\mathcal{O}(\cdot)}(\mathsf{crs}) \\ \pi \leftarrow \mathsf{Prv}(\mathsf{crs}, x, w) \end{array} \middle\| \, (x, w) \in \mathcal{R} \wedge \mathsf{Ver}(\mathsf{crs}, x, \pi) \neq \top \right] \leq \mathsf{negl}(\lambda),$$

Definition 16 (Zero-Knowledge). *A non-interactive zero-knowledge proof of knowledge* $\mathsf{NIZKPoK} = (\mathsf{Gen}, \mathsf{Prv}, \mathsf{Ver})$ *for relation* $\mathcal{R}$ *is said to be zero-knowledge, if there exists a pair of PPT algorithms* $(\mathsf{SimGen}, \mathsf{Sim}, \mathsf{Ext})$*, such that for any PPT adversary* $\mathcal{A}$*, it holds that*

$$\Pr\left[\begin{array}{r}(\mathsf{crs}, \mathsf{tSim}, \mathsf{tExt}) \leftarrow \mathsf{SimGen}(1^\lambda)\\ b \leftarrow \{0,1\}\\ g \leftarrow \mathcal{A}^{\mathcal{O}^{\textsc{SimOrReal}}_{\mathsf{tSim},b}(\cdot,\cdot),\mathcal{O}^{\textsc{Ext}}_{\mathsf{tExt}}(\cdot,\cdot),\mathcal{O}(\cdot)}(\mathsf{crs})\end{array}\middle\| \; g = b\right] \leq \frac{1}{2} + \mathsf{negl}(\lambda),$$

where the probability is taken over the random coins of the adversary and all involved algorithms and where the oracles are defined as follows:

$\pi \leftarrow \mathcal{O}^{\mathbf{SimOrReal}}_{\mathsf{tSim},b}(\cdot,\cdot)$**:** *The oracle takes pairs* (x, w) *as input and if* $(x, w) \notin \mathcal{R}$*, then it returns* $\perp$*. Otherwise, if* $b = 0$*, it returns* $\pi \leftarrow \mathsf{Sim}(\mathsf{tSim}, x)$ *and if* $b = 1$*, it returns* $\pi \leftarrow \mathsf{Prv}(\mathsf{crs}, x, w)$*. It adds* (x, π) *to* $\mathcal{Q}$*.*

$\top/\perp \leftarrow \mathcal{O}^{\mathbf{Ext}}_{\mathsf{tExt}}(x, \pi)$**:** *The oracle takes* (x, π) *as input and checks whether* $\mathsf{Ver}^{\mathcal{O}}(\mathsf{crs}, x, \pi) = \top$*. If not, it returns* $\perp$*. Otherwise, it proceeds as follows. If* $(x, \pi) \in \mathcal{Q}$*, it returns* $\top$ *or if* $(x, \pi) \notin \mathcal{Q}$*, it computes* $w \leftarrow \mathsf{Ext}^{\mathcal{O}}(\mathsf{tExt}, x, \pi)$*. If* $(x, w) \in \mathcal{R}$ *it returns* $\top$*, otherwise it returns* $\textsc{Fail}$*.*

Definition 17 (Weak Simulation-Extractability). *A non-interactive zero-knowledge proof of knowledge* $\mathsf{NIZKPoK} = (\mathsf{Gen}, \mathsf{Prv}, \mathsf{Ver})$ *for relation* $\mathcal{R}$ *is said to be weak simulation-extractable, if there exists a pair of PPT algorithms* $(\mathsf{SimGen}, \mathsf{Sim}, \mathsf{Ext})$*, such that for any PPT adversary* $\mathcal{A}$*, it holds that*

$$\Pr\left[\begin{array}{r}(\mathsf{crs}, \mathsf{tSim}, \mathsf{tExt}) \leftarrow \mathsf{SimGen}(1^\lambda)\\ \mathcal{A}^{\mathcal{O}^{\textsc{Sim}}_{\mathsf{tSim}}(\cdot,\cdot),\mathcal{O}^{\textsc{Ext}}_{\mathsf{tExt}}(\cdot,\cdot),\mathcal{O}(\cdot)}(\mathsf{crs})\end{array}\middle\| \; \mathcal{O}^{\textsc{Ext}}_{\mathsf{tExt}} \hookrightarrow \textsc{Fail}\right] = \mathsf{negl}(\lambda),$$

where the probability is taken over the random coins of the adversary and all involved algorithms and where the oracles are defined as follows:

$\pi \leftarrow \mathcal{O}^{\mathbf{Sim}}_{\mathsf{tSim}}(x, w)$**:** *The oracle takes pairs* (x, w) *as input and if* $(x, w) \notin \mathcal{R}$*, then it returns* $\perp$*. Otherwise, it computes* $\pi \leftarrow \mathsf{Sim}^{\mathcal{O}}(\mathsf{tSim}, x)$*, adds* (x, π) *to* $\mathcal{Q}$*, and returns* π*.*

$\top/\perp \leftarrow \mathcal{O}^{\mathbf{Ext}}_{\mathsf{tExt}}(x, \pi)$**:** *The oracle is as defined in Definition 16. We write* $\mathcal{O}^{\textsc{Ext}}_{\mathsf{tExt}} \hookrightarrow \textsc{Fail}$ *to denote the event that* $\mathcal{O}^{\textsc{Ext}}_{\mathsf{tExt}}$ *on some activation returned* $\textsc{Fail}$*.*

Definition 18 (GUC NIZK PoK). *A non-interactive zero-knowledge proof of knowledge* $\mathsf{NIZKPoK} = (\mathsf{Gen}, \mathsf{Prv}, \mathsf{Ver})$ *for relation* $\mathcal{R}$ *with an associated pair of simulation algorithms* $(\mathsf{SimGen}, \mathsf{Sim})$ *and extractor* Ext *is said to be UC NIZK PoK, if it simultaneously satisfies completeness, zero-knowledge, and weak simulation-extractability as defined in Definitions 15, 16, and 17 respectively.*

3 Anonymous Coin Friendly Encryption (ANCOs)

The first tool we need for building OCash is anonymous coin friendly encryption.

Definition 19 (Rerandomizable Public Key Encryption). *A rerandomizable public key encryption (RPKE) scheme is a tuple of PPT algorithms* $\mathsf{RPKE} = (\mathsf{Params}, \mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Ran})$*, which are defined as follows:*

$\mathsf{pp} \leftarrow \mathsf{Params}(1^\lambda)$**:** *The parameter generation algorithm takes security parameter* λ *as input and outputs public parameters* pp*.*

$(\mathsf{ek}, \mathsf{dk}) \leftarrow \mathsf{Gen}(\mathsf{pp})$**:** *The key generation algorithm takes public parameters* pp *as input and returns encryption key* ek *and corresponding decryption key* dk*.*

$\mathsf{ct} \leftarrow \mathsf{Enc}_{\mathsf{ek}}(m; \rho)$**:** *The encryption algorithm takes encryption key* ek*, message* $m \in \mathcal{M}$*, and randomizer* ρ *as input and returns ciphertext* $\mathsf{ct} \in \mathcal{C}$*.*

$m \leftarrow \mathsf{Dec}_{\mathsf{dk}}(\mathsf{ct})$**:** *The decryption algorithm takes decryption key* dk *and ciphertext* $\mathsf{ct} \in \mathcal{C}$ *as input and returns plaintext* $m \in \mathcal{M}$*.*

$\mathsf{ct}' \leftarrow \mathsf{Ran}(\mathsf{pp}, \mathsf{ct})$**:** *The rerandomization algorithm takes public parameter* pp *and ciphertext* ct *as input and returns ciphertext* ct'*.*

For an integer $k \in \mathbb{N}$, we write $\mathsf{Ran}^k(\mathsf{pp}, \mathsf{ct})$ to denote the k-fold application of the rerandomization procedure to ciphertext ct.

We consider encryption schemes in the presence of a relation $\mathcal{R}_{\text{ENC}}$ for statements $x = (\mathsf{pp}, \mathsf{ct}, m)$ and witnesses $w = \rho$. The pair (x, w) is in $\mathcal{R}_{\text{ENC}}$ if and only if there exists a key pair $(\mathsf{ek}, \mathsf{dk})$ in the support of $\mathsf{Gen}(\mathsf{pp})$, such that $\mathsf{ct} = \mathsf{Enc}_{\mathsf{ek}}(m; \rho)$. We require the relation to be *efficiently checkable*, when given only x and w, meaning that one can efficiently check whether a ciphertext is indeed a valid encryption *without knowing the public key.* This is crucial as anonymous payments will contain such an encryption for the receiver. We require our encryption scheme to satisfy several other properties stated in the following.

We require honestly generated ciphertexts to decrypt to the correct plaintext.

Definition 20 (Correctness). *We say an encryption scheme* $\mathsf{RPKE} = (\mathsf{Params}, \mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Ran})$ *is correct, if for all* $\lambda \in \mathbb{N}$*, all* $k \in \mathbb{N}$ *and all* $m \in \mathcal{M}$*:*

$$\Pr\left[\begin{array}{r} \mathsf{pp} \leftarrow \mathsf{Params}(1^\lambda) \\ (\mathsf{ek}, \mathsf{dk}) \leftarrow \mathsf{Gen}(\mathsf{pp}) \\ \mathsf{ct} \leftarrow \mathsf{Ran}^k(\mathsf{Enc}_{\mathsf{ek}}(m)) \end{array} \middle\| \ \mathsf{Dec}_{\mathsf{dk}}(\mathsf{ct}) = m \right] = 1,$$

where the probability is taken over the random coins of all involved algorithms.

The next property, called key-indistinguishability under rerandomization, requires that no adversary can determine under which key a given ciphertext was encrypted. Our notion is stronger than the classical notion of key-indistinguishability as originally formalized by Bellare *et al.* [BBDP01], since we allow the adversary to rerandomize the ciphertext an adversarially chosen amount of times.

Definition 21 (Key-Indistinguishability Under Rerandomization). *We say encryption scheme* $\mathsf{RPKE} = (\mathsf{Params}, \mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Ran})$ *is key-indistinguishable under rerandomization, if for any PPT adversary* $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ *and any* $\lambda \in \mathbb{N}$*:*

$$\Pr\left[\begin{array}{r} \mathsf{pp} \leftarrow \mathsf{Params}(1^\lambda) \\ (\mathsf{ek}_0, \cdot) \leftarrow \mathsf{Gen}(\mathsf{pp}); (\mathsf{ek}_1, \cdot) \leftarrow \mathsf{Gen}(\mathsf{pp}) \\ (m_0, m_1, k_0 \geq 1, k_1 \geq 1, \mathsf{st}) \leftarrow \mathcal{A}_1(\mathsf{pp}, \mathsf{ek}_0, \mathsf{ek}_1) \\ b \leftarrow \{0,1\} \\ \mathsf{ct}_b = \mathsf{Ran}^{k_b}(\mathsf{Enc}_{\mathsf{ek}_b}(m_b)) \\ b' \leftarrow \mathcal{A}_2(\mathsf{st}, \mathsf{ct}_b) \end{array} \middle\| \ b' = b \right] = \frac{1}{2} \pm \mathsf{negl}(\lambda),$$

where the probability is taken over the random coins of the involved algorithms and the adversary.

We require that any ciphertext can decrypt to at most one message, no matter which decryption key is used.

Definition 22 (Strong Message Binding). *We say an encryption scheme* $\mathsf{RPKE} = (\mathsf{Params}, \mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Ran})$ *with associated relation* $\mathcal{R}_{ENC}$ *is strongly message binding, if for any PPT adversary* $\mathcal{A}$ *and any* $\lambda \in \mathbb{N}$*:*

$$\Pr\left[\begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Params}(1^\lambda) \\ (c, m, w) \leftarrow \mathcal{A}(\mathsf{pp}) \end{array}\middle\| \begin{array}{l} ((\mathsf{pp}, \mathsf{ct}, m), w) \in \mathcal{R}_{ENC} \\ \qquad\qquad\qquad \Rightarrow \mathsf{Dec}.(\mathsf{ct}) \in \{m, \bot\} \end{array}\right] = 1 - \mathsf{negl}(\lambda) \ .$$

Definition 23 (Anonymous Coin Friendly Encryption Schemes). *We say a rerandomizable encryption scheme* $(\mathsf{Params}, \mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Ran})$ *is an anonymous coin friendly encryption scheme (ANCO), if it simultaneously satisfies correctness, key-indistinguishability under rerandomization, and strong message binding as defined in Definitions 20, 21 and 22.*

3.1 Constructing Anonymous Coin Friendly Encryption

We give our instantiation of a rerandomisable public key encryption scheme RPKE and prove that it is an ANCO. Let $\mathbb{G}$ be a group of prime order q where the DDH problem is hard. The public parameters will be of the form $g_0 \in \mathbb{G}$, where g_0 is a generator. A public key will be of the form $(g_0, h = g_0^x)$ where $x \in \mathbb{Z}_q$ is the secret key. A ciphertext will be of the form $\mathsf{ct} = (A, B, C, D)$ where $A \neq 1$ and $B = A^x$ for the unique secret key x for which the ciphertext is intended and $m = DC^{-x}$ is the message. Note that the part (A, B) of ciphertext uniquely fixes the secret key x and the part (C, D) is a normal ElGamal encryption for that secret key. The reason why we carry (A, B) along is to be able to rerandomise a ciphertext without knowing the public key: $(C', D') = (A^{\rho'}C, B^{\rho'}D)$. The part (A, B) will simply be a rerandomized version of the receiver's public key and will be rerandomized as $(A', B') = (A^\rho, B^\rho)$. To decrypt (A, B, C, D) using x we define the output to be $\bot$ if $B \neq A^x$. If $B = A^x$ the output is $m = DC^{-x}$. The scheme is summarised in Fig. 4. The ciphertext space is the set of all $\mathsf{ct} = (A, B, C, D) \in \mathbb{G}^4$ where $A \neq 1$. Note that this implies that A has order q and therefore the discrete logarithm $\mathrm{DL}_A : \mathbb{G} \rightarrow \mathbb{Z}_q$ is well-defined. We call $x = \mathrm{DL}_A B$ the secret key of the ciphertext. This defines the keyless decryption $\mathsf{RPKE.Dec}.(\mathsf{ct})$ as follows. Let $x = \mathrm{DL}_A B$ and then return $\mathsf{RPKE.Dec}_x(\mathsf{ct})$. Note that $\mathsf{RPKE.Dec}.(\mathsf{ct})$ is well defined but not poly-time.

Theorem 1. RPKE *in Fig. 4 is an ANCO (Definition 23) if DDH is hard in* $\mathbb{G}$.

Proof. We have to prove that (x, w) is in $\mathcal{R}_{\mathrm{ENC}}$ if and only if there exists a key pair $(\mathsf{ek}, \mathsf{dk})$ in the support of $\mathsf{Gen}(\mathsf{pp})$, such that $\mathsf{ct} = \mathsf{Enc}_{\mathsf{ek}}(m; w)$. This is easy to see, as $(A, B, C, D) = \mathsf{Enc}_{\mathsf{ek}}(m; (\rho, \rho'))$ implies that $(A, B) = (g_0^\rho, h^\rho)$, so $\mathsf{ek} = (A^{\rho^{-1}}, B^{\rho^{-1}})$.

It is clear that if ct is in the ciphertext space then $\mathsf{Ran}_{\mathsf{pp}}(\mathsf{ct})$ is also in the ciphertext space as $\rho \neq 0$ so $A^\rho \neq 1$. Finally, it follows that $\mathsf{Dec}.(\mathsf{Ran}(\mathsf{ct})) = \mathsf{Dec}.(\mathsf{ct})$ as $\mathrm{DL}_{A^\rho} B^\rho = \mathrm{DL}_A B$ and $B^{\rho'}D(A^{\rho'}C)^{-x} = DC^{-x}$ when $B = A^x$. From this it also follows that $\mathsf{Dec}_x(\mathsf{Ran}(\mathsf{ct})) = \mathsf{Dec}_x(\mathsf{ct})$, and we get correctness.

To show strong message binding we have to assume that $((\mathsf{pp}, \mathsf{ct}, m), w) \in \mathcal{R}_{\mathrm{ENC}}$ and show that $\mathsf{Dec}.(\mathsf{ct}) = m$. This is clear as $((\mathsf{pp}, \mathsf{ct}, m), w) \in \mathcal{R}_{\mathrm{ENC}}$ implies that $\mathsf{ct} = \mathsf{Enc}_{\mathsf{ek}}(m; (\rho, \rho'))$ so we can appeal to correctness.

Finally we have to prove that key-indistinguishability under rerandomization. We are given any PPT adversary $\mathcal{A}_1$ and run $(m_0, m_1, r_0 \geq 1, r_1 \geq 1, \mathsf{st}) \leftarrow \mathcal{A}_1(\mathsf{pp}, \mathsf{ek}_0, \mathsf{ek}_1)$ for uniformly random

```
Params(1^λ)
  1. Sample $(\mathbb{G}, g_0, q)$, where DDH is hard in the
     group $\mathbb{G}$ generated by $g_0$, and $\mathbb{G}$ has prime
     order $q$.
  2. Output $\mathsf{pp} = (g_0, q)$

Gen(pp)
  1. Input $\mathsf{pp} = (g_0, q)$
  2. $x \leftarrow \mathbb{Z}_q$
  3. $h = g_0^x$
  4. Output $(\mathsf{ek} = (g_0, h), \mathsf{dk} = x)$

Enc_ek(m)
  1. Input: $\mathsf{pp} = (g_0, q), \mathsf{ek} = (g, h), m$
  2. $\rho \leftarrow \mathbb{Z}_q^*$
  3. $\rho' \leftarrow \mathbb{Z}_q$
  4. Output $\mathsf{ct} = (g_0^{\rho}, h^{\rho}, g_0^{\rho'}, h^{\rho'} m)$

Dec_dk(ct)
  1. Input: $\mathsf{pp} = (g_0, q), \mathsf{dk} = x, \mathsf{ct} = (A, B, C, D)$
  2. If $B = A^x$ let $m = DC^{-x}$, otherwise let $m = \perp$.
  3. Output $m$

Ran(ct)
  1. Input: $\mathsf{pp} = (g_0, q), \mathsf{ct} = (A, B, C, D)$
  2. $\rho \leftarrow \mathbb{Z}_q^*$
  3. $\rho' \leftarrow \mathbb{Z}_q$
  4. Output $\mathsf{ct}' = (A^{\rho}, B^{\rho}, A^{\rho'} C, B^{\rho'} D)$

$\mathcal{R}_{\text{ENC}}(x = (\mathsf{pp}, \mathsf{ct}, m), w = (\rho, \rho'))$
  1. $(A, B, C, D) \leftarrow \mathsf{ct}$
  2. $(g, h) = \mathsf{ek} \leftarrow (A^{\rho^{-1}}, B^{\rho^{-1}})$
  3. Output $g \overset{?}{=} g_0 \wedge \mathsf{ct} \overset{?}{=} \mathsf{Enc}_{\mathsf{ek}}(m; (\rho, \rho'))$
```

Fig. 4. The ANCO RPKE

public keys ek_0 and ek_1. Then we compute $\mathsf{ct}_0 = \mathsf{Ran}^{r_0}(\mathsf{Enc}_{\mathsf{ek}_0}(m_0))$ and $\mathsf{ct}_1 = \mathsf{Ran}^{r_1}(\mathsf{Enc}_{\mathsf{ek}_1}(m_1))$ and have to argue that $\mathcal{A}_1$ cannot guess b given c_b. We have that the distributions of $\mathsf{ct}_0 = \mathsf{Enc}_{\mathsf{ek}_0}(m_0)$ and $\mathsf{ct}_1 = \mathsf{Enc}_{\mathsf{ek}_1}(m_1)$ are given by

$$\mathsf{ct}_0 = (g_0^{\rho_0}, h_0^{\rho_0}, g_0^{\rho_0'}, h^{\rho_0'} m_0)$$
$$\mathsf{ct}_1 = (g_1^{\rho_1}, h_0^{\rho_1}, g_1^{\rho_1'}, h^{\rho_1'} m_1)$$

for uniformly random $\rho_0, \rho_1 \in \mathbb{Z}_q^*$ and $\rho_0', \rho_1' \in \mathbb{Z}_q$. It is easy to see that because rerandomization uses a uniformly random $\rho \in \mathbb{Z}_q^*$ and uniformly random $\rho \in \mathbb{Z}_q$ the exact same distributions describe $\mathsf{Ran}^{r_0}(\mathsf{ct}_0)$ and $\mathsf{Ran}^{r_1}(\mathsf{ct}_1)$. It is therefore sufficient to prove that $\mathsf{ct}_0 \approx \mathsf{ct}_1$, where $\approx$ denotes computational indistinguishability. This clearly holds under the DDH assumption in $\mathbb{G}$, and follows using essentially the same proof as the proof of IND-CPA of ElGamal encryption. Namely, assume we get $(A, B, C, D) \in \mathbb{G}^4$ which in case A are four independent uniformly random elements and which in case B are three independent uniformly random elements A, B, C and $D = C^{\mathrm{DL}_A B}$. These two distributions are by definition computationally indistinguishable under the DDH assumption. Now input $\mathsf{ct} = (A, B, C, Dm_b)$ to $\mathcal{A}$ in the game instead of ct_b. In case A this information-theoretically hides b, so the adversary guesses b with probability exactly $\frac{1}{2}$. In case B this gives ct exactly the same distribution as ct_b. Since cases A and B are indistinguishable it follows that the adversary guesses b with probability negligibly close to $\frac{1}{2}$. Otherwise we could use $g \oplus b$ to distinguish case A or case B. □

4 Compressible Randomness Beacons (CRaBs)

In this section, we formilize the notion of compressible randomness beacons. We refer the reader to the technical overview in Section 1.3 for a discussion of what this primitive is and why we need it.

Definition 24. *A compressible randomness beacon (CRaB) with input domain* $[T]$ *and range* $\mathcal{Y}$ *is defined by a tuple of PPT algorithms* $\mathsf{CRaB} = (\mathsf{Gen}, \mathsf{Eval}, \mathsf{Prefix})$*, which are defined as follows:*

$k \leftarrow \mathsf{Gen}(1^\lambda)$**:** *The key generation algorithm takes the security parameter λ as input and returns a key k.*

$v \leftarrow \mathsf{Eval}(k, i)$**:** *The evaluation algorithm takes the key k and index $i \in [T]$ as input and returns an evaluation $v \in \mathcal{Y}$.*

$k^* \leftarrow \mathsf{Prefix}(k, 1^i)$**:** *The prefix algorithm takes a key k and index $1^i \in [T]$ in unary as input and returns a key k^*.*

Remark 1. We note that our prefix algorithm takes the index as a unary input, which may seem odd at first. This is done for technical reasons. In our main protocol, the polynomially bounded parties will only run for a polynomial number of time steps and therefore only input a polynomially large index, thus the provided interface of the prefix algorithm is sufficient. Looking ahead, in the proof of our compressible randomness beacon construction, we will need to guess the largest index provided by the adversary, but because there are only polynomially many that the adversary could query, we will be able to do this with a polynomial loss, instead of a super-polynomial loss. $\circ$

We would like our randomness beacon to be correct in the sense that the prefix keys produce evaluations that are consistent with the master secret key.

Definition 25 (Correctness). *We say $\mathsf{CRaB} = (\mathsf{Gen}, \mathsf{Eval}, \mathsf{Prefix})$ with input domain $[T]$ is correct, if for all $i, j \in [T]$ with $i \leq j$ it holds that*

$$\Pr\left[\begin{array}{r} k \leftarrow \mathsf{Gen}(1^\lambda) \\ k^* \leftarrow \mathsf{Prefix}(k, 1^j) \end{array} \middle\| \; \mathsf{Eval}(k, i) = \mathsf{Eval}(k^*, i)\right] = 1 \ ,$$

where the probability is taken over the algorithm's random coins.

From a security perspective, we want keys for prefixes to not reveal anything about outputs on indices outside that prefix. Given keys for prefixes up to index j, the adversary should not be able to predict any value at any index i with $i > j$. Conceptually, our notion is reminiscent of notions for constrained PRFs [BW13, KPTZ13, BGI14].

Definition 26 (Unpredictability). *We say $\mathsf{CRaB} = (\mathsf{Gen}, \mathsf{Eval}, \mathsf{Prefix})$ with input domain $[T]$ and output domain $[N]$ is unpredictable, if for any PPT adversary $\mathcal{A}$ it holds that*

$$\Pr\left[\begin{array}{r} k \leftarrow \mathsf{Gen}(1^\lambda) \\ (i, v) \leftarrow \mathcal{A}^{\mathsf{Eval}(k,\cdot), \mathsf{Prefix}(k,\cdot)}(1^\lambda) \end{array} \middle\| \begin{array}{r} \mathsf{Eval}(k, i) = v \wedge \\ \max Q_P < i \wedge \\ i \notin Q_E \end{array}\right] \leq \frac{1}{N} + \mathsf{negl}(\lambda),$$

where the randomness is taken over the coins of all algorithms and the adversary, Q_E is the set of inputs queried by the adversary to the oracle $\mathsf{Eval}(k, \cdot)$, and Q_P is the set of inputs to oracle $\mathsf{Prefix}(k, \cdot)$.

The property that makes CRaBs non-trivial to construct and interesting for our application is the δ-compressibility requirement, which states that prefix keys should be of size at most δ.

Definition 27 (Compressibility). *We say $\mathsf{CRaB} = (\mathsf{Gen}, \mathsf{Eval}, \mathsf{Prefix})$ with input domain $[T]$ and range $\mathcal{Y}$ is δ-compressing for some $\delta = \delta(T)$, if for all $k \leftarrow \mathsf{Gen}(1^\lambda)$ and $i \in [T]$ it holds that $|\mathsf{Prefix}(k, i)| \leq \delta$.*

Constructing CRaBs that are $\widetilde{\mathcal{O}}(T)$-compressing is trivial, but also not particularly interesting, so our focus lies on building CRaBs that are $o(T)$-compressing.

4.1 Constructing Compressible Randomness Beacon

Our construction of a compressible randomness beacon is a conceptually simple adaptation of existing approaches that construct puncturable PRFs [BW13, KPTZ13, BGI14] from the PRF construction of Goldreich, Goldwasser, and Micali (GGM) [GGM84].

Before explaining our construction, let us first recall the GGM construction, which constructs a PRF $F : \{0,1\}^\ell \to \{0,1\}^\lambda$ from a PRG $G : \{0,1\}^\lambda \to \{0,1\}^{2\lambda}$. For notational convenience, let us define G_0 and G_1 to be functions that output the first and second half of the output of G, i.e. $G(s) = G_0(s)\|G_1(s)$. On input $x = x_1\| \ldots \|x_\ell \in \{0,1\}^\ell$, the output of F with key k is defined to be $G_{x_\ell}(\ldots G_{x_2}(G_{x_1}(k))\ldots)$. Pictorially, one can view the evaluation of F as traversing a binary tree of depth ℓ from top to bottom via a route that is determined by input x and then returning the value of the leaf node that is reached. The root node's value is k and for any (non-leaf) node with value v, the left child's value is $G_0(v)$ and the right child's value is $G_1(v)$. On input $x = x_1\| \ldots \|x_\ell \in \{0,1\}^\ell$, one traverses the tree by iterating over the bits and going left if the current bit is zero and right if the current bit is one.

Our construction of a compressible randomness beacon will be based on the above construction and will make use of the following simple, but very powerful observation, which was already exploited in works that constructed puncturable PRFs [BW13, KPTZ13, BGI14]: one can take the PRF key k and produce a key, which only allows evaluating the PRF on a subset of inputs. Note that computing a certain leaf's value, i.e., a PRF output, requires knowing the value of at least one node between that leaf and the root node and if no such value is known, then the leaf's value is computationally hidden. As an example, one could provide the left child of the root node as a constrained key, which would allow for evaluating the PRF on all inputs starting with a zero bit, but prevent anybody from predicting the output values for any inputs that start with a one bit.

The idea behind our compressible randomness beacon is to view the i-th leaf (counted from the left) as the i-th random beacon output. Along with the random output, we will provide a *small* constrained key, which will allow for computing all previous outputs, but not any future ones. In the setting of puncturable PRFs based on the GGM construction, one can generally not reveal multiple arbitrary constrained keys as this would allow the adversary more outputs than they should. However, when the sequence of revealed keys is fixed to be the sequence which allows for evaluating growing prefixes of leafs, this is not a problem.

To formally present our construction, let us introduce some notation. We assume all nodes in the binary tree can be addressed by bit-strings, where the string $x_1\| \ldots \|x_k$ would point to the node that is reached by starting at the root node and going left or right depending on x_1, then making the same decision based on x_2 and so on. The root node has address $\perp$. For a node with address v, we refer to the node's value by $\mathsf{Value}(v)$. The parent of that node is $\mathsf{Parent}(v)$, the left child is $\mathsf{Left}(v)$, and the right child is $\mathsf{Right}(v)$. Let $\mathsf{Pred}(v)$ be the set of predecessors (along with their values) of v, i.e., the parent of v and the parent of the parent of v and so on. Let $\mathsf{Span}(v)$ be the set of all node addresses (and their values) that has v as a predecessor. For a given node v and a leaf index $i \in \mathsf{Span}$, let $\mathsf{Compute}(v, i)$ be the function that computes the value of node i according to the GGM construction by appropriately cutting off the prefix bits of i that would be "above" node v.

Theorem 2. *Let $\lambda, \ell \in \mathbb{N}$ with $\ell = \mathsf{poly}(\lambda)$. Let $G : \{0,1\}^\lambda \to \{0,1\}^{2\lambda}$ with $G_0 : \{0,1\}^\lambda \to \{0,1\}^\lambda$ and $G_1 : \{0,1\}^\lambda \to \{0,1\}^\lambda$, such that for all $s \in \{0,1\}^\lambda$, it holds that $G(s) = G_0(s)\|G_1(s)$ is a secure pseudorandom generator. Then the construction in Fig. 5 is a correct, unpredictable and $\mathcal{O}(\ell \cdot \lambda)$-compressing randomness beacon.*

```
Gen(1^λ)
1. $k \leftarrow \{0,1\}^\lambda$
2. Return $k$

Eval(k, i)
1. If $i \notin \mathsf{Span}(k)$
2.     Return $\perp$
3. Else
4.     Parse $i$ as $i_1 \| \dots \| i_\ell$
5.     Let $v \in k \cap \mathsf{Pred}(i)$
6.     Return $\mathsf{Compute}(v, i)$

Prefix(k, 1^i)
1. If $i = 2^\ell - 1$
2.     Return $k$
3. $\mathsf{cur} := \perp$
4. $S = \{i\}$
5. While $\mathsf{cur} \neq i$
6.     If $\mathsf{Left}(\mathsf{cur}) \in \mathsf{Pred}(i)$
7.         $\mathsf{cur} := \mathsf{Left}(\mathsf{cur})$
8.     If $\mathsf{Right}(\mathsf{cur}) \in \mathsf{Pred}(i)$
9.         $S := S \cup \{\mathsf{Left}(\mathsf{cur})\}$
10.        $\mathsf{cur} := \mathsf{Right}(\mathsf{cur})$
11. Return $S$
```

Fig. 5. Compressible randomness beacon construction.

Proof (Sketch). Let us start with observing that our construction is correct. Let k be an arbitrary key, let $i \in [2^\ell - 1]$, and let $S := \mathsf{Prefix}(k, 1^i)$. If $i = 2^\ell - 1$, then the algorithm just returns the root and from there clearly any leaf can be computed. So assume $i < 2^\ell - 1$ and let $j < i$ as the i-th leaf is always included in S by construction. Let v be the lowest common ancestor of node i and j. Since $i > j$, we know that reaching i requires going down the right and reaching j requires going down the left path. This means that during the computation of $\mathsf{Prefix}(k, i)$, the left child of v was included in S from which one can compute the value of node j.

To see that the construction is $\mathcal{O}(\ell \cdot \lambda)$-compressing, we observe that at each layer, we include at most one λ-bit long string into S. Since the depth of the tree is ℓ, the compression follows.

To argue unpredictability, we would like to follow the proof strategy of [GGM84], which reduces the security of their PRF to the security of a PRG, but we face an additional challenge. The adversary against the randomness beacon's unpredictability is additionally allowed to query the $\mathsf{Prefix}(k, \cdot)$ oracle and we thus may need to provide internal nodes of the tree, which allow for computing certain evaluations of the beacon. This means that during the proof, we cannot just simulate leafs by random values as this may then end up being inconsistent with the internal nodes we would need to provide.

Let $\mathcal{A}$ be a polynomial time adversary that runs in time $q(\lambda)$. By the bound on the runtime on $\mathcal{A}$, we know that any index queried to the prefix oracle is of size at most $q(\lambda)$. We consider a modified unpredictability experiment, where the challenger initially guesses the largest index $i^* \in [q(\lambda)]$, which the adversary $\mathcal{A}$ against the unpredictability will query to the prefix oracle. Since $\mathcal{A}$ is PPT, it means that the guess will be correct with an inverse polynomial probability. The challenger honestly picks uniformly random node values for the nodes in the set that would be returned by $\mathsf{Prefix}(k, i^*)$. Note that for any $i \leq i^*$ this allows consistently answering any query $\mathsf{Prefix}(k, i)$ or $\mathsf{Eval}(k, i^*)$ as both responses can be computed from the output of $\mathsf{Prefix}(k, i^*)$. For any $i > i^*$, any query to the Eval oracle with index i, we let the challenger pick a fresh uniformly random value (unless it was previously already picked in which case we use the already selected value) and return that value. Finally, at some point the adversary $\mathcal{A}$ will output a pair (i, v). If our guess for i^* was incorrect, then the challenger in our hybrid simply aborts and makes a random guess. If the challenger's guess was correct, then clearly the adversary can do no more than guess an output, since they are all uniformly random and independently chosen values.

What remains to show is that this modified experiment and the original experiment are indistinguishable from $\mathcal{A}$'s perspective. This is done by an argument essentially identical to the one of the GGM PRF, just applied to a part of the tree and not the full tree itself. Due to the guessing, our proof incurs an additional polynomial security loss. Since the proofs from this point on are virtually identical, we refer the interested reader to the original proof [GGM84]. □

5 Strongly Oblivious Read-Once Maps (SOROMs)

In this section, we formally introduce the notion of strongly oblivious read-once maps (SOROMs), which are defined by a tuple of PPT algorithms $\mathsf{SOROM} = (\mathsf{Pos}, \mathsf{Route}, \mathsf{Read})$. For a high-level discussion of what this primitive is and how it will be used in our context, we refer the reader to the technical overview in Section 1.3.

We model SOROMs as encrypted memory arrays. When inserting an element into the data structure, a random label L is chosen and assigned to the element. The element along with the label are encrypted and are always initially placed into position 0 of the memory.

To make sure that we can always insert new elements into position 0 , we need to route elements around in the memory after each insertion. For each $i \in \mathbb{N}$, after the i-th insertion operation, the memory positions indicated by $\mathsf{Pos}(i)$ will be the ones that are touched for routing purposes. Data in these positions may be moved around and data outside these positions stays where it is for the moment. The positions returned by $\mathsf{Pos}(i)$ and the labels of elements in those positions are given as input to Route, which determines how these elements should be moved around. More precisely, Route outputs a permutation π on $[|\mathsf{Pos}(i)|]$ specifying how data elements are permuted within the positions from $\mathsf{Pos}(i)$. Since the actual data elements are just dragged along with their corresponding labels and not important for the routing we leave them out of the definition. Only labels are routed around. The function $\mathsf{Read}(L)$ returns a set of memory positions, ensuring that one of those positions stores label L.

Informally, we define strong obliviousness to mean that L leaks nothing about when the data was inserted, even if the adversary gets to see the movements of a set of adversarially corrupted labels. Thus one can retrieve data by reading at locations $\mathsf{Read}(L)$ without leaking which was the corresponding write operation.

Definition 28. *A read-one map with label space $\mathcal{L}$ and memory size N is a tuple of PPT algorithms* $(\mathsf{Pos}, \mathsf{Route}, \mathsf{Read})$*, which are defined as follows:*

$S \leftarrow \mathsf{Pos}(i)$**:** *The positions algorithm takes an insertion counter $i \in \mathbb{N}$ as input and returns a set of memory positions $S \subset [N]$.*

$\pi \leftarrow \mathsf{Route}(j_1, \ldots, j_\ell, L_1, \ldots, L_\ell)$**:** *The routing algorithm takes memory indices $j_1, \ldots, j_\ell \in [N]$ and labels $L_1, \ldots, L_\ell \in \mathcal{L}$ as input and returns a permutation $\pi : [\ell] \to [\ell]$.*

$S \leftarrow \mathsf{Read}(L)$**:** *The read algorithm takes a label $L \in \mathcal{L}$ as input and returns a set of memory locations $S \subset [N]$.*

In the following security definitions, we view the memory $\mathcal{M}$ as a function mapping indices to labels, i.e., $\mathcal{M}\colon [N] \to \mathcal{L}$. For notational convenience for a set $S \subset [N]$, we write $\mathcal{M}(S) := \{L \mid \exists i \in S : \mathcal{M}(i) = L\}$.

