Z-Channel: Scalable and Efficient Scheme in Zerocash

Yuncong Zhang, Yu Long, Zhen Liu, Zhiqiang Liu, and Dawu Gu

Shanghai Jiao Tong University
Lab of Cryptology and Computer Security

Abstract. Decentralized ledger-based cryptocurrencies such as Bitcoin provide a means to construct payment systems without requiring a trusted bank, yet the anonymity of Bitcoin is proved to be far from enough. Zerocash is the first full-fledged anonymouse digital currency based on the blockchain technology, using zk-SNARK as the zero-knowledge module for the privacy preserving. Zerocash solves the privacy problem but also brings some other issues, including insufficient scalability as in Bitcoin. Meanwhile, Lightning network proves to be a nice solution to solve the scalability problem in Bitcoin. However, to employ the idea of lightning network in Zerocash is a great challenge due to the lack of programmability of Zerocash. We modify the Zerocash scheme to implement multisignature scheme and the lock time mechanism without compromising the privacy guarantee provided by Zerocash. With these two mechanisms, we present the construction of micropayment system Z-Channel on the basis of Zerocash. The Z-Channel system effectively solves the scalability and instant payment problems in Zerocash.

Keywords: Cryptocurrency, Zerocash, Scalability, Privacy, Instant payment

1 Introduction

Decentralized ledger-based cryptocurrencies such as Bitcoin [16] provide a means to construct payment systems without requiring a trusted bank. It was believed that Bitcoin is an anonymous digital currency which contains no information concerned with the personal identity of users. Unfortunately, the anonymity of Bitcoin is proved to be easily compromised [18]. Any user can analyze the transaction graph, values and dates in the ledger to possibly link Bitcoin addresses with real world identity. To break such linkability in Bitcoin, one can store his Bitcoin into a *mix*, which is a trusted central party aiming at mixing Bitcoins from different users and gives different coins back to them after sufficient amount of coins are mixed together. However, the delay in redeeming the coins and the trust to a central party can still be unacceptable to some users with strong motivation to hide information.

Much work has been done to implement a decentralized version of mix in the digital currency, such as TumbleBit [9], CoinSwap [14], CoinParty [24] and CoinShuffle[19] which is based on the work of CoinJoin [13]. Additionally, a lot of altcoins have been developed, including Zerocoin [15], BlindCoin [22] and its predecessor Mixcoin [3] and Pinocchio coin [4], etc.

Zerocash [20], which is based on Zerocoin, is the first full-fledged privacy preserving digital currency based on the blockchain technology, using zk-SNARK as the zero-knowledge module for the privacy protection. Zerocash is an instantiation of the Distributed Anonymous Payment (DAP) scheme. Compared with Zerocoin, zero-knowledge

proofs in Zerocash transactions are much more succinct and faster to verify, and direct payments are supported from coins to coins with arbitrary domination. Furthermore, the underlying zk-SNARK proof mechanism is flexible enough to support various additional policies. For example, it supports the scenario where a user is required to prove that he paid proper taxes on all transactions without even revealing the amount of taxes paid [7].

However, Zerocash still suffers scalability and performance problems like Bitcoin. In fact, the size of Zerocash transactions are larger and the time to verify zk-SNARK proof is significantly longer than verifying a Bitcoin transaction, which makes the scalability problem in Zerocash even worse than in Bitcoin. Meanwhile, many approaches have been proposed to solve the scalability problem of Bitcoin, such as changing the blocksize [1]. A most popular class of solutions are based on the idea of micropayment channel which supports high-frequency instant off-chain payments. Among them the Lightning network [17] proves to be on of the most promising. By transactions conducted securely off-chain using Bitcoin scripting, Lightning network enables Bitcoin to scale to billions of users without custodial risk or blockchain centralization.

Although Lightning network is promising in improving the scalability of ledger-based cryptocurrencies, transplanting the idea of Lightning network to Zerocash is a great challenge. The bidirectional micropayment channel employed in Lightning network makes heavy use of the scripting feature of Bitcoin, which Zerocash is particularly in lack of. Specifically, the micropayment channel relies on the features of multisignature and lock time mechanism. These features can be easily implemented by Bitcoin scripting language, but nontrivial to be embedded into Zerocash without compromising the privacy. In Zerocash, to allow the zk-SNARK module to verify transaction, the verification program has to be implemented into a circuit which is inputed in zk-SNARK at startup of the ledger. It is difficult to construct a general purpose circuit for a script language, which is almost equivalent to designing a computer chip. Such circuits will be significantly large in size and the resulting zero-knowledge proof would be intractable.

Related work. Decentralized cryptocurrency has been drawing much attention since Bitcoin was proposed in 2008. So far many digital currencies have been devised following this new trend such as [5, 10]. Efforts have been devoted to improving known cryptocurrencies or designing new schemes by analyzing the security and performance of Bitcoin [6, 12], proposing various consensus schemes suitable for different scenarios or improving existing consensus protocols to make them more powerful and applicable [21, 10], enhancing the scalability and efficiency of Bitcoin [23, 5], presenting mechanisms for privacy preserving of digital currencies [11, 22], etc.

Our contribution. In this work we address the above problems by the following contributions:

1. We present ideas to implement multisignature scheme in Zerocash. This multisignature scheme allows the following application of Zerocash: two or more parties need to

share an address, in the sense that a coin in the shared address can only be spent by cooperation of all parties. Others could verify that the public keys were committed in the coin by the zero-knowledge proof and that the signatures are valid by public signature scheme. Meanwhile, the privacy is still protected, which means others could not get any more information apart from the validity of the transaction.

2. We manage to implement the lock time mechanism in Zerocash, which allows to check whether the lock time of the input coin is effective or not, but reveals no information about the lock time length or create time of the input coin.
3. We give the definition of security of our modified scheme according to that of Zerocash in [20]. We prove that our scheme is secure under this definition. In addition, we find a weakness in the security model in [20], which we fix in our scheme.
4. We make use of the above mentioned mechanisms to transplant the lightning network micropayment channel into Zerocash, and develop **Z-Channel**. Z-Channel provides privacy protection and the instant off-chain payment features. Compared with Zerocash, Z-Channel significantly enhances the scalability, allowing a great number of users to perform high-frequency transactions off-chain in day-to-day routine, and the payment is made nearly instantly. Meanwhile, the Z-Channels are established and terminated with strong privacy guarantee.

Paper organization. The remainder of the paper is organized as follows. Section 2 introduces the preliminaries needed for our work. Section 3 improves the Zerocash scheme by embedding the multisignature and lock time mechanisms and gives the security proofs for the improved scheme. In Sect.4, we present the construction of Z-Channel based on our newly proposed scheme. Finally, Sect.5 concludes this paper.

2 Preliminaries

2.1 Background on zk-SNARKs

A zk-SNARK (Succinct Non-interactive ARgument of Knowledge) is a triple of algorithms (`KeyGen`, `Prove`, `Verify`).

Let C denote a circuit verifying an NP language $\mathcal{L}_C$ which takes as input an instance x and witness w, and outputs b indicating if w is a valid witness for x.

The algorithm `KeyGen` takes C as input and outputs a *proving key* `pk` and a *verification key* `vk`.

The algorithm `Prove` takes as input a instance of the NP problem x and a witness w, as well as `pk`, and generates a non-interactive proof π for the statement $x \in \mathcal{L}_C$.

The algorithm `Verify` takes as input the instance x and the proof π, as well as `vk`, and outputs b indicating if he is convinced that $x \in \mathcal{L}_C$.

A zk-SNARK is *correct* if the honest prover can convince the verifier. It has the quality of *proof of knowledge* if the verifier accepting a proof implies the prover knowing the witness. It has the quality of *perfect zero knowledge* if there exists a simulator which generates the same results for any instance $x \in \mathcal{L}_C$ without knowing witness w.

The work of Zerocash is based on a zk-SNARK implementation proposed in [2].

2.2 Zerocash DAP Scheme

The *decentralized anonymous payment scheme* (DAP scheme) is a full-fledged anonymous mechanism based on ledger based currency system. A DAP scheme is a tuple of six algorithms (`Setup`, `CreateAddress`, `Mint`, `Pour`, `VerifyTransaction`, `Receive`).

The algorithm `Setup` takes as input a security parameter λ and outputs public parameters `pp`.

The algorithm `CreateAddress` outputs a newly generated address key pair (addr_{pk}, addr_{sk}).

The algorithm `Mint` takes as input a value v and the destination address, and outputs a coin $\mathbf{c}$ and a mint transaction tx_{Mint}. A mint transaction consumes currency of the basecoin and outputs to a commitment. The transaction will be accepted only if the basecoin part of the transaction is valid and the commitment is correctly computed. The coin consists of the coin value and some secret values (for example, trapdoors for generating the commitment) and the destination address for the coin. The mint transaction reveals the value but nothing else of the coin.

The algorithm `Pour` takes as input two input coins, secrets for the input coins, two destination addresses and other infomration, and outputs two new coins and a pour transaction tx_{Pour}. A zero-knowledge proof π_{POUR} is appended to tx_{Pour} to prove the validity of this transaction, i.e. the validity of the input coins and the balance of this transaction, etc. The transaction reveals the unique serial numbers of the input coins to prevent double spending. To prove the existence of the input coins on the ledger, all the commitments on the ledger are maintained in a Merkle-tree, and `Pour` additionally takes as inputs a Merkle root `rt` in the Merkle-tree history, and the paths from the commitments to `rt`.

The algorithm `VerifyTransaction` takes as input the public parameters `pp` and a transaction tx_{Mint} or tx_{Pour} as well as a ledger, and outputs a bit b indicating if this transaction is valid to be appended on the ledger.

The algorithm `Receive` takes as input a pair of address keys ($\text{addr}_{\text{pk}}, \text{addr}_{\text{sk}}$) and a ledger, and outputs all unspent coins paid to the given address.

2.3 Micropayment Channel in Lightning Network

Micropayment channel allows two parties to make payments to each other without publishing Bitcoin transactions on the ledger.

To start a channel, the two parties first publish a *funding transaction* consuming their own Bitcoins and outputing to a shared address, which can be spent only by providing both signatures of the two parties.

Before they sign the funding transactions, they agree on a *commitment transaction* redeeming their Bitcoins from the funding transaction. After both of them receive the signature of the commitment transaction from the other, they sign the funding transaction and publish it on the ledger.

Each time when a payment is made, they agree on a new version of commitment transaction which redistributes the funding Bitcoins.

To prevent a malicious party from publishing an early version of commitment transaction, each time a new commitment transaction is agreed on, each party signs a *revocable transaction* as a penalty to ensure that he will not try to publish the previous commitment transaction. The revocable transaction allows the other party to get all the Bitcoins in the channel once this party publishes an earlier commitment transaction.

3 Implement Functions in Zerocash Without Script

Zerocash lacks programmability by nature [11]. To support micropayment channel in Zerocash, two key functionalities must be implemented: multisignature and lock time. We present the modification to the original Zerocash procedures to support these two functionalities.

3.1 Modification to Zerocash Scheme

In this subsection, we present our modifications to the original DAP scheme step by step, and finally obtain a new scheme supporting multisignature scheme and lock time mechanism.