Definition 29 (Correctness). *We say a read-one map* $(\mathsf{Pos}, \mathsf{Route}, \mathsf{Read})$ *with label space $\mathcal{L}$ and memory size N is correct, if at the end of an execution of running* ExecSOROM *(Fig. 6) with final insertion counter $i \in \mathbb{N}$, it holds that*

$$\Pr\left[\forall i' < i \ (\, L_{i'} \in \mathcal{M}[\mathsf{Read}(L_{i'})]\,)\right] \geq 1 - \mathsf{negl}(\lambda) \ .$$

Game $\text{ExecSOROM}_{\mathcal{A}}^{\mathsf{SOROM}}(1^\lambda)$.

1. Initialise an insertion counter $i = 0$, a set of corrupt insertions $\mathcal{C}$ and a set of honest insertions $\mathcal{H}$. Let $\mathcal{M}$ be the initial memory with $\mathcal{M}[j] = 0^\lambda$ for all j. Sample a uniformly random challenge bit b.
2. Run $\mathcal{A}$ to get an output o where $o \in \{\textsc{honest}, \textsc{corrupt}\}$ or $o = (\textsc{done}, g \in \{0, 1\})$.
3. If $o \in \{\textsc{honest}, \textsc{corrupt}\}$, proceed as follows:
   (a) Sample uniformly random $L_i \leftarrow \mathcal{L}$. If $L_i = L_{i'}$ for any $i' < i$ then stop the game with output 0 (meaning that the adversary wins).[a]
   (b) If $o = \textsc{corrupt}$ then add i to $\mathcal{C}$ and give L_i to $\mathcal{A}$.
   (c) If $o = \textsc{honest}$ then add i to $\mathcal{H}$. Furthermore, if $b = 0$ give L_i to $\mathcal{A}$. If $b = 1$ sample uniformly random $L_i^* \leftarrow \mathcal{L}$ and give L_i^* to $\mathcal{A}$.
   (d) Update $\mathcal{M}[0] = L_i$.
   (e) Compute the positions to route among $(j_1, \ldots, j_\ell) = \mathsf{Pos}(i)$.
   (f) For $h = 1, \ldots, \ell$ let $L_h = \mathcal{M}[j_h]$.
   (g) Sample $\pi \leftarrow \mathsf{Route}(j_1, \ldots, j_\ell, L_1, \ldots, L_\ell)$.
   (h) For $h = 1, \ldots, \ell$ let $\mathcal{M}[j_h] = L_{\pi(h)}$.
   (i) For all $j \in \mathcal{C}$ tell $\mathcal{A}$ the new position of L_j.
   (j) Let $i = i + 1$ and go to step 2.
4. If $o = (\textsc{done}, g)$ then terminate with output $g \oplus b$.[b]

[a] This is a technicality allowing us to proceed below under the assumption that all labels are unique.
[b] Note that output 0 means that the adversary guessed b correctly.

Fig. 6. Security game for SOROMs.

We define the strong obliviousness property that we require from our data structure through the game depicted in Fig. 6. In this game, the adversary is allowed to adaptively insert honest or corrupted data elements. The movements of the corrupt ones they can track, those of the honest ones they cannot. For the honest ones, the adversary either always gets the real corresponding labels or independent random labels. By just seeing the movements of the corrupt labels, the adversary then needs to decide in which of those two worlds it lives.

Definition 30 (Strong Obliviousness). *We say a read-once map* $(\mathsf{Pos}, \mathsf{Route}, \mathsf{Read})$ *with label space $\mathcal{L}$ and memory size N is strongly oblivious, if for any adversary $\mathcal{A}$, it holds that*

$$\Pr\left[\text{ExecSOROM}_{\mathcal{A}}(1^\lambda) = 1\right] \leq \frac{1}{2} + \mathsf{negl}(\lambda) ,$$

where $\text{ExecSOROM}_{\mathcal{A}}(1^\lambda)$ *is defined in Fig. 6 and the probability is taken over the random coins of the experiment and the adversary.*

5.1 Constructing Strongly Oblivious Read-Once Maps

Our construction of a strongly oblivious read-once map closely follows the ORAM construction of Shi *et al.* [SCSL11]. They arrange their memory of size $N = 2^D$ into a full binary tree of depth D, where each leaf and internal node is a bucket which can store up to B data elements. For an overview of the binary construction see Section 3 in [SCSL11]. Elements are inserted in the root bucket with an associated label L pointing to a uniformly random leaf of the tree. To get a SOROM we let the data element be L.

After each insertion an eviction algorithm $\mathsf{Evict}(\nu)$ is run, see Fig. 5 in [SCSL11]. From root to leaf, in each layer ν buckets b are chosen and from each b one data element L is pushed one level down towards leaf L. To the other child bucket a dummy element is pushed for obliviousness. For their asymptotical analysis they use $\nu = 2$, and we do the same. We will use the same construction but using instead an eviction rule $\mathsf{EvictAll}(\nu)$ which from each b pushes *all* elements in b one level down. I.e., $\mathsf{EvictAll}(\nu)$ simply runs $\mathsf{Evict}(\nu)$ B times on the bucket b. We use $D = \lambda$ and thus $N = 2^\lambda$. We use bucket size $\overline{B} = \sigma(\log_2 \lambda)(\log_2 \log_2 \lambda)$ for a statistical security parameter σ, which can be set at will under the restrictions that $2^{-\sigma} = \mathsf{negl}(\lambda)$.

Note that [SCSL11] also has protocols for handling what happens when an element is read (it is reinserted at the root). We do not use this. We only insert L in the root and then run $\mathsf{EvictAll}(\nu)$ once after each insert. The set $\mathsf{Pos}(i)$ is thus the positions in the buckets b selected for eviction in the i-th run of $\mathsf{EvictAll}(\nu)$ and $\mathsf{Route}(i)$ is given by the pushing down in the tree. The set $\mathsf{Read}(L)$ is the set of buckets from the root to L. We now proceed to analyse the construction. A bucket is said to overflow if we try to store more than $\overline{B}$ elements in it.

Lemma 1. *For the basic ORAM in [SCSL11] with the above parameters and using* $\mathsf{EvictAll}(\nu)$ *and under an access pattern which does a polynomial $M \leq N$ number of inserts to unique addresses $0, \ldots, M-1$, the probability that any bucket overflows is at most $2^{-\sigma}$.*

Proof. Consider polynomial $M = \lambda^m$ inserts to unique addresses, for $m \in \mathbb{N}$. Assume for now that we use $\mathsf{Evict}(\nu)$ and that we use bucket size $B = 2\sigma$. Let $D_M = \log_2 M = m \log_2 \lambda$. Change the eviction rule to not moving any elements out of buckets at depth D_M or deeper. Clearly the probability of overflowing any bucket only increases. Furthermore, the ORAM behaves *exactly* as the basic ORAM from [SCSL11] with ORAM capacity $N_D = 2^{D_M} = M$. Note that the address space is $\{0, \ldots, N_D - 1\}$ and $M = N_D$, so we can still inserts to the M distinct addresses. The significance of writing to distinct addresses is that then there will never be an overwrite of an address. Since there are no reads either, it follows that [SCSL11] like our SOROM just forever evicts elements towards leafs. So, by Lemma 2 on page 210 in [SCSL11] we have for the eviction rule $\mathsf{Evict}(\nu)$ with $\nu = 2$ that the probability that any given bucket at any given time overflows is at most $\delta = 2^{-B}$. Taking a union bound over $N_D = 2^{D_M}$ buckets and M operations this gives us an upper bound of $\gamma = M N_D 2^{-B} = 2^{2D_M - B} = 2^{2m(\log_2 \lambda) - B}$. From $2^{-\sigma} = \mathsf{negl}(\lambda)$ we have that $2m \log_2 \lambda \lesssim \sigma$, where we use $\lesssim$ to mean eventually $\leq$ for large enough λ. So, $\gamma \lesssim 2^{-\sigma}$. Since $2^{-\sigma}$ matches the conclusion of the lemma, let us assume for the rest of the proof that no bucket overflows. Then no bucket ever has more than B elements. As we do not push below depth D_M, it follows that no path ever has more than $D_M B$ elements. This clearly also holds if we increase the size of buckets to ∞, as $\mathsf{Evict}(\nu)$ does not depend on bucket size when there is no overflow.

Now switch from $\mathsf{Evict}(\nu)$ to $\mathsf{EvictAll}(\nu)$ and keep everything else the same, i.e., $B = \infty$ and we make the same random choices for which buckets to evict. Notice that $\mathsf{EvictAll}(\nu)$ always produces paths with *less* elements than $\mathsf{Evict}(\nu)$. To see this consider a bucket b on path (towards) L. If we use $\mathsf{Evict}(\nu)$ then one element is pushed from b and down L or it is pushed off L. The remaining elements from b stay on L, as b is on L. If we run $\mathsf{EvictAll}(\nu)$ then the remaining elements in b are all pushed down L or off L. This at most decreases the number of elements on L. What makes this work is that elements are never pushed *unto* a new path, as we work with a tree and always push down. Ergo, with $\mathsf{EvictAll}(\nu)$ no path will have more than $D_M B$ elements. So, clearly no *bucket* will have more than $D_M B$ elements either. We have that $D_M B = 2\sigma m \log_2 \lambda \lesssim \sigma(\log_2 \lambda)(\log_2 \log_2 \lambda) = \overline{B}$, where the inequality uses that $2m$ is a constant and thus $2m \lesssim (\log_2 \log_2 \lambda)$. So no bucket ever has more than $\overline{B}$ elements. Now change the eviction rule to push beyond level D_M again and use bucket size $\overline{B}$. After these changes we are exactly running the ORAM in the premise of the lemma.

Pushing beyond level D_M clearly only decreases the probability that a bucket overflows. This proves the lemma. $\square$

Theorem 3. *The construction described above with label space $\mathcal{L} = \{0,1\}^\lambda$ and bucket size of $\overline{B} = (\log_2 \lambda)^3 (\log_2 \log_2 \lambda)$ is a SOROM and for any polynomial number of operations is correct with probability $1 - \mathsf{negl}(\lambda)$. Furthermore, each invocation of* Route *needs to touch an expected $\mathcal{O}(\log_2(\lambda) \cdot \overline{B})$ labels.*

Proof. Setting $\sigma = (\log_2 \lambda)^2$ we get bucket size $\overline{B} = (\log_2 \lambda)^3(\log_2 \log_2 \lambda)$ and error probability $\lambda^{-\log_2 \lambda} = \mathsf{negl}(\lambda)$. Correctness follows from Lemma 1 as any adversary against the SOROM is polynomial and therefore does at most polynomially many inserts. When there are no overflows, correctness is straight forward. Strong obliviousness follows from the fact that elements move *independently* of each other: if an element is in a bucket chosen for eviction, then it will move one step down the tree.[13] Furthermore, the element moves towards its uniformly random L which is independent of other labels. As for complexity note that touching a path potentially touched $\lambda \overline{B}$ labels as there are λ levels and buckets have size $\overline{B}$. Note, however, that we only need to touch buckets which were activated in the following sense. To begin with say only the root is activated. Inductively say that if an activated bucket b is chosen for eviction by $\mathsf{EvictAll}(\nu)$, then its two children become activated. After M operations the were at most $2M$ distinct buckets chosen for eviction at each level $d-1$ as $\nu = 2$. So, at most $4M$ distinct buckets were activated at level d. So, since level d has width 2^d, when accessing all buckets on the part to a random label L there it probability $\leq 4M2^{-d}$ of touching an activated bucket at level d. For $d \leq \log_2(M) + 2$ this gives the uninteresting bound ≤ 1. For $d \geq \log_2(M) + 2 + \xi$ it gives $\leq 2^{-\xi}$. So, the expected number of active buckets touched is at most $\log_2(M) + 3 = \mathcal{O}(\log_2 \lambda)$. $\square$

6 Ideal Functionality for Anonymous Cryptocurrency

We start by giving a security definition for anonymous cryptocurrency by an ideal functionality $\mathcal{F}_{\mathsf{AnonPay}}$. The model is very idealised and is not an attempt to model all aspects which would be relevant to a real-life implementation. The model is meant to capture the fundamental security properties that we want the system to have by giving an ideal functionality with these properties and then defining security via the UC framework.

We consider an account based setting as it makes our definitions of anonymity more intuitive. We assume accounts might have a known association with real world identities and that transactions are between accounts. The motivation for the first assumption is that once an account was used to pay for a good at an online shop and the shop shipped the good to the user, the shop will know who owns the account. Our constructions can be applied to a UTXO setting too, but would require a somewhat more involved modelling, so we pick an account based model for simplicity. To mitigate that the owner of accounts might not be fully anonymous we will require that the sending account and receiving account of a transfer cannot be linked. The only thing which leaks about an account is how many sends and receives it made. So anonymity is required at the level of *transfers*. We consider two levels of anonymity. When the flavour is weak then the sender of a transfer may learn when the receiver collects the transfer. When the flavour is strong then the sender is oblivious of when the receiver collects the transfer.

[13] Note that this would not be true for $\mathsf{Evict}(\nu)$.

We use $\mathsf{Accounts}$ to denote the set of accounts. For concreteness think of the set of accounts as being the set of public keys for a signature scheme. We generally denote an account by A. For concreteness think of each account having an associated key pair $(\mathsf{pk}, \mathsf{sk})$ and $\mathsf{A} = \mathsf{pk}$ being the name of the account. In the discussion below, when using A to name the account we use sk_A to name the secret key. The part of $\mathcal{F}_{\mathsf{AnonPay}}$ related to account creation is given in Fig. 7.

Parameters The ideal functionality is parameterised by a flavour of anonymity $\mathsf{fla} \in \{\textsc{strong}, \textsc{weak}\}$ and an initial amount $a_0 \in \mathbb{N}$. On the first activation it asks the adversary $\mathcal{S}$ for a TID distribution Tid which is used for sampling transfer identifiers. The adversary must give a distribution Tid back with exponential collision-entropy, i.e., if S is a set of payment identifiers, then $\Pr[\mathsf{tid} \in S \mid \mathsf{tid} \leftarrow \mathsf{Tid}] \leq |S|2^{-\lambda}$. This is to ensure that we can ignore the event that a randomly sampled transfer identifier will hit an already used one.

Init When activated for the first time, proceed as follows. Let $\mathsf{Accounts} = \{\}$ be the initial set of accounts. Initialise a map $\mathsf{Balance} : \mathsf{Accounts} \to \mathbb{N}$ with $\mathsf{Balance}[\mathsf{A}] = 0$ for all A. Initialise the abstract transfer identifier $\mathsf{atid} = 0$.

Create Account On input $(\textsc{CreateAccount})$ from a party P, leak $(\textsc{CreateAccount}, \mathsf{P})$ to $\mathcal{S}$.

Callback Account Observable On a subsequent input $(\textsc{MakeAccountObservable}, \mathsf{P}, \mathsf{A})$ from $\mathcal{S}$, where $\mathsf{A} \notin \mathsf{Accounts}$, add A to $\mathsf{Accounts}$ and output $(\textsc{CreateAccount}, \mathsf{A})$ to P. From now on let P_A denote P and if we say that A does something, we mean that P_A does that thing.
If this was the first account created, i.e., $|\mathsf{Accounts}| = 1$, then let $\mathsf{FA} = \mathsf{A}$ and let $\mathsf{Balance}[\mathsf{FA}] = a_0$.

Fig. 7. Functionality $\mathcal{F}^{\mathsf{fla}, a_0}_{\mathsf{AnonPay}}$. Events related to creation.

Remark 2 (Where is the secret key?). When an account is created by party P the name A of the account is returned by the command, as opposed to letting P input the account name to the command. Another design choice is that only the account name is output, not any secret key material used for controlling the account. In terms of an implementation, think of the key material $(\mathsf{pk}, \mathsf{sk})$ as being generated as part of the account creation command. After the account was created the command returns $\mathsf{A} = \mathsf{pk}$ to the user on its API and the corresponding secret key sk_A is securely stored locally. When the party P_A, having created A, wants to use the account $\mathsf{A} = \mathsf{pk}$ then the secret key sk_A is recovered from storage. This prevents that we give sk_A as output and have to give it as input. Since it is the environment which sees outputs and gives inputs in the UC model and the environment talks to the adversary, having made the secret key sk an output would have created problems.

One might then be worried about how to authenticate access to sk_A. Since $\mathsf{A} = \mathsf{pk}$ is public knowledge and one identifies the payer in a payment $(\textsc{Pay}, \mathsf{A}, \mathsf{B}, a)$ just by naming A, cannot anyone walk up and ask to transfer in the name of A? The answer is no and is guaranteed by the party identifier logistics of the UC framework. In the UC framework, it is only the party with party identifier pid who can give inputs in the name of pid to an ideal functionality. In our case we call parties P which technically just means that the party identifier is $\mathsf{pid} = \mathsf{P}$. The ideal functionality therefore by UC design remembers which P received A, call it P_A, and then only takes commands to control A from P_A. In terms of an implementation, think of it as follows. Anyone can run the command $(\textsc{CreateAccount}, \mathsf{P})$. During this a key pair $(\mathsf{pk}, \mathsf{sk})$ is generated. By doing this the party having run $(\textsc{CreateAccount}, \mathsf{P})$ learns $(\mathsf{pk}, \mathsf{sk})$. It outputs $\mathsf{A} = \mathsf{pk}$ on the API so it can be used by outer protocols, but it stores $\mathsf{sk}_\mathsf{A} = \mathsf{sk}$ locally. This is what implements that only P can control sk. Since the standard corruption behaviour of the UC model is so-called *pid-wise corruption* it holds that the party P_A in an implementation of $\mathcal{F}_{\mathsf{AnonPay}}$ is corrupted if and only if the party calling party P_A on $\mathcal{F}_{\mathsf{AnonPay}}$ is corrupted. Therefore sk_A becomes known to the adversary if and only if the party with party identifier P_A is corrupted in the surrounding protocol. ◦

Remark 3 (Why does the adversary pick the account name?). We discuss a final subtlety of the model of account creation. In the ideal functionality we let the adversary pick the name A of the account. This might look weird, but it just models that the name of the account can be anything. In a proof of security it is the simulator which is the adversary towards $\mathcal{F}_{\mathsf{AnonPay}}$ and the simulator will just set A to be the public key that identifies the account in the implementation. If we did not allow the adversary to pick A then the simulator could not do this alignment of account names in the implementation and the ideal functionality. In terms of using $\mathcal{F}_{\mathsf{AnonPay}}$ as an ideal functionality in a larger construction the design choice means that an outer protocol should not rely on the account names having a particular form or distribution, which seems as a healthy design principle independently of the subtleties of UC modeling. ◦

Remark 4 (Could not the adversary steal the founding account?). Yes! Note that we simply say that the first account created is the funding account and it gets the initial amount a_0. This in principle allows a corrupted party to open the funding account. We picked this model for simplicity. Who is allowed to open the funding account is decided non-algorithmically and will in practice typically be decided even before the blockchain exists. We see no advantage in attempting a detailed model of this genesis ceremony. When using $\mathcal{F}_{\mathsf{AnonPay}}$ in some larger context one can simply assume that the environment only allows the intended party P to open the funding account.

We now describe and discuss how we model anonymous transfer. A transfer consists of a payment and a collection.

Transfer Anonymity. We assume that accounts can be associated to users, or rather, we do not assume that the user-account association can be reliably hidden from the adversary. It is therefore a conservative model to just assume that the adversary knows who owns which accounts. Anonymity will therefore be implemented by ensuring that if a payment is made from account A to account B, then A and B cannot be linked. When the amount is deducted from A the identity of B is hidden. When the amount is added to B the identity of A is hidden. We require a strong notion of anonymity which ensures that if an amount is deposited on B then this could be from any previous outgoing payment from any other account in the system. The adversary might have prior knowledge on who pays who, but observing the communication of the payment system should give it no additional knowledge. More concretely we want to require the following. First of all, we assume that the receiver of a payment is hidden for anyone but the sender and the receiver. We require that transferred amounts are hidden from anyone but the sender and receiver: this prevents linking outgoing and ingoing payments using the amounts. We allow that the receiver learns which account sent the payment and that the payer learns who is being payed.

Collection. In any anonymous payment system there will be some notion of the receiver "collecting" the payment later than when it is being made.[14] We cannot have that the account of

[14] In a UTXO system this would correspond to when a UTXO is being spent in the future.

the receiver is updated at the same time as the payment is being made. This would allow trivial traffic analysis attacks. Hence the "collection" must happen later. We do not want to add to the ideal functionality how long a collection is delayed. We consider this an external choice. What the ideal functionality allows is to completely decouple the deduction and collection events. The only information any party, not being sender or receiver, may learn is that the collection was after the deduction, which is an *a priori* fact.

Strong versus Weak Anonymity. We consider two flavours of anonymity, WEAK and STRONG. With weak anonymity the payer can see when its own payment is being "collected". This might not be tolerable in all cases as it allows timing attacks. Maybe the payer knows that the receiver collects coins only when at a specific location. With strong anonymity we also hide the time of collection from the payer.

Observability. The ideal functionality also has a notion of observability. Consider a customer paying anonymously in a pizza shop. There is a point where the customer initiates the payment, for instance by holding a smart phone next to a device in the shop and approving the amount. At some point the shop will learn that the payment went through. This might be well before it is collected, as discussed above. We therefore want to model explicitly this property that a payment has been observed to be "collectable". At this point the shop can safely hand out the pizza. It then chooses to collect the coins later to preserve the anonymity of the payer.

Transfer Identifiers. Once a payment has been initiated it gets a transfer identifier (TID) tid. This is just a common name the payer and collector can use to refer to the payment. If the same payer pays the same receiver the same amount twice about the same time, it is often necessary to have a way to distinguish the payments externally to the payment service. This is the role of tid. Since tid is the same for the sender A and the payer B it could be used to break anonymity. Therefore we do not leak tid. When considering only weak security we leak tid during collection. This will allow the sender to learn when the transfer was collected. Third parties learn nothing as we do not leak tid at the time of payment. To make it safe to use tid externally to the ideal functionality we want that tid in and of itself leaks nothing about the transfer, like the identifier of the parties, the amount, or time of payment or collection. A simple way to do this is to require that each tid is sampled from the same distribution Tid. This is the design choice we took. To ensure that the tid's are unique (except with negligible probability) we require that Tid has collision entropy λ, where λ is the security parameter. Under these restrictions we let the adversary pick the distribution. If the sender is corrupted we allow it to pick the payment identifier in a non-random manner. We enforce, however, that it cannot reuse a payment identifier, as this could lead to confusion and subtle attacks. We therefore make $\mathcal{F}_{\mathsf{AnonPay}}$ enforce that payment identifiers are unique even when picked by a corrupt sender. Note that this puts the same requirement on an implementation.

Discussion of Commands. The part of $\mathcal{F}_{\mathsf{AnonPay}}$ related to payment and collection is given in Fig. 8. We discuss some technicalities of how the ideal functionality is specified. We let $\mathcal{S}$ denote the adversary. The value atid is an abstract transfer identifier which is internal to the ideal functionality for book keeping only. This is just a way in which the adversary can denote a given payment for which it does not known the sender, receiver or amount. The value tid is a transfer identifier used

Initiate Pay On input $(\textsc{Pay}, \mathsf{A}, \mathsf{B}, a)$ from $\mathsf{P_A}$ where $\mathsf{B} \in \mathsf{Accounts}$ and $a \leq \mathsf{Balance}[\mathsf{A}]$, let $\mathsf{atid} = \mathsf{atid} + 1$ and leak $(\textsc{Pay}, \mathsf{A}, \mathsf{atid})$ to $\mathcal{S}$. Here A is the sender, B the receiver, a is the amount, and atid is an abstract transfer identifier used to refer to the transfer internally. If $\mathsf{P_A}$ or $\mathsf{P_B}$ is corrupted then instead leak $(\textsc{Pay}, \mathsf{A}, \mathsf{B}, a, \mathsf{atid})$. If $\mathsf{P_A}$ or $\mathsf{P_B}$ is corrupted, then ask $\mathcal{S}$ for a transfer identifier tid; It must specify a tid not used between A and B before, i.e., $(\mathsf{A}, \mathsf{B}, \mathsf{tid})$ must be unique. When $\mathsf{P_A}$ is corrupted, then $\mathcal{S}$ knows the previous tid's used, so it *can* pick a unique one. If $\mathcal{S}$ specified a tid used before, then $\mathcal{F}_{\mathsf{AnonPay}}$ ignores it and samples a random identifier $\mathsf{tid} \leftarrow \mathsf{Tid}$. Add $(\textsc{Pay}, \mathsf{atid}, \mathsf{tid}, \mathsf{A}, \mathsf{B}, a)$ to $\mathsf{BeingDeducted}$. If $\mathsf{P_A}$ or $\mathsf{P_B}$ is corrupt, then leak $(\textsc{Pay}, \mathsf{A}, \mathsf{B}, \mathsf{tid}, a)$ to $\mathcal{S}$.

Callback Deduct On input $(\textsc{MakeDeducted}, \widetilde{\mathsf{atid}})$ from $\mathcal{S}$, where some $(\textsc{Pay}, \widetilde{\mathsf{atid}}, \mathsf{tid}, \mathsf{A}, \mathsf{B}, a) \in \mathsf{BeingDeducted}$, remove $(\textsc{Pay}, \widetilde{\mathsf{atid}}, \mathsf{tid}, \mathsf{A}, \mathsf{B}, a)$ from $\mathsf{BeingDeducted}$. If $\mathsf{Balance}[\mathsf{A}] \geq a$, then let $\mathsf{Balance}[\mathsf{A}] = \mathsf{Balance}[\mathsf{A}] - a$ and add $(\textsc{Pay}, \widetilde{\mathsf{atid}}, \mathsf{tid}, \mathsf{A}, \mathsf{B}, a)$ to $\mathsf{Deducted}$. Output $(\textsc{Pay}, \mathsf{A}, \mathsf{B}, \mathsf{tid}, a)$ to $\mathsf{P_A}$.

Callback Observable On input $(\textsc{MakeObservable}, \widetilde{\mathsf{atid}})$ from $\mathcal{S}$, where some $(\textsc{Pay}, \widetilde{\mathsf{atid}}, \mathsf{tid}, \mathsf{A}, \mathsf{B}, a) \in \mathsf{Deducted}$, add $(\textsc{Pay}, \widetilde{\mathsf{atid}}, \mathsf{tid}, \mathsf{A}, \mathsf{B}, a)$ to $\mathsf{Observable}$.

Callback Collectable On input $(\textsc{MakeCollectable}, \widetilde{\mathsf{atid}})$ from $\mathcal{S}$ where some $(\textsc{Pay}, \widetilde{\mathsf{atid}}, \mathsf{tid}, \mathsf{A}, \mathsf{B}, a) \in \mathsf{Observable}$, add $(\textsc{Pay}, \widetilde{\mathsf{atid}}, \mathsf{tid}, \mathsf{A}, \mathsf{B}, a)$ to $\mathsf{Collectable}$.

Observe On input $(\textsc{Observe}, \mathsf{tid}, \mathsf{A}, \mathsf{B}, a)$ from $\mathsf{P_B}$, let $J = \top$ if some $(\textsc{Pay}, \widetilde{\mathsf{atid}}, \mathsf{tid}, \mathsf{A}, \mathsf{B}, a) \in \mathsf{Observable}$ and $J = \bot$ otherwise, and return $(\textsc{Pay}, \mathsf{tid}, \mathsf{A}, \mathsf{B}, a, J)$ to $\mathsf{P_B}$.

Collect On input $(\textsc{Collect}, \mathsf{tid}, \mathsf{A}, \mathsf{B}, a)$ from $\mathsf{P_B}$ proceed as follows.

- If $(\textsc{Pay}, \widetilde{\mathsf{atid}}, \mathsf{tid}, \mathsf{A}, \mathsf{B}, a) \notin \mathsf{Observable}$ ignore and return.
- If $(\textsc{Pay}, \widetilde{\mathsf{atid}}, \mathsf{tid}, \mathsf{A}, \mathsf{B}, a) \in \mathsf{Observable} \setminus \mathsf{Collectable}$, leak $(\textsc{Collect}, \mathsf{B}, \textsc{Too Early})$ to the adversary $\mathcal{S}$ and return.
- If $(\textsc{Pay}, \widetilde{\mathsf{atid}}, \mathsf{tid}, \mathsf{A}, \mathsf{B}, a) \in \mathsf{Collectable}$, let $\mathsf{atid} = \mathsf{atid} + 1$, leak $(\textsc{Collect}, \mathsf{B}, \mathsf{atid})$ to $\mathcal{S}$ (in case of $\mathsf{fla} = \textsc{weak}$ leak $(\textsc{Collect}, \mathsf{B}, \mathsf{atid}, \mathsf{tid})$ to $\mathcal{S}$), and add $(\textsc{Pay}, \mathsf{atid}, \mathsf{tid}, \mathsf{A}, \mathsf{B}, a)$ to $\mathsf{BeingCollected}$.

Callback Collected On input $(\textsc{MakeCollected}, \widetilde{\mathsf{atid}})$ from $\mathcal{S}$, where some $(\textsc{Pay}, \widetilde{\mathsf{atid}}, \mathsf{tid}, \mathsf{A}, \mathsf{B}, a) \in \mathsf{BeingCollected}$ and $(\textsc{Pay}, \mathsf{tid}, \mathsf{A}, \mathsf{B}, a) \notin \mathsf{Collected}$ remove $(\textsc{Pay}, \widetilde{\mathsf{atid}}, \mathsf{tid}, \mathsf{A}, \mathsf{B}, a)$ from $\mathsf{BeingCollected}$ and add $(\textsc{Pay}, \mathsf{tid}, \mathsf{A}, \mathsf{B}, a)$ to $\mathsf{Collected}$ and let $\mathsf{Balance}[\mathsf{B}] = \mathsf{Balance}[\mathsf{B}] + a$. Then output $(\textsc{Collect}, \mathsf{tid}, \mathsf{A}, \mathsf{B}, a)$ to $\mathsf{P_B}$.

Fig. 8. Functionality $\mathcal{F}^{\mathsf{fla}, a_0}_{\mathsf{AnonPay}}$. Events related to payment and collection.

by the sender and receiver to refer to the transfer. It will identify the transfer, so it must be hidden from $\mathcal{S}$ and cannot be used as a pointer for book keeping. We add a delay to some events to get a more realistic model. We let the adversary determine when events happen by giving a callback when the event should take place.

We now go over the commands and explain how they may relate to a real-life implementation. On input ($\textsc{CreateAccount}$) party P would start making the key material for an account and post it on the blockchain. Once the account has been posted it would be observable by other parties. This corresponds to the ($\textsc{MakeAccountObservable}, \mathsf{A}$) event.

On input ($\textsc{Pay}, \mathsf{A}, \mathsf{B}, a$) party P starts interacting with the blockchain to create the payment. The value atid does not appear in the implementation, it is only a pointer used for bookkeeping in the ideal functionality. Since P might have its secret key stored in several places and accidentally have different sites act at the same time, we do not assume that $\mathsf{P_A}$ knows that it did not try to spend more than $\mathsf{Balance}[\mathsf{A}]$. We assume that it checks that for a given transfer of a it holds that $a \geq \mathsf{Balance}[\mathsf{A}]$, but we allow that it might happen that several such payments are initiated concurrently and would create a negative balance if all were executed. We consider this possible honest behaviour and will not punish it. It seems a crucial security requirement that an honest party does not lose safety just because it accidentally attempts a “double spend”.

At some point the payment might get so far underway (if there are sufficient funds) that the account A is deducted the amount a. This is the callback event $\textsc{MakeDeducted}$. Here we will do the check that the balance will not get negative, and reject deductions that would cause this. Further down the road the payment will reach a state where B can observe that the payment was made and can be sure that it will eventually be able to collect it. This might happen before it is actually collectable, and it is an interesting event as it would allow the receiver to safely hand out goods or service. Therefore we make this a separate callback event $\textsc{MakeObservable}$. At the end of the payment process the transfer reaches a state where it can be collected at the discretion of the receiver. This is the callback event $\textsc{MakeCollectable}$.

Finally, once a coin is collectable, the receiver might chose to collect it. This is done using the command ($\textsc{Collect}, \mathsf{tid}, \mathsf{A}, \mathsf{B}, a$) from $\mathsf{P_B}$. It is leaked to the adversary when this happens and who B is. In case of weak anonymity we also leak the transfer identifier tid. But the connection to the payment is not revealed, as we did not leak tid during payment. We also use independent values of atid during payment and collection so the adversary cannot link payment and collection via the interaction with $\mathcal{F}_{\mathsf{AnonPay}}$. Once collection has begun it will eventually terminate. We say that the collection terminated when the funds become available to the receiver. This is modelled using the callback event $\textsc{MakeCollected}$.

Remark 5 (Why does the adversary specify the distribution of transfer identifiers?). We want that the distribution of transfer identifiers can depend on the implementation. Allowing the adversary to specify it will allow the simulator to simply set it to be the one of the implementation. ◦

Remark 6 (How does the receiver learn about the payment?). Note that to collect a payment the receiver has to input ($\textsc{Collect}, \mathsf{tid}, \mathsf{A}, \mathsf{B}, a$). This might raise the question of how it learns tid and a, and maybe even how it learns that $\mathsf{P_A}$ wants to pay $\mathsf{P_B}$ at all? Why not let $\mathcal{F}_{\mathsf{AnonPay}}$ signal these values to $\mathsf{P_B}$ when the payment takes place? We have chosen not to do this to not enforce implementations where $\mathsf{P_B}$ is online during payment or where $\mathsf{P_A}$ knows how to contact $\mathsf{P_B}$. It could be that $\mathsf{P_A}$ wants to pay an anonymous account B and does not know who the owner $\mathsf{P_B}$ is. It could also be that $\mathsf{P_A}$ wants to use a particular channel to inform $\mathsf{P_B}$ of the payment that we do not model in our setting. This could for instance be a text message or a channel not leaking the physical identity of $\mathsf{P_A}$ to $\mathsf{P_B}$. We therefore assume that there is some out-of-band way that $\mathsf{P_A}$ learns that a payment took place, and the workflow of the payment on $\mathcal{F}_{\mathsf{AnonPay}}$ resumes when $\mathsf{P_B}$ learned about the payment and inputs $(\textsc{Collect}, \mathsf{tid}, \mathsf{A}, \mathsf{B}, a)$. $\circ$

7 Modelling the Blockchain World

We now describe how to model the setting in which we want to implement the anonymous cryptocurrency. We will call this the blockchain world. The blockchain world will contain a public ledger $\mathcal{F}_{\text{LEDGER}}$ and an authenticated anonymous channel $\mathcal{F}_{\text{AAT}}$, both modelled as ideal functionalities.

The public ledger is an authenticated append-only ledger. We also assume it can do so-called filtering, i.e., a message can be posted along with a claim of having some given property ϕ. The property might depend on the current state, Ledger, of the ledger. Technically we will append (m, ϕ) and we only add this message to Ledger if $\phi(\mathsf{Ledger}, m) = \top$. We add ϕ to the ledger to model that it is known how a given m was filtered. An example of a property could be that m specifies a payment for which there currently is enough balance on Ledger.

The ideal functionality $\mathcal{F}_{\text{AAT}}$ models an authenticated anonymous channel between payers and receivers. This is a means by which a payer can send a message to the receiver without this event becoming visible on the public ledger, or to other senders or receivers.

The blockchain world will always contain a protocol $\mathit{\Pi}_{\text{ANONPAY}}$ for payment. This is a protocol using $\mathcal{F}_{\text{LEDGER}}$ and $\mathcal{F}_{\text{AAT}}$. The blockchain world might also contain a protocol $\mathit{\Pi}_{\text{SERVICE}}$ which has access to $\mathcal{F}_{\text{LEDGER}}$ and might do off-ledger computations and post messages to the ledger to aid the execution of the anonymous payment service. Concretely, in our implementation, they will perform the mixing of the coins. We can model this setting in the UC framework by letting $\mathcal{F}_{\text{LEDGER}}$ be a global sub-routine of both $\mathit{\Pi}_{\text{SERVICE}}$ and the protocol $\mathit{\Pi}_{\text{ANONPAY}}$ (cf. [BCH+20]). We now describe $\mathcal{F}_{\text{AAT}}$ and then $\mathcal{F}_{\text{LEDGER}}$.

Anonymous, Authenticated Channels. We formalise a notion of anonymous, authenticated channels where the sender can send a message to a receiver without anyone else being aware that a message is being sent.

What we mainly want to model is that there is an off-chain way to deliver a message between Sender and Receiver without *the parties running the blockchain* learning about the communication. This is a tricky notion to formalise, so for simplicity we assume the anonymity is perfect, i.e., no one learns anything about the transfer. In practice this is rarely possible to implement. Imagine for instance a setting where you are in a shop and bring your credit card close to a reader using Near Field Communication to read a message from the card. Or consider a situation where you send an e-mail to the address of the Receiver using a Tor like solution. In both these situation there are parties which learn some leakage. Another customer in the shop might know that the shop uses a blockchain based payment system and inspect the blockchain to see which payments were made at the particular point in time of your purchase. This might allow them to link the purchase to a particular account on the blockchain and thereby you. You do not necessarily want other customers to learn your identity. The Tor servers at the edge do learn some information on when certain entities were active and hence also learn some, possibly vague, information about the communication. But in both cases we can allow ourselves to ignore the leakage of the off-chain channel with open eyes and keep in mind that a scheme proven secure in our model would therefore have to be analysed for traffic analysis attacks when implemented in practice and using concrete real-world channels.

Init Let $\mathsf{did} = 0$ be a delivery id.

Drop Off On input $(\textsc{DropOff}, \mathsf{P}, \mathsf{Q}, \mathsf{mid}, m)$ from P, let $\mathsf{did} = \mathsf{did} + 1$, and send $(\textsc{DropOff}, \mathsf{did})$ to $\mathcal{S}$.

Callback DropOff On input $(\textsc{DropOff}, \mathsf{did})$ from $\mathcal{S}$ add $(\textsc{DropOff}, \mathsf{A}, \mathsf{B}, \mathsf{mid}, m)$ to $\mathsf{Dropped}$.

Collect On input $(\textsc{Collect}, \mathsf{P}, \mathsf{Q}, \mathsf{mid})$ from Q, where some $(\textsc{DropOff}, \mathsf{P}, \mathsf{Q}, \mathsf{mid}, m) \in \mathsf{Dropped}$, return $(\textsc{DropOff}, \mathsf{P}, \mathsf{Q}, \mathsf{mid}, m)$ to Q.

Fig. 9. Functionality $\mathcal{F}_{\text{AAT}}$.

We believe that a very precise and detailed model of these settings which would allow to catch and quantify the above problems would hide the big picture of our model which is meant to be simple and foundational. We therefore went for the very abstract models of the off-chain channels. The model is given in Fig. 9.

Ledger. At the most abstract level a blockchain is an append-only ledger with no associated secret state beyond the list of messages. It just allows several parties to broadcast messages and creates an agreed upon total order on the messages. This is also called total-ordered broadcast and atomic broadcast. We model this as an ideal functionality $\mathcal{F}_{\text{LEDGER}}$ in Fig. 10, where one can broadcast messages and lookup information on the ledger.

Any party can broadcast a message. A very unrealistic aspect of our model is that all parties see the same ledger at all times. This is physically impossible. There will always be some difference in the exact physical time at which two parties receiver the message making them update their view. We have for simplicity chosen to not model time and liveness in detail. However, for our construction one can see by inspection that there is no use of this perfect synchronisation feature. Our protocols can also be proven secure in more realistic models of time.

When posting a message m we also assume that a filtering/validity predicate ϕ is posted. It is used to reject invalid messages from being posted. When $\phi(\mathsf{Ledger}, m) = \top$ then we say that m is valid to be appended to Ledger, otherwise it is invalid. We assume that the ledger only post message in valid positions, i.e., for all ways to write the ledger as $\mathsf{Ledger} = \mathsf{Ledger}' \| (m, \phi) \| \mathsf{Ledger}''$ it holds that $\phi(\mathsf{Ledger}', m) = \top$. If we post a message without mentioning ϕ, then we are tacitly using the constantly true $\phi = \top$. We note that not all blockchains allow filtering. However, most modern blockchain allows some notion of smart contract. One can implement filtering by inputting the message to a smart contract which accept the message only if it is valid. The filtering functions we use are relatively simple, so it could be practical on most blockchains.

We have a read command, where an account holder can choose to read only part of the blockchain. This is done by submitting a function R and getting back $R(\mathsf{Ledger})$. The function R will be leaked to the adversary. In practice this could be implemented by the account holder submitting R to one or more reader nodes of the blockchain and getting back (an authenticated version of) $R(\mathsf{Ledger})$. For this implementation to be secure it is important that we let $\mathcal{F}_{\text{LEDGER}}$ leak R. We require that R is monotone in the following sense. The output of R is $\perp$ or a bit-string. If $R(\mathsf{Ledger}) \neq \perp$ then for all Ledger' which are extensions of Ledger it holds that

$R(\mathsf{Ledger}') = R(\mathsf{Ledger})$. Think of $\perp$ as having tried to read a part of the blockchain that did not exist yet. We call such R lookup functions.

The ledger $\mathcal{F}_{\text{LEDGER}}$ interacts with parties P and the adversary $\mathcal{A}$.

Init (Internal Command) Initialize an empty list Ledger and empty sets $\mathsf{InTransit}, \mathsf{Proven}$.

Broadcast On input $(\text{BROADCAST}, m, \phi)$ from P, where $m \in \{0,1\}^*$ is the message and $\phi : (\{0,1\}^*)^* \to \{\top, \perp\}$ is a PPT filtering predicate, leak $(\text{BROADCAST}, \mathsf{P}, m, \phi)$ to the adversary and add (P, m, ϕ) to $\mathsf{InTransit}$. We assume ϕ is given in some representation ensuring that it is PPT.

Callback Broadcast On input $(\text{BROADCAST}, \mathsf{P}, m, \phi)$ from $\mathcal{A}$, where $(\mathsf{P}, m, \phi) \in \mathsf{InTransit}$ and $\phi(\mathsf{Ledger}, m) = \top$ and $(\mathsf{P}, m, \phi) \notin \mathsf{Ledger}$, append (m, ϕ) to Ledger.

Read On input (READ, R) from P, where R is a lookup function, leak $(\text{READ}, \mathsf{P}, R)$ to the adversary and return $(\text{READ}, R, R(\mathsf{Ledger}))$ to P.

Prove Valid On input $(\text{PROVEVALID}, p)$ from P, where $\mathsf{Ledger}[p] \neq \perp$, send $(\text{PROVEVALID}, \mathsf{P}, p)$ to $\mathcal{A}$ and get back $\mathsf{TxProof} \in \{0,1\}^\lambda$. Add $(p, \mathsf{Ledger}[p], \mathsf{TxProof})$ to Proven and return $(\text{PROVEVALID}, p, \mathsf{TxProof})$ to P.

Verify Valid On input $(\text{VERVALID}, p, (m, \pi), \mathsf{TxProof})$ from P return $(\text{VERVALID}, p, (m, \pi), J \in \{\top, \perp\})$ to P where $J = \top$ if and only if $(p, (m, \phi), \mathsf{TxProof}) \in \mathsf{Proven}$.[a]

[a] This command by design does not leak information to the adversary.

Fig. 10. Blackbox Ledger $\mathcal{F}_{\text{LEDGER}}$

Main Theorem Statement. With the model in place we are ready to state or main theorem. Our protocol Π_{ANONPAY} follows the technical overview in the introduction. It uses several standard primitives formalised in Section 2 for completeness, including commitments, encryption, and zero-knowledge. It also uses several new primitives, namely a strongly oblivious read-once memory (SOROM), a compressible random beacon (CRaB), and an anonymous coin-friendly encryption scheme (ANCO).

Theorem 4 (informal). *Under the security of the primitives SOROM* SOROM*, UC ZK proof of knowledge* $\mathsf{NIZKPoK}$*, UC ZK proof of membership* NIZK*, perfectly hiding commitment scheme* Com*, IND-CCA secure public-key encryption scheme* PKE*, IND-CCA secure secret-key encryption scheme* SKE*, ANCO* RPKE*, CRaB* CRaB *the protocol* Π_{ANONPAY} *UC-securely implements* $\mathcal{F}_{\mathsf{AnonPay}}$ *in the hybrid world with ideal functionalities* $\mathcal{F}_{LEDGER}$*,* $\mathcal{F}_{AAT}$*, and* $\mathcal{F}_{SERVICE}$*.*

Unfortunately the page limit does not allow us to formalise all the new primitives nor their implementations. Their formalisations are given in the supplementary material. Here we formalise only the notion of SOROM which is central for giving anonymity against an adversary who can follow its own elements in an oblivious data structure and ANCO which specifies the encryption scheme for encrypting coins.

8 OCash: Anonymous Transfers from Oblivious RAM

We are now finally ready to present OCash. We describe a version of our construction that satisfies weak anonymity here and discuss how to achieve strong anonymity in Section 10.

8.1 Stateful Blockchains

We model a blockchain as an append-only datastructure, which in the following descriptions is a bit cumbersome. We note that we can talk about a blockchain containing a state or datastructure as follows. Consider a datastructure with dataspace D and initial state $d_0 \in D$, and update space U, and read space R. For a datastructure d and an update $u \in U$ we let $d' = \mathsf{Update}(d, u)$ be the datastructure obtained from d by applying u. For a datastructure d and a read $r \in R$ we let $v = \mathsf{Read}(d, r)$ be the value obtained by performing read operation r on d. For a sequence $u = (u_1, \ldots, u_m)$ we let $d = \mathsf{Update}(u)$ be $d = d_m$, where $d_i = \mathsf{Update}(d_{i-1}, u_i)$ for $i = 1, \ldots, m$.

To put a datastructure d on the blockchain we simply post the updates $u_1, \ldots, u_m$ as they are made, with appropriate meta data to signal which data structure they are updating. To perform read $r \in R$, read the blockchain with operation $(\textsc{Read}, L)$ where L is the following lookup function: Retrieve from the blockchain the sequence of updates $u = (u_1, \ldots, u_m)$ performed. Compute $d = \mathsf{Update}(u)$. Return $\mathsf{Read}(d, r)$. In practice one would of course not recompute d on each read. If many reads are performed by the same full node on the network it can maintain d. One can also imagine reading through a reader node which keeps d and returns r, possibly along with a proof that r is the correct value relative to some block on the blockchain. This could significantly reduce the communication complexity of the receiver. This would also be a secure implementation if the reader node is semi-honest, as a read operation $(\textsc{Read}, L)$ leaks L and the identity of the reader to the adversary. It is therefore simulatable to tell the reader node what read operation is being performed on which datastructure. To implement this efficiently will vary between concrete blockchains and is not in scope for present paper.

8.2 Overview

We run in a hybrid model with a ledger $\mathcal{F}_{\textsc{Ledger}}$ and anonymous, authenticated transfer $\mathcal{F}_{\mathrm{AAT}}$. For now $\Pi_{\textsc{Service}}$ is implemented as an ideal functionality $\mathcal{F}_{\textsc{Service}}$. We later discuss how to implement it using an MPC protocol. We also assume a random oracle $\mathcal{O} = \mathcal{G}_{\mathrm{roRO}}$ as formalised in [LR22].

Transfer will be done by sending coins (which are just commitments) containing amounts. When a coin of amount a is created the sender A deducts a from its account. When a coin of amount a is collected the receiver B adds a to its account.

Accounts will be of the form $(\mathsf{A}, c_{\mathsf{A}}, \mathsf{ek}_{\mathsf{A}}, \mathsf{nonce}_{\mathsf{A}})$, where the account holder knows the corresponding secret key $(b_{\mathsf{A}}, \rho_{\mathsf{A}}, \mathsf{dk}_{\mathsf{A}})$. The value A is the public identifier of the account and $c_{\mathsf{A}} = \mathsf{Com.Commit}_{\mathsf{ck}}(b_{\mathsf{A}}; \rho_{\mathsf{A}})$ is a perfectly hiding commitment to account balance b_{A} using randomness ρ_{A} and commitment key ck. The value ek_{A} is the encryption key of an ANCO scheme RPKE (Definition 23), dk_{A} is the corresponding decryption key, and $\mathsf{nonce}_{\mathsf{A}}$ is a nonce incremented for each payment done from account A.

Transfer identifiers will be of the form

$$\mathsf{tid} = \mathsf{Commit}_{\mathsf{ck}}((\mathsf{A}, \mathsf{B}, a, \mathsf{nonce}_{\mathsf{A}}); s),$$

where A is the account of the payer, B is the account of the receiver, ck is the global commitment key, a is the amount, $\mathsf{nonce}_{\mathsf{A}}$ is the unique nonce of the sender incremented whenever A does a payment, and s is a randomiser for the commitment algorithm.

Recall that we want tid's to be unique, which we ensure by enforcing that $\mathsf{nonce}_{\mathsf{A}}$ is fresh in each payment from A and that A actually knows an opening of tid. So, if A manages to use the same tid twice, then it must be the case that

$$\mathsf{Commit}_{\mathsf{ck}}((\mathsf{A},\mathsf{B},a,\mathsf{nonce}_{\mathsf{A}}),s) = \mathsf{Commit}_{\mathsf{ck}}((\mathsf{A},\mathsf{B}',a',\mathsf{nonce}'_{\mathsf{A}}),s')$$

for $\mathsf{nonce}_{\mathsf{A}} \neq \mathsf{nonce}'_{\mathsf{A}}$ and thus A would have broken the commitment's binding property. Similarly, if two different parties used the same tid, a collision must have been found as the account identifiers are in the commitment.

A coin will be an encryption of the tid under the receiver's public key. Each receiver B has an encryption key ek_{B} on the ledger. A coin is of the form

$$\mathsf{coin} \leftarrow \mathsf{RPKE.Enc}(\mathsf{ek}_{\mathsf{B}}, \mathsf{tid}).$$

The payer will put coin on the ledger and deduct its account by a. It proves in ZK that coin contains a tid which opens to a vector of the form $(\mathsf{A}',\mathsf{B}',a',\mathsf{nonce})$ where a' is the amount it deducted from its account and where $\mathsf{A}' = \mathsf{A}$, and $\mathsf{nonce} = \mathsf{nonce}_{\mathsf{A}}$. It then increments $\mathsf{nonce}_{\mathsf{A}}$ on the ledger. Since the commitment is perfectly hiding the proof that it contains consistent values need to be a proof of knowledge. A proof of membership for the statement would be trivial, as tid can in principle be opened to any value.

The key-indistinguishability property of the ANCO scheme RPKE ensures that encryptions under different encryption keys have indistinguishable distributions, thereby ensuring that coin does not leak B. Note that during payment, the payer does not prove that it used the correct key ek_{B} as this is not needed. If a malicious payer uses the incorrect key, then it simply burned a units. The amount will be deducted from the payer's account, but the receiver will not accept the payment.

To collect a payment the receiver B will put tid on the ledger and prove that some coin on the chain contains tid. This proof only needs to be a proof of membership, as the encryptions are perfectly binding. It adds a to its own account and then it shows in zero-knowledge that tid opens to a vector of the form $(\mathsf{A}',\mathsf{B}',a',\mathsf{nonce})$ where a' is the amount a they just added to its account and where $\mathsf{B}' = \mathsf{B}$. Showing that some coin contains tid, shows that at some previous point in time that amount a was paid by some other account. To prevent double spending we post tid during collections and only allow a given tid to be collected once. Since tid's are computationally unique, an honest receiver cannot be prevented from collecting a coin by some other parties using the same tid.

Up to this point, our design does not deviate that much from existing constructions like ZeroCash [BCG+14]. The big deviation is in how one proves that *some* coin on the chain contains tid, which is *a priori* a complicated statement that may naively involve all existing coins. As already outlined in Section 1.3, we rely on techniques from oblivious data structures literature to reduce the statement size. In OCash, a service will regularly mix coins by moving them around inside a SOROM. The service will regularly taking small sets of coins from the ledger, permuting them according to the SOROM, randomizing their encryptions, and writing them back to the ledger. When collecting a coin, we will only touch a small set of coins, instead of touching all of them, as is done by currencies like ZerOCash. The receiver will use their knowledge of a label associated to the coin they are looking for to determine the positions specified by $\mathsf{SOROM.Read}$ and collect the rerandomized coin coin' by revealing the identifier tid inside coin'. The collector then proves that tid is inside one of the coins specified by the label. The receiver now just has to prove in zero-knowledge that tid contains B and that the amount in the coin matches the amount added to the receiver's account. This last proof needs to be a proof of knowledge as tid is perfectly hiding.

8.3 Anonymous Coin-Flip on the Blockchain

Using an SOROM for shuffling, leaves a technical issue to be solved. For the security of a SOROM, it is important that the labels are chosen at random, meaning that we cannot let any one single party pick these labels. In fact, we cannot even let the sender *and* receiver pick the label jointly as they might both be corrupted. We therefore need that Π_{SERVICE} is involved in picking the label. At the same time, we would like to avoid that the payer A needs to wait for the service to come online and act to be able to pay. Ideally, payments should be as quick as posting a single message to the blockchain. Additionally, we ideally would also like a passive receiver. If Alice pays Bob, then Bob need not be online while Alice is. This is important for applications like anonymously paying to a smart contract. We also want Bob to be anonymous in the sense that one cannot link A and B by observing the ledger. Finally, we also do not want Bob to run a full node to receive a payment. Bob should be able to learn the label associated to the coin it receives by reading as little as possible from the blockchain. Let us now sketch the protocol flow of the coin flip protocol that we will use.

1. Initially service samples and stores $k \leftarrow \mathsf{CRaB.Gen}(1^\lambda)$. It also samples $(\mathsf{ek}, \mathsf{dk})$ for a PKE scheme and makes ek public on the ledger.
2. To initiate a coin-flip Alice samples $\mathsf{L_A} \leftarrow \{0,1\}^\lambda$ and broadcasts $d \leftarrow \mathsf{PKE.Enc}(\mathsf{ek}, (\mathsf{A}, \mathsf{L_A}); \rho_4)$. Alice waits for d to appear in some position p on the ledger and anonymously sends $(d, \mathsf{L_A}, \rho_4)$ along with a proof that d was posted in position p to Bob.
3. Bob rejects Alice's message, if $d \neq \mathsf{PKE.Enc_{ek}}((\mathsf{A}, \mathsf{L_A}); \rho_4)$ or if the proof that d is in position p is not valid. Note that Bob does not need to access the ledger for this.
4. Once d appears on the ledger in position p, the service broadcasts $k_p^* \leftarrow \mathsf{CRaB.Prefix}(k, p)$ on the ledger.
5. The service computes $\mathsf{L_S} = \mathsf{CRaB.Eval}(k, p)$, decrypts $(\mathsf{A}, \mathsf{L_A}) = \mathsf{PKE.Dec_{dk}}(d)$, and determines the label $\mathsf{L} = \mathsf{L_A} \oplus \mathsf{L_S}$.
6. Alice waits for k_p^* to appear on the ledger and computes $\mathsf{L_S} = \mathsf{CRaB.Eval}(k_p^*, p)$. Since Alice also knows $\mathsf{L_A}$, she can also determine $\mathsf{L} = \mathsf{L_A} \oplus \mathsf{L_S}$.
7. When Bob wants to learn some value L, he retrieves the latest $k_{p'}^*$ with $p' > p$ posted on the ledger. Bob can then compute $\mathsf{L_S} = \mathsf{CRaB.Eval}(k_{p'}^*, p)$ and since he received $\mathsf{L_A}$ from Alice, he can also compute $\mathsf{L} = \mathsf{L_A} \oplus \mathsf{L_S}$. Note that Bob reads the latest $k_{p'}^*$ and not k_p^*, thereby ensuring that the position p cannot be linked to Bob.

The above protocol has all the properties we need. Alice, Bob, and the service learn L, whereas any external observer does not. The coin-flip is random, even when both Alice and Bob are corrupt. Bob does not need to be active while Alice and the service are. Bob can learn the desired label L by reading the latest succinct prefix key from the ledger.

We note that in our formal description, we do not prove any properties of the above protocol in isolation, but rather as part of the overall security proof of OCash. We also note that the parties cannot learn a coin, until the service was active. In our payment application, this means Bob cannot collect a coin, until the service was active. This might seem disappointing, since we wanted to avoid that the service was active before the payment was done. There is, however, no avoiding this, as we want the outcome to be random, even if Alice and Bob are corrupt. It is therefore optimal to only have *collection* and not *payment* be blocked by a slow service. This is particularly the case, because anonymity requires that Bob waits some time before picking up the coin.

8.4 Relations for Zero-Knowledge

Our construction will make use of zero-knowledge proofs for several distinct relations. In the following, let us formally define those relations. The relation $\mathcal{R}_{\text{IsZero}}$ is for proving that a commitment indeed commits to 0, i.e.,

$$(x = (\mathsf{ck}, c), w = \rho) \in \mathcal{R}_{\text{IsZero}} \iff c = \mathsf{Com.Commit}_{\mathsf{ck}}(0, \rho)$$

and the relation $\mathcal{R}_{\text{IsFund}}$ is a generalization thereof that allows for proving that a commitment commits to a specific (non-zero) amount a_0, i.e.,

$$(x = (\mathsf{ck}, c), w = \rho) \in \mathcal{R}_{\text{IsFund}} \iff c = \mathsf{Com.Commit}_{\mathsf{ck}}(a_0, \rho).$$

The relation $\mathcal{R}_{\text{OrDec}}$ is for showing that one out of several ciphertexts contains a given plaintext:

$$(x = (\mathsf{pp}, \mathsf{ek}, \{c_j\}_{j\in 1}^{\ell}, m), w = \mathsf{dk}) \in \mathcal{R}_{\text{OrDec}} \iff \bigvee_{j=1}^{\ell} \mathsf{RPKE.Dec}_{\mathsf{dk}}(c_j) = m \ .$$

The relation $\mathcal{R}_{\text{Collect}}$ is for proving that the receiver updated their account correctly during collection. This is done by showing that a commitment c'_{B} is a valid commitment to the sum of the previous account balance committed in c_{B} and the amount of money committed in transaction tid, i.e.,

$$\begin{aligned}(x = (\mathsf{ck}, c_{\mathsf{B}}, \mathsf{B}, \mathsf{tid}, c'_{\mathsf{B}}), w = (b_{\mathsf{B}}, , (\mathsf{A}, a, \mathsf{nonce}_{\mathsf{A}}, \rho_1), \rho_{\mathsf{B}}, \rho'_{\mathsf{B}})) \in \mathcal{R}_{\text{Collect}}\\ \iff c_{\mathsf{B}} = \mathsf{Com.Commit}_{\mathsf{ck}}(b_{\mathsf{B}}, \rho_{\mathsf{B}}) \wedge\\ \mathsf{tid} = \mathsf{Com.Commit}_{\mathsf{ck}}(\mathsf{A}, \mathsf{B}, a, \mathsf{nonce}_{\mathsf{A}}, \rho_1) \wedge\\ c'_{\mathsf{B}} = \mathsf{Com.Commit}_{\mathsf{ck}}(b_{\mathsf{B}} + a, \rho'_{\mathsf{B}}) \ .\end{aligned}$$

Lastly, the relation $\mathcal{R}_{\text{Pay}}$ allows for proving correctness of a payment by showing that the payer's account balance c_{A} is being correctly updated to c'_{A} with respect to the amount of money a tied up in the generated coin coin, i.e.,

$$\begin{aligned}(x = (\mathsf{A}, \mathsf{nonce}_{\mathsf{A}}, \mathsf{ck}, c_{\mathsf{A}}, \mathsf{coin}, c'_{\mathsf{A}}), w = ((b_{\mathsf{A}}, \rho_{\mathsf{A}}), (\mathsf{B}, a, \rho_1, \rho_2), \rho'_{\mathsf{A}})) \in \mathcal{R}_{\text{Pay}} \iff\\ c_{\mathsf{A}} = \mathsf{Com.Commit}_{\mathsf{ck}}(b_{\mathsf{A}}, \rho_{\mathsf{A}}) \wedge\\ ((\mathsf{coin}, \mathsf{Com.Commit}_{\mathsf{ck}}(\mathsf{A}, \mathsf{B}, a, \mathsf{nonce}_{\mathsf{A}}; \rho_1)), \rho_2) \in \mathcal{R}_{\text{Enc}} \wedge\\ c'_{\mathsf{A}} = \mathsf{Com.Commit}_{\mathsf{ck}}(b_{\mathsf{A}} - a, \rho'_{\mathsf{A}}) \wedge b_{\mathsf{A}} \geq a \geq 0 \ .\end{aligned}$$

We provide concretely efficient proof systems, satisfying the properties we need, for all these relations in Section 11.

8.5 OCash Protocol

We now proceed to provide pseudocode for the OCash protocol. In our pseudocode we will assume that one can see who posts which messages on the ledger. In practice this would involve putting a public key of a signature scheme in each account and signing messages from the account using the corresponding secret key. For the sake of clarity, we do not deal with this explicitly in our pseudocode. We will use both a GUC proof of knowledge $\mathsf{NIZKPoK}$ and a UC proof of membership

NIZK. We have a single crs which is the pair of CRSs implicitly. We let both schemes use crs and tacitly assume they pick the part they need.

In Fig. 11, we show how the initialization part of the anonymizer service is done. In Fig. 12, we provide the pseudocode for creating accounts. Initially, there are no accounts and there is no money on the blockchain. For the sake of simplicity, we simply provide the first party that generates an account with an initial account balance of a_0. Once this account is created, all other parties can only create accounts that have an account balance of 0 initially. How money is bootstrapped into a blockchain in the real world is a non-cryptographic process and using our formal modeling outlined above, we have abstracted this process away in our work.

In Fig. 13 we give the pseudocode for initiating a payment by putting a coin on the ledger. In Fig. 14 we give the pseudocode for the part of the service doing the mixing of the coins. Finally, in Fig. 15 we give the code for observing and collecting a payment.

```
Init  When activated the first time do the following:
  1. Let $\mathsf{crs}^{\text{KNOW}} \leftarrow \mathsf{NIZKPoK.Gen}(1^\lambda)$, $\mathsf{crs}^{\text{MEMB}} \leftarrow \mathsf{NIZK.Gen}(1^\lambda)$, and $\mathsf{crs} = (\mathsf{crs}^{\text{KNOW}}, \mathsf{crs}^{\text{MEMB}})$.
     *This is the setup for proving statements to the ledger in ZK.*
  2. $\mathsf{ck} \leftarrow \mathsf{Com.Gen}(1^\lambda)$. *This is the key for creating transfer IDs and committing to balances.*
  3. $(\mathsf{ek}, \mathsf{dk}) \leftarrow \mathsf{PKE.Gen}(1^\lambda)$ and save $\mathsf{dk}$. *This is the key for sending secret messages to the committee.*
  4. $\mathsf{pp} \leftarrow \mathsf{RPKE.Params}(1^\lambda)$ *This is the public parameters for the randomisable public-key encryption used for encrypting coins.*
  5. Initialise an empty set $\mathsf{Spent}$ on $\mathcal{F}_{\text{LEDGER}}$. *This is the set of spent transaction identifiers.*
  6. Set up the key material for anonymous coin-flip on the blockchain.
     (a) $k \leftarrow \mathsf{CRaB.Gen}(1^\lambda)$ and save $k$.
     (b) $k^* \leftarrow \mathsf{Prefix}(k, 0)$.
     (c) Let $p_{\text{UPDATED}} = 0$ be the last updated position.
  7. Post $(\mathsf{crs}, \mathsf{ck}, \mathsf{ek}, \mathsf{pp}, k^*, p_{\text{UPDATED}})$ on $\mathcal{F}_{\text{LEDGER}}$.
  8. Set up the key material for running the SOROM on the blockchain.
     (a) $\iota = 0$.
     (b) $K \leftarrow \mathsf{SKE.Gen}(1^\lambda)$ and save $K$. *This is the key under which the service encrypt labels for the SOROM.*
     (c) Let $N = 2^\lambda$ and initialise on the blockchain an oblivious map $\mathsf{OM}$ where for each $i \in [N]$ the service store $\mathsf{OM}[i] = (\mathsf{lab}_i, \mathsf{coin}_i)$ where $\mathsf{lab}_i$ is an encryption under $K$ or $\perp$ and $\mathsf{coin}_i$ is a coin or $\perp$. Initially let $\mathsf{OM}[i] = (\perp, \perp)$ for all $i$.
```

Fig. 11. Ideal Functionality $\mathcal{F}_{\text{SERVICE}}$.

8.6 Proving Security in the UC Framework

We will prove the following statement:

Theorem 5. *Under the security of the primitives* SOROM, NIZKPoK, NIZK, Com, PKE, RPKE, CRaB, *and* SKE *the protocol* $\Pi_{AnonPay}$ *UC-securely implements* $\mathcal{F}_{\mathsf{AnonPay}}$ *with weak anonymity against static adversaries in the hybrid world with ideal functionalities* $\mathcal{F}_{LEDGER}$, $\mathcal{F}_{AAT}$, *and* $\mathcal{F}_{SERVICE}$.

The design rational for the protocol has already been discussed above. The proof of security follows the intuition fairly closely, but via an intricate sequence of hybrids. As TID distribution the simulator uses random commitments to 0, i.e., $\mathsf{Tid} \leftarrow \mathsf{Commit}_{\mathsf{ck}}(0)$. This will give honest tid

Create Account On input ($\textsc{CreateAccount}$) from P, where there are already other accounts on the ledger, it proceeds as follows.

1. Wait until ck, crs and pp appear on $\mathcal{F}_{\text{Ledger}}$.
2. $c_{\mathsf{A}} \leftarrow \mathsf{Com.Commit}_{\mathsf{ck}}(0; \rho)$.
3. $\pi \leftarrow \mathsf{NIZKPoK.Prv}_{\mathsf{crs}}(\mathcal{R}_{\text{IsZero}}, (\mathsf{ck}, c_{\mathsf{A}}), \rho_{\mathsf{A}})$.
4. $(\mathsf{ek}_{\mathsf{A}}, \mathsf{dk}_{\mathsf{A}}) \leftarrow \mathsf{RPKE.Gen}(\mathsf{pp})$.
5. Let $\mathsf{nonce}_{\mathsf{A}} = 0$.
6. Run $\mathcal{F}_{\text{Ledger}}.\textsc{Broadcast}(\mathsf{Tx}, \phi_{\text{Tx}})$ with $\mathsf{Tx} = (c_{\mathsf{A}}, \mathsf{ek}_{\mathsf{A}}, \mathsf{nonce}_{\mathsf{A}}, \pi)$ and
   Filtering Function $\phi_{\text{Tx}}(\mathsf{Ledger}, \mathsf{Tx}) \equiv$ Parse $(c_{\mathsf{A}}, \mathsf{ek}_{\mathsf{A}}, \mathsf{nonce}_{\mathsf{A}}, \pi) = \mathsf{Tx}$, fetch $(\mathsf{crs}, \mathsf{ck})$ from Ledger and check that $\mathsf{NIZKPoK.Ver}_{\mathsf{crs}}(\mathcal{R}_{\text{IsZero}}, (\mathsf{ck}, c_{\mathsf{A}}), \pi) = \top$, that $\mathsf{nonce}_{\mathsf{A}} = 0$, and that some other account already appears in Ledger.
7. When $\mathsf{Tx} = (c_{\mathsf{A}}, \mathsf{ek}_{\mathsf{A}}, \mathsf{nonce}_{\mathsf{A}}, \pi)$ appears on $\mathcal{F}_{\text{Ledger}}$ in some position p, compute $\mathcal{F}_{\text{Ledger}}.\textsc{ProveValid}(p) \rightarrow \mathsf{TxProof}$ and let $\mathsf{A} = (\mathsf{ek}_{\mathsf{A}}, \phi_{\text{Acc}}, p, \mathsf{TxProof})$. Then output $(\textsc{CreateAccount}, \mathsf{A})$ to P and save $(\mathsf{A}, b_{\mathsf{A}} = 0, c_{\mathsf{A}}, \rho_{\mathsf{A}}, \mathsf{ek}_{\mathsf{A}}, \mathsf{dk}_{\mathsf{A}}, \mathsf{nonce}_{\mathsf{A}})$. This is the initial secret state of the account. This establishes the invariant that for the current balance b_{A} of account A party P knows ρ_{A} such that $c_{\mathsf{A}} = \mathsf{Com.Commit}_{\mathsf{ck}}(b_{\mathsf{A}}, \rho_{\mathsf{A}})$.
   Store the updated account $(\mathsf{A}, c_{\mathsf{A}}, \mathsf{ek}_{\mathsf{A}}, \mathsf{nonce}_{\mathsf{A}}, \pi)$ on $\mathcal{F}_{\text{Ledger}}$.[a]

Create Account (Initial Funding Account) On input ($\textsc{CreateAccount}$) from P, where so far there are no other accounts on the ledger, it proceeds as above with the following changes.

2. $c_{\mathsf{A}} \leftarrow \mathsf{Com.Commit}_{\mathsf{ck}}(a_0; \rho)$.
3. $\pi \leftarrow \mathsf{NIZKPoK.Prv}_{\mathsf{crs}}(\mathcal{R}_{\text{IsFund}}, (\mathsf{ck}, c_{\mathsf{A}}), \rho_{\mathsf{A}})$.
6. Use
   Filtering Function $\phi_{\text{Tx}}(\mathsf{Ledger}, \mathsf{Tx}) \equiv$ Parse $(c_{\mathsf{A}}, \mathsf{ek}_{\mathsf{A}}, \mathsf{nonce}_{\mathsf{A}}, \pi) = \mathsf{Tx}$, fetch $(\mathsf{crs}, \mathsf{ck})$ from Ledger and check that $\mathsf{NIZKPoK.Ver}_{\mathsf{crs}}(\mathcal{R}_{\text{IsFund}}, (\mathsf{ck}, c_{\mathsf{A}}), \pi) = \top$, that $\mathsf{nonce}_{\mathsf{A}} = 0$, and that until now no other account appears in Ledger.
7. Save $(\mathsf{A}, b_{\mathsf{A}} = a_0, c_{\mathsf{A}}, \rho_{\mathsf{A}}, \mathsf{ek}_{\mathsf{A}}, \mathsf{dk}_{\mathsf{A}}, \mathsf{nonce}_{\mathsf{A}})$.

[a] When we say that we store a value which can already be computed from the values on the ledger, we mean that we use the "stateful ledger" abstraction from Section 8.1 to define and later fetch the value.