Step 1: Commit a public key in the coin. In Zerocash, a coin consists of a commitment `cm` and some secret information necessary for spending this coin. The commitment involves the following information: the destination address addr_{pk}, the value v and a random string ρ which is used by the payee to compute sequence number `sn` which is the unique identifier of the coin. To spend the coin, the sequence number `sn` is revealed to prevent double-spending, other information are kept secret, and a zero-knowledge proof π_{POUR} is needed to prove that the revealed `sn` is valid.

We modify this by committing a new piece of information, a public key `pk` into the commitment. Since zk-SNARK only supports fixed length input [2], in order to allow `pk` to be of arbitrary length, thus supporting arbitrary public key schemes, we commit the hash of `pk`, denoted by $h_{\texttt{pk}}$ instead of the original public key. To spend the coin, in addition to the zero-knowledge proof π_{POUR}, the payer has to provide a signature σ which can be verified by `pk`.

If the signature verification is carried out in zero-knowledge proof, however, the verification algorithm has to be coded in the circuit, which is fairly complex and would enlarge the circuit significantly. So we decide that the verification is done directly by public signature scheme. That is to say, the payer of this coin must publish the public key together in the transaction to allow verifiers to verify the signature. To prevent malicious users from modifying the public key, we add an additional statement for the zero-knowledge proof to prove that the hash of this public key is the $h_{\texttt{pk}}$ committed in the input coin.

Note that the anonymity is compromised by the revealed public key, since the payer would immediately perceive when the payee spends the coin, by identifying the public key published in the transaction. To solve this problem, we must require that the payer

does not know pk itself, but a commitment pkcm which is generated by the payee with a random trapdoor u with input $h_{\mathtt{pk}}$. Therefore, before the payer can send a coin to the payee, the payee has to generate a fresh pkcm randomly and sends to the payer together with the address public key. When the payee spends the coin, he proves that the hash of the revealed pk is committed into the coin with the trapdoor. The payer cannot connect the revealed public key to the public key commitment he put into the coin previously.

Step 2: Commit a public key list. Next, we replace the one public key pk by a list of public keys *pklist*. We still denote the hash of this list by $h_{\mathtt{pk}}$, and the commitment of $h_{\mathtt{pk}}$ by pkcm. To spend the coin, the payee has to publish the public key list and an index k specifying which public key to use, as well as the corresponding signature which could be verified by the specified public key.

In this step, the coin can be spent by a user knowing the some private key corresponding to any of the (one or more) public keys in the list.

Step 3: Distributed generation of public key and signature. If the public key is one of a multisignature scheme, a valid pour transaction can be generated only by cooperation of more than one parties.

Specifically, we require the scheme to support the following operations:

1. **Distributed key generation**. Multiple parties cooperate to generate a pair of public/private keys pk and sk. After the protocol is done, pk is known by all the parties, while sk is invisible to every one. Each party holds a share $\mathtt{sk}_i$ of the private key.
2. **Distributed signature generation**. Given a message M, the parties holding the pieces $\mathtt{sk}_i$ of the private key cooperate to generate a signature σ on M. Specifically, each party generates a piece σ_i of the signature alone and broadcasts it to other parties. Anyone obtaining all the pieces can recover the signature σ. This signature can be verified by pk and is indistinguishable from the signatures directly signed by sk.

Step 4: Commit a lock time in coin. In this step, we commit a time T into the coin, and try to design a mechanism such that the coin cannot be spent before time T has passed since the coin is on ledger. We hope this verification about time is done with privacy, i.e. the verifier does not even know the timestamp of the input coin or the length of the lock time. We also hope to allow a user to hold a transaction for a while before publishing it. In this situation the commitment must be unaffected by the timestamp. These requirements make enabling such lock time in Zerocash particularly tricky.

To link the timestamp with this coin without affecting the coin commitment, we consider making use of the public information related to the input coin. The one we consider is the Merkle-tree root rt, which is used to prove the existence of the input coin commitment. Denote by rt_i the Merkle-tree root formed by the previous i commitments on the ledger. We define the timestamp of rt_i to be that of $\mathtt{cm}_i$, the i'th commitment on the ledger.

To spend a coin with commitment $\texttt{cm}_i$ published at time t_i with lock time t_{lock}, the payer randomly select rt_j in the root history such that $j > i$ and $t_j + t_{lock} < t$ where t is the current time, and in the zero-knowledge proof proves that:

Given current timestamp t and root rt, I know t_{lock} such that

1. *t_{lock} is correctly committed in $\texttt{cm}$, and*
2. *$t_{lock} + rt.time \leq t$.*

Step 5: Different lock time for different party. The multisignature scheme and lock time could be combined to allow different lock times for different signatures.

Instead of committing a single t_{lock} into the coin, we commit a lock time list *tlist*. Then we let the index k indicating which public key to use specifies the lock time simultaneously. The related zero-knowledge proof statement then becomes:

Given current timestamp t, root rt and index k, I know tlist such that

1. *tlist is correctly committed in $\texttt{cm}$, and*
2. *$tlist[k] + rt.time \leq t$.*

3.2 Algorithms

A DAP' scheme is a tuple of polynomial-time algorithms (`Setup`, `CreateAddress`, `CreatePKCM`, `Mint'`, `Pour'`, `VerifyTransaction'`, `Receive'`) with the following syntax.

We first present the cryptographic algorithms utilized subsequently.

- Keyed pseudorandom functions $\texttt{PRF}^{addr}$ for generating addresses, $\texttt{PRF}^{sn}$ for serial numbers and $\texttt{PRF}^{pk}$ for binding public keys with addresses.
- Information hiding trapdoor commitment `COMM`.
- Fixed-input-length collision resistent hash function `CRH` and flexible-input-length hash function `Hash`.
- Zero-knowledge module zk-SNARK (`KeyGen`, `Prove`, `Verify`), where `KeyGen` generates a pair of proving key $\texttt{pk}_{\text{POUR}}$ and verification key $\texttt{vk}_{\text{POUR}}$, `Prove` generates a zero-knowledge proof π_{POUR} for an NP statement and `Verify` checks if a zero-knowledge proof is correct.
- Public signature scheme $(\mathcal{G}_{sig}, \mathcal{K}_{sig}, \mathcal{S}_{sig}, \mathcal{V}_{sig})$, where $\mathcal{G}_{sig}$ is for public parameter generation, $\mathcal{K}_{sig}$ is the key generation algorithm, $\mathcal{S}_{sig}$ is the signing algorithm and $\mathcal{V}_{sig}$ is the verification algorithm.
- Distributed public encryption scheme $(\mathcal{G}_{dst}, \mathcal{K}_{dst}, \mathcal{S}_{dst}, \mathcal{V}_{dst})$ is defined similarly, but the algorithms can be executed distributedly by more than one parties.
- Public encryption scheme $(\mathcal{G}_{enc}, \mathcal{K}_{enc}, \mathcal{E}_{enc}, \mathcal{D}_{enc})$, where $\mathcal{G}_{enc}$ is for public parameter generation, $\mathcal{K}_{enc}$ is the key generation algorithm, $\mathcal{E}_{enc}$ is the encryption algorithm and $\mathcal{D}_{enc}$ is the decryption algorithm.

The definitions of the algorithms for the new DAP' scheme is quite similar to the original DAP scheme in [20], we present the full definitions here for completeness.

System setup. The algorithm `Setup` generates a set of public parameters.

- **Input:** security parameter λ
- **Output:** public parameters `pp`

The `Setup` algorithm is executed by a trusted party only once at the startup of the ledger, and made public to all parties. Afterwards, no trusted party is needed.

To generate the public parameters, first invoke `KeyGen` algorithm to generate ($\mathtt{pk}_{\text{POUR}}$, $\mathtt{vk}_{\text{POUR}}$), then invoke algorithms $\mathcal{G}_{sig}$, $\mathcal{G}_{enc}$ and $\mathcal{G}_{dst}$ to obtain the public parameters for the public signature schemes and the public encryption scheme.

Create address. The algorithm `CreateAddress` generates a new pair of address key pair.

- **Input:** public parameters `pp`
- **Output:** address key pair $(\text{addr}_{\text{pk}}, \text{addr}_{\text{sk}})$

Each user may execute `CreateAddress` algorithm arbitrary number of times. The address public key addr_{pk} is used by other parites to send him coins.

To generate the key pair, first sample a random string a_{sk} and compute $a_{pk} = \mathtt{PRF}^{addr}_{a_{sk}}(0)$. Then, invoke $\mathcal{K}_{enc}$ algorithm to generate a pair of public/private key pairs $(\mathtt{pk}_{enc}, \mathtt{sk}_{enc})$. Finally, output $\text{addr}_{\text{pk}} = (a_{pk}, \mathtt{pk}_{enc})$ and $\text{addr}_{\text{sk}} = (a_{sk}, \mathtt{sk}_{enc})$.

Create public key commitment. The algorithm `CreatePKCM` generates a commitment for a public key list *pklist*.

- **Input:**
  - public parameters `pp`
  - integer $K > 0$
- **Output:**
  - K public/private key pairs
  - tuple $(pklist, u, \mathtt{pkcm})$

For complete anonymity, each time a payer tries to generate a coin (with `Mint'` or `Pour'` algorithm introduced later) for the payee, the payee invokes `CreatePKCM` algorithm to generate a fresh public key commitment `pkcm` and sends the `pkcm` to the payer.

To generate `pkcm`, invoke $\mathcal{K}_{dst}$ algorithm to generate and output K public/private key pairs. Assemble these K public keys into a list *pklist* and compute $h_{\mathtt{pk}} := \mathtt{Hash}(pklist)$. Randomly sample a commitment trapdoor u and compute $\mathtt{pkcm} := \mathtt{COMM}_u(h_{\mathtt{pk}})$. Output the tuple $(pklist, u, \mathtt{pkcm})$.

For privacy, each generated `pkcm` must be used only once. It is recommended that a user stores the output tuples $(pklist, u, \mathtt{pkcm})$ in a table `PKCM`. When receiving a coin from the ledger (as described in `Receive'` algorithm), check that the `pkcm` is in table `PKCM`, and delete it from the table after the coin using this `pkcm` is spent.

Mint coin. The `Mint'` algorithm generates a coin and a mint transaction.

- **Input:**
  - public parameter `pp`
  - coin value v
  - destination address addr_{pk}
  - **public key list commitment** `pkcm`
  - **lock time array** $tlist$
- **Output:**
  - coin **c**
  - mint transaction tx_{Mint}

The `Mint` algorithm in Zerocash is invoked to generate a `Mint` transaction which spends unspent output in basecoin and outputs to a commitment. We present the modified `Mint'` algorithm to further commit the hash of public key list $pklist$ and the corresponding lock-time list $tlist$ into the commitment.

The details of the `Mint'` algorithm are presented in Alg.1.