Fig. 12. Pseudocode of creator P. Part of $\mathit{\Pi}_{\text{AnonPay}}$.

```
Initiate Pay  On input $(\textsc{Pay}, \mathsf{A}, \mathsf{B}, a)$ to $\mathsf{P_A}$ proceeds as follows.
  1. Fetch $(\mathsf{A}, b_{\mathsf{A}}, c_{\mathsf{A}}, \rho_{\mathsf{A}}, \mathsf{ek_A}, \mathsf{dk_A}, \mathsf{nonce_A})$ from local storage and terminate if $a > b_{\mathsf{A}}$. Get
     $(\mathsf{A}, b_{\mathsf{A}}, c_{\mathsf{A}}, \rho_{\mathsf{A}}, \mathsf{ek_A}, \mathsf{dk_A}, \mathsf{nonce'_A})$ from $\mathcal{F}_{\textsc{Ledger}}$ and terminate if $\mathsf{nonce'_A} \neq \mathsf{nonce_A}$. *Check that there
     is balance and that the local information is up to date. This can happen not to be the case if two
     payments were started in parallel.*
  2. Get the public values $(\mathsf{crs}, \mathsf{ek}, \mathsf{ck}, k^*)$ of the service from $\mathcal{F}_{\textsc{Ledger}}$.
  3. Parse $(\mathsf{ek_B}, \phi_{\textsc{Acc}}, p, \mathsf{TxProof_B}) = \mathsf{B}$ and run $\mathcal{F}_{\textsc{Ledger}}.\textsc{VerValid}(p, (\mathsf{ek_B}, \phi_{\textsc{Acc}}), \mathsf{TxProof_B}) \to J$. If $J = \bot$,
     then abort. *Check* $\mathsf{B}$*'s account exists.*
  4. Compute the transfer identifier $\mathsf{tid} = \mathsf{Com.Commit_{ck}}((\mathsf{A}, \mathsf{B}, a, \mathsf{nonce_A}); \rho_1)$.
  5. Compute the coin $\mathsf{coin} = \mathsf{RPKE.Enc_{ek_B}}(\mathsf{tid}; \rho_2)$.
  6. Let $b'_{\mathsf{A}} = b_{\mathsf{A}} - a$, compute $c'_{\mathsf{A}} = \mathsf{Com.Commit_{ck}}(b'_{\mathsf{A}}, \rho'_{\mathsf{A}})$.
  7. $\pi \leftarrow \mathsf{NIZKPoK.Prv_{crs}}(\mathcal{R}_{\textsc{Pay}}, (c_{\mathsf{A}}, \mathsf{coin}, c_{\mathsf{A'}}, \mathsf{A}, \mathsf{nonce_A}), ((b_{\mathsf{A}}, \rho_{\mathsf{A}}), (\mathsf{B}, a, \rho_1, \rho_2), \rho'_{\mathsf{A}}))$.
  8. Sample $\mathsf{L_A} \leftarrow \{0,1\}^{\lambda}$ , compute $d \leftarrow \mathsf{PKE.Enc_{ek}}((\mathsf{A}, \mathsf{L_A}); \rho_4)$.
  9. Run $\mathcal{F}_{\textsc{Ledger}}.\textsc{Broadcast}(\mathsf{Tx}, \phi_{\textsc{Pay}})$ with $\mathsf{Tx} = (\mathsf{A}, c'_{\mathsf{A}}, \mathsf{coin}, \pi, d)$ and
     Filtering Function $\phi_{\textsc{Pay}}(\mathsf{Ledger}, \mathsf{Tx}) \equiv$ Parse $(\mathsf{A}, c'_{\mathsf{A}}, \mathsf{coin}, \pi, d) = \mathsf{Tx}$,
         fetch $(\mathsf{crs}, \mathsf{ek}, k^*)$ and $(\mathsf{A}, c_{\mathsf{A}}, \mathsf{ek_A}, \mathsf{nonce_A})$ from $\mathsf{Ledger}$ and check that
         $\mathsf{NIZKPoK.Ver_{crs}}(\mathcal{R}_{\textsc{Pay}}, (c_{\mathsf{A}}, \mathsf{coin}, c_{\mathsf{A'}}, \mathsf{A}, \mathsf{nonce_A}), \pi) = \top$.
 10. Wait for $(\mathsf{Tx}, \phi_{\textsc{Pay}})$ to appear on $\mathcal{F}_{\textsc{Ledger}}$ in some position $p$. When this happens the blockchain stateful
     abstraction layer replaces $(\mathsf{A}, c_{\mathsf{A}}, \mathsf{ek_A}, \mathsf{nonce_A})$ by $(\mathsf{A}, c'_{\mathsf{A}}, \mathsf{ek_A}, \mathsf{nonce'_A} = \mathsf{nonce_A} + 1)$. *Note that if* $\mathsf{Tx}$ *is
     rejected or never posted, the transaction deadlocks here.*
 11. The first time when $\mathsf{P_A}$ is activated again, store $(\mathsf{A}, b'_{\mathsf{A}}, c'_{\mathsf{A}}, \rho'_{\mathsf{A}}, \mathsf{ek_A}, \mathsf{dk_A}, \mathsf{nonce'_A} = \mathsf{nonce_A} + 1)$.
 12. Run $\mathcal{F}_{\textsc{Ledger}}.\textsc{ProveValid}(p) \to \mathsf{TxProof}$.
 13. Run $\mathcal{F}_{\textsc{AAT}}.\textsc{DropOff}(\mathsf{P_A}, \mathsf{P_B}, \mathsf{tid}, m)$ where

         $m = ((\mathsf{Tx}, p, \mathsf{TxProof}, (\mathsf{nonce_A}, \rho_1, \rho_2)), (\mathsf{L_A}, \rho_4))$ .

 14. Output $(\textsc{Pay}, \mathsf{A}, \mathsf{B}, a, \mathsf{tid})$ to $\mathsf{P_A}$.
```

Fig. 13. Pseudocode of payer A. Part of $\Pi_{\textsc{AnonPay}}$.

```
Receive Coin  Whenever a new payment $(\mathsf{Tx}, \phi_{\text{PAY}})$ appears in some position $p$, proceed as follows. Transactions must be consumed in order of increasing $p$.
    1. Parse $(\mathsf{A}, c'_{\mathsf{A}}, \mathsf{coin}, \pi, d) \leftarrow \mathsf{Tx}$.
    2. Keep track of the number of payments so far: $\iota = \iota + 1$.
    3. Compute $\mathsf{P_A}$'s contribution to the coinflip: $(\mathcal{C}, \mathsf{L_A}) = \mathsf{PKE.Dec_{dk}}(d)$. If $\mathcal{C} \neq \mathsf{A}$ then terminate. [a]
    4. Service's contribution to the coinflip: $\mathsf{L_S} = \mathsf{CRaB.Eval}(k, p)$, where $p$ is the position of $\mathsf{Tx}$ on the ledger.
    5. Coinflip: $\mathsf{L} = \mathsf{Hash}(\mathsf{L_A} \oplus \mathsf{L_S})$.
    6. Encrypted coinflip: $\mathsf{lab} = \mathsf{SKE.Enc}_K(L)$.
    7. Updated CRaB key: $k^* \leftarrow \mathsf{CRaB.Prefix}(k, p)$ and $p_{\text{UPDATED}} = p$.
    8. Update $k^*$, $p_{\text{UPDATED}}$, and $\mathsf{OM}[0] = (\mathsf{lab}, \mathsf{coin})$ on $\mathcal{F}_{\text{LEDGER}}$.
    9. Go to Route
Route
    1. Compute $(j_1, \ldots, j_\ell) = \mathsf{SOROM.Pos}(\iota)$.
    2. For $k = 1, \ldots, \ell$ read $(\mathsf{lab}_k, \mathsf{coin}_k) \leftarrow \mathsf{OM}[j_k]$ from $\mathcal{F}_{\text{LEDGER}}$.
    3. For $k = 1, \ldots, \ell$ let $L_k = \mathsf{SKE.Dec}_K(\mathsf{lab}_k)$.
    4. Compute the routing permutation: $\pi = \mathsf{SOROM.Route}(\iota, L_1, \ldots, L_\ell)$.
    5. For $k = 1, \ldots, \ell$ let $\mathsf{lab}'_k \leftarrow \mathsf{SKE.Enc}_K(L_{\pi(k)})$.
    6. For $k = 1, \ldots, \ell$ let $\mathsf{coin}'_k \leftarrow \mathsf{RPKE.Ran}(\mathsf{coin}_{\pi(k)})$.
    7. For $k = 1, \ldots, \ell$ update $\mathsf{OM}[j_k] \leftarrow (\mathsf{lab}'_k, \mathsf{coin}'_k)$ on $\mathcal{F}_{\text{LEDGER}}$.
```

[a] This prevents replay attacks with the d's.

Fig. 14. Pseudocode of service $\mathcal{F}_{\text{SERVICE}}$

the same distribution as in the protocol. During the simulation the simulator will simulate the proofs that the honest tid contain the right values. For corrupted parties the simulator uses the tid produced in the simulated protocol. The use of binding commitments to compute tid ensures that tid's cannot be reused. This in turn ensures that if B observes a payment it will also eventually be able to pick it up. The correctness of SOROM ensures that no encryptions of new tid's can be introduced. Therefore each tid collected can be linked to a unique payment. The proofs ensure that it is of the same amount a. We can use extractability of the proofs to extract the link between payments and let the simulator input these to $\mathcal{F}_{\mathsf{AnonPay}}$ in the simulation to make it do the same transfers as the simulated protocol. The fact that tid is perfectly hiding ensures that tid cannot be linked to A or a. The security of the coin-flip into the well ensures that the labels are random, which ensures the correctness and strong obliviousness of the SOROM. In proving the coin-flipping secure it is crucial that the PKE is IND-CPA and that the encryption contains the name of A as it ensures that A's contribution $\mathsf{L_A}$ is independent of the contribution of other parties. Therefore the label are random and *independent*. In doing the reduction to the strong obliviousness of the SOROM we model Hash as a random oracle in $\mathsf{L} = \mathsf{Hash}(\mathsf{L_A} \oplus \mathsf{L_S})$, which allows us to embed the label we get from to SOROM game into the simulation by programming Hash. The strong obliviousness of the SOROM ensures that the label L posted in collection does not leak anything about when the collected coin was added to the SOROM. In the simulation we can therefore for transfers between honest parties make dummy payments and collections of 0 and randomly map collections to payments. This will have the same distribution in the view of the environment. We defer the full proof to Section 9.

Observe On input $(\textsc{Observe}, \mathsf{tid}, \mathsf{A}, \mathsf{B}, a)$ to $\mathsf{P_B}$ it proceeds as follows. Fetch $(\mathsf{B}, b_\mathsf{B}, c_\mathsf{B}, \rho_\mathsf{B}, \mathsf{ek_B}, \mathsf{dk_B}, \mathsf{nonce_B})$ from local storage. Run $\mathcal{F}_{\text{AAT}}.\textsc{Collect}(\mathsf{P_A}, \mathsf{P_B}, \mathsf{tid}) \to m$. If

$$m = ((\mathsf{Tx} = (\mathsf{A}, \cdot, \mathsf{coin}, \cdot, d), p, \mathsf{TxProof}, (\mathsf{nonce_A}, \rho_1, \rho_2)), (\mathsf{L_A}, \rho_4))$$

where

$$\mathcal{F}_{\text{Ledger}}.\textsc{VerValid}((\mathsf{Tx}, p, \phi_{\text{Pay}}), \mathsf{TxProof}) \to \top$$
$$\mathsf{tid} = \mathsf{Com.Commit_{ck}}((\mathsf{A}, \mathsf{B}, a, \mathsf{nonce_A}), \rho_1)$$
$$\mathsf{coin} = \mathsf{RPKE.Enc_{ek_B}}(\mathsf{tid}, \rho_2)$$
$$d = \mathsf{PKE.Enc_{ek}}((\mathsf{A}, \mathsf{L_A}), \rho_4)$$

then return $(\textsc{Observe}, \mathsf{tid}, \mathsf{A}, \mathsf{B}, a, \top)$.

Collect On input $(\textsc{Collect}, \mathsf{tid}, \mathsf{A}, \mathsf{B}, a)$ to $\mathsf{P_B}$ it proceeds as follows.

1. First run $(\textsc{Observe}, \mathsf{tid}, \mathsf{A}, \mathsf{B}, a)$ if this was not already done. If the result is $(\textsc{Observe}, \mathsf{tid}, \mathsf{A}, \mathsf{B}, \top)$, then proceed as below, using the values defined while running $(\textsc{Observe}, \mathsf{tid}, \mathsf{A}, \mathsf{B}, a)$. [a]
2. Fetch the $\mathsf{L_A}$ where $d = \mathsf{PKE.Enc_{ek}}((\mathsf{A}, \mathsf{L_A}), \rho_4)$ and the position p.
3. From $\mathcal{F}_{\text{Ledger}}$ fetch the most recent CRaB key k^* and position p_{updated}. If $p_{\text{updated}} < p$ terminate. [b]
4. Let $\mathsf{L_S} = \mathsf{CRaB.Eval}(k^*, p)$.
5. Let $\mathsf{L} = \mathsf{Hash}(\mathsf{L_A} \oplus \mathsf{L}_S)$.
6. Compute $\mathsf{pos} = \mathsf{SOROM.Pos}(\mathsf{L})$.
7. Fetch the data at positions $j \in \mathsf{pos}$ in OM from $\mathcal{F}_{\text{Ledger}}$ and for $j \in \mathsf{pos}$ let $\mathsf{OM}[j] = (\cdot, \mathsf{coin}_j)$. Let $j_0 \in \mathsf{pos}$ be the position where $\mathsf{OM}[j_0] = (\cdot, \mathsf{coin}')$ and $\mathsf{RPKE.Dec_{dk_B}}(\mathsf{coin}') = \mathsf{tid}$.
8. $\pi_1 \leftarrow \mathsf{NIZK.Prv_{crs}}(\mathcal{R}_{\text{OrDec}}, (\mathsf{ck}, \{\mathsf{coin}_j\}_{j \in \mathsf{pos}}, \mathsf{tid}), \mathsf{dk_B})$.
9. Let $b'_\mathsf{B} = b_\mathsf{B} + a$, compute $c'_\mathsf{B} = \mathsf{Com.Commit}(b'_\mathsf{B}, \rho'_\mathsf{B})$.
10. $\pi_2 \leftarrow \mathsf{NIZKPoK.Prv_{crs}}(\mathcal{R}_{\text{Collect}}, (\mathsf{ck}, c_\mathsf{B}, \mathsf{B}, \mathsf{tid}, c'_\mathsf{B}), ((b_\mathsf{B}, \rho_\mathsf{B}), (\mathsf{A}, a, \mathsf{nonce_A}, \rho_1), \rho'_\mathsf{B}))$.
11. Run $\mathcal{F}_{\text{Ledger}}.\textsc{Broadcast}(\mathsf{Tx}, \phi_{\text{Collect}})$ with $\mathsf{Tx} = (\mathsf{B}, \mathsf{L}, \mathsf{tid}, \pi_1, \pi_2)$ and
    Filtering Function $\phi_{\text{Collect}}(\mathsf{Ledger}, \mathsf{Tx}) \equiv$ Parse $(\mathsf{B}, \mathsf{L}, \mathsf{tid}, \pi_1, \pi_2) = \mathsf{Tx}$, fetch Spent, OM, $(\mathsf{crs}, \mathsf{ek}, \mathsf{ck}, k^*)$ and $(\mathsf{B}, c_\mathsf{B}, \mathsf{ek_B}, \mathsf{nonce_B})$ from Ledger along with $\{\mathsf{coin}_j\}_{j \in \mathsf{pos}}$ for $\mathsf{pos} = \mathsf{SOROM.Pos}(\mathsf{L})$ and check that

$$\mathsf{NIZK.Ver_{crs}}(\mathcal{R}_{\text{OrDec}}, (\mathsf{ck}, \{\mathsf{coin}_j\}_{j \in \mathsf{pos}}, \mathsf{tid}), \pi_1) = \top$$
$$\mathsf{NIZKPoK.Ver_{crs}}(\mathcal{R}_{\text{Collect}}, (\mathsf{ck}, c_\mathsf{B}, \mathsf{B}, \mathsf{tid}, c'_\mathsf{B}), \pi_2) = \top$$
$$(\mathsf{B}, \mathsf{tid}) \notin \mathsf{Spent}$$

12. When the transaction is posted the blockchain stateful abstraction layer replaces $(\mathsf{B}, c_\mathsf{B}, \mathsf{ek_B}, \mathsf{nonce_B})$ by $(\mathsf{B}, c'_\mathsf{B}, \mathsf{ek_B}, \mathsf{nonce_B})$ and $\mathsf{Spent} = \mathsf{Spent} \cup \{\mathsf{tid}\}$.
13. Store $(\mathsf{B}, b'_\mathsf{B}, c'_\mathsf{B}, \rho'_\mathsf{B}, \mathsf{ek_B}, \mathsf{dk_B}, \mathsf{nonce_B})$ on local storage.

[a] If the result $(\textsc{Observe}, \mathsf{tid}, \mathsf{A}, \mathsf{B}, \top)$ is not returned it corresponds to the case where $\mathcal{F}_{\mathsf{AnonPay}}$ aborts the collection.

[b] This corresponds to the case where $\mathcal{F}_{\mathsf{AnonPay}}$ returns Too Early.

Fig. 15. Pseudocode of receiver B. Part of Π_{AnonPay}.

9 Proof of Theorem 5

In this section, we provide the full proof of Theorem 5. We only give full simulation-based security proofs for the core features and weak anonymity. We separately discuss how to extend the analysis to cover strong anonymity and some other extensions. Recall that to prove security in the UC framework we need to construct a simulator $\mathcal{S}$ such that for all environments $\mathcal{E}$ it holds that

$$\mathrm{Exec}_{\mathcal{F}_{\mathsf{AnonPay}},\mathcal{S},\mathcal{E}}(1^\lambda) \approx \mathrm{Exec}_{\Pi_{\textsc{AnonPay}},\mathcal{F}_{\textsc{Ledger}},\mathcal{F}_{\mathrm{AAT}},\mathcal{F}_{\textsc{Service}},\mathcal{E}}(1^\lambda)\ . \tag{2}$$

Note that we prove security for the dummy adversary and therefore leave it out of the notation.

In $\mathrm{Exec}_{\mathcal{F}_{\mathsf{AnonPay}},\mathcal{S},\mathcal{E}}(1^\lambda)$ it is the environment which gives inputs to $\mathcal{F}_{\mathsf{AnonPay}}$ and receives outputs from $\mathcal{F}_{\mathsf{AnonPay}}$. In $\mathrm{Exec}_{\Pi_{\textsc{AnonPay}},\mathcal{F}_{\textsc{Ledger}},\mathcal{F}_{\mathrm{AAT}},\mathcal{F}_{\textsc{Service}},\mathcal{E}}(1^\lambda)$ the inputs from $\mathcal{E}$ goes to $\Pi_{\textsc{AnonPay}}$ and the outputs of the parties go back to $\mathcal{E}$. In $\mathrm{Exec}_{\Pi_{\textsc{AnonPay}},\mathcal{F}_{\textsc{Ledger}},\mathcal{F}_{\mathrm{AAT}},\mathcal{F}_{\textsc{Service}},\mathcal{E}}(1^\lambda)$ the leakage of $\mathcal{F}_{\textsc{Ledger}}$, $\mathcal{F}_{\mathrm{AAT}}$, and $\mathcal{F}_{\textsc{Service}}$ goes to $\mathcal{E}$ and it is $\mathcal{E}$ giving commands to these ideal functionalities as adversary. In $\mathrm{Exec}_{\mathcal{F}_{\mathsf{AnonPay}},\mathcal{S},\mathcal{E}}(1^\lambda)$ it is simulator $\mathcal{S}$ which gets the leakage from $\mathcal{F}_{\mathsf{AnonPay}}$ and which gives adversarial commands to $\mathcal{F}_{\mathsf{AnonPay}}$. The simulator $\mathcal{S}$ also interacts with $\mathcal{E}$ and presents $\mathcal{E}$ with the same adversarial interface as $\mathcal{F}_{\textsc{Ledger}}$, $\mathcal{F}_{\mathrm{AAT}}$, and $\mathcal{F}_{\textsc{Service}}$ have. It tries to convince $\mathcal{E}$ that $\mathcal{E}$ is observing a run of $\mathrm{Exec}_{\Pi_{\textsc{AnonPay}},\mathcal{F}_{\textsc{Ledger}},\mathcal{F}_{\mathrm{AAT}},\mathcal{F}_{\textsc{Service}},\mathcal{E}}(1^\lambda)$. It must produce the same leakage to $\mathcal{E}$ as the $\mathcal{F}_{\textsc{Ledger}}$, $\mathcal{F}_{\mathrm{AAT}}$, and $\mathcal{F}_{\textsc{Service}}$ would in

$$\mathrm{Exec}_{\Pi_{\textsc{AnonPay}},\mathcal{F}_{\textsc{Ledger}},\mathcal{F}_{\mathrm{AAT}},\mathcal{F}_{\textsc{Service}},\mathcal{E}}(1^\lambda)$$

and it must receive adversarial commands from $\mathcal{E}$ to $\mathcal{F}_{\textsc{Ledger}}$, $\mathcal{F}_{\mathrm{AAT}}$, and $\mathcal{F}_{\textsc{Service}}$ and translate these into adversarial commands to $\mathcal{F}_{\mathsf{AnonPay}}$ with the same effects. It must in particular make $\mathcal{F}_{\mathsf{AnonPay}}$ give the same outputs as the protocol $\Pi_{\textsc{AnonPay}}$. The main challenge in constructing $\mathcal{S}$ is that in $\mathrm{Exec}_{\mathcal{F}_{\mathsf{AnonPay}},\mathcal{S},\mathcal{E}}(1^\lambda)$ it does not receive the inputs of honest parties, only the limited leakage provided by $\mathcal{F}_{\mathsf{AnonPay}}$.

We only prove static security, i.e., it is decided before the execution which parties are corrupted.

In the simulation there are two types of differences from the protocol to handle. First of all, there is *structural simulation*, i.e., we construct the simulator to send values which are structured as in the protocol and are sent at the same time as the protocol, and construct it to give appropriate inputs to $\mathcal{F}_{\mathsf{AnonPay}}$ to make it give the same outputs as the protocol at the same times. There is also *content simulation*, i.e., some of the values sent at the correct times during the simulation might have a different distribution than in the protocol, for instance an encryption of a dummy label instead of the correct label. We prove indistinguisability of structure and contents slight differently. We argue during the presentation of the simulator, inside the pseudo-code, that the simulation has the correct temporal structure, as we think this helps to understand why the simulator is constructed as it is and therefore makes it easier to read. After that we then argue that the messages sent in the simulation are also indistinguishable from those of the protocol using a hybrids argument.

9.1 Observations

We first make some observations about the protocol which help better understand the structure of the simulator and prove it secure.

In the below we will change to a hybrid where the CRS $\mathsf{crs}^{\mathrm{KNOW}}$ for the proof of knowledge is generated by $\mathcal{F}_{\textsc{Service}}$ as follows:

$$(\mathsf{crs}^{\mathrm{KNOW}}, \mathsf{tSim}, \mathsf{tExt}) \leftarrow \mathsf{NIZKPoK.SimGen}(1^\lambda)\ .$$

By Zero-Knowledge in Section 2.5 it follows that this change will be indistinguishable to the adversary and environment. Furthermore, by Weak Simulation Extractability in Section 2.5 it follows that we can use tExt to extract accepting proofs given by the adversary, and this will yield a witness except with negligible probability. This is because we are not using tSim to simulate any proofs and therefore in particular are simulating no false statements. The reason why we need to extract some proofs from the adversary will become apparent by Definition 31 below.

Lemma 2 (No Account Collisions). *If $\mathcal{F}_{\textsc{Ledger}}$ accepts two accounts A_1 and A_2 then $\mathsf{A}_1 \neq \mathsf{A}_2$.*

Proof. The position p_i at which A_i appears in Ledger is part of A_i and $p_1 \neq p_2$.

Recall that we model Hash as a random oracle.

Lemma 3 (Hidden Query Point). *For all executions it holds except with negligible probability that whenever Step 5 **Receive Coin** in Fig. 14 is executed, then Hash has not yet been queried on $\mathsf{L} = \mathsf{L}_\mathsf{A} \oplus \mathsf{L}_\mathsf{S}$. Furthermore, if both A and B in a payment $(\textsc{Pay}, \mathsf{A}, \mathsf{B}, \cdots)$ are honest, then Hash has not yet been queried on $\mathsf{L} = \mathsf{L}_\mathsf{A} \oplus \mathsf{L}_\mathsf{S}$ by the adversary when Step 5 in **Collect** in Fig. 15 is executed.*

Proof. We can set up the execution such that we only use the CRaB key k via blackbox access to $\mathsf{Eval}(k, \cdot)$ and $\mathsf{Prefix}(k, \cdot)$. At a given point let p_0 be the maximal value we queried $\mathsf{Eval}(k, \cdot)$ or $\mathsf{Prefix}(k, \cdot)$ on. Let p be the position of $(\mathsf{Tx}, \phi_{\mathsf{Tx}})$ in Fig. 14. It is not hard to see that just before we execute Step 5 in Fig. 14 we have that $p > p_0$. So if we can compute $(p, \mathsf{L}_\mathsf{S})$ such that $\mathsf{CRaB.Eval}(k, p) = \mathsf{L}_\mathsf{S}$ with probability non-negligibly better than $2^{-\lambda}$ then we can win Definition 26, which we have assumed that we cannot. Note that before Step 5 in Fig. 14 we can compute the, possibly corrupted, P_A's contribution to the coinflip as $(\mathsf{A}, \mathsf{L}_\mathsf{A}) = \mathsf{PKE.Dec}_{\mathsf{dk}}(d)$. Keep track of the points Q on which Hash was queried and compute $Q \oplus \mathsf{L}_\mathsf{A} = \{x \oplus \mathsf{L}_\mathsf{A} | x \in Q\}$. Assume for the sake of contradiction that Hash was queried on $\mathsf{L} = \mathsf{L}_\mathsf{A} \oplus \mathsf{L}_\mathsf{S}$. This is the same as $\mathsf{L}_\mathsf{A} \oplus \mathsf{CRaB.Eval}(k, p) \in Q$ which is equivalent to $\mathsf{CRaB.Eval}(k, p) \in Q \oplus \mathsf{L}_\mathsf{A}$. This allows us to guess $\mathsf{CRaB.Eval}(k, p)$ with noticeable probability $|Q|^{-1}$ by outputting a uniformly random point from $Q \oplus \mathsf{L}_\mathsf{A}$. But we have argued that this cannot be the case. Then note that the adversary gets no additional information on L until Step 5 in **Collect** in Fig. 15 when both the sender and receiver are honest. □

When describing the simulator, and in the proofs, we will assume that whenever Step 5 in **Receive Coin** in Fig. 14 is executed then Hash has not been queried on $\mathsf{L} = \mathsf{L}_\mathsf{A} \oplus \mathsf{L}_\mathsf{S}$. Similarly for Step 5 in **Collect** in Fig. 15 when both the sender and receiver are honest. We can without loss of generality ignore the negligible probability that this is not the case.

We then prove a lemma about when an honest payment can fail. Let us first see that there is something to do. Note that in Step 1 in Fig. 13 the payer checks that $a \leq b_\mathsf{A}$. However, when the transaction is sent in Step 9 then the control goes back to the environment which may now initiate a new payment for the same A. This payment would be made with the same $\mathsf{nonce}_\mathsf{A}$ so it will be in contradiction to the previous payment. Assume for sake of argument that both payments are of the amount $a = b_\mathsf{A}$ and that $a > 0$. Then clearly it is a feature that at most one should go through. We could change the code to avoid that such parallel payments are done. The honest user could set a local bit indicating that a payment is in process. This is, however, not a practical solution. In practice keys often exist in several places, say in a mobile wallet and a desktop wallet, so it is hard to implement such a semaphore. Users also often make multiple retries on purpose. It can for instance happen that a user makes a payment, but during the payment the payment app crashes and/or the payment takes a very long time to arrive on the blockchain. The user in these cases often will try to make the payment again until it goes through. We therefore want to allow multiple conflicting payments being in progress as a feature. The first to arrive on the chain will be the one accepted.

Notice that the payment transaction $(\mathsf{Tx}, \phi_{\mathsf{Tx}})$ contains $\mathsf{nonce}_{\mathsf{A}}$. Call this value $\mathsf{nonce}(\mathsf{Tx}, \phi_{\mathsf{Tx}})$.

Lemma 4 (Unique NONCE). *If* A *is honest then for each* $\mathsf{nonce}_{\mathsf{A}}$ *at most one payment* $(\mathsf{Tx}, \phi_{\mathsf{Tx}})$ *with* $\mathsf{nonce}(\mathsf{Tx}, \phi_{\mathsf{Tx}}) = \mathsf{nonce}_{\mathsf{A}}$ *is posted. Furthermore, for each* $\mathsf{nonce}_{\mathsf{A}}$ *it is always possible that some payment will go through for* $\mathsf{nonce}_{\mathsf{A}}$*, at least as long as* $b_{\mathsf{A}} > 0$ *and a covered payment is made (i.e., one where* $a \leq b_{\mathsf{A}}$*) while* $\mathsf{nonce}_{\mathsf{A}}$ *is the value stored on* $\mathcal{F}_{LEDGER}$*. Furthermore, the* $(\mathsf{Tx}, \phi_{\mathsf{Tx}})$ *to go through for* $\mathsf{nonce}_{\mathsf{A}}$ *is the first covered one to be scheduled by the adversary for which* $\mathsf{nonce}(\mathsf{Tx}, \phi_{\mathsf{Tx}}) = \mathsf{nonce}_{\mathsf{A}}$*.*

Proof. We first prove that at most one transaction is posted for each $\mathsf{nonce}_{\mathsf{A}}$. Note that if any $(\mathsf{Tx}, \phi_{\mathsf{Tx}})$ initiated with $\mathsf{nonce}_{\mathsf{A}} = \mathsf{nonce}$ is posted on $\mathcal{F}_{\text{LEDGER}}$, then nonce is part of $(\mathsf{Tx}, \phi_{\mathsf{Tx}})$. Let us write $\mathsf{nonce}(\mathsf{Tx}, \phi_{\mathsf{Tx}}) = \mathsf{nonce}$. Assume $(\mathsf{Tx}, \phi_{\mathsf{Tx}})$ is posted on $\mathcal{F}_{\text{LEDGER}}$. Let $\mathsf{nonce}_{\mathsf{A}}$ be the value on $\mathcal{F}_{\text{LEDGER}}$ when $(\mathsf{Tx}, \phi_{\mathsf{Tx}})$ is posted. It is easy to see that by construction of ϕ_{Tx} and soundness of $\mathsf{NIZKPoK}$ it holds that $\mathsf{nonce}(\mathsf{Tx}, \phi_{\mathsf{Tx}}) = \mathsf{nonce}_{\mathsf{A}}$ when $(\mathsf{Tx}, \phi_{\mathsf{Tx}})$ is posted. And by construction it will hold that $\mathsf{nonce}_{\mathsf{A}} = \mathsf{nonce}(\mathsf{Tx}, \phi_{\mathsf{Tx}}) + 1$ right after. So for each A and each $\mathsf{nonce}_{\mathsf{A}}$ at most one $(\mathsf{Tx}, \phi_{\mathsf{Tx}})$ with $\mathsf{nonce}(\mathsf{Tx}, \phi_{\mathsf{Tx}}) = \mathsf{nonce}_{\mathsf{A}}$ is posted.

We then prove that for each $\mathsf{nonce}_{\mathsf{A}}$ it is always possible that some covered payment will go through for $\mathsf{nonce}_{\mathsf{A}}$ and that it is the first one scheduled by the adversary for which $\mathsf{nonce}(\mathsf{Tx}, \phi_{\mathsf{Tx}}) = \mathsf{nonce}_{\mathsf{A}}$. Say that A is *locally consistent* when the values stored locally by A match the values on the ledger, i.e., $c_{\mathsf{A}} = \mathsf{Commit}_{\mathsf{ck}}(b_{\mathsf{A}}; \rho_{\mathsf{A}})$ and the value of $\mathsf{nonce}_{\mathsf{A}}$ in local storage is the same as the one on $\mathcal{F}_{\text{LEDGER}}$. There is only one period of time where A is not locally consistent and that is from when Tx is posted on $\mathcal{F}_{\text{LEDGER}}$ in Step 10 and A updates its state in Step 11. In Step 10 the blockchain stateful layer updates to $\mathsf{nonce}'_{\mathsf{A}} = \mathsf{nonce}_{\mathsf{A}} + 1$ and this is only done by A in Step 11. We show that if a covered payment is made for $\mathsf{nonce}_{\mathsf{A}}$ then one will eventually go through. Assume first the payment is made when A is *not* locally consistent. Note that any payment initiated when A is not locally consistent will terminate already in Step 1 as the check $\mathsf{nonce}_{\mathsf{A}} \neq \mathsf{nonce}'_{\mathsf{A}}$ will be true. This can only happen if already some other payment with $\mathsf{nonce}_{\mathsf{A}}$ was initiated. And this payment will terminate when A is again scheduled to run from Step 11 and then a payment was made for $\mathsf{nonce}_{\mathsf{A}}$. Assume then the payment is made when A *is* locally consistent. Then by construction $(\mathsf{Tx}, \phi_{\mathsf{Tx}})$ with $\mathsf{nonce}(\mathsf{Tx}, \phi_{\mathsf{Tx}}) = \mathsf{nonce}_{\mathsf{A}}$ is posted. Furthermore, when the first such $(\mathsf{Tx}, \phi_{\mathsf{Tx}})$ is scheduled, then the ledger updates $\mathsf{nonce}'_{\mathsf{A}} = \mathsf{nonce}_{\mathsf{A}} + 1$ and no other such payment can now get posted. Once A is scheduled again it will execute from Step 11 and the payment has gone through. □

We now define the "information inside a transfer", which is not a straight-forward notion as the commitments are information theoretically hiding. We instead use extraction of the proof system to get the information. This is the first place where we use that we set up $\mathsf{crs}^{\text{KNOW}}$ with known tExt.

Definition 31 (Transfer Information). *In the protocol we can inspect* $\mathcal{F}_{SERVICE}$ *to learn* tExt*. This allows us to define what information is in the coin of a payment as follows. When during a payment a* $(\mathsf{Tx}, \phi_{PAY})$ *appears on the ledger in position* p*, get the proof* π *from* Tx*. By construction of* ϕ_{PAY} *we know that* $\mathsf{NIZKPoK.Ver}_{\mathsf{crs}}(\mathcal{R}_{PAY}, (c_{\mathsf{A}}, \mathsf{coin}, c_{\mathsf{A}'}, \mathsf{A}, \mathsf{nonce}_{\mathsf{A}}), \pi) = \top$*. By definition of* $\mathcal{R}_{PAY}$ *we can therefore use* tExt *to compute a witness* $w = ((b_{\mathsf{A}}, \rho_{\mathsf{A}}), (\mathsf{B}, a, \rho_1, \rho_2), \rho'_{\mathsf{A}})$ *such that* $c_{\mathsf{A}} = \mathsf{Com.Commit}_{\mathsf{ck}}(b_{\mathsf{A}}, \rho_{\mathsf{A}})$ *and* $((\mathsf{coin}, \mathsf{Com.Commit}_{\mathsf{ck}}(\mathsf{A}, \mathsf{B}, a, \mathsf{nonce}_{\mathsf{A}}; \rho_1)), \rho_2) \in \mathcal{R}_{ENC}$ *and*

$c'_{\mathsf{A}} = \mathsf{Com.Commit}_{\mathsf{ck}}(b_{\mathsf{A}} - a, \rho'_{\mathsf{A}})$*, and* $b_{\mathsf{A}} \geq a \geq 0$*. We say that* coin *is a payment from* A *to* B *of amount* a *with transfer identifier* $\mathsf{tid} = \mathsf{Com.Commit}_{\mathsf{ck}}(\mathsf{A}, \mathsf{B}, a, \mathsf{nonce}_{\mathsf{A}}; \rho_1)$*, nonce* $\mathsf{nonce}_{\mathsf{A}}$ *and appearing in position* p*. We write* $\mathsf{PayInfo}_{\mathsf{tExt}}(\mathsf{Tx}) = (\mathsf{A}, \mathsf{B}, \mathsf{tid}, a, \mathsf{nonce}_{\mathsf{A}}, p)$*. Similarly for a collection* $(\mathsf{Tx}, \phi_{\mathit{COLLECT}})$ *in position* p*. By construction of* ϕ_{COLLECT} *we have that*

$$\mathsf{NIZKPoK.Ver}_{\mathsf{crs}}(\mathcal{R}_{\mathit{COLLECT}}, (\mathsf{ck}, c_{\mathsf{B}}, \mathsf{B}, \mathsf{tid}, c'_{\mathsf{B}}), \pi_2) = \top \ ,$$

so we can use tExt *to compute an opening* $\mathsf{tid} = \mathsf{Com.Commit}_{\mathsf{ck}}(\mathsf{A}, \mathsf{B}, a, \mathsf{nonce}_{\mathsf{A}}, \rho_1)$*. We write* $\mathsf{CollInfo}_{\mathsf{tExt}}(\mathsf{Tx}) = (\mathsf{A}, \mathsf{B}, \mathsf{tid}, a, \mathsf{nonce}_{\mathsf{A}}, p)$*.*

Note that the extractions needed in the above definition will indeed succeed by weak simulation extraction. Since we do not simulate any false statements or extract any simulated proofs, the failure of an extraction would allow us to win the game in **Weak Simulation Extraction** in Definition 17.

Note that $\mathsf{PayInfo}_{\mathsf{tExt}}(\mathsf{Tx})$ can be computed in PPT given tExt. We can assume without loss of generality that $\mathsf{NIZKPoK.Extract}^{\mathcal{O}}$ is deterministic, making $\mathsf{PayInfo}_{\mathsf{tExt}}(\mathsf{Tx})$ a function. Even if $\mathsf{NIZKPoK.Extract}^{\mathcal{O}}$ was randomised it is easy to see that if different computations of $\mathsf{PayInfo}_{\mathsf{tExt}}(\mathsf{Tx})$ would lead to different information, then we would have broken computational binding of $\mathsf{Com.Commit}_{\mathsf{ck}}$. Similarly for $\mathsf{CollInfo}$.