Algorithm 1: Mint' Algorithm

Parse addr_{pk} as $a_{pk}, \texttt{pk}_{enc}$;
Randomly sample a $\texttt{PRF}^{sn}$ seed ρ;
Randomly sample three `COMM` **trapdoors** r, s, t;
Compute $m := \texttt{COMM}_r(a_{pk} \| \rho)$;
Compute $H := \texttt{CRH}(\texttt{pkcm} \| tlist)$;
Compute $k := \texttt{COMM}_s(H \| m)$;
Compute $\texttt{cm} := \texttt{COMM}_t(v \| k)$;
Set $\mathbf{c} := (\text{addr}_{\text{pk}}, v, \rho, r, s, t, \texttt{cm}, tlist, \texttt{pkcm})$;
Set $tx_{\text{Mint}} := (\texttt{cm}, v, *)$ where $* := (k, t)$;
Output **c** and tx_{Mint}.

Pour algorithm. The `Pour'` algorithm transfers values from two input coins into two new coins, and optionally reveal part of the input value. Pouring allows parties to subdivide coins, merge coins or transfer ownership. `Pour'` generates two coins and a pour transaction.

- **Input:**
  - public parameter `pp`
  - the Merkle roots rt_1 and rt_2
  - old coins $\mathbf{c}_1^{old}$ and $\mathbf{c}_2^{old}$
  - old addresses secret keys $addr_{sk,1}^{old}$ and $addr_{sk,2}^{old}$
  - path $path_1$ and $path_2$ from commitments to roots rt_1 and rt_2 respectively
  - **old public key lists and trapdoors** $(\texttt{pkcm}_1^{old}, pklist_1^{old}, u_1)$ **and** $(\texttt{pkcm}_2^{old}$**,** $pklist_2^{old}$**,** $u_2)$

  - **old lock time array** $tlist_1^{old}$**,** $tlist_2^{old}$
  - **public key indices** k_1**,** k_2
  - **secret key** sk **for** $pklist_i^{old}[k_i]$
  - new values v_1^{new}, v_2^{new}
  - new addresses public keys $addr_{pk,1}^{new}$, $addr_{pk,2}^{new}$
  - **new public key list commitments** $\texttt{pkcm}_1^{new}$**,** $\texttt{pkcm}_2^{new}$
  - **new lock time array** $tlist_1^{new}$**,** $tlist_2^{new}$
  - public value v_{pub}
  - transaction info `info`
- **Output:**
  - coins $\mathbf{c}_1$, $\mathbf{c}_2$
  - pour transaction tx_{Pour}

We modify the original `Pour` algorithm to `Pour'` to publish the public key lists *pklist* previously committed in the input coins, and (if one of the *pklist* is nonempty) append the index of which public key to use, and the corresponding signatures.

To allow parties to sign a pour transaction long time before the transaction is published on ledger, we specify that the signature of the distributed signature scheme only applies to the most significant part which we call the *kernel* of the pour information. Specifically, we define the kernel to be $\mathbf{Ker} = (\texttt{cm}_1^{new}, \texttt{cm}_2^{new}, \texttt{sn}_1^{old}, \texttt{sn}_2^{old})$.

The details of the `Pour'` algorithm are presented in Alg.2.

Verify Transaction Algorithm. The `VerifyTransaction'` algorithm outputs a bit b indicating if a given transaction is valid on a ledger.

- **Input:**
  - public parameters `pp`
  - mint/pour transaction tx
  - ledger L
- **Output:** bit b indicating if the transaction is valid

We modify the original `VerifyTransaction` algorithm to `VerifyTransaction'` to verify the zero-knowledge proof for the new NP statement, and additionally verify the signatures on the kernel of the pour transaction.

The details of `VerifyTransaction'` algorithm are presented in Alg.3.

Receive Algorithm. The `Receive'` algorithm scans the ledger and outputs coins on the ledger belonging to a given address public key.

- **Input:**
  - public parameters `pp`
  - recipient address key pair $(\text{addr}_{\text{pk}}, \text{addr}_{\text{sk}})$
  - public key commitment tuple set `PKCM`
  - ledger L
- **Output:** set of received coins

Algorithm 2: Pour' Algorithm

for $i \in \{1,2\}$ **do**
  Parse $\mathbf{c}_i^{old}$ as $(addr_{pk,i}^{old}, v_i^{old}, \rho_i^{old}, r_i^{old}, s_i^{old}, t_i^{old}, \mathtt{cm}_i^{old}, tlist_i^{old}, \mathtt{pkcm}_i^{old})$;
  Compute $h_{pk,i}^{old} = \mathtt{Hash}(pklist_i^{old})$ **for** $i = 1, 2$;
  Verify that $\mathtt{pkcm}_i^{old} = \mathtt{COMM}_{u_i^{old}}(h_{pk,i}^{old})$ **for** $i = 1, 2$;
  Parse $addr_{sk,i}^{old}$ as $a_{sk,i}, \mathtt{sk}_{enc,i}$;
  Compute $\mathtt{sn}_i^{old} := \mathtt{PRF}^{sn}_{a_{sk,i}^{old}}(\rho_i^{old})$;
  Parse $addr_{pk,i}^{new}$ as $(a_{pk,i}^{new}, \mathtt{pk}_{enc,i}^{new})$;
  Randomly sample a $\mathtt{PRF}^{sn}$ seed ρ_i^{new};
  Randomly sample three $\mathtt{COMM}$ **trapdoors** $r_i^{new}, s_i^{new}, t_i^{new}$;
  Compute $m_i^{new} := \mathtt{COMM}_{r_i^{new}}(a_{pk,i}^{new} \| \rho_i^{new})$;
  Compute $H_i^{new} := \mathtt{CRH}(\mathtt{pkcm}_i^{new} \| tlist_i^{new})$;
  Compute $k_i^{new} := \mathtt{COMM}_{s_i^{new}}(H_i^{new} \| m_i^{new})$;
  Compute $\mathtt{cm}_i^{new} := \mathtt{COMM}_{t_i^{new}}(v_i^{new} \| k_i^{new})$;
  Set $\mathbf{c}_i^{new} := (addr_{pk,i}^{new}, v_i^{new}, \rho_i^{new}, r_i^{new}, s_i^{new}, t_i^{new}, \mathtt{cm}_i^{new}, tlist_i^{new}, \mathtt{pkcm}_i^{new})$;
  Set $\mathbf{C}_i := \mathcal{E}_{enc}(\mathtt{pk}_{enc,i}^{new}, (v_i^{new}, \rho_i^{new}, r_i^{new}, s_i^{new}, t_i^{new}, tlist_i^{new}, \mathtt{pkcm}_i^{new}))$;
end
Obtain *timestamp* **from** $\mathtt{info}$;
Generate $(\mathtt{pk}_{sig}, \mathtt{sk}_{sig}) := \mathcal{K}_{sig}(\mathbf{pp}_{sig})$;
Compute $h_{sig} := \mathtt{CRH}(\mathtt{pk}_{sig})$;
Compute $h_i := \mathtt{PRF}^{pk}_{a_{sk,i}^{old}}((i-1) \| h_{sig})$ for $i = 1, 2$;
Set $\boldsymbol{x} := (rt_1, rt_2, \mathtt{sn}_1^{old}, \mathtt{sn}_2^{old}, h_{pk,1}^{old}, h_{pk,2}^{old}, \mathtt{cm}_1^{new}, \mathtt{cm}_2^{new}, v_{pub}, h_{sig}, h_1, h_2, k_1, k_2, timestamp)$;
Set $\boldsymbol{a} := (path_1, path_2, \mathbf{c}_1^{old}, \mathbf{c}_2^{old}, addr_{sk,1}^{old}, addr_{sk,2}^{old}, \mathbf{c}_1^{new}, \mathbf{c}_2^{new}, \mathtt{pkcm}_1^{new}, \mathtt{pkcm}_2^{new}, u_1^{old}, u_2^{old})$;
Compute $\pi_{\mathrm{POUR}} := \mathtt{Prove}(\mathtt{pk}_{\mathrm{POUR}}, \boldsymbol{x}, \boldsymbol{a})$;
Set $M := (\boldsymbol{x}, \pi_{\mathrm{POUR}}, info, \mathbf{C}_1, \mathbf{C}_2, pklist_1^{old}, pklist_2^{old})$;
Compute $\sigma := \mathcal{S}_{sig}(\mathtt{sk}_{sig}, M)$;
Set Ker $:= (\mathtt{cm}_1^{new}, \mathtt{cm}_2^{new}, \mathtt{sn}_1^{old}, \mathtt{sn}_2^{old})$;
for $i \in \{1,2\}$ **do**
  if *IsEmpty* $(pklist_i^{old})$ **then**
    Set $\sigma_i = \perp$;
  else
    Compute $\sigma_i = \mathcal{S}_{dst}(sk_i, \mathbf{Ker})$
  end
end
Set $tx_{\mathrm{Pour}} := (rt_1, rt_2, \mathtt{sn}_1^{old}, \mathtt{sn}_2^{old}, \mathtt{cm}_1^{new}, \mathtt{cm}_2^{new}, v_{pub}, info, *)$, where $* := (\mathtt{pk}_{sig}, h_1, h_2, \pi_{\mathrm{POUR}}, \mathbf{C}_1, \mathbf{C}_2, \sigma, \sigma_1, \sigma_2, pklist_1^{old}, pklist_2^{old}, k_1, k_2)$;
Output $\mathbf{c}_1^{new}, \mathbf{c}_2^{new}, tx_{\mathrm{Pour}}$;

We modify the original `Receive` algorithm to `Receive'` to additionally check if the public key commitment `pkcm` is one previously generated by `CreatePKCM` and never used before.

The details of the `Receive'` algorithm are presented in Alg.4.

The NP Statement. Finally, we modify the NP statement `POUR` to be proved by the zk-SNARK module to add a claim that the public key list *pklist* and the lock times *tlist* have been correctly committed, and that the lock times have run out. Following is the

Algorithm 3: Verify' Algorithm

if *tx is of type* tx_{Mint} **then**
  Parse tx_{Mint} as $(\mathtt{cm}, v, *)$ and $*$ as (k, t);
  Set $\mathtt{cm}' := \mathtt{COMM}_t(v \| k)$;
  Output $b := 1$ if $\mathtt{cm} = \mathtt{cm}'$, else output $b := 0$.
else
  Parse tx_{Pour} as (rt_1, rt_2, $\mathtt{sn}_1^{old}$, $\mathtt{sn}_2^{old}$, $\mathtt{cm}_1^{new}$, $\mathtt{cm}_2^{new}$, v_{pub}, $info$, $*$) and $*$ as ($\mathtt{pk}_{sig}$, h_1, h_2, π_{POUR}, $\mathbf{C}_1$, $\mathbf{C}_2$, σ, σ_1, σ_2, $pklist_1$, $pklist_2$, k_1, k_2);
  If $\mathtt{sn}_1^{old}$ or $\mathtt{sn}_2^{old}$ appears on L or $\mathtt{sn}_1^{old} = \mathtt{sn}_2^{old}$, output $b := 0$ and exit;
  If the Merkle tree root rt_1 or rt_2 does not appear on L, output $b := 0$ and exit;
  Compute $h_{sig} := \mathtt{CRH}(\mathtt{pk}_{sig})$;
  Compute $h_{pk,i}^{old} := \mathtt{Hash}(pklist_i)$ **for** $i \in \{1, 2\}$;
  Set $\boldsymbol{x} := (rt_1, rt_2, \mathtt{sn}_1^{old}, \mathtt{sn}_2^{old}, h_{pk,1}^{old}, h_{pk,2}^{old}, \mathtt{cm}_1^{new}, \mathtt{cm}_2^{new}, v_{pub}, h_{sig}, h_1, h_2, k_1, k_2, timestamp)$;
  Set $M := (\boldsymbol{x}, \pi_{\text{POUR}}, info, \mathbf{C}_1, \mathbf{C}_2, pklist_1^{old}, pklist_2^{old})$;
  If $\mathcal{V}_{sig}(\mathtt{pk}_{sig}, M, \sigma) = 0$ output $b := 0$ and exit;
  If $\mathtt{Verify}(\mathtt{vk}_{\text{POUR}}, \boldsymbol{x}, \pi_{\text{POUR}}) = 0$ output $b := 0$ and exit;
  Set Ker $:= (\mathtt{cm}_1^{new}, \mathtt{cm}_2^{new}, \mathtt{sn}_1^{old}, \mathtt{sn}_2^{old})$;
  for $i \in \{1, 2\}$ **do**
    If $\mathtt{NotEmpty}$ $(pklist_i)$ **and** $\mathcal{V}_{dst}(pklist_i[k_i], \mathbf{Ker}, \sigma_i) = 0$ **output** $b := 0$ **and exit**;
  end
  Output $b := 1$;
end

Algorithm 4: Receive' Algorithm

Parse addr_{pk} as $(a_{pk}, \mathtt{pk}_{enc})$, addr_{sk} as $(a_{sk}, \mathtt{sk}_{enc})$;
for *each Pour transaction* tx_{Pour} *on* L **do**
  Parse tx_{Pour} as (rt_1, rt_2, $\mathtt{sn}_1$, $\mathtt{sn}_2$, $\mathtt{cm}_1$, $\mathtt{cm}_2$, v_{pub}, $info$, $*$);
  for *each* $i \in \{1, 2\}$ **do**
    Compute $(v, \rho, r, s, t, tlist, \mathtt{pkcm}) := \mathcal{D}_{enc}(\mathtt{sk}_{enc}, \mathbf{C}_i)$;
    if $\mathcal{D}_{enc}$ *does not output* $\perp$ **then**
      Verify that $\mathtt{cm}_i = \mathtt{COMM}_t(v \| \mathtt{COMM}_s(H \| \mathtt{COMM}_r(a_{pk} \| \rho)))$
      where $H = \mathtt{CRH}(\mathtt{pkcm} \| tlist)$;
      Check that $\mathtt{pkcm}$ is in $\mathtt{PKCM}$ and never appears in other coins, if so, output
        $\mathbf{c} := (\text{addr}_{\text{pk}}, v, \rho, r, s, t, \mathtt{cm}_i, tlist, \mathtt{pkcm})$;
    end
  end
end

detail of the modified NP statement $\mathtt{POUR}$ for the zero-knowledge proof. Given

$$\boldsymbol{x} = (rt_1, rt_2, \mathtt{sn}_1^{old}, \mathtt{sn}_2^{old}, h_{pk,1}^{old}, h_{pk,2}^{old}, \mathtt{cm}_1^{new}, \mathtt{cm}_2^{new}, v_{pub}, h_{sig}, h_1, h_2, k_1, k_2, timestamp),$$

where $h_{pk,i}^{old} = \mathtt{Hash}(pklist_i^{old})$, for $i \in \{1, 2\}$, I know

$$\boldsymbol{a} = (path_1, path_2, \mathbf{c}_1^{old}, \mathbf{c}_2^{old}, addr_{sk,1}^{old}, addr_{sk,2}^{old}, \mathbf{c}_1^{new}, \mathbf{c}_2^{new}, \mathtt{pkcm}_1^{new}, \mathtt{pkcm}_2^{new}, u_1^{old}, u_2^{old}),$$

such that:

1. For each $i \in \{1, 2\}$:
   (a) The $path_i$ is a valid authentication path for leaf $\mathtt{cm}_i^{old}$ with respect to root rt_i, in a CRH-based Merkle tree.
   (b) The private key $addr_{sk,i}^{old}$ matches the public address of $addr_{pk,i}^{old}$.
   (c) The serial number $\mathtt{sn}_i^{old}$ is computed correctly, i.e. $\mathtt{sn}_i^{old} = \mathtt{PRF}^{sn}_{a_{sk,i}^{old}}(\rho_i^{old})$.
   (d) **The coin $\mathbf{c}_i^{old}$ is well formed, i.e.**
   $\mathtt{cm}_i^{old} = \mathtt{COMM}_{t_i^{old}}(v_i^{old} \| \mathtt{COMM}_{s_i^{old}}(H \| \mathtt{COMM}_{r_i^{old}}(a_{pk,i}^{old} \| \rho_i^{old})))$
   where $H = \mathtt{CRH}(\mathtt{COMM}_{u_i^{old}}(h_{pk,i}^{old}) \| tlist_i^{old})$.
   (e) **The coin $\mathbf{c}_i^{new}$ is well formed, i.e.**
   $\mathtt{cm}_i^{new} =$
   $\mathtt{COMM}_{t_i^{new}}(v_i^{new} \| \mathtt{COMM}_{s_i^{new}}(H \| \mathtt{COMM}_{r_i^{new}}(a_{pk,i}^{new} \| \rho_i^{new})))$
   where $H = \mathtt{CRH}(\mathtt{pkcm}_i^{new} \| tlist_i^{new})$.
   (f) The address secret key ties h_{sig} to h_i, i.e.
   $h_i = \mathtt{PRF}^{pk}_{a_{sk,i}^{old}}((i-1) \| h_{sig})$.
   (g) **The lock time is up, i.e.**
   $time(rt_i) + tlist_i^{old}[k_i] \leq timestamp$.
2. Balance is preserved: $v_1^{new} + v_2^{new} + v_{pub} = v_1^{old} + v_2^{old}$

3.3 Instantiation

To instantiate the new DAP' scheme, we need to consider the instantiation of the following components and procedures:

1. The public key list;
2. The lock time list;
3. The cryptographic algorithms:
   - The hash functions `CRH` and `Hash`.
   - The committing procedures $\mathtt{COMM}_s$, $\mathtt{COMM}_t$ and $\mathtt{COMM}_u$.
   - The pseudorandom functions.
   - The distributed key generation and distributed signature generation schemes.
   - The public signature scheme and public encryption scheme.

For simplicity, we leave the instantiation of pseudorandom functions, public signature scheme and public encryption scheme unmodified. We mention here that in Zerocash the public signature scheme takes ECDSA and the public encryption scheme is ECIES.

For instantiation of the distributed key generation and distribuetd signature generation sheme, we adopt the threshold Schnorr signature scheme using JF-DKG [8], and take the elliptic curve version of the schnorr signature scheme.

We follow the instantiation of Zerocash and take the compression function of SHA256 as `CRH`, which compresses 512 bits into 256 bits. And the flexible-input-length hash function `Hash` takes SHA256 directly.

The lock time list is fixed to 256 bits. We use 32 bits to store each lock time by seconds, allowing 8 lock times to be committed in the coin, which is sufficient in most

cases. The index indicating which public key and lock time to use starts from 0. The size of public key list, however, is unbounded. we encode this list by the Distinguished Encoding Rules (DER). For nine or more public keys, the corresponding lock time is default to zero for index larger than 7.

We instantiate $\texttt{COMM}_s$ by applying CRH twice on s, H and m: $\texttt{COMM}_s(H\|m) := \texttt{CRH}$ (CRH $(H \|s) \|m)$. $\texttt{COMM}_t$ is instantiated in the same way as the second commitment in Zerocash: $\texttt{COMM}_t(\ v\|k) := \texttt{CRH}(k \ \|0^{192} \ \|v)$. Note that t is actually ignored in this instantiation, because k already provides enough randomness. Finally $\texttt{COMM}_u$ is instantiated directly by: $\texttt{COMM}_u(h_{\texttt{pk}}) := \texttt{CRH}(h_{\texttt{pk}}\|u)$.

3.4 Completeness and Security

The completeness is defined similar to that in [20] by the experiment INCOMP. The security is similarly defined by *ledger indistinguishability*, *transaction non-malleability* and *balance*, which are defined by modifications of the experiments L-IND, TR-NM and BAL respectively.

Definition 1. *We say that a DAP' scheme Π = (Setup, CreateAddress, Mint', Pour', VerifyTransaction', Receive') is complete, if no polynomial-size adversary $\mathcal{A}$ wins INCOMP with more than negligible probability.*

Definition 2. *We say that a DAP' scheme Π = (Setup, CreateAddress, Mint', Pour', VerifyTransaction', Receive') is secure, if it is secure under experiment L-IND, TR-NM and BAL.*

In the INCOMP experiment, an adversary $\mathcal{A}$ sends the challenger $\mathcal{C}$ a ledger L and two coins $\mathbf{c}_1^{old}, \mathbf{c}_2^{old}$, and parameters needed to spend the coins. The challenger $\mathcal{C}$ tries to spend the two coins and gets a pour transaction tx_{Pour}. The adversary $\mathcal{A}$ wins if the L is a valid ledger, the parameters are valid with respect to L, the transaction tx_{Pour} is consistent to the parameters, but tx_{Pour} cannot be verified on the ledger. The completeness requires that $\mathcal{A}$ wins with negligible probability.

In the L-IND experiment, the challenger $\mathcal{C}$ samples a random bit b establishes two oracles $\mathcal{O}_0^{DAP}$ and $\mathcal{O}_1^{DAP}$, each of which maintains a DAP' scheme on a ledger L_0 and L_1 respectively. In each step the adversary is presented with the two ledgers L_b and L_{b-1} and issues a pair of queries (Q, Q') to the challenger, which will be forwarded to the oracles $\mathcal{O}_0^{DAP}$ and $\mathcal{O}_1^{DAP}$ respectively. The queries Q and Q' satisfy *public consistency* that they matches in type and reveals the same information to $\mathcal{A}$. Finally, $\mathcal{A}$ outputs a guess b' and wins when $b' = b$. The ledger indistinguishability requires that the advantage of $\mathcal{A}$ is negligible.

In the TR-NM experiment, $\mathcal{A}$ interacts with one DAP' scheme oracle and then outputs a pour transaction tx'_{Pour}, and wins if there is a pour transaction $tx_{\text{Pour}} \neq tx'_{\text{Pour}}$ on the ledger such that tx_{Pour} reveals the same serial number of tx'_{Pour} and that if tx'_{Pour} takes the place of tx_{Pour} the ledger is still valid. The transaction non-malleability requires that $\mathcal{A}$ wins with negligible probability.

In the BAL experiment, $\mathcal{A}$ interacts with one DAP' scheme oracle and wins the game if the total value he can spend or has spent is greater than the value he has minted or received. The balance requires that $\mathcal{A}$ wins with negligible probability.

For the definitions of security under the above experiments, refer to Appendix A. Regarding the experiments L-IND, TR-NM and BAL, we design them similarly to those in [20], and the major modifications are listed below.