Lemma 5 (Correct Nonce). *Let* $(\mathsf{Tx}, \phi_{\mathit{PAY}})$ *be a payment on* $\mathcal{F}_{\mathit{LEDGER}}$*. Let* $\mathsf{PayInfo}_{\mathsf{tExt}}(\mathsf{Tx}) = (\mathsf{A}, \mathsf{B}, \mathsf{tid}, a, \mathsf{nonce}_{\mathsf{A}}, p)$*. Then except with negligible probability* $\mathsf{nonce}_{\mathsf{A}}$ *is the value of* $\mathsf{nonce}_{\mathsf{A}}$ *from* A*'s account on* $\mathcal{F}_{\mathit{LEDGER}}$ *just before* $(\mathsf{Tx}, \phi_{\mathit{PAY}})$ *was posted.*

Proof. This follows by the fact that $\mathsf{NIZKPoK.Ver}_{\mathsf{crs}}(\mathcal{R}_{\mathrm{PAY}}, (c_{\mathsf{A}}, \mathsf{coin}, c_{\mathsf{A}'}, \mathsf{A}, \mathsf{nonce}_{\mathsf{A}}), \pi) = \top$, soundness of $\mathsf{NIZKPoK}$ and construction of $\mathcal{R}_{\mathrm{PAY}}$. □

Lemma 6 (Unique Coins and Identifiers). *The following holds in all executions except with negligible probability. Let* $(\mathsf{Tx}^1, \phi^1_{\mathit{PAY}})$ *and* $(\mathsf{Tx}^2, \phi^2_{\mathit{PAY}})$ *be two payments on* $\mathcal{F}_{\mathit{LEDGER}}$ *containing coins* coin^1 *and* coin^2 *respectively. Then* $\mathsf{coin}^1 \neq \mathsf{coin}^2$*. Furthermore, if we let* $\mathsf{PayInfo}_{\mathsf{tExt}}(\mathsf{Tx}^1) = (\cdot, \cdot, \mathsf{tid}^1, \cdots)$ *and* $\mathsf{PayInfo}_{\mathsf{tExt}}(\mathsf{Tx}^2) = (\cdot, \cdot, \mathsf{tid}^2, \cdots)$*, then* $\mathsf{tid}^1 \neq \mathsf{tid}^2$*.*

Proof. Let $\mathsf{RPKE.Dec.}(\mathsf{coin}^i) = \mathsf{tid}^i$ define the transfer identifier tid^i inside coin^i. Note that if $\mathsf{coin}^1 = \mathsf{coin}^2$ then $\mathsf{tid}^1 = \mathsf{tid}^2$, so it is enough to prove that $\mathsf{tid}^1 \neq \mathsf{tid}^2$. Assume for the sake of contradiction that $\mathsf{tid}^1 = \mathsf{tid}^2 =: \mathsf{tid}$. As in Definition 31 we can use tExt to open coin from the proofs posted on $\mathcal{F}_{\mathrm{LEDGER}}$ in PPT. This allows us to compute tid in PPT and by definition of $\mathcal{R}_{\mathrm{ENC}}$ we get that $\mathsf{tid} = \mathsf{Com.Commit}_{\mathsf{ck}}(\mathsf{A}, \mathsf{B}, a, \mathsf{nonce}_{\mathsf{A}}; \rho_1)$. So $\mathsf{PayInfo}_{\mathsf{tExt}}(\mathsf{Tx}^1) = (\mathsf{A}, \mathsf{B}, \mathsf{tid}, a, \mathsf{nonce}_{\mathsf{A}}, \cdots)$ and $\mathsf{PayInfo}_{\mathsf{tExt}}(\mathsf{Tx}^2) = (\mathsf{A}, \mathsf{B}, \mathsf{tid}, a, \mathsf{nonce}_{\mathsf{A}}, \cdots)$. This, however, cannot be the case. One of the two payments were posted first, say Tx^1. When this happened $\mathsf{nonce}_{\mathsf{A}}$ was the value of $\mathsf{nonce}_{\mathsf{A}}$ from A's account on $\mathcal{F}_{\mathrm{LEDGER}}$ just before Tx^1 was posted (by Lemma 5). By inspection of Fig. 13 it can be seen that when Tx^1 was posted $\mathcal{F}_{\mathrm{LEDGER}}$ the value of $\mathsf{nonce}_{\mathsf{A}}$ from A's account on $\mathcal{F}_{\mathrm{LEDGER}}$ was incremented to $\mathsf{nonce}_{\mathsf{A}} + 1$. And this value never decreases. Therefore $\mathsf{PayInfo}_{\mathsf{tExt}}(\mathsf{Tx}^2) = (\mathsf{A}, \mathsf{B}, \mathsf{tid}, a, \mathsf{nonce}_{\mathsf{A}})$ is in contradiction with Lemma 5.

From Lemma 6 we get that if there is a coin coin appearing in a payment then except with negligible probability it appears in a unique Tx. We can therefore make the definition

$$\mathsf{PayInfo}(\mathsf{coin}) = (\mathsf{A}, \mathsf{B}, \mathsf{tid}, a) \ ,$$

where $\mathsf{PayInfo}_{\mathsf{tExt}}(\mathsf{Tx}) = (\mathsf{A}, \mathsf{B}, \mathsf{tid}, a, \mathsf{nonce}_{\mathsf{A}})$ and Tx is the unique Tx in which coin appears. This definition is well-formed except with negligible probability. We will tacitly ignore the executions where this definition is not well-defined.

Lemma 7 (TID Distribution). *Consider the distribution* Tid *obtained by sampling* $\mathsf{tid} \leftarrow \mathsf{Com.Commit}_{\mathsf{ck}}(0)$*. All* tid *produced by honest parties have this distribution. Furthermore, for all transfers* Tx *produced by corrupted parties and* $\mathsf{PayInfo}_{\mathsf{tExt}}(\mathsf{Tx}) = (\mathsf{A}, \mathsf{B}, \mathsf{tid}, \cdots)$ *it holds that* tid *is different from all other* tid *used by honest or corrupt parties, except with negligible probability.*

Proof. The transfer identifier used by an honest party is computed as

$$\mathsf{tid} = \mathsf{Com.Commit}_{\mathsf{ck}}((\mathsf{A}, \mathsf{B}, a, \mathsf{nonce}_{\mathsf{A}}); \rho_1)$$

for a uniformly random ρ_1. It follows from perfect hiding that this has the same distribution as Tid. The second part of the lemma follows from Lemma 6.

Definition 32 (Account Information). *For an account* $\mathsf{Account} = (\mathsf{A}, c_{\mathsf{A}}, \mathsf{ek}_{\mathsf{A}}, \mathsf{nonce}_{\mathsf{A}})$ *on* $\mathcal{F}_{\text{LEDGER}}$ *we let* $\mathsf{AccountInfo}(\mathsf{Account}) = (\mathsf{A}, b_{\mathsf{A}}, \rho_{\mathsf{A}})$ *where* $c_{\mathsf{A}} = \mathsf{Com.Commit}_{\mathsf{ck}}(b_{\mathsf{A}}; \rho_{\mathsf{A}})$*. Note that if* A *is honest then* $\mathsf{AccountInfo}(\mathsf{Account})$ *can be read from the local storage of* P_{A}*. In the protocol we can inspect* $\mathcal{F}_{\text{SERVICE}}$ *to learn* tExt*. This allows us to compute* $\mathsf{AccountInfo}(\mathsf{Account})$ *for corrupt* A *in PPT as follows. In* ***Create Account*** *use* tExt *to extract the proof* π *for* $\mathcal{R}_{\text{IsZero}}$ *to learn* $b_{\mathsf{A}} = 0$ *and* ρ_{A} *such that* $c_{\mathsf{A}} = \mathsf{Com.Commit}_{\mathsf{ck}}(b_{\mathsf{A}}; \rho_{\mathsf{A}})$*. Similarly if the account is the funding account. In* ***Initiate Pay*** *use* tExt *to extract the proof* π *for* $\mathcal{R}_{\text{PAY}}$ *to learn* b'_{A} *and* ρ'_{A} *such that* $c'_{\mathsf{A}} = \mathsf{Com.Commit}_{\mathsf{ck}}(b'_{\mathsf{A}}; \rho'_{\mathsf{A}})$*. In* ***Collect*** *use* tExt *to extract the proof* π *for* $\mathcal{R}_{\text{COLLECT}}$ *to learn* b'_{A} *and* ρ'_{A} *such that* $c'_{\mathsf{A}} = \mathsf{Com.Commit}_{\mathsf{ck}}(b'_{\mathsf{A}}; \rho'_{\mathsf{A}})$*. All these proofs can be extracted as they were given using* $\mathsf{NIZKPoK}$*.*

Since A is unique for the accounts, as the position p is part of A, we have a well-defined map from A to $\mathsf{Account} = (\mathsf{A}, c_{\mathsf{A}}, \mathsf{ek}_{\mathsf{A}}, \mathsf{nonce}_{\mathsf{A}})$ on $\mathcal{F}_{\text{LEDGER}}$. We therefore also have a well-defined map from to A to the balance b_{A}, computed as follows. Given A, find $\mathsf{Account} = (\mathsf{A}, c_{\mathsf{A}}, \mathsf{ek}_{\mathsf{A}}, \mathsf{nonce}_{\mathsf{A}})$, compute $\mathsf{AccountInfo}(\mathsf{Account}) = (\mathsf{A}, b_{\mathsf{A}}, \rho_{\mathsf{A}})$, and output b_{A}. We call this map $\mathsf{Balance}^{\text{REAL}}$ below.

Lemma 8 (Coin Matching). *Any collection can be matched to a unique, earlier payment with the same transfer information about parties, amount and nonce. In a bit more detail, let* P *be the set of payments, i.e., all transactions* Tx *which appear as part of some* $(\mathsf{Tx}, \phi_{\text{PAY}})$ *on* Ledger*. Similarly let* C *be the set of collections, i.e., transactions* Tx *appearing as part of some* $(\mathsf{Tx}, \phi_{\text{COLLECT}})$ *on* Ledger*. Then except with negligible probability we can use* tExt *to efficiently compute a map* $\mathsf{CoinMap} : C \to P$ *such that* $\mathsf{CoinMap}$ *is injective and for all* $\mathsf{Tx} = (\cdot, \cdot, \mathsf{tid}, \cdot, \cdot)$ *and* $\mathsf{CoinMap}(\mathsf{Tx}) = \mathsf{Tx}' = (\cdot, \cdot, \mathsf{coin}, \cdot, \cdot)$ *it holds that* $\mathsf{RPKE.Dec}_{\mathsf{dk}}(\mathsf{coin}) = \mathsf{tid}$*. Furthermore, if* $\mathsf{CollInfo}_{\mathsf{tExt}}(\mathsf{Tx}) = (\mathsf{A}, \mathsf{B}, \mathsf{tid}, a, \mathsf{nonce}_{\mathsf{A}}, p)$ *and* $\mathsf{PayInfo}_{\mathsf{tExt}}(\mathsf{Tx}') = (\mathsf{A}', \mathsf{B}', \mathsf{tid}', a', \mathsf{nonce}'_{\mathsf{A}}, p')$ *then* $p' < p$ *and* $(\mathsf{A}', \mathsf{B}', \mathsf{tid}', a', \mathsf{nonce}'_{\mathsf{A}}) = (\mathsf{A}, \mathsf{B}, \mathsf{tid}, a, \mathsf{nonce}_{\mathsf{A}})$*.*

Proof. Let $\mathsf{Tx} = (\mathsf{B}, \mathsf{L}, \mathsf{tid}, \pi_1, \pi_2)$ be any collection. By construction of ϕ_{COLLECT} if we fetch the coins $\{\mathsf{coin}_j\}_{j \in \mathsf{pos}}$ for $\mathsf{pos} = \mathsf{SOROM.Pos}(\mathsf{L})$ then

$$\begin{aligned}
\mathsf{NIZK.Ver}_{\mathsf{crs}}(\mathcal{R}_{\text{OrDec}}, (\mathsf{ck}, \{\mathsf{coin}_j\}_{j \in \mathsf{pos}}, \mathsf{tid}), \pi_1) &= \top \\
\mathsf{NIZKPoK.Ver}_{\mathsf{crs}}(\mathcal{R}_{\text{COLLECT}}, (\mathsf{ck}, c_{\mathsf{B}}, \mathsf{B}, \mathsf{tid}, c'_{\mathsf{B}}), \pi_2) &= \top \\
\mathsf{tid} \notin \mathsf{Spent}\ .&
\end{aligned}$$

From $\mathsf{NIZK.Ver}_{\mathsf{crs}}(\mathcal{R}_{\text{OrDec}}, (\mathsf{ck}, \{\mathsf{coin}_j\}_{j \in \mathsf{pos}}, \mathsf{tid}), \pi_1) = \top$ and simulation soundness of NIZK we get that for some j it holds that $\mathsf{tid} = \mathsf{RPKE.Dec}_{\mathsf{dk}}(\mathsf{coin}_j)$. By inspection of $\mathcal{F}_{\text{Service}}$ it can be seen that each coin_j is either a coin posted in a payment transaction or such a coin rerandomized some number of times using $\mathsf{RPKE.Ran}$. By correctness of RPKE this means that we can find an earlier payment $\mathsf{Tx}' = (\mathsf{A}, c'_{\mathsf{A}}, \mathsf{coin}, \pi, d)$ such that $\mathsf{RPKE.Dec}_{\mathsf{dk}}(\mathsf{coin}) = \mathsf{RPKE.Dec}_{\mathsf{dk}}(\mathsf{coin}_j) = \mathsf{tid}$. By Unique Coins and Identifiers there is in fact a unique such Tx', but this is not central to the proof. We can pick any such Tx' and let $\mathsf{CoinMap}(\mathsf{Tx}) = \mathsf{Tx}'$. Then we have that $\mathsf{RPKE.Dec}_{\mathsf{dk}}(\mathsf{coin}) = \mathsf{tid}$ as desired.

That the map is injective except with negligible probability follows from the fact that ϕ_{Collect} checks that $\mathsf{tid} \notin \mathsf{Spent}$, so for each collection Tx the value $\mathsf{Tx.tid}$ is unique. Therefore two different collections Tx_1 and Tx_2 will also map to different payments $\mathsf{Tx}'_1 = \mathsf{CoinMap}(\mathsf{Tx}_1)$ and $\mathsf{Tx}'_2 = \mathsf{CoinMap}(\mathsf{Tx}_2)$ as $\mathsf{RPKE.Dec}_{\mathsf{dk}}(\mathsf{Tx}'_1.\mathsf{coin}) = \mathsf{Tx}_1.\mathsf{tid} \neq \mathsf{Tx}_2.\mathsf{tid} = \mathsf{RPKE.Dec}_{\mathsf{dk}}(\mathsf{Tx}'_2.\mathsf{coin})$. Now let $\mathsf{CollInfo}_{\mathsf{tExt}}(\mathsf{Tx}) = (\mathsf{A}, \mathsf{B}, \mathsf{tid}, a, \mathsf{nonce}_{\mathsf{A}}, p)$ and $\mathsf{PayInfo}_{\mathsf{tExt}}(\mathsf{Tx}') = (\mathsf{A}', \mathsf{B}', \mathsf{tid}', a', \mathsf{nonce}'_{\mathsf{A}}, p')$. The fact that $p' < p$ follows from coin_j being a coin already on the ledger when Tx was posted, so it is a rerandomization of an earlier payment. Finally, $(\mathsf{A}', \mathsf{B}', \mathsf{tid}', a', \mathsf{nonce}'_{\mathsf{A}}) = (\mathsf{A}, \mathsf{B}, \mathsf{tid}, a, \mathsf{nonce}_{\mathsf{A}})$ follows from $\mathsf{tid}' = \mathsf{RPKE.Dec}_{\mathsf{dk}}(\mathsf{Tx}'.\mathsf{coin}) = \mathsf{tid}$, computational binding of $\mathsf{Commit}_{\mathsf{ck}}$ and the fact that we can compute openings $\mathsf{tid} = \mathsf{Commit}_{\mathsf{ck}}(\mathsf{A}, \mathsf{B}, \mathsf{tid}, a, \mathsf{nonce}_{\mathsf{A}}; \rho_1)$ and $\mathsf{tid}' = \mathsf{Commit}_{\mathsf{ck}}(\mathsf{A}', \mathsf{B}', \mathsf{tid}', a', \mathsf{nonce}'_{\mathsf{A}}; \rho'_1)$, cf. the definition of Transfer Information. □

Since each collection Tx has a unique tid we can define a function $\mathsf{CoinMap}(\mathsf{tid}) = \mathsf{CoinMap}_{\mathsf{tExt}}(\mathsf{Tx})$ giving for each tid in a collection the corresponding payment.

Lemma 9 (Balance Correctness). *Except with negligible probability balances, as defined by* $\mathsf{AccountInfo}(\mathsf{A})$*, are updated according to the logic of* $\mathcal{F}_{\mathsf{AnonPay}}$ *as if run on the payment information* $\mathsf{PayInfo}(\mathsf{coin})$ *in the protocol. In a bit more detail, consider a run of the protocol* $\Pi_{\textit{AnonPay}}$ *and keep a copy for* $\mathcal{F}_{\mathsf{AnonPay}}$ *synchronised with it as follows. Whenever there is an account* A *created in the protocol by* P *ask* P *to create an account on* $\mathcal{F}_{\mathsf{AnonPay}}$ *and reply on behalf of* $\mathcal{S}$ *with the account name* A*. Whenever there is a payment* Tx *in* $\Pi_{\textit{AnonPay}}$ *with* $\mathsf{PayInfo}(\mathsf{Tx}) = (\mathsf{A}, \mathsf{B}, \mathsf{tid}, a, \cdots)$ *ask* P_{A} *on* $\mathcal{F}_{\mathsf{AnonPay}}$ *to do the same payment using input* $(\textit{Pay}, \mathsf{A}, \mathsf{B}, a)$ *and make the payment deducted. Then remember that this is the payment corresponding to* tid*. Note that it might get some other* tid' *on* $\mathcal{F}_{\mathsf{AnonPay}}$*, which does not matter. Whenever there is a collection* Tx *in the protocol with transfer identifier* tid *let* $\mathsf{Tx}' = \mathsf{CoinMap}(\mathsf{tid})$ *be the corresponding payment. Find the corresponding payment in* $\mathcal{F}_{\mathsf{AnonPay}}$*, make it observable, let* pid' *be the payment identifier it has in* $\mathcal{F}_{\mathsf{AnonPay}}$*, input* $(\textit{Collect}, \mathsf{tid}', \mathsf{A}, \mathsf{B}, a)$ *to* P_{B} *and then use the callback* *MakeCollected* *to make the call collected. For all accounts* A *let* $\mathcal{F}_{\mathsf{AnonPay}}.\mathsf{Balance}[\mathsf{A}]$ *be the balance in* $\mathcal{F}_{\mathsf{AnonPay}}$*. Recall that we defined the map* $b_{\mathsf{A}} = \mathsf{Balance}^{\textit{REAL}}(\mathsf{A})$ *above giving the balance in the protocol via the account information function* $\mathsf{AccountInfo}$*. Except with negligible probability it holds for all points of time in the execution and all* A *that* $\mathsf{Balance}^{\textit{REAL}}(\mathsf{A}) = \mathcal{F}_{\mathsf{AnonPay}}.\mathsf{Balance}[\mathsf{A}]$*.*

Proof. It can be seen that in both $\mathcal{F}_{\mathsf{AnonPay}}$ and Π_{AnonPay} the first account will have balance a_0 and all other accounts will have balance 0. We then show that payments and collections move the balances in synchrony. This follows by the fact that $\mathsf{NIZKPoK}$ is simulation extractable, Coin Matching, the proofs for $\mathcal{R}_{\text{Collect}}$ and $\mathcal{R}_{\text{Pay}}$, and the binding of $\mathsf{Com.Commit}_{\mathsf{ck}}$. We give a bit more details for payment and collection separately.

During payment, in the proof of $\mathcal{R}_{\text{Pay}}$ we get a witness opening the old c_{A} to some b_{A}. This must be the $b_{\mathsf{A}} = \mathsf{Balance}^{\text{REAL}}(\mathsf{A})$ or we broke binding of the commitment scheme. By induction we can assume that $b_{\mathsf{A}} = \mathcal{F}_{\mathsf{AnonPay}}.\mathsf{Balance}[\mathsf{A}]$. Furthermore, the coin is opened to the a in $\mathsf{PayInfo}(\mathsf{coin})$, or we broke binding of the commitment scheme. This is the a that we use when doing the corresponding payment $(\textsc{Pay}, \mathsf{A}, \mathsf{B}, a)$ on $\mathcal{F}_{\mathsf{AnonPay}}$. The relation $\mathcal{R}_{\textsc{Pay}}$ then ensures that the balance b'_{A} in the new c'_{A} will be $b'_{\mathsf{A}} = b_{\mathsf{A}} - a$, or we broke binding of the commitment scheme. I.e., in the new state of $\Pi_{\textsc{AnonPay}}$ we have that $\mathsf{Balance}^{\textsc{real}}(\mathsf{A}) = b_{\mathsf{A}} - a$. The same update will happen to $\mathcal{F}_{\mathsf{AnonPay}}.\mathsf{Balance}[\mathsf{A}]$ when we make the payment deducted.

During collection, in the proof of $\mathcal{R}_{\textsc{Collect}}$ we get a witness opening the old c_{A} to some b_{A}. This must be the $b_{\mathsf{A}} = \mathsf{Balance}^{\textsc{real}}(\mathsf{A})$ or we broke binding of the commitment scheme. By induction we can assume that $b_{\mathsf{A}} = \mathcal{F}_{\mathsf{AnonPay}}.\mathsf{Balance}[\mathsf{A}]$. Furthermore, the transfer identifier tid is opened to the a in $\mathsf{CollInfo}(\mathsf{tid})$, or we broke binding of the commitment scheme. This is also the a that we used when doing the corresponding payment on $\mathcal{F}_{\mathsf{AnonPay}}$ by Coin Matching. The relation $\mathcal{R}_{\textsc{Collect}}$ then ensures that the balance b'_{A} in the new c'_{A} will be $b'_{\mathsf{A}} = b_{\mathsf{A}} + a$, i.e., in the new state of $\Pi_{\textsc{AnonPay}}$ we have that $\mathsf{Balance}^{\textsc{real}}(\mathsf{A}) = b_{\mathsf{A}} + a$, or we broke the binding of the commitment scheme. The same update will happen to $\mathcal{F}_{\mathsf{AnonPay}}.\mathsf{Balance}[\mathsf{A}]$ when we make the payment collected, as it uses the a from the corresponding payment $(\textsc{Pay}, \mathsf{A}, \mathsf{B}, a)$. □

The following lemma follows form Coin Matching.

Lemma 10 (UCHP). *Let I be the number of payments initiated from an honest A to an honest B and let C be the number of payments collected from an honest A to an honest B. Then it is always that case that $C \leq I$.*

Finally we prove a helping lemma about when a coin is collectable. First a definition.

Definition 33 (Observable, Collectable). *If B is honest and there is a coin in position p on $\mathcal{F}_{\textsc{Ledger}}$ with $\mathsf{PayInfo}(\mathsf{coin}) = (\mathsf{A}, \mathsf{B}, \mathsf{tid}, a)$ then we call it* observable *if a call to $(\textsc{Observe}, \mathsf{tid}, \mathsf{A}, \mathsf{B})$ would make B return $(\textsc{Observe}, \mathsf{tid}, \mathsf{A}, \mathsf{B}, \top)$. We call it* collected *if $\mathcal{F}_{\textsc{Ledger}}$ contains $(\mathsf{Tx}, \phi_{\textsc{Collect}})$ with tid in Tx. We call it* collectable *if it is collected or a call to $(\textsc{Collect}, \mathsf{tid}, \mathsf{A}, \mathsf{B})$ would make B post $(\mathsf{Tx}, \phi_{\textsc{Collect}})$ with tid in Tx and which would be accepted by $\mathcal{F}_{\textsc{Ledger}}$ if scheduled by the adversary.*

Lemma 11 (Observable implies Collectable). *If B is honest and there is a coin in position p on $\mathcal{F}_{\textsc{Ledger}}$ with $\mathsf{PayInfo}(\mathsf{coin}) = (\mathsf{A}, \mathsf{B}, \mathsf{tid}, a)$ and the current value of $p_{\textsc{updated}}$ on $\mathcal{F}_{\textsc{Ledger}}$ is such that $p_{\textsc{updated}} \geq p$, then coin is collectable whenever it is observable.*

Proof. If $p_{\textsc{updated}} \geq p$ then coin has been added to OM so by completeness it can be found at one of the positions $\mathsf{SOROM.Pos}(\mathsf{L})$. And if **Observe** returns $\top$ then B can learn all the values needed to construct the proofs for Tx. □

Note that the above lemma shows that if a coin is observable, then it eventually becomes collectable as the service will eventually ensure that $p_{\textsc{updated}} \geq p$.

9.2 Simulator

We first construct $\mathcal{S}$ and then analyse it to prove (2). Our strategy will be to run a copy of

$$\mathrm{Exec}_{\Pi_{\textsc{AnonPay}}, \mathcal{F}_{\textsc{Ledger}}, \mathcal{F}_{\textsc{AAT}}, \mathcal{F}_{\textsc{Service}}, \cdot}(1^\lambda)$$

inside $\mathcal{S}$ and present the adversarial interface of $\mathcal{F}_{\textsc{Ledger}}$, $\mathcal{F}_{\textsc{AAT}}$, $\mathcal{F}_{\textsc{Service}}$ to $\mathcal{E}$. We call this the dummy execution. Note that since it is $\mathcal{S}$ running $\mathcal{F}_{\textsc{Ledger}}$, $\mathcal{F}_{\textsc{AAT}}$, $\mathcal{F}_{\textsc{Service}}$ it will see all inputs to these and may run them in deviation from their real code, as long as this it not noticed by $\mathcal{E}$. Note that $\mathcal{O}$ is a global ideal functionality, as formalised in [LR22], so it is not the simulator running it. However, the simulator has access to programming it and seeing all queries made by the environment so it can extract all the proofs made by the environment (by definition of the proof being a GUC NIZK PoK).

When an honest party does a transfer to an honest party we will not learn the amount nor the receiver. We will therefore run on dummy inputs and use the ZK simulator to cheat with the proofs. The simulator $\mathcal{S}$ runs $\mathcal{F}_{\text{SERVICE}}$.**Init** as in the protocol and saves all values, with the one difference that it computes $\mathsf{crs}^{\text{KNOW}}$ as $(\mathsf{crs}^{\text{KNOW}}, \mathsf{tSim}, \mathsf{tExt}) \leftarrow \mathsf{NIZKPoK.SimGen}(1^\lambda)$. Note in particular that $\mathcal{S}$ learns the simulation trapdoor tSim of $\mathsf{crs}^{\text{KNOW}}$ along with the extraction trapdoor tExt. Note also that this allows us to apply the observations in the previous section to the simulation.

Init The simulator runs a copy of $\mathcal{F}_{\text{LEDGER}}$, $\mathcal{F}_{\text{AAT}}$ and $\mathcal{F}_{\text{SERVICE}}$. These ideal functionalities are mostly run honestly with some deviations described below. Notice that hereby the simulator knows all secret keys generated by the service. It also runs a simulated version of Π_{ANONPAY} with many deviations described below. When $\mathcal{F}_{\mathsf{AnonPay}}$ asks for a distribution Tid on transaction identifiers return the following distribution: $\mathsf{tid} \leftarrow \mathsf{Com.Commit}_{\mathsf{ck}}(0)$.

Create Account On input $(\text{CREATEACCOUNT}, \mathsf{P})$ from $\mathcal{F}_{\mathsf{AnonPay}}$ for an honest P run $\Pi_{\text{ANONPAY}}.\mathsf{CreateAccount}$ honestly. This is possible as there are no secret inputs to P. Store all secrets generated in $\Pi_{\text{ANONPAY}}.\mathsf{CreateAccount}$ for later use in the simulation. Let A be the generated account name. When $(\mathsf{A}, \ldots)$ appears on the simulated $\mathcal{F}_{\text{LEDGER}}$ input $(\text{MAKEACCOUNTOBSERVABLE}, \mathsf{P}, \mathsf{A})$ to $\mathcal{F}_{\mathsf{AnonPay}}$. This will be a perfect simulation unless $\mathsf{A} \in \mathsf{Accounts}$ already, but this cannot happen by Lemma 2.

Honest/Corrupt Payment We call a payment *honest* when both the sender and receiver are honest. We call it *corrupt* if either the sender or the receiver is corrupt. Note that if the payment is corrupt then the ideal functionality leaks some $(\text{PAY}, \mathsf{A}, \mathsf{B}, \mathsf{atid}, a)$. We therefore know all inputs and can run the honest party (if there is any) according to the protocol. We will do this. We therefore only specify how to simulate honest parties in honest payments and corrupted parties in corrupted payments.
We let UCHP be the set of uncollected, collectible honest payments, initially empty. These will be used for simulating collection of honest payments, where we do not know the sender. We learn in the ideal functionality that some collectible payment was collected by B, but not which one. We will therefore in the simulation let B collect some payment from UCHP.

Service Simulate $\mathcal{F}_{\text{SERVICE}}$ by running it honestly as in Fig. 14.

Fig. 16. Simulator $\mathcal{S}$ (Init, Creation, Service)

The simulation of initialisation, account creation and the service is given in Fig. 16. The simulation of payment by an honest sender is given in Fig. 17. If the receiver is corrupt we learn the information to do the payment honestly. If the receiver is honest we do not learn the payment information, so we use ZK to just do a dummy payment of 0. The simulation of observation of a payment is given in Fig. 18, but there is not really anything to simulate as the protocol does not generate communication during observation. The simulation of collection between honest sender and honest receiver is given in Fig. 19. This is the interesting case as it is not known to the simulator which payment is collected. We simply collect some previously uncollected and currently collectible payment by some honest party. The simulation of a corrupted payment is given in Fig. 20. Here we use extraction to find the payment information and ask $\mathcal{F}_{\mathsf{AnonPay}}$ do the same payment on behalf of the corrupt sender. The collection of a corrupt payment by an honest receiver is given in Fig. 21 by just following the protocol. Finally, collection by a corrupt party is simulated in Fig. 22. We just

Initiate Pay (Honest Sender, Corrupt Receiver) On leakage $(\textsc{Pay}, \mathsf{A}, \mathsf{B}, a, \mathsf{atid})$ from $\mathcal{F}_{\mathsf{AnonPay}}$, i.e., where B is corrupted, simulate by running A according to the protocol in Fig. 13. Once tid has been computed, specify this tid to $\mathcal{F}_{\mathsf{AnonPay}}$ as the one to use for atid.

Initiate Pay (Honest Sender, Honest Receiver) On input $(\textsc{Pay}, \mathsf{A}, \mathsf{atid})$ from $\mathcal{F}_{\mathsf{AnonPay}}$ proceed as below. In this case both A and the unknown B are honest. We know that some $(\textsc{Pay}, \mathsf{A}, \mathsf{B}, a)$ was input to P_{A} on $\mathcal{F}_{\mathsf{AnonPay}}$ but we do not get a nor B. Simulate as follows.

1. Fetch $(\mathsf{A}, b_{\mathsf{A}}, c_{\mathsf{A}}, \rho_{\mathsf{A}}, \mathsf{ek}_{\mathsf{A}}, \mathsf{dk}_{\mathsf{A}}, \mathsf{nonce}_{\mathsf{A}})$ from local storage.[a]
2. Get the public values $(\mathsf{crs}, \mathsf{ek}, \mathsf{ck}, k^*)$ of the service from $\mathcal{F}_{\textsc{Ledger}}$.
3. If not done before generate $(\mathsf{ek}_{\mathcal{S}}, \mathsf{dk}_{\mathcal{S}}) \leftarrow \mathsf{RPKE.Gen}(\mathsf{pp})$ and let $\mathsf{ek}'_{\mathsf{B}} = \mathsf{ek}_{\mathcal{S}}$.[b]
4. Let $a' = 0$ and $\mathsf{B}' = \mathsf{A}$ and let $\mathsf{tid}' = \mathsf{Com.Commit}_{\mathsf{ck}}((\mathsf{A}, \mathsf{B}', a', \mathsf{nonce}_{\mathsf{A}}); \rho_1)$. [c]
5. Compute the coin $\mathsf{coin} = \mathsf{RPKE.Enc}_{\mathsf{ek}'_{\mathsf{B}}}(\mathsf{tid}'; \rho_2)$.
6. Let $b'_{\mathsf{A}} = b_{\mathsf{A}} - a'$, compute $c'_{\mathsf{A}} = \mathsf{Com.Commit}_{\mathsf{ck}}(b'_{\mathsf{A}}, \rho'_{\mathsf{A}})$.
7. $\pi \leftarrow \mathsf{NIZKPoK.Sim}^{\mathcal{O}}_{\mathsf{tSim}}(\mathcal{R}_{\textsc{Pay}}, (c_{\mathsf{A}}, \mathsf{coin}, c_{\mathsf{A}'}, \mathsf{A}, \mathsf{nonce}_{\mathsf{A}})$.
8. Sample $\mathsf{L}_{\mathsf{A}} \leftarrow \{0,1\}^{\lambda}$, compute $d \leftarrow \mathsf{PKE.Enc}_{\mathsf{ek}}((\mathsf{A}, \mathsf{L}_{\mathsf{A}}); \rho_4)$.
9. Run $\mathcal{F}_{\textsc{Ledger}}.\textsc{Broadcast}(\mathsf{Tx}, \phi_{\textsc{Pay}})$ with $\mathsf{Tx} = (\mathsf{A}, c'_{\mathsf{A}}, \mathsf{coin}, \pi, d)$ and the same $\phi_{\textsc{Pay}}$ as the protocol.
10. Wait for $(\mathsf{Tx}, \phi_{\textsc{Pay}})$ to appear on $\mathcal{F}_{\textsc{Ledger}}$ in some position p. When this happens the blockchain stateful abstraction layer replaces $(\mathsf{A}, c_{\mathsf{A}}, \mathsf{ek}_{\mathsf{A}}, \mathsf{nonce}_{\mathsf{A}})$ by $(\mathsf{A}, c'_{\mathsf{A}}, \mathsf{ek}_{\mathsf{A}}, \mathsf{nonce}'_{\mathsf{A}} = \mathsf{nonce}_{\mathsf{A}} + 1)$.
11. Store $(\mathsf{A}, b'_{\mathsf{A}}, c'_{\mathsf{A}}, \rho'_{\mathsf{A}}, \mathsf{ek}_{\mathsf{A}}, \mathsf{dk}_{\mathsf{A}}, \mathsf{nonce}'_{\mathsf{A}} = \mathsf{nonce}_{\mathsf{A}} + 1)$.
12. Run $\mathcal{F}_{\textsc{Ledger}}.\textsc{ProveValid}(p) \rightarrow \mathsf{TxProof}$.
13. Run $\mathcal{F}_{\textsc{AAT}}.\textsc{DropOff}(\mathsf{P}_{\mathsf{A}}, \mathsf{P}_{\mathsf{B}'}, \mathsf{tid}, m)$, where $m = ((\mathsf{Tx}, p, \mathsf{TxProof}, (\mathsf{nonce}_{\mathsf{A}}, \rho_1, \rho_2)), (\mathsf{L}_{\mathsf{A}}, \rho_4))$.
14. Input $(\textsc{MakeDeducted}, \mathsf{atid})$ to $\mathcal{F}_{\mathsf{AnonPay}}$.[d]

Callback Observe (Honest Sender, Honest Receiver) When the m sent via $\mathcal{F}_{\textsc{AAT}}.\textsc{DropOff}$ gets delivered by the adversary input $(\textsc{MakeObservable}, \mathsf{atid})$ to $\mathcal{F}_{\mathsf{AnonPay}}$. [e]

Callback Collect (Honest Sender, Honest Receiver) When it happens on $\mathcal{F}_{\textsc{Ledger}}$ that $p_{\textsc{updated}} \geq p$ input $(\textsc{MakeCollectable}, \mathsf{atid})$ to $\mathcal{F}_{\mathsf{AnonPay}}$ and add atid to UCHP. [f]

---

[a] Note that at this step the protocol terminates if $a > b_{\mathsf{A}}$. In the simulation we would never make it here when $a > b_{\mathsf{A}}$ as we only get $(\textsc{Pay}, \mathsf{A}, \mathsf{atid})$ from $\mathcal{F}_{\mathsf{AnonPay}}$ if $a \leq b_{\mathsf{A}}$. So the protocol and simulation check $a \leq b_{\mathsf{A}}$ at the same time and behave the same when $a > b_{\mathsf{A}}$.

[b] In this step the protocol retrieves ek_{B} and we know this would work as the ideal functionality checked $\mathsf{B} \in \mathsf{Accounts}$. However, we do not know ek_{B} in the simulation. To mitigate this we generate a set of keys for the simulator and use the simulators key for all unknown receivers.

[c] Here the protocol computes tid. Instead now tid is generated by $\mathcal{F}_{\mathsf{AnonPay}}$ using the distribution Tid that we gave. We do not learn tid, so we use a dummy transfer identifier tid' committing to dummy values.

[d] Here the protocol outputs $(\textsc{Pay}, \mathsf{A}, \mathsf{B}, a, \mathsf{tid})$ to P_{A} if the payment was accepted by the blockchain and in the simulation we only reach this step if $(\mathsf{Tx}, \phi_{\textsc{Pay}})$ appeared on $\mathcal{F}_{\textsc{Ledger}}$. Inputting $(\textsc{MakeDeducted}, \mathsf{atid})$ to $\mathcal{F}_{\mathsf{AnonPay}}$ makes it also output $(\textsc{Pay}, \mathsf{A}, \mathsf{B}, a, \mathsf{tid})$ to P_{A}.

[e] This is a perfect simulation as in the protocol when the m is delivered for an honest-honest payment then a call to $\textsc{Observe}$ on the receiver will return $(\textsc{Observe}, \mathsf{tid}, \mathsf{A}, \mathsf{B}, a, \top)$. Inputting $(\textsc{MakeObservable}, \mathsf{atid})$ to $\mathcal{F}_{\mathsf{AnonPay}}$ makes it behave the same.