Modifications to the experiments L-IND, TR-NM and BAL.

1. Assume that $\mathcal{O}^{DAP}$ maintains three tables PKCM, OLDPKCM and PK. We add a new kind of query CreatePKCM as follows:
   - $Q = (\textbf{CreatePKCM}, \text{addr}_{\text{pk}}, K)$
     (a) Invoke CreatePKCM (pp,K) to obtain the tuple $(pklist, u, \texttt{pkcm})$ and a set of K key pairs.
     (b) Store $(pklist, u, \texttt{pkcm})$ in table PKCM.
     (c) Store the K keypairs in table.
     (d) Output pkcm.
2. We modify the queries Mint, Pour as follows:
   - For each $addr_{pk,i}^{old}$, the adversary provides an index k_i to indicate which public key and lock time to use to unlock the coin.
   - The index k_i in Q and Q' must be the same for each input coin, and the selected lock time must be less than current time.
   - The number of public keys committed in $\texttt{pkcm}_1$ and $\texttt{pkcm}_2$ must be the same.
   - For addr_{pk} in Mint query or each $addr_{pk,i}^{new}$ in Pour query, the adversary provides a public key commitment $\texttt{pkcm}_i^{new}$ and a lock time list $tlist_i^{new}$.
   - If the address is in ADDR, $\mathcal{O}^{DAP}$ checks that $\texttt{pkcm}_i^{new}$ is in PKCM and not in OLDPKCM, and aborts if the check fails.
   - If the address is not in ADDR, $\mathcal{O}^{DAP}$ checks that $\texttt{pkcm}_i^{new}$ is not in either PKCM or OLDPKCM, and aborts if the check fails.
   - If the Mint or Pour query is successful, $\mathcal{O}^{DAP}$ removes all $\texttt{pkcm}^{new}$ mentioned from PKCM and stores the tuple $(\text{addr}_{\text{pk}}, \texttt{pkcm}, u, pklist)$ in OLDPKCM.
   - For Pour query, $\mathcal{O}$ looks up the table OLDPKCM to find the tuple ($addr_{pk,i}^{old}$, $\texttt{pkcm}_i^{old}$, u_i^{old}, $pklist_i^{old}$) for each $addr_{pk,i}^{old}$, include $pklist_i^{old}$ in the pour transaction tx_{Pour}. Then $\mathcal{O}$ checks that $tlist_i^{old}[k_i]$ is less than current time, aborts if check fails. Then $\mathcal{O}$ signs the transaction with the corresponding secret key of $pklist_i^{old}[k_i]$ (looked up from PK) and include the signature in tx_{Pour}.
3. We remove the Receive query in the original definition of $\mathcal{O}^{DAP}$ for the following reasons:
   - The Receive query does not model a proper attacking scenario in real life. In fact, this query allows the adversary to identify the coins belonging to an address for which the adversary does not hold the secret key, which is *unreasonable* in real life.

– The `Receive` query compromises the ledger indistinguishability. We devise the following attack to the `L-IND` game making use of the information provided by `Receive` query. First, the adversary $\mathcal{A}$ issues two pairs of `CreateAddress` queries to receive two address public keys, for simplicity we denote the two addresses by Alice and Bob respectively. Then, $\mathcal{A}$ issues a pair of `Mint` queries to generate a coin for Alice in both ledgers. Next, $\mathcal{A}$ issues a pair of `Pour` queries (Q, Q') to the challenger. In Q $\mathcal{A}$ specifies that Alice pays her coin to Bob, while in Q' Alice pays the coin to herself. Finally, $\mathcal{A}$ issues a pair of `Receive` queries on Alice, and obtains the lists of coin commitments for the ledgers respectively. The oracle that returns an empty commitment list is the one maintaining ledger L_0. Thus $\mathcal{A}$ wins `L-IND` game with 100 percent probability.

4. We modify the `Insert` query as follows:
   – For each output coin, check that the `pkcm` in the coin is stored in `PKCM`, abort if not so; remove the corresponding tuple from `PKCM` and add to `OLDPKCM`.

The following theorem claims that our construction of DAP' scheme is complete and secure under the above definitions.

Theorem 1. *The tuple (`Setup`, `CreateAddress`, `Mint'`, `Pour'`, `VerifyTransaction'`, `Receive'`) is a complete and secure DAP' scheme.*

The proof is similar to that of Theorem 4.1 in [20]. Here we only present the modifications to the original one. For the complete proof refer to Appendix B.

1. **Modify the simulation experiment.** The simulated experiment $\Im_{\mathbf{sim}}$ proceed as in [20], except for the following modification:
   (a) **Answering `CreatePKCM` queries.** To answer Q, $\mathcal{C}$ behaves as in `L-IND`, except for the following modification: after obtaining $(pklist, u, \mathtt{pkcm})$, $\mathcal{C}$ replaces `pkcm` with a random string of the appropriate length; then, $\mathcal{C}$ stores the tuple in `PKCM` and returns `pkcm` to $\mathcal{A}$. Afterwards, $\mathcal{C}$ does the same for Q'.
   (b) **Answering `Mint` queries.** Compute $k = \mathtt{COMM}_s(\tau)$ for a random string τ of the suitable length, instead of $k = \mathtt{COMM}_s(H\|m)$. Afterwards, $\mathcal{C}$ does the same for Q'.

   Remark 1. There is no need to modify the `Pour` queries except for the modifications mentioned in [20], which already discard the information of `pkcm` and *tlist* in the commitment $\mathtt{cm}_i^{new}$ and ciphertext $\mathbf{C}_i^{new}$. For each $addr_{pk,i}^{old}$ the simulated oracle puts the original *pklist* looked up from `OLDPKCM` in tx_{Pour}. It makes no difference to replace it by a newly generated one, since the one stored in the table is independent from the randomly string replacing $\mathtt{pkcm}_i^{old}$.

2. **Difference between $\Im_{\mathbf{sim}}$ and hybrid experiment $\Im_3$.** Let $q_{\mathbf{CP}}$ be the total number of `CreatePKCM` queries issued by $\mathcal{A}$. In addition to those described in [20] appendix D.1, we additionally let the experiment $\Im_{\mathbf{sim}}$ modifies $\Im_3$ in the following ways:
   – Each time $\mathcal{A}$ issues a `CreatePKCM` query, the commitment `pkcm` is substituted with a random string of suitable length.

– Each time $\mathcal{A}$ issues a `Mint` query, the commitment k in tx_{Mint} is substituted with a commitment to a random input.

Then we modify the Lemma D.3 in [20] appendix D.1 as follows:

$$\left|\mathbf{Adv}^{\mathrm{sim}} - \mathbf{Adv}^{\mathrm{3}}\right| \leq (q_{\mathbf{M}} + 4 \cdot q_{\mathbf{P}} + q_{\mathbf{CP}}) \cdot \mathbf{Adv}^{\mathtt{COMM}}$$

3.5 Efficiency Analysis of DAP' Scheme

The time of signature verification is negligible compared with that of the zero-knowledge proof. Moreover, the NP statement `POUR` in our scheme is similar to that in Zerocash, and by the succinctness of the zk-SNARK, it is reasonable to estimate that the generation and verification time for the zero-knowledge proof is comparable to that of Zerocash. Therefore, our new scheme is almost as efficient as the orginal DAP scheme.

4 Z-Channel

With above mechanisms, we can now implement a micropayment system in Zerocash, which is a transplant of the idea of lightning network [17] to Zerocash. This will significantly improve the scalability and the instant payment capability of Zerocash, allowing numberous payments conducted and confirmed off-chain in short periods of time.

The algorithms mentioned in this section are executed together by parties P_1 and P_2. Before we present the definition of the algorithms, we clarify some concepts and denotions used in the algorithms.

– **Signing on a piece of information**: for simplicity, we use the verb "sign" to represent both the operation of generating the signature piece or the entire signature, when the context is clear.
– **Shared address** $(\mathrm{addr}_{\mathrm{pk}}, \mathrm{addr}_{\mathrm{sk}})$: during the lifetime of a Z-Channel, all the coins involved in the channel are targeted at one shared address $\mathrm{addr}_{\mathrm{pk}}$. The secret key $\mathrm{addr}_{\mathrm{sk}}$ and all the "secret" trapdoors are shared between the parties, except the private keys (or shares) of the distributed signature scheme.
– **Kernel of a `Pour` transaction**: a structure indicating the inputs and outputs of a pour transaction. Recall that in the previous section we define the kernel of a pour transaction as the tuple $(\mathtt{cm}_1^{new}, \mathtt{cm}_2^{new}, \mathtt{sn}_1^{old}, \mathtt{sn}_2^{old})$. For simplicity and clarity we denote such kernel by $\mathbf{Ker}(\mathbf{c}_1^{old}, \mathbf{c}_2^{old}) \to (\mathbf{c}_1^{new}, \mathbf{c}_2^{new})$, and if any of the coins is empty (exists as a placeholder), we directly omit it in the denotion. Since all trapdoors are public between parties, any party holding a pour kernel signed by the other party can easily generate the full `Pour` transaction. For simplicity we express this procedure by saying *extending* a pour kernel.
– Similarly, denote by $\mathtt{Pour}(\mathbf{c}_1^{old}, \mathbf{c}_2^{old}) \to (\mathbf{c}_1^{new}, \mathbf{c}_2^{new})$ a pour transaction consuming $\mathbf{c}_1^{old}$ and $\mathbf{c}_2^{old}$ and outputing $\mathbf{c}_1^{new}$ and $\mathbf{c}_2^{new}$.

4.1 Establish Z-Channel

The algorithm `EstablishChannel` establishes a Z-Channel between the two parties P_1 and P_2 executing this algorithm.

- Initially, two parties P_1 and P_2 wish to establish a micropayment channel, and agree to put v_1 and v_2 value of Zerocash into the channel respectively.
- After the algorithm is finished,
  - two *funding transactions* $tx_{\text{fund,1}}$ and $tx_{\text{fund,2}}$ are published on ledger, outputing coins $\mathbf{c}_{\text{fund,1}}$ and $\mathbf{c}_{\text{fund,2}}$ respectively
  - a *share transaction* $tx_{\text{Pour,share}} = \texttt{Pour}(\mathbf{c}_{\text{fund,1}}, \mathbf{c}_{\text{fund,2}}) \rightarrow (\mathbf{c}_{\text{share}})$
  - for $i \in \{1, 2\}$ (let $j = 3 - i$), party P_i holds a *terminating pour kernel* $\mathbf{Ker}(\mathbf{c}_{\text{share}}) \rightarrow (\mathbf{c}_{\text{redeem,i}}, \mathbf{c}_{\text{redeem,j}})$ signed by P_j

We present the details of `EstablishChannel` as follows.

- Let T be the penalty time.

1. The parties agree on a shared address addr_{pk} and addr_{sk}; then they run distributed $\mathcal{K}_{enc}$ algorithm to generate shared public keys $\texttt{pk}_{1,2}^{share}$, $\texttt{pk}_{1,2}^{term}$ and corresponding private keys $\texttt{sk}_{1,2}^{share}$, $\texttt{sk}_{1,2}^{term}$; after that, each party P_i invokes $\mathcal{K}_{enc}$ privately to generate public/private key pairs $\texttt{pk}_i^{fund}$, $\texttt{pk}_i^{term}$ and $\texttt{sk}_i^{fund}$, $\texttt{pk}_i^{term}$.
2. Each party P_i invokes `Mint'` or `Pour'` to generate the fund coin $\mathbf{c}_{\text{fund,i}}$ and the funding transaction $tx_{\text{fund},i}$, where the *pklist* is $(\texttt{pk}_i^{fund})$, and *tlist* is (0).
3. Each party P_i publishes $tx_{\text{fund},i}$ on the ledger, and sends $\mathbf{c}_{\text{fund,i}}$ to P_j; on receiving $\mathbf{c}_j$ from the P_j, checks that $\mathbf{c}_j$ is a valid coin, the $pklist, tlist, a_{pk}, v_j$ are correctly committed in it, and $tx_{\text{fund},j}$ publishes the same commitment as $\mathbf{c}_{\text{fund,j}}$; if any of the checks fails, redeems $\mathbf{c}_{\text{fund,i}}$ immediately and aborts.
4. If both $tx_{\text{fund,1}}$ and $tx_{\text{fund,2}}$ are correctly published, they agree on a pour kernel $\mathbf{Ker}(\mathbf{c}_{\text{fund,i}}, \mathbf{c}_{\text{fund,j}}) \rightarrow (\mathbf{c}_{\text{share}})$ with $pklist = (\texttt{pk}_{1,2}^{share})$ and $tlist = (0)$ in $\mathbf{c}_{\text{share}}$.
5. Each party P_i generates and signs (with share of $\texttt{sk}_{1,2}^{share}$) a pour kernel $\mathbf{Ker}(\mathbf{c}_{\text{share}}) \rightarrow (\mathbf{c}_{\text{redeem,i}}, \mathbf{c}_{\text{redeem,j}})$ with $pklist_i = (\texttt{pk}_i^{term})$, $tlist_i = (0)$ for $\mathbf{c}_{\text{redeem,i}}$, $pklist_j = (\texttt{pk}_j^{term}, \texttt{pk}_{1,2}^{term})$, $tlist_j = (T, 0)$ for $\mathbf{c}_{\text{redeem,j}}$ and the values of the coins are v_i and v_j respectively, and sends the kernel with the signature piece to P_j, along with the trapdoors used to generate the commitment; on receiving the pour kernel from P_j, checks that the kernel has been generated as expected and the signature piece is valid, if not, redeems the fund coin and aborts immediately.
6. Each party P_i signs $\mathbf{Ker}(\mathbf{c}_{\text{fund,i}}, \mathbf{c}_{\text{fund,j}}) \rightarrow (\mathbf{c}_{\text{share}})$ and sends P_j the signature; on receiving the signature piece from the other, one of the parties extends the kernel to pour transaction $tx_{\text{Pour,share}}$ and publishes it on the ledger.

4.2 Make Payments in Z-Channel

The algorithm `Pay` updates the terminating pour kernels to redistribute the redeem values.

– At the beginning of this algorithm,
  - a *share transaction* $tx_{\text{Pour,share}} = \texttt{Pour}(\mathbf{c}_{\text{fund,1}}, \mathbf{c}_{\text{fund,2}}) \to (\mathbf{c}_{\text{share}})$
  - for $i \in \{1, 2\}$ party P_i holds a terminating pour kernel $\mathbf{Ker}(\mathbf{c}_{\text{share}}) \to (\mathbf{c}^{old}_{\text{redeem,i}}, \mathbf{c}^{old}_{\text{redeem,j}})$ signed by P_j
– After the algorithm is finished,
  - for $i \in \{1, 2\}$, party P_i holds an updated terminating pour kernel $\mathbf{Ker}(\mathbf{c}_{\text{share}}) \to (\mathbf{c}^{new}_{\text{redeem,i}}, \mathbf{c}^{new}_{\text{redeem,j}})$ signed by P_j
  - for $i \in \{1, 2\}$, party P_i holds a *penalty pour kernel* $\mathbf{Ker}(\mathbf{c}^{old}_{\text{redeem,j}}) \to (\mathbf{c}_i)$ signed by P_j

Each time one party tries to pay another, they agree on a new distribution of the shared coin, v_i^{new} and v_j^{new}. Then they try to generate new terminating pour kernel with the updated distribution. After that, each party signs a pernalty pour kernel for another to prevent any of them from trying to publish outdated terminating pour kernel. The detail is as follows.

– The two parties agrees on the update values v_i^{new}, v_j^{new}.

1. The parties run distrbuted $\mathcal{K}_{enc}$ algorithm to generate shared public/private key pair $\texttt{pk}_{1,2}^{term,new}$ and $\texttt{sk}_{1,2}^{term,new}$; then each party P_i invokes $\mathcal{K}_{enc}$ privately to generate public/private key pairs $\texttt{pk}_i^{term,new}$, $\texttt{pk}_i^{pen}$ and $\texttt{sk}_i^{term,new}$, $\texttt{sk}_i^{pen}$.
2. Each party P_i generates and signs (with share of $\texttt{sk}_{1,2}^{share}$) a pour kernel $\mathbf{Ker}(\mathbf{c}_{\text{share}}) \to (\mathbf{c}^{new}_{\text{redeem,i}}, \mathbf{c}^{new}_{\text{redeem,j}})$ with $pklist_i = (\texttt{pk}_i^{term})$, $tlist_i = (0)$ for $\mathbf{c}^{new}_{\text{redeem,i}}$, $pklist_j = (\texttt{pk}_j^{term}, \texttt{pk}_{1,2}^{term})$, $tlist_j = (T, 0)$ for $\mathbf{c}^{new}_{\text{redeem,j}}$ and the values of the coins are v_i^{new} and v_j^{new} respectively, and sends the kernel with the signature piece to P_j, along with the trapdoors used to generate the commitment; on receiving the pour kernel from P_j, checks that the kernel has been generated as expected and the signature piece is valid, if not, terminates the channel (by extending $\mathbf{Ker}(\mathbf{c}_{\text{share}}) \to (\mathbf{c}^{old}_{\text{redeem,i}}, \mathbf{c}^{old}_{\text{redeem,j}})$ and publishing the result pour transaction) and aborts immediately.
3. Each party P_i generates and signs (with share of $\texttt{sk}_{1,2}^{term}$) a pour kernel $\mathbf{Ker}(\mathbf{c}_{\text{redeem,i}}) \to (\mathbf{c}_j)$ with $pklist = (\texttt{pk}_j^{term})$ and $tlist = (0)$ for $\mathbf{c}_j$, and sends the kernel with the signature piece to P_j; on receiving the pour kernel from P_j, checks that the kernel has been generated as expected and the signature piece is valid, if not, terminates the channel and aborts immediately.

4.3 Terminate Z-Channel

The algorithm `Terminate` terminates the channel by publishing one terminating transaction for this channel on ledger.

– At the beginning of this algorithm,
  - a *share transaction* $tx_{\text{Pour,share}} = \texttt{Pour}(\mathbf{c}_{\text{fund,1}}, \mathbf{c}_{\text{fund,2}}) \to (\mathbf{c}_{\text{share}})$
  - for $i \in \{1, 2\}$ party P_i holds a terminating pour kernel $\mathbf{Ker}(\mathbf{c}_{\text{share}}) \to (\mathbf{c}_{\text{redeem,i}}, \mathbf{c}_{\text{redeem,j}})$ signed by P_j

– After this algorithm is finished,
  - a *terminating transaction* $tx_{\text{Pour,term}} = \texttt{Pour}(\mathbf{c}_{\text{share}}) \rightarrow (\mathbf{c}_{\text{redeem,1}}, \mathbf{c}_{\text{redeem,2}})$ is published on ledger
  - two pour transactions are published on ledger consuming $\mathbf{c}_{\text{redeem,1}}$ and $\mathbf{c}_{\text{redeem,2}}$

After terminating transaction is published, party P_1 and P_2 can redeem the Zerocash from $\mathbf{c}_{\text{redeem,1}}$ and $\mathbf{c}_{\text{redeem,2}}$ respectively, by another pour transaction to their own addresses. The one who takes the action to terminate the channel, however, has to wait time T before redeeming the coin. If the terminating transaction is an outdated one, the other party has time T to extend and publish the penalty transaction to take away both of the redeem coins. The detail of `Terminate` is as follows.

– One of the parties (assume P_i) decides to terminate the channel.

1. P_i extends $\mathbf{Ker}(\mathbf{c}_{\text{share}}) \rightarrow (\mathbf{c}_{\text{redeem,i}}, \mathbf{c}_{\text{redeem,j}})$ to terminating transaction $tx_{\text{Pour,term}}$ and publishes it.
2. P_j invokes `Pour'` algorithm to pour $\mathbf{c}_{\text{redeem,j}}$ to his own account.
3. P_j checks if $tx_{\text{Pour,term}}$ is the most updated, if not, P_j extends and publishes the penalty kernel $\mathbf{Ker}(\mathbf{c}_{\text{redeem,i}}) \rightarrow (\mathbf{c}_j)$.
4. If the check succeeds, and P_j agrees to help P_i to redeem his coin:
   (a) P_i sends $\mathbf{Ker}(\mathbf{c}_{\text{redeem,i}}) \rightarrow (\mathbf{c}_i)$ to P_j.
   (b) P_j signs this kernel by his share of $\mathtt{sk}_{1,2}^{term}$ corresponding to this terminating transaction, and sends the signature back to P_i.
   (c) P_i extends and publishes the kernel.
5. If the check succeeds, and P_j disagree to help P_i:
   (a) P_i waits for time T.
   (b) P_i invokes `Pour'` algorithm to pour $\mathbf{c}_{\text{redeem,i}}$ to his own account.

5 Conclusion and Future Work

Our work aimed at improving the scalability and efficiency of Zerocash while maintaining the privacy. We modified the original DAP scheme for Zerocash to support more features which are essential in implementing micropayment channel. Our modification enabled Zerocash to support shared address by a group of parties. Specifically, a user can generate a coin which can be spent only by cooperation of all the group members, while in the view of others the coin and the transaction spending it look the same as normal transactions issued by a single user. Our new scheme also made it possible to lock a coin by a specific period of time. The verifiers can check whether the input coin of a transaction is unlocked or not, while information about the timestamp of the input coin and lock time are well protected. Moreover, a coin can be locked with different lock time for different parties, this feature permits various kinds of applications.

We then transplanted the idea of Lightning network to develop a micropayment channel Z-Channel. Compared with the micropayment channel in Lightning network, the identities of the parties and the amount of coins involved in the channels are kept secret.

Furthermore, others cannot perceive even the existence of the channel by observing the ledger. Compared with the original Zerocash, Z-Channel significantly improves the scalability and the instant payment capability. As a matter of fact, Z-Channel allows numberous payments conducted and confirmed off-chain in short periods of time.