[f] This is a perfect simulation as in the protocol the party B rejects if and only if $p_{\textsc{updated}} < p$, cf. Lemma 11.

Fig. 17. Simulator $\mathcal{S}$ (Pay, Honest Sender)

Observe (Honest Receiver) Here there is nothing to simulate. The cases **Callback Observable (Honest Sender, Honest Receiver)** and **Callback Observable (Corrupt Sender, Honest Receiver)** are constructed to make the protocol and $\mathcal{F}_{\mathsf{AnonPay}}$ give the same replies.

Observe (Corrupt Receiver) Here there is nothing to simulate. In the simulation the command does not affect the state of $\mathcal{F}_{\mathsf{AnonPay}}$ and in the (simulated) protocol there is no notion of the adversary having carried out this command. It can do this simply by looking at Ledger.

Fig. 18. Simulator $\mathcal{S}$ (Observe)

Collect (Honest Sender, Honest Receiver, Too Early) On input $(\textsc{Collect}, \mathsf{B}, \mathsf{atid}, \textsc{Too Early})$ simulate as follows.

1. Simulate the run of $\textsc{Observe}$ by doing nothing. This is a perfect simulation as $\mathcal{F}_{\text{AAT}}$ does not leak anything on collection.[a]
2. Simulate the fetching of the unknown L_A and p by doing nothing.
3. From $\mathcal{F}_{\textsc{Ledger}}$ fetch the most recent CRaB key k^* and position $p_{\textsc{updated}}$. Then simulate that we learned that $p_{\textsc{updated}} < p$ by terminating, as would the protocol.

Collect (Honest Sender, Honest Receiver, Timely) On input $(\textsc{Collect}, \mathsf{B}, \mathsf{atid}, \mathsf{tid})$ simulate as follows.

1. If UCHP is empty then terminate the simulation. By Lemma 10 this happens with negligible probability, so we can ignore it. Otherwise, pick the lexicographically smallest $\mathsf{atid}' \in \mathsf{UCHP}$ and remove it from UCHP.[b] Let $(\textsc{Pay}, \mathsf{A}', \mathsf{atid}')$ be the input from $\mathcal{F}_{\mathsf{AnonPay}}$ that made $\mathcal{S}$ add atid' to UCHP and $m' = ((\mathsf{Tx} = (\mathsf{A}', \cdot, \mathsf{coin}, \cdot, d'), p', \mathsf{TxProof}', (\mathsf{nonce}'_\mathsf{A}, \rho'_1, \rho'_2)), (\mathsf{L}_{\mathsf{A}'}, \rho'_4))$ be the message sent in that payment.
2. Fetch the $\mathsf{L}'_{\mathsf{A}'}$ where $d' = \mathsf{PKE.Enc}_{\mathsf{ek}}((\mathsf{A}', \mathsf{L}_{\mathsf{A}'}), \rho'_4)$ and the position p' from m'.
3. From $\mathcal{F}_{\textsc{Ledger}}$ fetch the most recent CRaB key k^* and position $p_{\textsc{updated}}$. Since $\mathsf{atid}' \in \mathsf{UCHP}$ we have that $p_{\textsc{updated}} \geq p'$.
4. Let $\mathsf{L}'_\mathsf{S} = \mathsf{CRaB.Eval}(k^*, p')$.
5. Let $\mathsf{L}' = \mathsf{Hash}(\mathsf{L}'_{\mathsf{A}'} \oplus \mathsf{L}'_S)$.
6. Compute $\mathsf{pos} = \mathsf{SOROM.Pos}(\mathsf{L}')$.
7. Fetch the data at positions $j \in \mathsf{pos}$ in OM from $\mathcal{F}_{\textsc{Ledger}}$ and for $j \in \mathsf{pos}$ let $\mathsf{OM}[j] = (\cdot, \mathsf{coin}_j)$.
8. $\pi_1 \leftarrow \mathsf{NIZK.Sim}_{\mathsf{tSim}}(\mathcal{R}_{\textsc{OrDec}}, (\mathsf{ck}, \{\mathsf{coin}_j\}_{j \in \mathsf{pos}}, \mathsf{tid}))$.
9. Let $a' = 0$ and let $b'_\mathsf{B} = b_\mathsf{B} + a'$, compute $c'_\mathsf{B} = \mathsf{Com.Commit}(b'_\mathsf{B}, \rho'_\mathsf{B})$
10. $\pi_2 \leftarrow \mathsf{NIZKPoK.Sim}^{\mathcal{O}}_{\mathsf{tSim}}(\mathcal{R}_{\textsc{Collect}}, (\mathsf{ck}, c_\mathsf{B}, \mathsf{B}, \mathsf{tid}, c'_\mathsf{B}))$.
11. Run $\mathcal{F}_{\textsc{Ledger}}.\textsc{Broadcast}(\mathsf{Tx}, \phi_{\textsc{Collect}})$ with $\mathsf{Tx} = (\mathsf{B}, \mathsf{L}', \mathsf{tid}, \pi_1, \pi_2)$ and the same $\phi_{\textsc{Collect}}$ as in the protocol.
12. When the transaction is posted the blockchain stateful abstraction layer replaces $(\mathsf{B}, c_\mathsf{B}, \mathsf{ek}_\mathsf{B}, \mathsf{nonce}_\mathsf{B})$ by $(\mathsf{B}, c'_\mathsf{B}, \mathsf{ek}_\mathsf{B}, \mathsf{nonce}_\mathsf{B})$ and lets $\mathsf{Spent} = \mathsf{Spent} \cup \{\mathsf{tid}\}$.
13. Store $(\mathsf{B}, b'_\mathsf{B}, c'_\mathsf{B}, \rho'_\mathsf{B}, \mathsf{ek}_\mathsf{B}, \mathsf{dk}_\mathsf{B}, \mathsf{nonce}_\mathsf{B})$ on local storage.

[a] Since we get input $(\textsc{Collect}, \mathsf{B}, \textsc{Too Early})$ we know that we are in the case where $(\textsc{Pay}, \widetilde{\mathsf{atid}}, \mathsf{tid}, \mathsf{A}, \mathsf{B}, a) \in \mathsf{Observable} \setminus \mathsf{Collectable}$. Note that by construction of **Callback Collect** we have that $(\textsc{Pay}, \widetilde{\mathsf{atid}}, \mathsf{tid}, \mathsf{A}, \mathsf{B}, a) \in \mathsf{Collectable}$ if and only if $(\textsc{Pay}, \widetilde{\mathsf{atid}}, \mathsf{tid}, \mathsf{A}, \mathsf{B}, a) \in \mathsf{Observable}$ and $p_{\textsc{updated}} \geq p$, where p is the position of the payment on $\mathcal{F}_{\textsc{Ledger}}$. So we conclude that $p_{\textsc{updated}} < p$. But we do not know p nor which simulated payment is being collected.

[b] Since we get input $(\textsc{Collect}, \mathsf{B}, \mathsf{atid}, \mathsf{tid})$ (as opposed to $(\textsc{Collect}, \mathsf{B}, \textsc{Too Early})$) we know that we are in the case where $(\textsc{Pay}, \widetilde{\mathsf{atid}}, \mathsf{tid}, \mathsf{A}, \mathsf{B}, a) \in \mathsf{Collectable}$. Note that by construction of **Callback Collect** we have that $(\textsc{Pay}, \widetilde{\mathsf{atid}}, \mathsf{tid}, \mathsf{A}, \mathsf{B}, a) \in \mathsf{Collectable}$ if and only if $(\textsc{Pay}, \widetilde{\mathsf{atid}}, \mathsf{tid}, \mathsf{A}, \mathsf{B}, a) \in \mathsf{Observable}$ and $p_{\textsc{updated}} \geq p$. By inspection of Steps 1–3 in **Collect** in Fig. 15 we see that the protocol will proceed to collect a coin in this case. The simulation should do the same, but we do not know A or which simulated payment corresponds to tid. So we collect some other collectible coin instead.

Fig. 19. Simulator $\mathcal{S}$ (Collect, Honest Sender / Honest Receiver)

Initiate Pay (Corrupt Sender) If in the simulation $(\mathsf{Tx}, \phi_{\text{PAY}})$ appears on $\mathcal{F}_{\text{LEDGER}}$ in some position p where $\mathsf{Tx} = (\mathsf{A}, c'_{\mathsf{A}}, \mathsf{coin}, \pi, d)$ and where $\mathsf{P_A}$ is corrupted, then proceed as follows.

1. Use the extraction trapdoor tExt for $\mathsf{NIZKPoK}$ to compute a witness $w = ((b_{\mathsf{A}}, \rho_{\mathsf{A}}), (\mathsf{B}, a, \rho_1, \rho_2), \rho'_{\mathsf{A}})$ such that $((c_{\mathsf{A}}, \mathsf{coin}, c_{\mathsf{A}'}, \mathsf{A}, \mathsf{nonce_A}), w) \in \mathcal{R}_{\text{PAY}}$. This is possible by construction of ϕ_{PAY} and simulation extraction of $\mathsf{NIZKPoK}$. Note that this in particular means that
$$c_{\mathsf{A}} = \mathsf{Com.Commit_{ck}}(b_{\mathsf{A}}, \rho_{\mathsf{A}}) \wedge$$
$$((\mathsf{coin}, \mathsf{Com.Commit_{ck}}(\mathsf{A}, \mathsf{B}, a, \mathsf{nonce_A}; \rho_1)), \rho_2) \in \mathcal{R}_{\text{ENC}} \wedge$$
$$c'_{\mathsf{A}} = \mathsf{Com.Commit_{ck}}(b_{\mathsf{A}} - a, \rho'_{\mathsf{A}}) \wedge b_{\mathsf{A}} \geq a \geq 0$$
2. Input $(\text{PAY}, \mathsf{A}, \mathsf{B}, a)$ to $\mathsf{P_A}$ on $\mathcal{F}_{\mathsf{AnonPay}}$ and get back $(\text{PAY}, \mathsf{A}, \mathsf{atid})$. In response to this $\mathcal{F}_{\mathsf{AnonPay}}$ will ask $\mathcal{S}$ for a transfer identifier. Use
$$\mathsf{tid} = \mathsf{Com.Commit_{ck}}(\mathsf{A}, \mathsf{B}, a, \mathsf{nonce_A}; \rho_1) \ .$$
If $\mathcal{F}_{\mathsf{AnonPay}}$ rejects tid because tid was not fresh, then abort the simulation. By Lemma 7 we can ignore this event.
3. Input $(\text{MAKEDEDUCTED}, \mathsf{atid})$ to $\mathcal{F}_{\mathsf{AnonPay}}$ to add $(\text{PAY}, \mathsf{atid}, \mathsf{tid}, \mathsf{A}, \mathsf{B}, a)$ to $\mathsf{Deducted}$.

Callback Observable (Corrupt Sender, Honest Receiver) If **Initiate Pay (Corrupt Sender)** was executed and B is honest and the adversary delivers on $\mathcal{F}_{\text{AAT}}$ to B a message m with message identifier tid such that the check in **Observe** in Fig. 15 would go through, then input $(\text{MAKEOBSERVABLE}, \mathsf{atid})$ to $\mathcal{F}_{\mathsf{AnonPay}}$. [a]

Callback Collectable (Corrupt Sender, Honest Reiver) If **Initiate Pay (Corrupt Sender)** and **Callback Observable (Corrupt Sender, Honest Receiver)** was executed and Tx appears in position p and it happens that $p_{\text{UPDATED}} \geq p$ then input $(\text{MAKECOLLECTABLE}, \mathsf{atid})$ to $\mathcal{F}_{\mathsf{AnonPay}}$. [b]

[a] This is a perfect simulation as $\mathcal{F}_{\mathsf{AnonPay}}$ and the protocol would now respond identical.
[b] This is a perfect simulation as $\mathcal{F}_{\mathsf{AnonPay}}$ and the protocol would now respond identical, cf. Lemma 11.

Fig. 20. Simulator $\mathcal{S}$ (Pay, Corrupt Sender).

Collect (Corrupt Sender, Honest Receiver, Too Early) As in the protocol.

Collect (Corrupt Sender, Honest Receiver, Timely) As in the protocol. This will be a perfect simulation by Lemma 11.

Fig. 21. Simulator $\mathcal{S}$ (Collect, Corrupt Sender / Honest Receiver)

Collect (Corrupt Receiver) If $(\mathsf{Tx}, \phi_{\textsc{Collect}})$ appears on $\mathcal{F}_{\textsc{Ledger}}$ with $\mathsf{Tx} = (\mathsf{B}, \mathsf{L}, \mathsf{tid}, \pi_1, \pi_2)$ then simulate as follows.

1. Use the extraction trapdoor tExt to extract witnesses such that
$$((\mathsf{ck}, c_{\mathsf{B}}, \mathsf{B}, \mathsf{tid}, c'_{\mathsf{B}}), ((b_{\mathsf{B}}, \rho_{\mathsf{B}}), (\mathsf{A}, a, \mathsf{nonce}_{\mathsf{A}}, \rho_1), \rho'_{\mathsf{B}})) \in \mathcal{R}_{\textsc{Collect}}$$
and $\mathsf{tid} \notin \mathsf{Spent}$.
2. Abort the simulation if the following happens: 1) A is corrupt and there was no previous execution of **Initiate Pay (Corrupt Sender)** where $\mathcal{S}$ input $(\textsc{Pay}, \mathsf{A}, \mathsf{B}, a)$ to $\mathcal{F}_{\mathsf{AnonPay}}$ got back some atid and then specified the transfer identifier tid to be used by $\mathcal{F}_{\mathsf{AnonPay}}$ or 2) A is honest and there was no previous execution of **Initiate Pay (Honest Sender, Corrupt Receiver)** where $\mathcal{S}$ got input $(\textsc{Pay}, \mathsf{A}, \mathsf{B}, a, \mathsf{atid})$ from $\mathcal{F}_{\mathsf{AnonPay}}$ and then specified tid. This happens with negligible probability by Lemma 9.
3. Input $(\textsc{MakeObservable}, \mathsf{atid})$ followed by $(\textsc{MakeCollectable}, \mathsf{atid})$ to $\mathcal{F}_{\mathsf{AnonPay}}$, and then input $(\textsc{Collect}, \mathsf{tid}, \mathsf{A}, \mathsf{B}, a)$ to $\mathcal{F}_{\mathsf{AnonPay}}$ on behalf of P_{B}. This is allowed as B is corrupted.
This maintains the amount invariant if $(\textsc{Pay}, \cdot, \mathsf{tid}, \mathsf{A}, \mathsf{B}, a) \notin \mathsf{Collected}$. This is the case as $\mathsf{tid} \notin \mathsf{Spent}$.

Fig. 22. Simulator $\mathcal{S}$ (Collect, Corrupt Receiver).

run the protocol honestly, but introduce a few cases where we abort the simulation for use in the following hybrid arguments. All these cases occur with negligible probability.

9.3 Analysis

We now prove (2). We do the proof by a hybrid argument where we define distributions $H_1, \ldots, H_{11}$ and prove that
$$\mathrm{Exec}_{\mathcal{F}_{\mathsf{AnonPay}}, \mathcal{S}, \mathcal{E}}(1^{\lambda}) \approx H_1 \tag{3}$$
and $H_i \approx H_{i+1}$ for $i = 1, \ldots, 10$, and
$$H_{11} \approx \mathrm{Exec}_{\Pi_{\textsc{AnonPay}}, \mathcal{F}_{\textsc{Ledger}}, \mathcal{F}_{\textsc{AAT}}, \mathcal{F}_{\textsc{Service}}, \mathcal{O}, \mathcal{E}}(1^{\lambda}) \ . \tag{4}$$

Letting $H_1 = \mathrm{Exec}_{\mathcal{F}_{\mathsf{AnonPay}}, \mathcal{S}, \mathcal{E}}(1^{\lambda})$ and $H_{11} = \mathrm{Exec}_{\Pi_{\textsc{AnonPay}}, \mathcal{F}_{\textsc{Ledger}}, \mathcal{F}_{\textsc{AAT}}, \mathcal{F}_{\textsc{Service}}, \mathcal{O}, \mathcal{E}}(1^{\lambda})$ makes the end cases trivial. We now look at the steps.

Hybrid 2 Let H_2 be defined as the simulation $\mathrm{Exec}_{\mathcal{F}_{\mathsf{AnonPay}}, \mathcal{S}, \mathcal{E}}(1^{\lambda})$ except that:

Change 2.1. In Step 2 in **Collect (Honest Sender, Honest Receiver, Timely)** in Fig. 19, instead of getting $\mathsf{L}'_{\mathsf{A}'}$ from d' we use the atid' from UCHP to find the matching execution of **Initiate Pay (Honest Sender, Honest Receiver)** in Fig. 17 and then we fetch $\mathsf{L}'_{\mathsf{A}'}$ from Step 8 in that execution.

Change 2.2. Run the $\mathcal{F}^{(2)}_{\textsc{Service}}$ in Fig. 23 instead of $\mathcal{F}_{\textsc{Service}}$.

Lemma 12. $H_2 \approx H_1$.

Proof. This follows from correctness of PKE. Whether we decrypt the ciphertexts d' and d or recall what plaintext we have put inside them does not matter. □

Receive Coin Whenever a new payment $(\mathsf{Tx}, \phi_{\text{PAY}})$ appears in position p, proceed as follows. Transactions must be consumed in increasing order.

1. Same: Parse $(\mathsf{A}, c'_{\mathsf{A}}, \mathsf{coin}, \pi, d) \leftarrow \mathsf{Tx}$.
2. Same: Let $\iota = \iota + 1$.
3. If d comes from an execution of **Initiate Pay (Honest Sender, Honest Receiver)** then inspect that execution and find the identity $\mathcal{C}$ of the sender and the label $\mathsf{L}_{\mathcal{C}}$ encrypted in d. If $\mathcal{C} \neq \mathsf{A}$ then terminate. Otherwise let $\mathsf{L}_{\mathsf{A}} = \mathsf{L}_{\mathcal{C}}$. If d does not come from an execution of **Initiate Pay (Honest Sender, Honest Receiver)** then let $(\mathcal{C}, \mathsf{L}_{\mathsf{A}}) = \mathsf{PKE.Dec}_{\mathsf{dk}}(d)$. If $\mathcal{C} \neq \mathsf{A}$ then terminate.
4. All other steps are as in $\mathcal{F}_{\text{SERVICE}}$.

Fig. 23. The $\mathcal{F}^{(2)}_{\text{SERVICE}}$ used in H_2.

Receive Coin Whenever a new payment $(\mathsf{Tx}, \phi_{\text{PAY}})$ appears in position p, proceed as follows. Transactions must be consumed in increasing order.

1. Parse $(\mathsf{A}, c'_{\mathsf{A}}, \mathsf{coin}, \pi, d) \leftarrow \mathsf{Tx}$.
2. Let $\iota = \iota + 1$.
3. If d comes from an execution of **Initiate Pay (Honest Sender, Honest Receiver)** with sender $\mathcal{C}$ where d was supposed to contain $(\mathcal{C}, \mathsf{L}_{\mathcal{C}})$, then proceed as follows. If $\mathcal{C} \neq \mathsf{A}$ then terminate. This prevents replay attacks if and when that protocols does the same. If $\mathcal{C} = \mathsf{A}$ then let $\mathsf{L}_{\mathsf{A}} = \mathsf{L}_{\mathcal{C}}$. When d does not come from an execution of **Initiate Pay (Honest Sender, Honest Receiver)** then let $(\mathcal{C}, \mathsf{L}_{\mathsf{A}}) = \mathsf{PKE.Dec}_{\mathsf{dk}}(d)$. If $\mathcal{C} \neq \mathsf{A}$ then terminate.
4. Let $\mathsf{L}_{\mathsf{S}} = \mathsf{CRaB.Eval}(k, p)$, where p is the position of Tx on the ledger.
5. Let $\mathsf{L} = \mathsf{Hash}(\mathsf{L}'_{\mathsf{A}} \oplus \mathsf{L}_{\mathsf{S}})$.
6. Let $\mathsf{lab} = \mathsf{SKE.Enc}_K(0^\lambda)$ and recall that lab was supposed to encrypt using a map $\mathsf{Plain}(\mathsf{lab}) = \mathsf{L}$.
7. Update CRaB key: $k^* \leftarrow \mathsf{CRaB.Prefix}(k, p)$ and $p_{\text{UPDATED}} = p$.
8. Update k^*, p_{UPDATED}, and $\mathsf{OM}[0] = (\mathsf{lab}, \mathsf{coin})$ on $\mathcal{F}_{\text{LEDGER}}$.
9. Go to **Route**

Route

1. Compute $(j_1, \ldots, j_\ell) = \mathsf{SOROM.Pos}(\iota)$.
2. For $k = 1, \ldots, \ell$ read $(\mathsf{lab}_k, \mathsf{coin}_k) \leftarrow \mathsf{OM}[j_k]$ from $\mathcal{F}_{\text{LEDGER}}$.
3. For $k = 1, \ldots, \ell$ let $L_k = \mathsf{Plain}(\mathsf{lab}_k)$.
4. Compute the routing permutation $\pi = \mathsf{SOROM.Route}(\iota, L_1, \ldots, L_\ell)$.
5. For $k = 1, \ldots, \ell$ let $\mathsf{lab}'_k \leftarrow \mathsf{SKE.Enc}_K(0^\lambda)$ and $\mathsf{Plain}(\mathsf{lab}'_k) = L_{\pi(k)}$.
6. For $k = 1, \ldots, \ell$ let $\mathsf{coin}'_k \leftarrow \mathsf{RPKE.Ran}(\mathsf{coin}_{\pi(k)})$.
7. For $k = 1, \ldots, \ell$ update $\mathsf{OM}[j_k] \leftarrow (\mathsf{lab}'_k, \mathsf{coin}'_k)$ on $\mathcal{F}_{\text{LEDGER}}$.

Fig. 24. The $\mathcal{F}^{(3)}_{\text{SERVICE}}$ used in H_3 and H_4.

Hybrid 3 Let H_3 be defined as H_2 except that:

Change 3.1. In Step 8 in **Initiate Pay (Honest Sender, Honest Receiver)** in Fig. 17 we let d be an encryption of $\perp$ instead of $(\mathsf{A}, \mathsf{L}_\mathsf{A})$. However, recall that d was supposed to contain $(\mathsf{A}, \mathsf{L}_\mathsf{A})$.

Change 3.2. In Step 5 in **Initiate Pay (Honest Sender, Honest Receiver)** in Fig. 17 we encrypt $\perp$ instead of tid'.

Change 3.3. We replace the run of $\mathcal{F}^{(2)}_{\text{SERVICE}}$ by $\mathcal{F}^{(3)}_{\text{SERVICE}}$ in Fig. 24.

Lemma 13. *$H_3 \approx H_2$.*

Proof. Change 3.1 does not matter by IND-CCA security of PKE. We no longer decrypt the d's from Step 8 in **Initiate Pay (Honest Sender, Honest Receiver)** anywhere in the simulation. So we could get them from an IND-CCA oracle encrypting either $\perp$ or $(\mathsf{A}, \mathsf{L}_\mathsf{A})$ and embed them in the simulation.

Change 3.2 does not matter by IND-CPA of RPKE as we do not decrypt coin anywhere. Whereever coin is used, we give simulated proofs. We could therefore get encryptions of either tid' or $\perp$ and embed them in the simulation.

After the above two changes Change 3.3 does not matter by correctness of SKE and IND-CPA of SKE. Whether we decrypt a ciphertext or recall what is it in does not matter by correcteness. And after that we can change what it encrypts to 0^λ by IND-CPA. □

Hybrid 4 Let H_4 be defined as H_3 except that:

Change 4.1. In Step 6 in Fig. 24 we do the following instead. Let $\mathsf{lab} = \mathsf{SKE.Enc}_K(0^\lambda)$. Then if the payment is from honest A to some (unknown) honest B sample a uniformly random $\mathsf{L}^* \leftarrow \mathcal{L}$ and let $\mathsf{Plain}(\mathsf{lab}) = \mathsf{L}^*$. Otherwise let $\mathsf{Plain}(\mathsf{lab}) = \mathsf{L}$.

Note that this just means that for honest-honest payments we route a fresh uniformly random label and not the one returned by Hash.

Lemma 14. *$H_4 \approx H_3$.*

Proof. We can prove this via an easy reduction to the SOROM game. We play the role of adversary in the SOROM game. We then produce a version of the simulation via our blackbox access to the SOROM game. When $b = 0$ in the SOROM game we produce H_3. When $b = 1$ in the SOROM game we produce H_4. So, if H_3 and H_4 could be distinguished we could use this to win the SOROM game. This shows that $H_4 \approx H_3$ when the SOROM construction is secure.

Let $\mathsf{lab} = \mathsf{SKE.Enc}_K(0^\lambda)$ and update the map Plain as follows. If the payment is honest-honest then let $\mathsf{Plain}(\mathsf{lab}) = \perp$. Otherwise let $\mathsf{Plain}(\mathsf{lab}) = \mathsf{L}$. The reason for this definition is that for the corrupt lab we know the corresponding label L being routed by the SOROM game. But for honest-honest labels we do not know the label being routed. If $b = 0$ it is the L we were returned above. But if $b = 1$ it is another independent and uniformly random label. So, read $\mathsf{Plain}(\mathsf{lab}) = \perp$ as "label unknown".

Then replace the **Route** part of $\mathcal{F}^{(3)}_{\text{SERVICE}}$ by the following.

1. Compute $(j_1, \ldots, j_\ell) = \mathsf{SOROM.Pos}(\iota)$. These will be the same as the positions used by the SOROM game as Pos is deterministic.

2. For $k = 1, \dots, \ell$ read $(\mathsf{lab}_k, \mathsf{coin}_k) \leftarrow \mathsf{OM}[j_k]$ from $\mathcal{F}_{\text{LEDGER}}$.
3. For $k = 1, \dots, \ell$ let $L_k = \mathsf{Plain}(\mathsf{lab}_k)$.
4. The routing permutation $\pi = \mathsf{SOROM.Route}(\iota, L_1, \dots, L_\ell)$ is computed by the SOROM game. For the corrupted labels L_k it tells us where they are routed to, and we can compute π' moving the corrupted label correctly. It can move the honest labels in any way, say fill up empty slots in lexicographic order.
5. For $k = 1, \dots, \ell$ let $\mathsf{lab}'_k \leftarrow \mathsf{SKE.Enc}_K(0^\lambda)$ and $\mathsf{Plain}(\mathsf{lab}'_k) = L_{\pi'(k)}$.
6. For $k = 1, \dots, \ell$ let $\mathsf{coin}'_k \leftarrow \mathsf{RPKE.Ran}(\mathsf{coin}_{\pi'(k)})$.
7. For $k = 1, \dots, \ell$ update $\mathsf{OM}[j_k] \leftarrow (\mathsf{lab}'_k, \mathsf{coin}'_k)$ on $\mathcal{F}_{\text{LEDGER}}$.

Notice that we can indeed run the above process in poly-time as to produce H_4 we never used the value of $\mathsf{Plain}(\mathsf{lab})$ anywhere except in **Route**, so the fact that we do not know the value in the above will not become a problem. Note, in particular, that in Step 5 in Fig. 19 when we simulate picking up a coin we get the label from Hash and not from the SOROM.

It is easy to see that when $b = 0$ we produce exactly H_3 and when $b = 1$ we produce exactly H_4. □

Hybrid 5 Let H_5 be defined as H_4 except that:

Change 5.1. In Step 1 in **Collect (Honest Sender, Honest Receiver, Timely)** in Fig. 19 we pick atid' from UCHP differently. We inspect $\mathcal{F}_{\mathsf{AnonPay}}$ to learn which payment $(\text{PAY}, \mathsf{A}, \mathsf{B}, a)$ created tid and then we find the atid in $\mathcal{F}_{\mathsf{AnonPay}}$ corresponding to this payment, i.e., the atid such that $(\text{PAY}, \mathsf{atid}, \mathsf{tid}, \mathsf{A}, \mathsf{B}, a) \in \mathsf{Collectable}$. Then we let $\mathsf{atid}' = \mathsf{atid}$. Note that this atid is in fact in UCHP by Lemma 9, as it cannot have been collected by another party and still has not been collected by B. □

Lemma 15. $H_5 \approx H_4$.

Proof. This holds information theoretically. At this point in the hybrids all uncollected honest-honest payments look exactly the same to the adversary. The SOROM are routing uniformly random labels independent of the labels $\mathsf{Hash}(\mathsf{L_A} \oplus \mathsf{L}_S)$ used by the transfers and these labels are not leaked anywhere else until used for collection. In more detail, after **Change 3.1** where in Step 8 in **Initiate Pay (Honest Sender, Honest Receiver)** in Fig. 17 we let d be an encryption of $\perp$ instead of $(\mathsf{A}, \mathsf{L_A})$ we can defer even defining $\mathsf{L_A}$ until the payment is picked up. So as part of initiating the payment we do not sample $\mathsf{L_A}$. Only when we do the collection do we sample $\mathsf{L_A}$ and then we *define* $\mathsf{Hash}(\mathsf{L_A} \oplus \mathsf{L_S}) = \mathsf{L}$. This is possible by Lemma 3. To do this we do not even have to decide on which corresponding payment $\mathsf{L_A}$ was made. Therefore the only thing connecting a collection to the corresponding payment is the point p that we use to compute $\mathsf{L_S}$. But information of p only goes into $\mathsf{L_S}$ and even if we gave $\mathsf{L_A} \oplus \mathsf{L_S}$ to the adversary, this would be a one-time encryption of $\mathsf{L_S}$. Hence p is information theoretically hidden from the adversary. □

Hybrid 6 In H_6 we reverse all changes again, except that we keep the change from H_5, i.e., we now have the following situation.

Change 6.1. Everything is run as in $H_1 = \text{Exec}_{\mathcal{F}_{\mathsf{AnonPay}}, \mathcal{S}, \mathcal{E}}(1^\lambda)$, except that:

Change 6.2. In Step 1 in **Collect (Honest Sender, Honest Receiver, Timely)** in Fig. 19 we pick atid' from UCHP differently. We inspect $\mathcal{F}_{\mathsf{AnonPay}}$ to learn which payment $(\textsc{Pay}, \mathsf{A}, \mathsf{B}, a)$ created tid and then we find the atid in $\mathcal{F}_{\mathsf{AnonPay}}$ corresponding to this payment, i.e., the atid such that $(\textsc{Pay}, \mathsf{atid}, \mathsf{tid}, \mathsf{A}, \mathsf{B}, a) \in \mathsf{Collectable}$. Then we let $\mathsf{atid}' = \mathsf{atid}$. Note that this atid is in fact in UCHP by Lemma 9, as it cannot have been collected by another party and still have not been collected by B.

The following holds by using the same arguments as when we introduced the changes that we now reversed.

Lemma 16. *$H_6 \approx H_5$.*

Hybrid 7 Let H_7 be defined as H_6 except that:

Change 7.1. In Fig. 17 **Initiate Pay (Honest Sender, Honest Receiver)** inspect the copy of $\mathcal{F}_{\mathsf{AnonPay}}$ to find the correct value of the receiver B and amount a. Note that the UC simulator is not allowed to do this, but we are describing a hybrid and is free to do as we please for sake of argument. Then in Step 3 use $\mathsf{ek}'_{\mathsf{B}} = \mathsf{ek}_{\mathsf{B}}$ instead of $\mathsf{ek}'_{\mathsf{B}} = \mathsf{ek}_{\mathcal{S}}$. And in Step 3 we use $a' = a$ and $\mathsf{B}' = \mathsf{B}$ instead of $a' = 0$ and $\mathsf{B}' = \mathsf{A}$.

Lemma 17. *$H_7 \approx H_6$.*

Proof. The change is indistinguishable to the adversary by hiding of Com and key anonymity of RPKE. □

The reason why we do not yet replace the second simulated proof is that the instance might not be true yet. We are still not updating accounts correctly when picking up payments, so the accounts might not have sufficient balance. We fix this in the next two hybrids.

Hybrid 8 Let H_8 be defined as H_7 except that:

Change 8.1. We change $\mathcal{F}_{\mathsf{AnonPay}}$ such that when it picks tid for an honest-honest payment then it uses $\mathsf{tid} = \mathsf{tid}'$ for the $\mathsf{tid}' = \mathsf{Com.Commit}_{\mathsf{ck}}((\mathsf{A}, \mathsf{B}', a', \mathsf{nonce}_{\mathsf{A}}); \rho_1)$ computed in Step 4 in Fig. 17.

Lemma 18. *$H_8 \approx H_7$.*

Proof. This follows from hiding of Com. We could simply get tid or tid' from an oracle at this point. We do not use the opening of the transfer identifier anywhere, as collection is still simulated.

Hybrid 9 Let H_9 be defined as H_8 except that:

Change 9.1. In Fig. 19 in Step 8 compute

$$\pi_1 \leftarrow \mathsf{NIZK.Prv}_{\mathsf{crs}}(\mathcal{R}_{\textsc{OrDec}}, (\mathsf{ck}, \{\mathsf{coin}_j\}_{j \in \mathsf{pos}}, \mathsf{tid}), \mathsf{dk}_{\mathsf{B}})$$

instead of

$$\pi_1 \leftarrow \mathsf{NIZK.Sim}_{\mathsf{tSim}}(\mathcal{R}_{\textsc{OrDec}}, (\mathsf{ck}, \{\mathsf{coin}_j\}_{j \in \mathsf{pos}}, \mathsf{tid})) \ .$$

This is possible as now the instance is true and we know the witness.

Change 9.2. In Step 9 use replace $a' = 0$ by $a' = a$ where a is the correct value being transferred. We can learn this value by inspecting $\mathcal{F}_{\mathsf{AnonPay}}$ and it is by the changes in Hybrid 7 also the value inside the coin we are picking up.

Change 9.3. In Fig. 19 in Step 10 compute

$$\pi_2 \leftarrow \mathsf{NIZKPoK.Prv}^{\mathcal{O}}_{\mathsf{crs}}(\mathcal{R}_{\text{Collect}}, (\mathsf{ck}, c_{\mathsf{B}}, \mathsf{B}, \mathsf{tid}, c'_{\mathsf{B}}), ((b_{\mathsf{B}}, \rho_{\mathsf{B}}), (\mathsf{A}, a', \mathsf{nonce}_{\mathsf{A}}, \rho_1), \rho'_{\mathsf{B}}))$$

instead of

$$\pi_2 \leftarrow \mathsf{NIZKPoK.Sim}^{\mathcal{O}}_{\mathsf{tSim}}(\mathcal{R}_{\text{Collect}}, (\mathsf{ck}, c_{\mathsf{B}}, \mathsf{B}, \mathsf{tid}, c'_{\mathsf{B}}))\ .$$

Note that by now the tid given as input is the tid' produced in Step 4 in Fig. 17, so we indeed know an opening of tid to $a' = a$, so we have a true instance and we know the witness.

Lemma 19. *$H_9 \approx H_8$.*

Proof. The second change is indistinguishable to the adversary by hiding of Com. The two other changes are indistinguishable to the adversary using zero-knowledge of $\mathsf{NIZKPoK}$. □

Hybrid 10 Finally let H_{10} be like H_9 except that

Change 10.1. In Step 1 in **Initiate Pay (Honest Sender, Honest Receiver)** in Fig. 17 we inspect $\mathcal{F}_{\mathsf{AnonPay}}$ and learn the amount a being transferred. If $a > b_{\mathsf{A}}$ then we stop the simulation.

Change 10.2. In Step 7 in **Initiate Pay (Honest Sender, Honest Receiver)** in Fig. 17 we compute

$$\pi \leftarrow \mathsf{NIZKPoK.Prv}_{\mathsf{crs}}(\mathcal{R}_{\text{Pay}}, (c_{\mathsf{A}}, \mathsf{coin}, c_{\mathsf{A}'}, \mathsf{A}, \mathsf{nonce}_{\mathsf{A}}), ((b_{\mathsf{A}}, \rho_{\mathsf{A}}), (\mathsf{B}, a', \rho_1, \rho_2), \rho'_{\mathsf{A}}))$$

instead of

$$\pi \leftarrow \mathsf{NIZKPoK.Sim}_{\mathsf{tSim}}(\mathcal{R}_{\text{Pay}}, (c_{\mathsf{A}}, \mathsf{coin}, c_{\mathsf{A}'}, \mathsf{A}, \mathsf{nonce}_{\mathsf{A}})\ .$$

Lemma 20. *$H_{10} \approx H_9$.*

Proof. After H_9 all commitments to account values are updated as in the protocol. Therefore, by Lemma 9, the balances committed to for honest parties is the same as the balances held by $\mathcal{F}_{\mathsf{AnonPay}}$. And $\mathcal{F}_{\mathsf{AnonPay}}$ would only have given the simulator input $(\text{Pay}, \mathsf{A}, \mathsf{atid})$ if $a \leq b_{\mathsf{A}}$ in $\mathcal{F}_{\mathsf{AnonPay}}$. Hence the check $a \leq b_{\mathsf{A}}$ only fails with negligible probability in the (simulated) protocol. The second change is indistinguishable to the adversary by zero-knowledge of $\mathsf{NIZKPoK}$, as we now know a correct witness. We clearly know the witnesses needed and they satisfy $\mathcal{R}_{\text{Pay}}$ as $a \leq b_{\mathsf{A}}$, so we can do a reduction to zero-knowledge. □

For clarity we did the changes in Hybrids 9 and 10 in two steps, but note that technically we need to give a single reduction to the ZK of the GUC NIZK PoK proof system, as the ZK notion does not allow to replace some simulated proofs with real proofs and some not. We can do this as follows. First we do the changes in **Change 9.1**, **Change 9.2** and **Change 10.1** and then we do the changes in **Change 9.3** and **Change 10.2** and prove indistinguishability of this step using a single reduction.