It is intriguing (though challenging) to transplant the idea of Hashed Timelock Contract (HTLC) in Lightning network to Z-Channel to form a more scalable network. Another direction worth considering is to connect the public keys by logic relations AND and OR in our modified Zerocash scheme. The applications and security of such schemes remain to be explored and analyzed.

A Definition of Security

We define the completeness, ledger indistinguishability, transaction non-malleability and balance in a way similar to definitions B.1, C.1 C.2 and C.3 in [20].

Definition 3. *We say that a DAP' scheme* Π *= (*`Setup`*,* `CreateAddress`*,* `Mint'`*,* `Pour'`*,* `VerifyTransaction'`*,* `Receive'`*) is complete, if for every poly(*λ*)-size adversary* $\mathcal{A}$ *and sufficiently large* λ*,* $\mathbf{Adv}^{INCOMP}_{\Pi,\mathcal{A}}(\lambda) < negl(\lambda)$*, where* $\mathbf{Adv}^{INCOMP}_{\Pi,\mathcal{A}}(\lambda) := 2 \cdot Pr[$`INCOMP` $(\Pi, \mathcal{A}, \lambda) = 1] - 1$ *is* $\mathcal{A}$*'s advantage in the* `INCOMP` *experiment.*

Definition 4. *We say that a DAP' scheme* Π *= (*`Setup`*,* `CreateAddress`*,* `Mint'`*,* `Pour'`*,* `VerifyTransaction'`*,* `Receive'`*) is* `L-IND` *secure, if for every poly(*λ*)-size adversary* $\mathcal{A}$ *and sufficiently large* λ*,* $\mathbf{Adv}^{L\text{-}IND}_{\Pi,\mathcal{A}}(\lambda) < negl(\lambda)$*, where* $\mathbf{Adv}^{L\text{-}IND}_{\Pi,\mathcal{A}}(\lambda) := 2 \cdot Pr[$`L-IND`$($ $\Pi, \mathcal{A}, \lambda) = 1] - 1$ *is* $\mathcal{A}$*'s advantage in the* `L-IND` *experiment.*

Definition 5. *We say that a DAP' scheme* Π *= (*`Setup`*,* `CreateAddress`*,* `Mint'`*,* `Pour'`*,* `VerifyTransaction'`*,* `Receive'`*) is* `TR-NM` *secure, if for every poly(*λ*)-size adversary* $\mathcal{A}$ *and sufficiently large* λ*,* $\mathbf{Adv}^{TR\text{-}NM}_{\Pi,\mathcal{A}}(\lambda) < negl(\lambda)$*, where* $\mathbf{Adv}^{TR\text{-}NM}_{\Pi,\mathcal{A}}(\lambda) := 2 \cdot Pr[$`TR-NM` $(\Pi, \mathcal{A}, \lambda) = 1] - 1$ *is* $\mathcal{A}$*'s advantage in the* `TR-NM` *experiment.*

Definition 6. *We say that a DAP' scheme* Π *= (*`Setup`*,* `CreateAddress`*,* `Mint'`*,* `Pour'`*,* `VerifyTransaction'`*,* `Receive'`*) is* `BAL` *secure, if for every poly(*λ*)-size adversary* $\mathcal{A}$ *and sufficiently large* λ*,* $\mathbf{Adv}^{BAL}_{\Pi,\mathcal{A}}(\lambda) < negl(\lambda)$*, where* $\mathbf{Adv}^{BAL}_{\Pi,\mathcal{A}}(\lambda) := 2 \cdot Pr[$`BAL` $(\Pi, \mathcal{A}, \lambda) = 1] - 1$ *is* $\mathcal{A}$*'s advantage in the* `BAL` *experiment.*

In each of the experiments, one or more oracles of the DAP scheme $\mathcal{O}^{DAP}$ receives queries and output answers. A challenger $\mathcal{C}$ interacts with an adversary $\mathcal{A}$, forwards the queries from $\mathcal{A}$ to $\mathcal{O}^{DAP}$ and the answers back to $\mathcal{A}$, and performs sanity checks. We modify the mechanism of the original $\mathcal{O}^{DAP}$ in [20] to suit our new DAP' scheme. Below, we first describe how this new oracle $\mathcal{O}^{DAP}$ works.

The oracle $\mathcal{O}^{DAP}$ is initialized by a list of public parameters `pp` and maintains state. Internally, $\mathcal{O}^{DAP}$ stores the following:

(i) L, a ledger;
(ii) `ADDR`, a set of address key pairs;
(iii) `COIN`, a set of coins;
(iv) `PK`, a set of public/private key pairs;
(v) `PKCM`, a set of tuples of $(pklist, u, \texttt{pkcm})$;
(vi) `OLDPKCM`, a set of tuples of $(\text{addr}_{\text{pk}}, \texttt{pkcm}, u, pklist)$.

Initially, L, `ADDR`, `COIN`, `PK`, `PKCM`, `OLDPKCM` start out empty. The oracle $\mathcal{O}^{DAP}$ accepts various types of queries, and each type of query modifes L, `ADDR`, `COIN`, `PK`, `PKCM`, `OLDPKCM` in different ways and outputs differently. We now describe each type of query Q.

– $Q =$ (`CreateAddress`)
  1. Compute $(\text{addr}_{\text{pk}}, \text{addr}_{\text{sk}}) :=$ `CreateAddress` (`pp`).

2. Add the address key pair (addr_{pk}, addr_{sk}) to `ADDR`.
3. Output the address public key addr_{pk}.

Other internal storages apart from `ADDR` stay unchanged.

- Q = (`CreatePKCM`,addr_{pk},K)
  1. Randomly sample u.
  2. Randomly sample a public key list $pklist$ (with secret key list being $sklist$) of size K.
  3. Compute $\texttt{pkcm} = \texttt{COMM}_u(\texttt{Hash}(pklist))$.
  4. Store $(pklist, u, \texttt{pkcm})$ in table `PKCM`.
  5. Store $(pklist[i], sklist[i])$ in table `PK` for $i \in \{1, ..., K\}$.
  6. Output `pkcm`.

  Other internal storages apart from `PK`, `PKCM` stay unchanged.
- Q = (`Mint`, v, addr_{pk}, `pkcm`, $tlist$)
  1. Compute ($\mathbf{c}$, tx_{Mint}) := `Mint` (`pp`, v, addr_{pk}, $tlist$).
  2. Add the coin $\mathbf{c}$ to `COIN`.
  3. If addr_{pk} is in `ADDR`, find tuple $(pklist, u, \texttt{pkcm})$ in table `PKCM`, aborts if cannot find, then removes the tuple from `PKCM` and stores $(\text{addr}_{\text{pk}}, \texttt{pkcm}, u, pklist)$ in `OLDPKCM`;
  4. If addr_{pk} is not in `ADDR`, but `pkcm` can be found in `PKCM` or `OLDPKCM`, aborts;
  5. Add the mint transaction tx_{Mint} to L.
  6. Output $\perp$.

  The internal storage `ADDR` stay unchanged.
- Q = (`Pour`, idx_1^{old} , idx_2^{old} , $addr_{pk,1}^{old}$, $addr_{pk,2}^{old}$, k_1, k_2, `info`, v_1^{new} , v_2^{new} , $addr_{pk,1}^{new}$, $addr_{pk,2}^{new}$, $\texttt{pkcm}_1^{new}$, $\texttt{pkcm}_2^{new}$, $tlist_1^{new}$, $tlist_2^{new}$, v_{pub})
  1. Let $timestamp$ be the current time.
  2. For each $i \in \{1, 2\}$:
     (a) Let $\texttt{cm}_i^{old}$ be the idx_i^{old}-th coin commitment in L.
     (b) Let tx_i be the mint/pour transaction in L that contains $\texttt{cm}_i^{old}$.
     (c) Let $\mathbf{c}_i^{old}$ be the first coin in `COIN` with coin commitment $\texttt{cm}_i^{old}$.
     (d) Let $\texttt{pkcm}_i^{old}$ be the public key commitment stored in $\mathbf{c}_i^{old}$.
     (e) Let $(pklist_i^{old}, u_i^{old}, \texttt{pkcm}_i^{old})$ be the first tuple in `OLDPKCM` with public key commitment $\texttt{pkcm}_i^{old}$.
     (f) Let ($addr_{pk,i}^{old}$,$addr_{sk,i}^{old}$) be the first key pair in `ADDR` with $addr_{pk,i}^{old}$ being $\mathbf{c}_i^{old}$'s address.
     (g) Let ($pklist[k_i]$,sk_i) be the first key pair in `PK` with $pklist[k_i]$ being the public key.
     (h) Let $tlist_i$ be the lock time stored in $\mathbf{c}_i^{old}$.
     (i) Let rt_i be the a randomly selected root in the Merkle tree root history later than $\texttt{cm}_i^{old}$ in L such that $rt_i.time + tlist_i[k_i] < timestamp$.
     (j) Compute $path_i$, the authentication path from $\texttt{cm}_i^{old}$ to rt_i.
     (k) If $addr_{pk,i}^{new}$ is in `ADDR`, checks that $\texttt{pkcm}_i^{new}$ is in `PKCM` and not in `OLDPKCM`, and aborts if the check fails. Let $(pklist_i^{new}, u_i^{new}, \texttt{pkcm}_i^{new})$ be the tuple found in `PKCM`. Remove $\texttt{pkcm}_i^{new}$ from `PKCM` and stores ($addr_{pk,i}^{new}$, $\texttt{pkcm}_i^{new}$, u_i^{new}, $pklist_i^{new}$) in `OLDPKCM`.

(l) If $addr_{pk,i}^{new}$ is not in ADDR, checks that $\mathtt{pkcm}_i^{new}$ is not in either PKCM or OLDPKCM, and aborts if the check fails.

3. Compute $(\mathbf{c}_1^{new}, \mathbf{c}_2^{new}, tx_{\text{Pour}}) :=$ Pour (pp, rt_1, rt_2, $\mathbf{c}_1^{old}$, $\mathbf{c}_2^{old}$, $addr_{sk,1}^{old}$, $addr_{sk,2}^{old}$, $path_1$, $path_2$, $pklist_1^{old}$, $pklist_2^{old}$, u_1^{old}, u_2^{old}, $tlist_1^{old}$, $tlist_2^{old}$, k_1, k_2, sk_1, sk_2, v_1^{new}, v_2^{new}, $addr_{pk,1}^{new}$, $addr_{pk,2}^{new}$, v_{pub}, $\mathtt{pkcm}_1^{new}$, $\mathtt{pkcm}_2^{new}$, $tlist_1^{new}$, $tlist_2^{new}$, info).
4. Verify that VerifyTransaction (pp, tx_{Pour}, L) outputs 1.
5. Add the coin $\mathbf{c}_1^{new}$ to COIN.
6. Add the coin $\mathbf{c}_2^{new}$ to COIN.
7. Add the pour transaction tx_{Pour} to L.
8. Output $\perp$.

If any of the above operations fail, the output is $\perp$ (and L, ADDR, COIN, PK, PKCM, OLDPKCM remain unchanged).