Hybrid 10 Finally let $H_{11} = \text{Exec}_{\Pi_{\text{AnonPay}},\mathcal{F}_{\text{Ledger}},\mathcal{F}_{\text{AAT}},\mathcal{F}_{\text{Service}},\mathcal{E}}(1^\lambda)$ be the execution of the protocol.

Lemma 21. $H_{11} \approx H_{10}$.

Proof. By H_{10} the honest parties are being run according to the protocol except for syntactical difference which do not matter. Furthermore, $\mathcal{F}_{\mathsf{AnonPay}}$ and the simulated protocol have been aligned. We argued that structurally $\mathcal{F}_{\mathsf{AnonPay}}$ and the protocol gives the same types of outputs at the same time. Collections are mapped to the correct payments. Furthermore, by Lemma 9 the balances in $\mathcal{F}_{\mathsf{AnonPay}}$ and in the protocol are now the same except with negligible probability. And the protocol now uses the same tid's as the ideal functionality. Therefore the protocol and the ideal functionality gives exactly the same outputs at exactly the same times, except with negligible probability. The only difference is that in H_{10} we compute $\mathsf{crs}^{\text{KNOW}}$ using $\mathsf{NIZKPoK.SimGen}$ and in H_{11} we use $\mathsf{NIZKPoK.Gen}$, but this is indistinguishable by Zero Knowledge of $\mathsf{NIZKPoK}$. □

10 Adding Strong Anonymity

In this section, we sketch how to extend our OCash construction to satisfy strong anonymity. OCash, as described in Section 8, currently leaks tid during payment. This allows the user, who knows tid, to learn when the shop collected its coin. To mitigate this we will instead let the shop post what is essentially $\mathsf{PRF}_K(\mathsf{tid})$ for a key K bound to the shop. When K is fixed then $\mathsf{PRF}_K(\mathsf{tid})$ repeats whenever tid repeats. Furthermore, if the shop is honest then K is random and unknown and therefore $\mathsf{PRF}_K(\mathsf{tid})$ leaks nothing about tid.

10.1 PRF Key Registration

To make this proof concretely efficient, we use a PRF which is a slightly modified version of the Dodis-Yampolskiy VRF [DY05]. We extend the public parameters of the commitmnent scheme with an independent generator g_5. Each shop commits to a key $K = s$ by putting

$$\mathsf{vk} = g_5^K \tag{5}$$

in its account information. Then, during **Create Account** they give a proof using $\mathsf{NIZKPoK}$ for the relation $\mathcal{R}_{\text{KEY}}(\mathsf{vk}, K) \equiv \mathsf{vk} = g_5^K$. Clearly $\Phi(K) = g_5^K$ is a group homomorphism, so we can use the proofs described in Section 11.2 to make this proof efficient. During simulation, the simulator will for honest parties not learn K, but instead use g_5^ρ for uniformly random ρ. This will be indistinguishable by the DDH assumption. It then uses the $\mathsf{NIZKPoK}$ simulator to simulate the proof for vk, which will again be indistinguishable. For the corrupted parties, it will use the $\mathsf{NIZKPoK}$ extractor to learn K.

The key K defines a pseudorandom function

$$\mathsf{PRF}_K : \mathbb{Z}_q \to \mathbb{G}\ , \quad y = \mathsf{PRF}_K(x) = g_5^{1/(K+x)}\ . \tag{6}$$

Using the Diffie-Hellman inversion (DHI) assumption, one can show that the function is a PRF, even if one is given vk. This was proven by Dodis and Yampolskiy [DY05], but their reduction runs in exponential time. The specific reduction in [DY05] runs in time 2^{160} and asymptotically runs in time 2^λ, where λ is the output length of a collision resistant hash function[15]. To withstand an

[15] Observe that $a(k)$ in Theorem 1 in [DY05] needs to be λ and also see also Remark 1 [DY05].

adversary running in time 2^τ we must have $\lambda > 2\tau$ because of the birthday bound. In our setting this is not satisfactory, as it would force us to use a group $\mathbb{G}$ of size $2^{2\tau}$, where in practice one would hope to get by using a group of size about 2^τ. In Section 12 we give a more fine grained reduction allowing us to do a poly-time reduction for our use of DY. The reduction is not novel, but formalizes what seem to be folklore in the way DY is used in the literature.

10.2 Hashed Identifier

We furthermore add a new component $\mathsf{hid} = \mathsf{Hash}(m_\mathsf{U}, n_\mathsf{U})$, called the *hashed identifier*, to the commitment tid. Here m_U is the user number of U. Each registration appears on the total ordered ledger, so we can order the users as $1, 2, \ldots$ according to when their registration appears. We need that Hash maps two polynomially large m and n collision resistantly into $\mathbb{Z}_q$. In fact, since q is exponential the map is easy to make injective. When the coin is posted it has the form

$$c = (G = g^\rho, H = h^\rho, G^r, H^r \cdot g_0^s \cdot g_1^\mathsf{U} \cdot g_2^\mathsf{S} \cdot g_3^{n_\mathsf{U}} \cdot g_4^a \cdot g_5^\mathsf{hid}) \ . \tag{7}$$

The user will prove that hid was computed as $\mathsf{hid} = \mathsf{Hash}(m_\mathsf{U}, n_\mathsf{U})$. This can be done basically by opening the g_5 component and checking. We discuss how to extend $\mathcal{R}_{\text{PAY}}$ in Section 11.5. This is then maintained during re-randomizations and therefore does not have to be reproven when the coin is collected. The current proof already proves that when a coin is collected, then the committed values are the same as in some coin constructed during a payment, cf. Lemma 8. Therefore, when the coin is collected a rerandomized version is posted of the following form:

$$c' = (G = g^{\rho'}, H = h^{\rho'}, G^{r'}, H^{r'} \cdot g_0^s \cdot g_1^\mathsf{U} \cdot g_2^\mathsf{S} \cdot g_3^{n_\mathsf{U}} \cdot g_4^a \cdot g_5^\mathsf{hid}) \ , \tag{8}$$

where $\mathsf{hid} = \mathsf{Hash}(m_\mathsf{U}, n_\mathsf{U})$.

10.3 Rerandomized TID

The next modification we make is that the shop will put

$$\mathsf{tid}' = \mathsf{tid} \cdot g_0^\rho$$

and

$$h = g_1^\rho$$

on the ledger instead of tid, as tid is known by the user. Here ρ is uniform in $\mathbb{Z}_q$. Therefore, by DDH, (tid', h) is indistinguishable from (tid', h') for uniformly random $h' \in \mathbb{G}$. And in this distribution tid' is uniform in $\mathbb{G}$ and independent of tid and h', which can essily be seen to give strong anonymity. Note that after this

$$\mathsf{tid}' = g_0^{s'} \cdot g_1^\mathsf{U} \cdot g_2^\mathsf{S} \cdot g_3^{n_\mathsf{U}} \cdot g_4^a \cdot g_5^\mathsf{hid} \ .$$

for $s' = s + \rho \bmod q$.

10.4 Pseudononymous Hashed Identifier

What remains is to prove that the coin was not collected before. Note that the value $(m_{\mathsf{U}}, n_{\mathsf{U}})$ is unique as n_{U} is incremented for each payment by U. It is therefore sufficient to check that $(m_{\mathsf{U}}, n_{\mathsf{U}})$ was not used before by the shop. By collision resistance of Hash and the fact that hid was computed correctly, this is the same as checking that hid was not used before. Let vk be the public key of the shop and let K be its key such that $\mathsf{vk} = g_5^K$. Since $\mathsf{PRF}_K : \mathbb{Z}_q \to \mathbb{G}$ is an injective function and K is bound to the shop is it therefore enough to check that the pseudononymous hashed identifier $\mathsf{otid} = \mathsf{PRF}_K(\mathsf{hid})$ was not used before. We check this simply by letting the shop post

$$\mathsf{otid} = g_5^{1/(K+\mathsf{hid})}$$

and prove that this values was constructed correctly. The collection is ignored if this proof fails or otid was used before.

10.5 Improved Reduction to DHI

Let us return to the security loss in the reduction to DHI. In Section 12 we provide an improved analysis for the PRF, using a reduction to DHI with running time $|X| \operatorname{poly}(\lambda)$, where X is a set which can be computed before the PRF key K is sampled and where it is guaranteed that all queries to PRF_K will be from X. The reduction essentially does $|X|$ group operations in $\mathbb{G}$.

Note that in our construction we only need PRF_K to be a PRF when the collector is honest. Furthermore, in this case we apply it to $\mathsf{Hash}(m, n)$ where $m \in [U]$ and $n \in [P]$, where U is a upper bound on the maximal number of users in the lifetime of the system and P an upper bound on the number of payments per user. So we set $X = \mathsf{Hash}([U], [P])$. These bounds are both polynomial and we do not need to estimate them for the construction, only the reduction. In the reduction we can set $U = P$ to be the running time of the environment. A poly-time environment can start at most poly-many collections, so the reduction is asymptotically poly-time.

Our construction seems to be the first to use the DY PRF for strong anonymity while applying it only to a polynomial domain. For instance [CHK23] applies the PRF to the coins c, which are harder to control as they can be constructed by the adversary. Constructions, like [CHK23] apply it to an exponential domain and therefore *a priori* suffer the exponential security loss. We note, however, that the complexity of the reduction seems to be controllable using a programmable random oracle. In [CHK23], the PRF is applied to $\mathsf{Hash}(c)$, where Hash could be modelled as a programable random oracle. Before the reduction is run one could then sample a large set X of uniformly random values and then when the random oracle is queried by the adversary return a fresh value from X. Now it is known that the PRF will only be applied to elements from X. However, at the time of writing the SHA256 hash function is being computed about 2^{75} times per second by the Bitcoin network. Over a ten year period this is more than 2^{100} hashes. This approach therefore still gives a substantial security loss, though still asymptotically polynomial.

10.6 Extending the Relations and ZK Proofs

We show how to extend our proof systems to the new coin format in Section 11.5.

11 ZK Proof System Instantiations

In this section, we show how to instantiate the proof systems that are needed for our OCash construction with weak and strong anonymity.

11.1 Discussion of GUC NIZK PoK Definitions

In Section 2.5 we gave a slight reformulation of the notion of GUC NIZK Proof of Knowledge in [LR22]. We here sketch why it is equivalent to Definition 11 in [LR22].

In [LR22] the definition of GUC NIZK PoK is given indirectly by letting the NIZK PoK Ideal Functionality from their Definition 8 generate simulated proofs for honest parties and generate a failure event observable by the environment if it cannot at the same time extract all accepting proofs (which are not among the simulated ones). In their Definition 11 it is then required that this ideal functionality is indistinguishable from the real protocol, where the honest proofs are generated correctly and there are no attempts at extracting accepting proofs and therefore no observable failure events. This implies that the failure event must be negligible. Namely, it never occurs in the real world, so if it occurred in the simulation with non-negligible probability simulation would be impossible. This gives what is normally called *simulation extractability.* At the same time, in the real world real proofs are given and in the simulation the proofs are simulated (see **Prove** in their Definition 8). This means that it follows from their Definition 8 in conjunction with their Definition 11 that real proofs must be indistinguishable from simulated proofs. This gives what is normally called *zero-knowledge.* Finally, **Prove** in their Definition 8 creates an observable failure if a simulated proof does not verify. Since no such failure event is possible in the real world it follows from their Definition 8 in conjunction with their Definition 11 that simulated proofs verify except with negligible probability. Since simulated proofs are indistinguishable from real proofs, it follows that real proofs verify except with negligible probability, which is normally called *completeness.*

In Section 2.5 we explicitly state these three properties. They are equivalent to Definition 11 in [LR22] qua the above discussion.

11.2 Proofs for Group Homomorphisms

We will need several Σ-protocols with strong special soundness for relations defined via collision resistant group homomorphisms, and we describe the general theory here. We first present the canonical Σ-protocol for group homomorphisms. Let $(G, +)$ and $(H, \cdot)$ be Abelian groups, we write G additively and H multiplicatively. Assume G has a known prime order q. For $g, h \in G$ and $e \in \mathbb{Z}$ let eg and h^e denote the usual group actions of adding/multiplying the element with itself e times.

The relation is given by $\mathcal{R} \subset H \times G$ with $x = \Phi(w)$. The first message is computed as $a = \Phi(r)$ for uniformly random $r \in G$. The challenge is space is $E = \mathbb{Z}_q$. The reply is computed as $z = ew + r$, and the verification is $\Phi(z) = x^e a$. Completeness is clear: $\Phi(ew + r) = \Phi(w)^e \Phi(r) = x^e a$. To prove SHVZK we pick $z \in G$ uniformly at random and let $a = x^{-e}\Phi(z)$. This has a distribution identical to the basic run, as z is uniform in both distributions and $a = x^{-e}\Phi(z)$ in both distributions.

For special soundness note that if $\Phi(z) = x^e a$ and $\Phi(z') = x^{e'} a$, then $\Phi(z - z') = \Phi(z)/\Phi(z') = x^{e-e'}$. Therefore, if we let $d = (e - e')^{-1} \bmod q$ and $w = d(z - z')$, then $\Phi(w) = x^{d(e-e')} = x$.

Now let a system parameter trapdoor be any element $z \in G \setminus \{0\}$ such that $\Phi(z) = 1 \in H$. Formally, $T(t) \equiv t \in \Phi^{-1}(1) \setminus \{0\}$. We can then argue strong special soundness. Assume that $(e', z') \neq (e, z)$ and $\Phi(z) = x^e a$ and $\Phi(z') = x^{e'} a$. If $e' \neq e$ then we get w such that $(x, w) \in R$ from special soundness. If $e = e'$, then $z' \neq z$ and therefore $\Phi(z' - z) = (x^{e'} a)(x^e a)^{-1} = (x^e a)(x^e a)^{-1} = 1$ is a system parameter trapdoor.

To illustrate the definition, we consider Pedersen vector commitments. The system parameters are uniform and independent generators $g_1, \ldots, g_\ell$ of a group H of order q. We let $G = (\mathbb{Z}_q)^\ell$ and $\Phi(w_1, \ldots, w_\ell) = \prod_{i=1}^{\ell} g_i^{w_i}$. Clearly G has order q and Φ is a group homomorphism $G \to H$. A system parameter trapdoor is a non-zero element $(w_1, \ldots, w_\ell) \in \mathbb{Z}_q^\ell$ such that $\prod_i g_i^{w_i} = 1$. It is well-known and straight-forward to verify that finding such a non-trivial representation of 1 is equivalent to the discrete logarithm problem in H under poly-time reduction.

Consider then a Pedersen vector commitment $c = \prod_{i=1}^{\ell} g_i^{w_i}$, where for instance w_ℓ is the randomizer. Proving knowledge of an opening of c is the same as proving knowledge of $w \in G$ such that $\Phi(w) = c$. By the above discussion, we have a strong special sound Σ-protocol for this with a system parameter trapdoor which is computationally hard to find under the DL protocol in H.

In Section 8.4 we formalise the relations for which we need ZK proof in our construction. In Section 11.4 we give instantiations of strong simulation sound Σ-protocols for these relations.

11.3 $\bigvee$-Construction Maintains Strong Special Soundness

We now recall that the OR construction of Σ-protocols is strong special sound. This is proven in [KS22, LR22] and we recall it here for completeness.

Consider two Σ protocols

$$\Sigma_0 = (\mathcal{R}_0, A_0, \mathcal{E}, Z_0, V_0, W_0, S_0, T_0)$$

and

$$\Sigma_1 = (\mathcal{R}_1, A_1, \mathcal{E}, Z_1, V_1, W_1, S_1, T_1)$$

with the same challenge space, which is also an Abelian group. Here W_i is the witness extractor, S_i the simulator, and T_i the system trapdoor predicate. We can define the disjunction between these to be $(\mathcal{R}_\vee, A_\vee, \mathcal{E}, Z_\vee, W_\vee, V_\vee, S_\vee, T_\vee)$, where the relation $((x_0, x_1), w) \in \mathcal{R}_\vee$ is given by $(x_0, w) \in \mathcal{R}_0$ or $(x_1, w) \in \mathcal{R}_1$ and $T_\vee(t) \equiv T_0(t) \vee T_1(t)$.

1. The input to the prover is $((x_0, x_1), w) \in \mathcal{R}_\vee$, i.e., $(x_0, w) \in \mathcal{R}_0$ or $(x_1, w) \in \mathcal{R}_1$. The input to the verifier is (x_0, x_1). Below we proceed for the case $(x_0, w) \in \mathcal{R}_0$. The other case is symmetric.
2. Let $a_0 \leftarrow A_0(x_0, w; r)$, sample $e_1 \in \mathcal{E}$ uniformly at random, and let $(a_1, z_1) \leftarrow S_1(x_1, e_1)$.
3. The verifier samples and sends a uniformly random *challenge* $e \in \mathcal{E}$.
4. The prover computes $e_0 = e - e_1$, computes $z_0 \leftarrow Z_0(x_0, w, e_0, r)$ and sends $z = ((e_0, e_1), (z_0, z_1))$
5. The verifier checks that $e_0 + e_1 = e$ and $V_1(x_0, a_0, e_0, z_0) = V_2(x_1, a_1, e_1, z_1) = \top$.

Completeness is straight-forward and SHVZK follows from just simulating both tracks, so we focus on strong special soundness. Consider two transcripts $((a_0, a_1), e, z) = ((e_0, e_1), (z_0, z_1))$, where $e_0 + e_1 = e$ and $V_1(x_0, a_0, e_0, z_0) = V_2(x_1, a_1, e_1, z_1) = \top$, and $((a_0, a_1), e', z') = ((e'_0, e'_1), (z'_0, z'_1))$, where $e'_0 + e'_1 = e'$ and $V_1(x_0, a_0, e'_0, z'_0) = V_2(x_1, a_1, e'_1, z'_1) = \top$.

We can assume that $(e', z') \neq (e, z)$ and has to compute a witness or a system parameter trapdoor. We prove in two cases $e' \neq e$ and $e' = e$. If $e' \neq e$ then $e'_0 \neq e_0$ or $e'_1 \neq e_1$. This gives us that $(e'_0, z'_0) \neq (e_0, z_0)$ or $(e'_1, z'_1) \neq (e_1, z_1)$. This either allows to use W_0 to compute w_0 such that $(x_0, w_0) \in \mathcal{R}_0$ or $T_0(w_0) = \top$ or allows to use W_1 to compute w_1 such that $(x_1, w_1) \in \mathcal{R}_1$ or $T_1(w_1) = \top$. This gives use a witness $w \in \{w_0, w_1\}$ for $\mathcal{R}_\vee$ or t such that $T(t) = \top$. If $e = e'$ then it follows from $(e', z') \neq (e, z)$ that $z' \neq z$, i.e., $((e'_0, e'_1), (z'_0, z'_1)) \neq ((e_0, e_1), (z_0, z_1))$. If $e'_0 \neq e_0$ then we are done by the above reasoning. If $e'_0 = e_0$ then $e' = e$ implies that $e'_1 = e_1$. We must therefore have that $(z'_0, z'_1) \neq (z_0, z_1)$. So for some b we have that $z'_b \neq z_b$, and then we are done by strong special soundness of Σ_b as above.

11.4 Σ-Protocols for Relations

We now give the Σ-protocols for the relations used in OCash. Note that for the relations where we give proof of knowledge (using $\mathsf{NIZKPoK}$) we need the protocols to have strong special soundness (cf. Section 2.5). This is all relation but $\mathcal{R}_{\text{OrDec}}$.

We first consider the relation $(x = (\mathsf{ck}, c), w = \rho) \in \mathcal{R}_{\text{IsZero}} \iff c = \mathsf{Com.Commit}_{\mathsf{ck}}(0, \rho)$. Let $\Psi(\rho) = \mathsf{Commit}_{\mathsf{ck}}(0; \rho)$. This is clearly a group homomorphism $\Psi(\rho_1 + \rho_2) = \Psi(\rho_1)\Psi(\rho_2)$, so we can use the Σ-protocol from Section 11.2, which has strong special soundness.

Consider then the relation $(x = (\mathsf{ck}, c), w = \rho) \in \mathcal{R}_{\text{IsFund}} \iff c = \mathsf{Com.Commit}_{\mathsf{ck}}(a_0, \rho)$. We have that $c = \mathsf{Commit}_{\mathsf{ck}}(a_0, \rho) \iff \mathsf{Com.Commit}_{\mathsf{ck}}(0, \rho) = c \cdot g_4^{-a_0}$, so we can use the Σ-protocol for $\mathcal{R}_{\text{IsZero}}$.

We then consider $(x = (\mathsf{pp}, \mathsf{ek}, \{c_j\}_{j \in 1}^{\ell}, m), w = \mathsf{dk}) \in \mathcal{R}_{\text{OrDec}} \iff \bigvee_{j=1}^{\ell} \mathsf{RPKE.Dec}_{\mathsf{dk}}(c_j) = m$. Note that here we do not need a proof of knowledge, just a proof of membership. We have that $\mathsf{ek} = g_0^{\mathsf{sk}}$ and for $c = (A, B, C, D)$ we have that $\mathsf{RPKE.Dec}_{\mathsf{dk}}(c) = \bot$ if $B \neq A^{\mathsf{sk}}$, and otherwise $\mathsf{RPKE.Dec}_{\mathsf{dk}}(c) = DC^{-\mathsf{sk}}$. Therefore $\mathsf{RPKE.Dec}_{\mathsf{dk}}(c) = m$ is equivalent to the existence of x such that $\mathsf{ek} = g_0^x$ and $B = A^x$ and $Dm^{-1} = C^x$. Consider the group $\overline{\mathbb{G}} = \mathbb{G}^3$ and for fixed $\mathfrak{g} = (g_0, A, C) \in \overline{\mathbb{G}}$ consider the group homomorphism $\Psi : \mathbb{Z}_q \to \overline{\mathbb{G}}$ given by $\Psi(w) = \mathfrak{g}^w = (g_0^w, A^w, C^w)$. Let $\mathfrak{h} = (\mathsf{ek}, B, Dm^{-1})$. Then $\mathsf{RPKE.Dec}_{\mathsf{dk}}(c) = m$ is equivalent to $\exists w \, \Psi(w) = \mathfrak{h}$. So we can use the Σ-protocol from Section 11.2. We could then use the $\bigvee$-construction from Section 11.3 to get a Σ-protocol for $\mathcal{R}_{\text{OrDec}}$, but the communication would be linear in ℓ. We can do better than this using [GK15] as discussed now. For $j = 1, \ldots, \ell$ let $c_j = (A_j, B_j, C_j, D_j)$, $\mathfrak{g}_j = (g_0, A_j, C_j) \in \overline{\mathbb{G}}$, and $\mathfrak{h}_j = (\mathsf{ek}, B_j, D_j m^{-1}) \in \overline{\mathbb{G}}$. Then

$$(x = (\mathsf{pp}, \mathsf{ek}, \{c_j\}_{j \in 1}^{\ell}, m), w = \mathsf{dk}) \in \mathcal{R}_{\text{OrDec}} \iff \bigvee_{j=1}^{\ell} \mathfrak{h}_j = \mathfrak{g}_j^w \; .$$

We can therefore directly use the one-out-of-many DL Σ-protocol in [GK15]. The communication complexity is in the order of $\log \ell$ times that of a single proof for Ψ.

Consider then

$$\begin{aligned} &(x = (\mathsf{ck}, c_{\mathsf{B}}, \mathsf{B}, \mathsf{tid}, c'_{\mathsf{B}}), w = ((b_{\mathsf{B}}, \rho_{\mathsf{B}}), (\mathsf{A}, a, \mathsf{nonce}_{\mathsf{A}}, \rho_1), \rho'_{\mathsf{B}})) \in \mathcal{R}_{\text{Collect}} \\ \iff &c_{\mathsf{B}} = \mathsf{Com.Commit}_{\mathsf{ck}}(b_{\mathsf{B}}, \rho_{\mathsf{B}}) \land \\ &\mathsf{tid} = \mathsf{Com.Commit}_{\mathsf{ck}}(\mathsf{A}, \mathsf{B}, a, \mathsf{nonce}_{\mathsf{A}}, \rho_1) \land \\ &c'_{\mathsf{B}} = \mathsf{Com.Commit}_{\mathsf{ck}}(b_{\mathsf{B}} + a, \rho'_{\mathsf{B}}). \end{aligned}$$

This is equivalent to

$$c_{\mathsf{B}} = g_0^{\rho_{\mathsf{B}}} g_4^{b_{\mathsf{B}}} \land \mathsf{tid} g_4^{-\mathsf{nonce}_{\mathsf{A}}} g_2^{-\mathsf{B}} = g_0^{\rho_1} g_1^{\mathsf{A}} g_2^{a} \land c'_{\mathsf{B}} = g_0^{\rho'_{\mathsf{B}}} g_4^{b_{\mathsf{B}}+a} \; .$$

Let $\mathfrak{h} = (c_{\mathsf{B}}, \mathsf{tid} g_4^{-\mathsf{nonce}_{\mathsf{A}}} g_2^{-\mathsf{B}}, c'_{\mathsf{B}})$ and consider the group homomorphism $\Psi : \mathbb{Z}_q^6 \to \mathbb{G}^3$ given by

$$\Psi(\rho_{\mathsf{B}}, b_{\mathsf{B}}, \rho_1, \mathsf{A}, a, \rho'_{\mathsf{B}}) = (g_0^{\rho_{\mathsf{B}}} g_4^{b_{\mathsf{B}}}, \, g_0^{\rho_1} g_1^{\mathsf{A}} g_2^{a}, \, g_0^{\rho'_{\mathsf{B}}} g_4^{b_{\mathsf{B}}+a}) \; .$$

Let $w = (\rho_{\mathsf{B}}, b_{\mathsf{B}}, \rho_1, \mathsf{A}, a, \rho'_{\mathsf{B}})$. Then $\mathcal{R}_{\text{Collect}}$ is equivalent to $\Phi(w) = \mathfrak{h}$, so we can use the Σ-protocol from Section 11.2, which has strong special soundness.

We finally consider

$$\begin{aligned}
&(x = (\mathsf{A}, \mathsf{nonce}_\mathsf{A}, \mathsf{ck}, c_\mathsf{A}, \mathsf{coin}, c'_\mathsf{A}), w = ((b_\mathsf{A}, \rho_\mathsf{A}), (\mathsf{B}, a, \rho_1, \rho_2), \rho'_\mathsf{A})) \in \mathcal{R}_{\text{PAY}} \\
\iff & c_\mathsf{A} = \mathsf{Com.Commit}_\mathsf{ck}(b_\mathsf{A}, \rho_\mathsf{A}) \wedge \\
&((\mathsf{coin}, \mathsf{Com.Commit}_\mathsf{ck}(\mathsf{A}, \mathsf{B}, a, \mathsf{nonce}_\mathsf{A}; \rho_1)), \rho_2) \in \mathcal{R}_{\text{ENC}} \wedge \\
&c'_\mathsf{A} = \mathsf{Com.Commit}_\mathsf{ck}(b_\mathsf{A} - a, \rho'_\mathsf{A}) \wedge b_\mathsf{A} \geq a \geq 0 \ .
\end{aligned}$$

Let $\mathsf{coin} = (A, B, C, D)$ and $\mathsf{tid} = \mathsf{Com.Commit}_\mathsf{ck}(\mathsf{A}, \mathsf{B}, a, \mathsf{nonce}_\mathsf{A}; \rho_1)$. As part of the proof we want to show that $((\mathsf{coin}, \mathsf{tid}), \rho_2) \in \mathcal{R}_{\text{ENC}}$. We use that the sender knows r such that $C = A^r$ and $D = B^r \mathsf{rid}$, (cf. Fig. 4) and let $\rho_2 = r$. Then $((\mathsf{coin}, \mathsf{tid}), r) \in \mathcal{R}_{\text{ENC}}$ is equivalent to $C = A^r \wedge D = B^r g_0^{\rho_1} g_1^\mathsf{A} g_2^\mathsf{B} g_3^a g_4^{\mathsf{nonce}_\mathsf{A}}$. What we want to prove is therefore knowledge of $((b_\mathsf{A}, \rho_\mathsf{A}), (\mathsf{B}, a, \rho_1, r), \rho'_\mathsf{A})$ such that

$$c_\mathsf{A} = g_0^{\rho_\mathsf{A}} g_4^{b_\mathsf{A}} \wedge C = A^r \wedge D = B^r g_0^{\rho_1} g_1^\mathsf{A} g_2^\mathsf{B} g_3^a g_4^{\mathsf{nonce}_\mathsf{A}} \wedge c'_\mathsf{A} = g_0^{\rho'_\mathsf{A}} g_4^{b_\mathsf{A} - a} \wedge b_\mathsf{A} \geq a \geq 0 \ . \tag{9}$$

Let $\mathfrak{h} = (c_\mathsf{A}, C, D g_1^{-\mathsf{A}} g_4^{-\mathsf{nonce}_\mathsf{A}}, c'_\mathsf{A})$ and define the group homomorphism

$$\Psi(\rho_\mathsf{A}, b_\mathsf{A}, r, \rho_1, \mathsf{B}, a, \rho'_\mathsf{A}) = (g_0^{\rho_\mathsf{A}} g_4^{b_\mathsf{A}},\ A^r,\ B^r g_0^{\rho_1} g_2^\mathsf{B} g_3^a,\ g_0^{\rho'_\mathsf{A}} g_4^{b_\mathsf{A} - a}) \ ,$$

then ignoring the condition $b_\mathsf{A} \geq a \geq 0$ what we have to prove is knowledge of $(\rho_\mathsf{A}, b_\mathsf{A}, r, \rho_1, \mathsf{B}, a, \rho'_\mathsf{A})$ such that $\Psi(\rho_\mathsf{A}, b_\mathsf{A}, r, \rho_1, \mathsf{B}, a, \rho'_\mathsf{A}) = \mathfrak{h}$. We do this using the Σ-protocol from Section 11.2, which has strong special soundness.

We then focus on $b_\mathsf{A} \geq a \geq 0$. This is the same as showing knowledge of

$$w' = (b'_\mathsf{A}, a, \rho'_\mathsf{A}, \rho''_\mathsf{A})$$

such that

$$c'_\mathsf{A} = g_0^{\rho'_\mathsf{A}} g_4^{b'_\mathsf{A}} \wedge c_\mathsf{A}/c'_\mathsf{A} = g_0^{\rho''_\mathsf{A}} g_4^a \wedge b'_\mathsf{A} \geq 0 \wedge a \geq 0 \ . \tag{10}$$

It is sufficient to give a *separate* proof for this fact. I.e., we give one proof of knowledge of $w = (\rho_\mathsf{A}, b_\mathsf{A}, r, \rho_1, \mathsf{B}, a, \rho'_\mathsf{A})$ such that $\Psi(\rho_\mathsf{A}, b_\mathsf{A}, r, \rho_1, \mathsf{B}, a, \rho'_\mathsf{A}) = \mathfrak{h}$ and at the same time we give a proof of knowledge for w' such satisfying Eq. (10) without, e.g., trying to prove that $b'_\mathsf{A} = b_\mathsf{A}$. However, collision resistance will ensure that the shared values between w and w' will be identical. Assume namely that we have a witness w' for Eq. (10) and a witness $(\rho_\mathsf{A}, b_\mathsf{A}, r, \rho_1, \mathsf{B}, a, \rho'_\mathsf{A})$ such that $\Psi(\rho_\mathsf{A}, b_\mathsf{A}, r, \rho_1, \mathsf{B}, a, \rho'_\mathsf{A}) = \mathfrak{h}$. Under the DL assumption this implies that $b_\mathsf{A} \geq a \geq 0$. To see this, note that from $(\rho_\mathsf{A}, b_\mathsf{A}, r, \rho_1, \mathsf{B}, a, \rho'_\mathsf{A})$ such that $\Psi(\rho_\mathsf{A}, b_\mathsf{A}, r, \rho_1, \mathsf{B}, a, \rho'_\mathsf{A}) = \mathfrak{h}$ we can compute a potential witness

$$\overline{w} = (\overline{b}'_\mathsf{A}, \overline{a}, \overline{\rho}'_\mathsf{A}, \overline{\rho}''_\mathsf{A}) = (b_\mathsf{A} - a, a, \rho'_\mathsf{A}, \rho_\mathsf{A}/\rho'_\mathsf{A}) \ .$$

It follows from Eq. (9) that $c'_\mathsf{A} = g_0^{\rho'_\mathsf{A}} g_4^{b'_\mathsf{A}}$ and $c'_\mathsf{A} = g_0^{\rho'_\mathsf{A}} g_4^{b_\mathsf{A} - a}$ which implies that

$$c'_\mathsf{A} = g_0^{\overline{\rho}'_\mathsf{A}} g_4^{\overline{b}'_\mathsf{A}} \wedge c_\mathsf{A}/c'_\mathsf{A} = g_0^{\overline{\rho}''_\mathsf{A}} g_4^{\overline{a}} \ .$$

From the DL assumption it then follows that $\overline{w} = w'$. Therefore it follows from w' being a witness for Eq. (9) that

$$\overline{b}'_\mathsf{A} \geq 0 \wedge \overline{a} \geq 0 \ .$$

By construction $\overline{b}'_{\mathsf{A}} = b_{\mathsf{A}} - a$ and $\overline{a} = a$ which gives us that

$$b_{\mathsf{A}} - a \geq 0 \wedge a \geq 0 \ ,$$

which gives the desired conclusion that $b_{\mathsf{A}} \geq a \geq 0$.

Then note that since a witness w' for Eq. (10) can be computed from Eq. (9) we do not need that the proof for Eq. (10) can be extracted using tExt. If we can use tExt to extract a witness $(\rho_{\mathsf{A}}, b_{\mathsf{A}}, r, \rho_1, \mathsf{B}, a, \rho'_{\mathsf{A}})$ such that $\Psi(\rho_{\mathsf{A}}, b_{\mathsf{A}}, r, \rho_1, \mathsf{B}, a, \rho'_{\mathsf{A}}) = \mathfrak{h}$. From this it can compute $\overline{w}$ as above with $c'_{\mathsf{A}} = g_0^{\overline{\rho}'_{\mathsf{A}}} g_4^{\overline{b}'_{\mathsf{A}}}$ and $c_{\mathsf{A}}/c'_{\mathsf{A}} = g_0^{\overline{\rho}''_{\mathsf{A}}} g_4^{\overline{a}}$. If in addition $\overline{b}'_{\mathsf{A}} \geq 0$ and $\overline{a} \geq 0$ then we have a witness for $\mathcal{R}_{\text{PAY}}$ and is done. If it is not the case that $\overline{b}'_{\mathsf{A}} \geq 0$ and $\overline{a} \geq 0$ then we can break DL as follows. We can rewind the entire UC experiment to compute another witness w' for *Eq.* (10).[16] Since $b'_{\mathsf{A}} \geq 0$ and $a \geq 0$ except with negligible probability by soundness of the proof it follows that $w' \neq \overline{w}$. But then we broke DL, a contradiction. We therefore except with negligible probability also get a witness for $\mathcal{R}_{\text{PAY}}$ when we extract the witness for Eq. (9).

To recap, we give the proof for $\mathcal{R}_{\text{PAY}}$ by giving a proof for

$$\begin{aligned}
&(x = (\mathsf{A}, \mathsf{nonce}_{\mathsf{A}}, \mathsf{ck}, c_{\mathsf{A}}, \mathsf{coin}, c'_{\mathsf{A}}), w = ((b_{\mathsf{A}}, \rho_{\mathsf{A}}), (\mathsf{B}, a, \rho_1, \rho_2), \rho'_{\mathsf{A}})) \in \mathcal{R}^1_{\text{PAY}} \\
\iff &c_{\mathsf{A}} = \mathsf{Com.Commit}_{\mathsf{ck}}(b_{\mathsf{A}}, \rho_{\mathsf{A}}) \wedge \\
&((\mathsf{coin}, \mathsf{Com.Commit}_{\mathsf{ck}}(\mathsf{A}, \mathsf{B}, a, \mathsf{nonce}_{\mathsf{A}}; \rho_1)), \rho_2) \in \mathcal{R}_{\text{ENC}} \wedge \\
&c'_{\mathsf{A}} = \mathsf{Com.Commit}_{\mathsf{ck}}(b_{\mathsf{A}} - a, \rho'_{\mathsf{A}})
\end{aligned}$$

using $\mathsf{NIZKPoK}$ and giving a separate proof for

$$\begin{aligned}
&(x^2 = (\mathsf{ck}, c_{\mathsf{A}}, c'_{\mathsf{A}}), w' = (b'_{\mathsf{A}}, \rho'_{\mathsf{A}}, a, \rho''_{\mathsf{A}})) \in \mathcal{R}^2_{\text{PAY}} \\
\iff &c'_{\mathsf{A}} = \mathsf{Com.Commit}_{\mathsf{ck}}(b'_{\mathsf{A}}, \rho'_{\mathsf{A}}) \wedge \\
&c_{\mathsf{A}}/c'_{\mathsf{A}} = \mathsf{Com.Commit}_{\mathsf{ck}}(a, \rho''_{\mathsf{A}}) \wedge b'_{\mathsf{A}} \geq 0 \wedge a \geq 0 \ .
\end{aligned}$$

using NIZK, letting the honest prover use $\rho''_{\mathsf{A}} = \rho_{\mathsf{A}} - \rho'_{\mathsf{A}}$. Furthermore, a witness for $\mathcal{R}^1_{\text{PAY}}$ will also be a witness for $\mathcal{R}_{\text{PAY}}$ by global extractability of the proof for $\mathcal{R}^2_{\text{PAY}}$ and the DL assumption.

We then turn to how we make a proof for $\mathcal{R}^2_{\text{PAY}}$ for NIZK with global extraction. We will use that $a_0 \geq b'_{\mathsf{A}} \geq 0$, where a_0 is the initial amount. This is maintained by invariant in the protocol. For the same reason on $a_0 \geq a \geq 0$ makes sense. We can therefore pick the order q of $\mathbb{G}$ such that $q > 2^\lambda a_0$. It is then sufficient to give a proof that $b'_{\mathsf{A}}, a_0 \in [0, 2^\lambda a_0 - 1]$ where the honest prover always has $b'_{\mathsf{A}}, a_0 \in [0, a_0]$. We can therefore in principle use any off-the-shelf range proof. However, using the range proof from [AC20] we can get a Σ-protocol secure under the DL assumption. We can then apply Section 2.4 to get a UC NIZK PoM NIZK for this Σ-protocol.

11.5 Extended Proof Systems for Strong Anonymity

We now discuss how to extend the relations $\mathcal{R}_{\text{PAY}}$, $\mathcal{R}_{\text{ORDEC}}$, $\mathcal{R}_{\text{COLLECT}}$, and their proofs to the case with strong anonymity.

We first look at payment. In the proof for $\mathcal{R}^1_{\text{PAY}}$ we simply use the target value

$$\mathfrak{h}' = (c_{\mathsf{A}}, C, D g_1^{-\mathsf{A}} g_4^{-\mathsf{nonce}_{\mathsf{A}}} g_5^{-\mathsf{Hash}(m_{\mathsf{A}}, \mathsf{nonce}_{\mathsf{A}})}, c'_{\mathsf{A}})$$

[16] Of course the UC simulator cannot rewind the UC execution, but we as the provers of security is allowed to do this as a *Gedankenspiel*.

instead of

$$\mathfrak{h} = (c_{\mathsf{A}}, C, D g_1^{-\mathsf{A}} g_4^{-\mathsf{nonce}_{\mathsf{A}}}, c'_{\mathsf{A}}) \ .$$

This ensures that the correct hid was used. We then extrend $\mathcal{R}_{\text{OrDec}}$. Note that the ρ used in $\mathsf{tid}' = \mathsf{tid} \cdot g_0^\rho$ used is fixed by g_1 and $h = g_1^\rho$. Therefore tid is fixed by g_1 and h and tid'. What remains is to prove that this well-defined tid is in one of the encryptions. We can prove this with a proof for the relation

$$(x = (\mathsf{pp}, \mathsf{ek}, \{c_j\}_{j\in 1}^{\ell}, m), w = (\mathsf{dk}, \rho)) \in \mathcal{R}_{\text{OrDec}} \iff h = g_1^\rho \wedge \bigvee_{j=1}^{\ell} (\mathsf{RPKE.Dec}_{\mathsf{dk}}(c_j))\, g_0^\rho = m \ ,$$

by using $m = \mathsf{tid}'$ and ρ being the witness that $m = \mathsf{tid} g_0^\rho$. Note that here we do not need a proof of knowledge, just a proof of membership.

We have that $\mathsf{ek} = g_0^{\mathsf{sk}}$ so for a ciphertext $c = (A, B, C, D)$ we have that $\mathsf{RPKE.Dec}_{\mathsf{dk}}(c) = \bot$ if $B \neq A^{\mathsf{sk}}$, and otherwise $\mathsf{RPKE.Dec}_{\mathsf{dk}}(c) = DC^{-\mathsf{sk}}$. Therefore $\mathsf{RPKE.Dec}_{\mathsf{dk}}(c) g_1^\rho = m$ is equivalent to the existence of ρ and x such that $h = g_1^\rho$ and $\mathsf{ek} = g_0^x$ and $B = A^x$ and $DC^{-x} = m g_0^{-\rho}$, where the last equaiton can be rewritten as $Dm^{-1} = g_0^{-\rho} C^x$. Consider the group $\overline{\mathbb{G}} = \mathbb{G}^4$ and the group homomorphism $\Psi : \mathbb{Z}_q \times \mathbb{Z}_q \to \overline{\mathbb{G}}$ given by $\Psi(w, \rho) = (g_1^\rho, g_0^w, A^w, g_0^{-\rho} C^w)$. Let $\mathfrak{h} = (h, \mathsf{ek}, B, Dm^{-1})$. Then $\mathsf{RPKE.Dec}_{\mathsf{dk}}(c) C^\rho = m$ is equivalent to $\exists w\, \Psi(w) = \mathfrak{h}$. So we can use the Σ-protocol from Section 11.2 and the one-out-of-many DL Σ-protocol in [GK15] to get communication complexity is in the order of $\log \ell$ times that of a single proof for Ψ.

We finally look at collection. During payment the shop posts

$$y = \mathsf{PRF}_K(\mathsf{hid}) = g_5^{1/(K+\mathsf{hid})} \ , \tag{11}$$

where $1/(K+\mathsf{hid})$ denotes multiplicative inverse modulo the order of $\mathbb{G}$. It then shows that it knows $(s', \mathsf{U}, \mathsf{S}, n_{\mathsf{U}}, a, \mathsf{hid}, K)$ such that

$$\mathsf{tid}' = g_0^{s'} \cdot g_1^{\mathsf{U}} \cdot g_2^{\mathsf{S}} \cdot g_3^{n_{\mathsf{U}}} \cdot g_4^{a} \cdot g_5^{\mathsf{hid}} \wedge \mathsf{vk} = g_5^K \wedge y = g_5^{1/(K+\mathsf{hid})} \ . \tag{12}$$

We do not know a Σ-protocol with strong special soundness for this relation, so we will use the same trick as for $\mathcal{R}_{\text{PAY}}$ where we gave a proof for $\mathcal{R}_{\text{PAY}}^1$ using $\mathsf{NIZKPoK}$ and gave a proof of $\mathcal{R}_{\text{PAY}}^2$ with global extraction using NIZK. In the present case we will give a proof for

$$\mathsf{tid}' = g_0^{s'} \cdot g_1^{\mathsf{U}} \cdot g_2^{\mathsf{S}} \cdot g_3^{n_{\mathsf{U}}} \cdot g_4^{a} \cdot g_5^{\mathsf{hid}} \wedge \mathsf{vk} = g_5^K \tag{13}$$

using $\mathsf{NIZKPoK}$ by using a straight-forward group homomorphism proof. In parallel we give a proof for Eq. (12) using a NIZK proof with global extraction. This proof is about a multiplicative relation in the exponent and can therefore be constructed using [AFK22]. If the witness extracted from $\mathsf{NIZKPoK}$ for Eq. (13) by the UC simulator does not fulfill Eq. (12), then we can use global extraction of NIZK to get another witness which *does* fulfill Eq. (12) and therefore also Eq. (13). But clearly, computing two different openings of Eq. (13) in poly-time breaks DL using a stand poly-time reduction.

12 Generalised Dodis-Yampolskiy Theorem

In this section we give a generalised proof of the security of the Dodis-Yampolskiy VRF without using a random oracle. In our work, we only treat it as a PRF as this is all we need from it. Dodis and

Yampolskiy [DY05] prove their VRF construction secure under the Diffie-Hellman inversion (DHI) problem, but their reduction has an exponential security loss, meaning that the DHI assumption needs to hold with exponential security. More concretely this means that the DHI problem needs to be secure against an adversary making 2^λ queries, where λ is the output length of a collision-resistant hash function. The reduction at some point iterates over all possible output of the hash function used.

At the time of writing, most groups used in practice for DHI are shorter than the length of hash functions that are considered collision resistance. Because of the birthday paradox this would appear to be a staying situation. Here we do a more fine grained version of the DY reduction which allows us to avoid the exponential security assumption on DHI. We emphasize that we are not adding anything fundamentally different, we merely squeeze concretely better parameters out of the existing proof technique by iterating only over some controlled subsets of the outputs of the hash function.

Definition 34 (DHI). *We say that $\mathbb{G}$ is (ϵ, ℓ)-DHI against $\mathcal{A}$ if $\mathsf{Adv}^{DHI}_{\mathcal{A}} \le \epsilon$, where*

$$\mathsf{Adv}^{DHI}_{\mathcal{A},\ell} := \Pr\left[\begin{array}{r} g \leftarrow \mathbb{G} \\ \beta \leftarrow \mathbb{Z}_q \\ G_0 \leftarrow g^{\beta^{-1} \bmod q} \\ G_1 \leftarrow \mathbb{G} \\ b \leftarrow \{0,1\} \\ c \leftarrow \mathcal{A}(g, g^\beta, \ldots, g^{\beta^\ell}, G_b) \end{array} \middle\| \; c = b \right] - \frac{1}{2} \; .$$

We make a definition which is equivalent to the DY VRF being adaptive secure.

Definition 35 (DY). *We say that $\mathbb{G}$ is ϵ-DY secure against $\mathcal{A}$ if $\mathsf{Adv}^{DY}_{\mathcal{A}} \le \epsilon$, where*

$$\mathsf{Adv}^{DY} := \Pr\left[\begin{array}{r} h \leftarrow \mathbb{G} \\ \alpha \leftarrow \mathbb{Z}_q \\ G \leftarrow h^\alpha \\ \mathcal{O}_0(\mathtt{x}) := \begin{cases} \textit{add } \mathtt{x} \textit{ to initially empty } Q \\ \textit{return } h^{(\mathtt{x}+\alpha)^{-1} \bmod q} \end{cases} \\ x^* \leftarrow \mathcal{A}^{\mathcal{O}_0}(h, G) \\ X_1 \leftarrow \mathbb{G} \\ X_0 \leftarrow \begin{cases} X_1 & \textit{if } x^* \in Q \\ h^{(x^*+\alpha)^{-1} \bmod q} & \textit{otherwise} \end{cases} \\ \mathcal{O}_1(\mathtt{x}) := \begin{cases} \perp & \textit{if } \mathtt{x} = x^* \\ h^{(\mathtt{x}+\alpha)^{-1} \bmod q} & \textit{otherwise} \end{cases} \\ b \leftarrow \{0,1\} \\ c \leftarrow \mathcal{A}^{\mathcal{O}_1}(h, G, X_b) \end{array} \middle\| \; c = b \right] \le \frac{1}{2} \pm \mathsf{negl}(\lambda) \; .$$

Our goal is to reduce DY to DHI. As a stepping stone we use another assumption which we call DYZ (Dodis-Yampolskiy with Zero challenge). It is equivalent to the DY VRF being secure if the adversary is always challenging on $m^* = 0$

Definition 36 (DY zero). *We say that $\mathbb{G}$ is ϵ-DYZ against $\mathcal{A}$ if $\mathsf{Adv}^{DYZ}_{\mathcal{A}} \leq \epsilon$, where*

$$\mathsf{Adv}^{DYZ}_{\mathcal{A}} := \Pr\left[\begin{array}{r|l} h \leftarrow \mathbb{G} & \\ \beta \leftarrow \mathbb{Z}_q & \\ H \leftarrow h^{\beta} & \\ H_0 \leftarrow h^{\beta^{-1} \bmod q} & \\ H_1 \leftarrow \mathbb{G} & g = b \\ \mathcal{O}(\mathtt{x}) := \begin{cases} \perp & \textit{if } \mathtt{x} = 0 \\ h^{(\mathtt{x}+\beta)^{-1} \bmod q} & \textit{otherwise} \end{cases} & \\ b \leftarrow \{0,1\} & \\ g \leftarrow \mathcal{A}^{\mathcal{O}}(h, H, H_b) & \end{array}\right] - \frac{1}{2} .$$

Let X be a set such that we are guaranteed that $Q \subseteq X$. Ultimatly we could set $X = \mathbb{Z}_q$ but we might be in a setting where we can limit the query set further. Let Y be a set such that it is guaranteed that $x^* \in Y$ in the DY game. We call $\mathcal{A}$ for the DY game $(\ell_{\mathrm{X}}, \ell_{\mathrm{Y}})$-limited it it starts by outputting (X, Y) which limits it as above and where $|X| \leq \ell_{\mathrm{X}}$ and $|Y| \leq \ell_{\mathrm{Y}}$. We call $\mathcal{A}$ for the DYZ game ℓ_{X}-limited it it starters by outputting X with $|X| \leq \ell_{\mathrm{X}}$.

Theorem 6. *For all $(\ell_{\mathrm{X}}, \ell_{\mathrm{Y}})$-limited adversary $\mathcal{A}$ for the DY game there exists an adversary $\mathcal{B}$ for the ξ-DHI game, with $\xi = \ell_{\mathrm{X}} + 1$, which runs $\mathcal{A}$ once plus some $\widetilde{O}(\ell_X)$ operations in the group $\mathbb{G}$, and such that*

$$\mathsf{Adv}^{DY}_{\mathcal{A}} \leq \ell_Y \cdot \mathsf{Adv}^{DHI}_{\mathcal{B},\xi}$$

Lemma 22. *For all $(\ell_{\mathrm{X}}, \ell_{\mathrm{Y}})$-limited adversary $\mathcal{A}$ for the DY game there exists an ℓ_{X}-limited adversary $\mathcal{B}$ for the DYZ game, which runs $\mathcal{A}$ once and such that*

$$\mathsf{Adv}^{DY}_{\mathcal{A}} \leq \ell_Y \cdot \mathsf{Adv}^{DYZ}_{\mathcal{B}} .$$

Proof. Assume we have an adversary $\mathcal{A}$ for the DY game. We construct $\mathcal{B}$ for the DYZ game. First run $\mathcal{A}$ to get (X, Y). Sample a uniformly random $x_0 \leftarrow Y$. This is our guess at what x^* will be. Output $X' = \{x - x_0 \,|\, x \in X\}$. Now we receive (h, H, H_b) from the DYZ game where $H = h^{\beta}$, $H_0 \leftarrow h^{\beta^{-1} \bmod q}$, and $H_1 \leftarrow \mathbb{G}$. Define

$$\alpha := \beta - x_0 \bmod q$$

such that $\beta = x_0 + \alpha \bmod q$. Compute

$$G \leftarrow h^{-x_0} = h^{\beta} h^{-x_0} = h^{\beta - x_0} = h^{\alpha} .$$

Let

$$\mathcal{O}_0(\mathtt{x}) = \mathcal{O}(\mathtt{x} - x_0) .$$

Note that if $x \in X$ then $x - x_0 \in X'$, so $\mathcal{B}$ is also ℓ_{X}-limited. Note also that

$$\mathcal{O}_0(x) = \mathcal{O}(x - x_0) = h^{((x-x_0)+\beta)^{-1} \bmod q} = h^{(x+\alpha)^{-1} \bmod q} .$$

Run

$$x^* \leftarrow \mathcal{A}^{\mathcal{O}_0}(h, G) .$$

If $x^* \neq x_0$ then output a random guess $c \leftarrow \{0,1\}$. Otherwise, proceed as follows. Below we assume x^* is fresh, i.e., $x^* \notin Q$. This is without loss of generality. Let $\mathcal{O}_1(\mathtt{x}) = \bot$ if $\mathtt{x} = x^*$ and let $\mathcal{O}_1(\mathtt{x}) = \mathcal{O}(\mathtt{x} - x_0)$ otherwise. For $d = 0, 1$ define

$$X_d = H_d \ .$$

Note that we know X_b and that

$$X_0 = h^{\beta^{-1} \bmod q} = h^{(x_0+\alpha)^{-1} \bmod q} = h^{(x^*+\alpha)^{-1} \bmod q}$$

$$X_1 = H_1 \leftarrow \mathbb{G} \ .$$

By construction, when $x^* = x_0$ then the values shown to $\mathcal{A}$ are exactly as in the DY game. Since $x^* \in Y$ we have that $x_0 = x^*$ correctly with probability $1/\ell_{\mathrm{Y}}$. Therefore

$$\mathsf{Adv}_{\mathcal{B}}^{\mathrm{DYZ}} = \mathsf{Adv}_{\mathcal{A}}^{\mathrm{DY}} / \ell_{\mathrm{Y}} \ .$$

Lemma 23. *For all ℓ_{X}-limited adversary $\mathcal{A}$ for the DYZ game there exists an adversary $\mathcal{B}$ for the ξ-DHI game, with $\xi = \ell_{\mathrm{X}} + 1$, which runs $\mathcal{A}$ once plus some $\widetilde{O}(\ell_{\mathrm{X}})$ operations in the group $\mathbb{G}$ and such that*

$$\mathsf{Adv}_{\mathcal{A}}^{DYZ} \leq \mathsf{Adv}_{\mathcal{B},\xi}^{DHI} \ .$$

Proof. We are given $\mathcal{A}$ for the DYZ game and describe $\mathcal{B}$ for the DHI game. Run $\mathcal{A}$ to get X such $|X| \leq \ell_{\mathrm{X}}$ and such that all queries to $\mathcal{O}$ are in X and $0 \notin X$. Let $\xi = \ell_{\mathrm{X}} + 1$ and be given

$$g, g_1 = g^{\beta}, g_2 = g^{\beta^2}, \ldots, g_{\xi} = g^{\beta^{\xi}}, G_b \ ,$$

where $G_0 = g^{\beta^{-1} \bmod q}$ and $G_1 \leftarrow \mathbb{G}$. Define

$$f(\mathtt{x}) := \prod_{z \in X} (\mathtt{x} + z) \ .$$

Compute coefficients $c_0, c_1, \ldots, c_{\ell_{\mathrm{X}}}$ such that

$$f(\mathtt{x}) = \sum_{i=0}^{\ell_{\mathrm{X}}} c_i \mathtt{x}^i \ .$$

Define

$$h := \prod_{i=0}^{\ell_{\mathrm{X}}} g_i^{c_i} = g^{\sum_{i=0}^{\ell_{\mathrm{X}}} c_i \beta^i} = g^{f(\beta)} \ .$$

Note that

$$h^{\beta} = g^{\left(\sum_{i=0}^{\ell_{\mathrm{X}}} c_i \beta^i\right)\beta} = g^{\sum_{i=0}^{\ell_{\mathrm{X}}} c_i \beta^{i+1}} \ .$$

So, we can compute H as in the DYZ game as follows:

$$H = \prod_{i=0}^{\ell_{\mathrm{X}}} g_{i+1}^{c_i} = h^{\beta} \ .$$

This is the place where we use $\xi = \ell_{\mathrm{X}} + 1$. To answer $\mathcal{O}(x)$ as in the DYZ game we proceed as follows. Define the polynomial

$$f_x(\mathtt{x}) \coloneqq \prod_{z \in X \setminus \{x\}} (\mathtt{x} + z) = f(\mathtt{x})/(\mathtt{x} + x) \ .$$

Compute coefficients $d_0, d_1, \ldots, d_{\ell_{\mathrm{X}}-1}$ such that

$$f_x(\mathtt{x}) = \sum_{i=0}^{\ell_{\mathrm{X}}-1} d_i \mathtt{x}^i \ .$$

As above we can compute

$$\mathcal{O}(x) = \prod_{i=0}^{\ell_{\mathrm{X}}-1} g_i^{d_i} = g^{\sum_{i=0}^{\ell_{\mathrm{X}}-1} d_i \beta^i} = g^{f_x(\beta)} = g^{f(\beta)/(\beta+x)} = h^{(\beta+x)^{-1} \bmod q} \ .$$

Finally we address how to compute the challenges. Define

$$f_0(\mathtt{x}) \coloneqq f(\mathtt{x})/\mathtt{x} \ .$$

We have that

$$f_0(\mathtt{x}) = c_0/\mathtt{x} + \sum_{i=1}^{\ell_{\mathrm{X}}} c_i \mathtt{x}^{i-1} \ ,$$

$$f_0(\beta) \bmod q = f(\beta)\beta^{-1} \bmod q \ .$$

Compute

$$H_b \leftarrow G_b^{c_0} \prod_{i=1}^{\ell_{\mathrm{X}}} g_{i-1}^{c_i} \ .$$

Note that

$$H_0 = G_0^{c_0} \prod_{i=1}^{\ell_{\mathrm{X}}} g_{i-1}^{c_i} = g^{c_0 \beta^{-1} + \sum_{i=1}^{\ell_{\mathrm{X}}} c_i \beta^{i-1}} = g^{f_0(\beta)} = g^{f(\beta)/\beta} = h^{\beta^{-1} \bmod q} \ ,$$

as in the DY game. Since $x \notin X$ we have that $c_0 \neq 0$. Therefore it follows from G_1 being uniform that

$$H_1 = G_1^{c_0} \prod_{i=1}^{\ell_{\mathrm{X}}} g_{i-1}^{c_i}$$

is uniform, as desired. We compute $c = \mathcal{A}^{\mathcal{O}}(h, H, H_b)$ and return c. It is easy to see that all values by construction are as in the DY game, so $\mathsf{Adv}_{\mathcal{B},\xi}^{\mathrm{DHI}} = \mathsf{Adv}_{\mathcal{A}}^{\mathrm{DYZ}}$. $\square$

References

AC20. Thomas Attema and Ronald Cramer. Compressed Σ-protocol theory and practical application to plug & play secure algorithmics. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part III*, volume 12172 of *LNCS*, pages 513–543. Springer, Heidelberg, August 2020.

AFK22. Thomas Attema, Serge Fehr, and Michael Klooß. Fiat-shamir transformation of multi-round interactive proofs. In Eike Kiltz and Vinod Vaikuntanathan, editors, *Theory of Cryptography - 20th International Conference, TCC 2022, Chicago, IL, USA, November 7-10, 2022, Proceedings, Part I*, volume 13747 of *Lecture Notes in Computer Science*, pages 113–142. Springer, 2022.

BBDP01. Mihir Bellare, Alexandra Boldyreva, Anand Desai, and David Pointcheval. Key-privacy in public-key encryption. In Colin Boyd, editor, *ASIACRYPT 2001*, volume 2248 of *LNCS*, pages 566–582. Springer, Heidelberg, December 2001.

BCG$^+$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 Computer Society Press, May 2014.

BCH$^+$20. Christian Badertscher, Ran Canetti, Julia Hesse, Björn Tackmann, and Vassilis Zikas. Universal composition with global subroutines: Capturing global setup within plain UC. In Rafael Pass and Krzysztof Pietrzak, editors, *Theory of Cryptography - 18th International Conference, TCC 2020, Durham, NC, USA, November 16-19, 2020, Proceedings, Part III*, volume 12552 of *Lecture Notes in Computer Science*, pages 1–30. Springer, 2020.

BCK$^+$23. Donald Beaver, Konstantinos Chalkias, Mahimna Kelkar, Lefteris Kokoris-Kogias, Kevin Lewi, Ladi de Naurois, Valeria Nikolaenko, Arnab Roy, and Alberto Sonnino. STROBE: streaming threshold random beacons. In Joseph Bonneau and S. Matthew Weinberg, editors, *5th Conference on Advances in Financial Technologies, AFT 2023, October 23-25, 2023, Princeton, NJ, USA*, LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

BDPR98. Mihir Bellare, Anand Desai, David Pointcheval, and Phillip Rogaway. Relations among notions of security for public-key encryption schemes. In Hugo Krawczyk, editor, *CRYPTO'98*, volume 1462 of *LNCS*, pages 26–45. Springer, Heidelberg, August 1998.

BGG$^+$20. Fabrice Benhamouda, Craig Gentry, Sergey Gorbunov, Shai Halevi, Hugo Krawczyk, Chengyu Lin, Tal Rabin, and Leonid Reyzin. Can a public blockchain keep a secret? In Rafael Pass and Krzysztof Pietrzak, editors, *TCC 2020, Part I*, volume 12550 of *LNCS*, pages 260–290. Springer, Heidelberg, November 2020.

BGI14. Elette Boyle, Shafi Goldwasser, and Ioana Ivan. Functional signatures and pseudorandom functions. In Hugo Krawczyk, editor, *PKC 2014*, volume 8383 of *LNCS*, pages 501–519. Springer, Heidelberg, March 2014.

Blu82. Manuel Blum. Coin flipping by telephone. In *Proc. IEEE Spring COMPCOM*, pages 133–137, 1982.

BMRS20. Joseph Bonneau, Izaak Meckler, Vanishree Rao, and Evan Shapiro. Coda: Decentralized cryptocurrency at scale. Cryptology ePrint Archive, Report 2020/352, 2020. `https://eprint.iacr.org/2020/352`.

BW13. Dan Boneh and Brent Waters. Constrained pseudorandom functions and their applications. In Kazue Sako and Palash Sarkar, editors, *ASIACRYPT 2013, Part II*, volume 8270 of *LNCS*, pages 280–300. Springer, Heidelberg, December 2013.

Can01. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.

Can20. Ran Canetti. Universally composable security. *J. ACM*, 67(5):28:1–28:94, 2020.

CFH$^+$22. Matteo Campanelli, Dario Fiore, Semin Han, Jihye Kim, Dimitris Kolonelos, and Hyunok Oh. Succinct zero-knowledge batch proofs for set accumulators. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 455–469. ACM Press, November 2022.

CGKS95. Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *36th FOCS*, pages 41–50. IEEE Computer Society Press, October 1995.

CH22. Matteo Campanelli and Mathias Hall-Andersen. Veksel: Simple, efficient, anonymous payments with large anonymity sets from well-studied assumptions. In Yuji Suga, Kouichi Sakurai, Xuhua Ding, and Kazue Sako, editors, *ASIA CCS '22: ACM Asia Conference on Computer and Communications Security, Nagasaki, Japan, 30 May 2022 - 3 June 2022*, pages 652–666. ACM, 2022.

Cha82. David Chaum. Blind signatures for untraceable payments. In David Chaum, Ronald L. Rivest, and Alan T. Sherman, editors, *CRYPTO'82*, pages 199–203. Plenum Press, New York, USA, 1982.

CHK23. Matteo Campanelli, Mathias Hall-Andersen, and Simon Holmgaard Kamp. Curve trees: Practical and transparent zero-knowledge accumulators. In Joseph A. Calandrino and Carmela Troncoso, editors, *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023.* USENIX Association, 2023.

CKS11. Jan Camenisch, Stephan Krenn, and Victor Shoup. A framework for practical universally composable zero-knowledge protocols. In Dong Hoon Lee and Xiaoyun Wang, editors, *Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and*

Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings, volume 7073 of *Lecture Notes in Computer Science*, pages 449–467. Springer, 2011.

Cra97. Ronald Cramer. *Modular Design of Secure yet Practical Cryptographic Protocols.* PhD thesis, January 1997.

DH76. Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

DPSZ12. Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 643–662. Springer, Heidelberg, August 2012.

DS21. Dominic Deuber and Dominique Schröder. CoinJoin in the wild - an empirical analysis in dash. In Elisa Bertino, Haya Shulman, and Michael Waidner, editors, *ESORICS 2021, Part II*, volume 12973 of *LNCS*, pages 461–480. Springer, Heidelberg, October 2021.

DY05. Yevgeniy Dodis and Aleksandr Yampolskiy. A verifiable random function with short proofs and keys. In Serge Vaudenay, editor, *PKC 2005*, volume 3386 of *LNCS*, pages 416–431. Springer, Heidelberg, January 2005.

ElG85. Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.

Fis05. Marc Fischlin. Communication-efficient non-interactive proofs of knowledge with online extractors. In Victor Shoup, editor, *CRYPTO 2005*, volume 3621 of *LNCS*, pages 152–168. Springer, Heidelberg, August 2005.

Fis06. Marc Fischlin. Round-optimal composable blind signatures in the common reference string model. In Cynthia Dwork, editor, *CRYPTO 2006*, volume 4117 of *LNCS*, pages 60–77. Springer, Heidelberg, August 2006.

FMMO19. Prastudy Fauzi, Sarah Meiklejohn, Rebekah Mercer, and Claudio Orlandi. Quisquis: A new design for anonymous cryptocurrencies. In Steven D. Galbraith and Shiho Moriai, editors, *ASIACRYPT 2019, Part I*, volume 11921 of *LNCS*, pages 649–678. Springer, Heidelberg, December 2019.

FS87. Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *CRYPTO'86*, volume 263 of *LNCS*, pages 186–194. Springer, Heidelberg, August 1987.

GGM84. Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions (extended abstract). In *25th FOCS*, pages 464–479. IEEE Computer Society Press, October 1984.

GHK$^+$21. Craig Gentry, Shai Halevi, Hugo Krawczyk, Bernardo Magri, Jesper Buus Nielsen, Tal Rabin, and Sophia Yakoubov. YOSO: You only speak once - secure MPC with stateless ephemeral roles. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part II*, volume 12826 of *LNCS*, pages 64–93, Virtual Event, August 2021. Springer, Heidelberg.

GK15. Jens Groth and Markulf Kohlweiss. One-out-of-many proofs: Or how to leak a secret and spend a coin. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 253–280. Springer, Heidelberg, April 2015.

Gol87. Oded Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In Alfred Aho, editor, *19th ACM STOC*, pages 182–194. ACM Press, May 1987.

HF16. Martin Harrigan and Christoph Fretter. The unreasonable effectiveness of address clustering. In *2016 intl ieee conferences on ubiquitous intelligence & computing, advanced and trusted computing, scalable computing and communications, cloud and big data computing, internet of people, and smart world congress (uic/atc/scalcom/cbdcom/iop/smartworld)*, pages 368–373. IEEE, 2016.

JBWD18. Marc Jourdan, Sebastien Blandin, Laura Wynter, and Pralhad Deshpande. Characterizing entities in the bitcoin blockchain. In *2018 IEEE international conference on data mining workshops (ICDMW)*, pages 55–62. IEEE, 2018.

KFTS17. Amrit Kumar, Clément Fischer, Shruti Tople, and Prateek Saxena. A traceability analysis of monero's blockchain. In Simon N. Foley, Dieter Gollmann, and Einar Snekkenes, editors, *ESORICS 2017, Part II*, volume 10493 of *LNCS*, pages 153–173. Springer, Heidelberg, September 2017.

KPTZ13. Aggelos Kiayias, Stavros Papadopoulos, Nikos Triandopoulos, and Thomas Zacharias. Delegatable pseudorandom functions and applications. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 669–684. ACM Press, November 2013.

KS22. Yashvanth Kondi and Abhi Shelat. Improved straight-line extraction in the random oracle model with applications to signature aggregation. In Shweta Agrawal and Dongdai Lin, editors, *Advances in Cryptology - ASIACRYPT 2022 - 28th International Conference on the Theory and Application of Cryptology* *and Information Security, Taipei, Taiwan, December 5-9, 2022, Proceedings, Part II*, volume 13792 of *Lecture Notes in Computer Science*, pages 279–309. Springer, 2022.

KY00. Jonathan Katz and Moti Yung. Complete characterization of security notions for probabilistic private-key encryption. In *32nd ACM STOC*, pages 245–254. ACM Press, May 2000.

Lin15. Yehuda Lindell. An efficient transform from sigma protocols to NIZK with a CRS and non-programmable random oracle. In Yevgeniy Dodis and Jesper Buus Nielsen, editors, *TCC 2015, Part I*, volume 9014 of *LNCS*, pages 93–109. Springer, Heidelberg, March 2015.

LN18. Kasper Green Larsen and Jesper Buus Nielsen. Yes, there is an oblivious RAM lower bound! In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 523–542. Springer, Heidelberg, August 2018.

LR22. Anna Lysyanskaya and Leah Namisa Rosenbloom. Universally composable $\varsigma $-protocols in the global random-oracle model. In Eike Kiltz and Vinod Vaikuntanathan, editors, *Theory of Cryptography - 20th International Conference, TCC 2022, Chicago, IL, USA, November 7-10, 2022, Proceedings, Part I*, volume 13747 of *Lecture Notes in Computer Science*, pages 203–233. Springer, 2022.

LWW04. Joseph K. Liu, Victor K. Wei, and Duncan S. Wong. Linkable spontaneous anonymous group signature for ad hoc groups (extended abstract). In Huaxiong Wang, Josef Pieprzyk, and Vijay Varadharajan, editors, *ACISP 04*, volume 3108 of *LNCS*, pages 325–335. Springer, Heidelberg, July 2004.

Max13. Greg Maxwell. Coinjoin: Bitcoin privacy for the real world. *Post on Bitcoin Forum*, 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.

Nie17. Jesper Buus Nielsen. Universally composable zero-knowledge proof of membership. Cryptology ePrint Archive, Paper 2017/362, 2017. `https://eprint.iacr.org/2017/362`.

OMJ$^+$13. Damien Octeau, Patrick D. McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective inter-component communication mapping in android: An essential step towards holistic security analysis. In Samuel T. King, editor, *USENIX Security 2013*, pages 543–558. USENIX Association, August 2013.

Ost90. Rafail Ostrovsky. An efficient software protection scheme (rump session). In Gilles Brassard, editor, *CRYPTO'89*, volume 435 of *LNCS*, pages 610–611. Springer, Heidelberg, August 1990.

Ped92. Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In Joan Feigenbaum, editor, *CRYPTO'91*, volume 576 of *LNCS*, pages 129–140. Springer, Heidelberg, August 1992.

RH13. Fergal Reid and Martin Harrigan. An analysis of anonymity in the bitcoin system. In *Security and Privacy in Social Networks*, 2013.

RST01. Ronald L. Rivest, Adi Shamir, and Yael Tauman. How to leak a secret. In Colin Boyd, editor, *ASIACRYPT 2001*, volume 2248 of *LNCS*, pages 552–565. Springer, Heidelberg, December 2001.

SCSL11. Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 197–214. Springer, Heidelberg, December 2011.

STW23. Srinath T. V. Setty, Justin Thaler, and Riad S. Wahby. Unlocking the lookup singularity with lasso. *IACR Cryptol. ePrint Arch.*, page 1216, 2023.

SV15. Berry Schoenmakers and Meilof Veeningen. Universally verifiable multiparty computation from threshold homomorphic cryptosystems. In Tal Malkin, Vladimir Kolesnikov, Allison Bishop Lewko, and Michalis Polychronakis, editors, *ACNS 15*, volume 9092 of *LNCS*, pages 3–22. Springer, Heidelberg, June 2015.

Val08. Paul Valiant. Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In Ran Canetti, editor, *TCC 2008*, volume 4948 of *LNCS*, pages 1–18. Springer, Heidelberg, March 2008.

Vij23. Saravanan Vijayakumaran. Analysis of cryptonote transaction graphs using the dulmage-mendelsohn decomposition. In *5th Conference on Advances in Financial Technologies, AFT 2023, October 23-25, 2023, Princeton, NJ, USA*, volume 282 of *LIPIcs*, pages 28:1–28:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

Wik09. Douglas Wikström. A commitment-consistent proof of a shuffle. In Colin Boyd and Juan Manuel González Nieto, editors, *Information Security and Privacy, 14th Australasian Conference, ACISP 2009, Brisbane, Australia, July 1-3, 2009, Proceedings*, volume 5594 of *Lecture Notes in Computer Science*, pages 407–421. Springer, 2009.

YAY$^+$19. Zuoxia Yu, Man Ho Au, Jiangshan Yu, Rupeng Yang, Qiuliang Xu, and Wang Fat Lau. New empirical traceability analysis of CryptoNote-style blockchains. In Ian Goldberg and Tyler Moore, editors, *FC 2019*, volume 11598 of *LNCS*, pages 133–149. Springer, Heidelberg, February 2019.

ZBK^{+}22. Arantxa Zapico, Vitalik Buterin, Dmitry Khovratovich, Mary Maller, Anca Nitulescu, and Mark Simkin. Caulk: Lookup arguments in sublinear time. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 3121–3134. ACM Press, November 2022.