– $Q =$ (Insert, tx)

1. Verify that VerifyTransaction (pp, tx, L) outputs 1. (Else, abort.)
2. Add the mint/pour transaction tx to L.
3. Run Receive' for all addresses addr_{pk} in ADDR;
4. For each output coin from Receive'
   (a) Let pkcm be the public key commitment stored in it.
   (b) Let $(pklist, u, \mathtt{pkcm})$ be the first tuple in PKCM with the public key commitment pkcm (if not exists, aborts).
   (c) Remove this tuple from PKCM;
   (d) Add $(\text{addr}_{\text{pk}}, \mathtt{pkcm}, u, pklist)$ to OLDPKCM.
5. Output $\perp$.

The address set ADDR stays unchanged.

With the above described oracle $\mathcal{O}^{DAP}$, the definitions of ledger indistinguishability, transaction non-malleability and balance are defined by three games respectively: L-IND, TR-NM and BAL. We now describe the above mentioned L-IND experiment. The other experiments TR-NM and BAL are similar to the original ones, refer to [20] for the details.

Given a DAP' scheme Π, adversary $\mathcal{A}$, and security parameter λ, the (probabilistic) experiment L-IND $(\Pi, \mathcal{A}, \lambda)$ consists of a series of interactions between $\mathcal{A}$ and a challenger $\mathcal{C}$. At the end of this experiment, $\mathcal{C}$ outputs a bit in $\{0, 1\}$ indicating whether $\mathcal{A}$ succeeds.

At the start of the experiment, $\mathcal{C}$ samples $b \in \{0, 1\}$ at random, samples pp $\leftarrow$Setup (1^λ), and sends pp to $\mathcal{A}$; using pp, $\mathcal{C}$ initializes two DAP' oracles $\mathcal{O}_0^{DAP}$ and $\mathcal{O}_1^{DAP}$.

Now $\mathcal{A}$ and $\mathcal{C}$ start interaction in steps. In each step, $\mathcal{C}$ provides to $\mathcal{A}$ two ledgers (L_{Left}, L_{Right}), where $L_{Left} := L_b$ is the current ledger in $\mathcal{O}_b^{DAP}$ and $L_{Right} := L_{1-b}$ the ledger in $\mathcal{O}_{1-b}^{DAP}$; then $\mathcal{A}$ sends to $\mathcal{C}$ a pair of queries (Q, Q'), which must be of the same type of query. $\mathcal{C}$ acts differently on differnt types of queries, as follows:

– If the query is of type Insert, $\mathcal{C}$ forwards Q to $\mathcal{O}_b^{DAP}$, and Q' to $\mathcal{O}_{1-b}^{DAP}$. If the inserted query is a Pour query with one of the target address addr_{pk} in ADDR, the public key commitment pkcm committed in the coin must not be one generated by CreatePKCM previously.

– For the other query types, $\mathcal{C}$ ensures that Q, Q' are *publicly consistent*, and then forwards Q to $\mathcal{O}_0^{DAP}$, and Q' to $\mathcal{O}_1^{DAP}$; assume the two oracle answer (a_0, a_1), $\mathcal{C}$ forwards to $\mathcal{A}$ (a_b, a_{1-b}).

At the end, $\mathcal{A}$ sends $\mathcal{C}$ a guess $b' \in \{0, 1\}$. If $b = b'$, $\mathcal{C}$ outputs 1; else, $\mathcal{C}$ outputs 0.

Public consistency. As mentioned above, the pairs of queries $\mathcal{A}$ sends $\mathcal{C}$ must be of the same type and publicly consistent. We now define the public consistency. If Q, Q' are of type `CreateAddress`, the queries are automatically public consistent; further more, we require that in this case the address generated in both oracles are identity. If they are of type `CreatePKCM`, the queries are automatically public consistent. If they are of type `Mint`, then the minted value v in Q must equal the value in Q'. Finally, if they are `Pour` query, we require the following restrictions.

First, each of Q, Q' must be well-formed:

(i) the coins $\mathbf{c}_1^{old}$, $\mathbf{c}_2^{old}$ corresponding to the coin commitments (reference by the two indices idx_1^{old}, idx_2^{old}) in Q must appear in the coin table `COIN`, similar requirement for Q';
(ii) the coins $\mathbf{c}_1^{old}$, $\mathbf{c}_2^{old}$ referenced in Q must be unspent, similar requirement for Q';
(iii) the address public keys $\text{addr}_{\text{pk},1}$ and $\text{addr}_{\text{pk},2}$ in Q must match those in $\mathbf{c}_1^{old}$, $\mathbf{c}_2^{old}$, similar requirement for Q';
(iv) the balance equations must hold;
(v) the lock times of the old coins must be up;
(vi) the public key commitments $\texttt{pkcm}_i$ must be one generated by `PKCM` previously and never used in previous queries and each must be unique in these queries Q and Q'.

Furthermore, Q, Q' must be consistent with respect to public information and $\mathcal{A}$'s view:

(i) the public values in Q and Q' must equal;
(ii) for each $i \in \{1, 2\}$, the number K of public keys committed in $\texttt{pkcm}_i$ in Q and Q' must equal;
(iii) for each $i \in \{1, 2\}$, if the i-th recipient addresses in Q is not in `ADDR`, then v_i^{new} in Q and Q' must equal (vice versa for Q');
(iv) for each $i \in \{1, 2\}$, the i-th public key index k_i in Q must equal the corresponding index in Q' (vice versa for Q');
(v) for each $i \in \{1, 2\}$, if the i-th index in Q references a coin commitment in a transaction from a previously posted `Insert` query, then the corresponding index in Q' must also reference a coin commitment in a transaction posted in `Insert` query; additionally, v_i^{old} in Q and Q' must equal (vice versa for Q').

B Proof of Security

Here we present the complete proof of Theorem 1. The proofs to transaction non-malleability and balance are trivially similar to the ones in [20], we omit them here. For proof of ledger indistingsuishability, we construct a simulation $\beth_{\mathbf{sim}}$ in which the adversary $\mathcal{A}$ interacts with a challenger $\mathcal{C}$, as in the L-IND experiment. However $\beth_{\mathbf{sim}}$ modifies the L-IND experiment in a critical way: all answers sent by $\mathcal{C}$ to $\mathcal{A}$ are independent from the bit b, so the advantage of $\mathcal{A}$'s in $\beth_{\mathbf{sim}}$ is 0. Then we show that $\mathbf{Adv}^{\text{L-IND}}_{\Pi,\mathcal{A}}(\lambda)$ is only negligibly larger than $\mathcal{A}$'s advantage in $\beth_{\mathbf{sim}}$.

B.1 The simulation experiment.

The simulation $\beth_{\mathbf{sim}}$ works as follows. First, $\mathcal{C}$ samples $b \in \{0,1\}$ and pp $\leftarrow$Setup (1^λ), with the following modifications: the zk-SNARK keys are generated by ($\mathtt{pk}_{\text{POUR}}$, $\mathtt{vk}_{\text{POUR}}$, trap)$\leftarrow$ Sim$(1^\lambda, C_{\texttt{POUR}})$, instead of the usual way. Then, $\mathcal{C}$ sends pp to $\mathcal{A}$, and initializes two DAP' oracles $\mathcal{O}_0^{DAP}$ and $\mathcal{O}_1^{DAP}$.

Afterwards, $\beth_{\mathbf{sim}}$ proceeds in steps and at each step $\mathcal{C}$ present $\mathcal{A}$ two ledgers (L_{Left}, L_{Right}), where $L_{Left} := L_b$ is the current ledger in $\mathcal{O}_b^{DAP}$ and $L_{Right} := L_{1-b}$ the ledger in $\mathcal{O}_{1-b}^{DAP}$; then $\mathcal{A}$ sends to $\mathcal{C}$ a message (Q, Q'), which consist of two queries of the same type. The requirement to these two queries is the same to that in L-IND. The reaction of challenger $\mathcal{C}$ is different from that in L-IND, as described as follows:

- **Answering CreateAddress queries**. In this case, $Q = Q' =$ CreateAddress. To answer Q, $\mathcal{C}$ behaves as in L-IND, except for the following modification: after obtaining (addr_{pk}, addr_{sk})$\leftarrow$CreateAddress (pp), $\mathcal{C}$ replaces a_{pk} in addr_{pk} with a random string of the appropriate length; then, $\mathcal{C}$ stores (addr_{pk},addr_{sk}) in ADDR and returns addr_{pk} to $\mathcal{A}$. Afterwards, $\mathcal{C}$ does the same for Q'.
- **Answering CreatePKCM queries.** In this case, $Q = Q' =$ CreatePKCM. To answer Q, $\mathcal{C}$ behaves as in L-IND, except for the following modification: after obtaining ($pklist, u,$ pkcm), $\mathcal{C}$ replaces pkcm with a random string of the appropriate length; then, $\mathcal{C}$ stores the tuple in PKCM and returns pkcm to $\mathcal{A}$. Afterwards, $\mathcal{C}$ does the same for Q'.
- **Answering Mint queries**. In this case, $Q =$ (Mint, v, addr_{pk}) and $Q' =$ (Mint, v, addr'_{pk}). To answer Q, $\mathcal{C}$ behaves as in L-IND, except for the following modification: Compute $k = \texttt{COMM}_s(\tau)$ for a random string τ of the suitable length, instead of $k = \texttt{COMM}_s(H\|m)$. Afterwards, $\mathcal{C}$ does the same for Q'.
- **Answering Pour queries**. In this case, Q and Q' both have the form (Pour, idx_1^{old}, idx_2^{old}, $addr_{pk,1}^{old}$, $addr_{pk,2}^{old}$, k_1, k_2, info, v_1^{new}, v_2^{new}, $addr_{pk,1}^{new}$, $addr_{pk,2}^{new}$, $\texttt{pkcm}_1^{new}$, $\texttt{pkcm}_2^{new}$, $tlist_1^{new}$, $tlist_2^{new}$, v_{pub}). To answer Q, $\mathcal{C}$ modifies in the following ways:
  1. For each $j \in \{1,2\}$:
     (a) Uniformly sample random $\texttt{sn}_j^{old}$.
     (b) Randomly sample a list of pairs of public/private keys $pklist_j$, compute $h_{pk,j}^{old}$:= Hash($pklist_j$).
     (c) If $addr_{pk,j}^{new}$ is in ADDR:

i. sample a coin commitment $\mathtt{cm}_j^{new}$ on a random input;
ii. run $\mathcal{K}_{enc}(\mathbf{pp}_{enc}) \rightarrow (\mathtt{pk}_{enc}, \mathtt{sk}_{enc})$ and compute $\mathbf{C}_j^{new} := \mathcal{E}_{enc}(\mathtt{pk}_{enc}, r)$ for a random r of suitable length.

(d) Otherwise, calculate $(\mathtt{cm}_j^{new}, \mathbf{C}_j^{new})$ as in the `Pour` algorithm.

2. Set h_1 and h_2 to be random strings of suitable length.
3. Compute all other values as in the `Pour` algorithm.
4. The pour proof is computed as $\pi_{\text{POUR}} := \mathtt{Sim}(\mathtt{trap}, x)$, where $x := (rt_1, rt_2, \mathtt{sn}_1^{old}, \mathtt{sn}_2^{old}, h_{pk,1}^{old}, h_{pk,2}^{old}, \mathtt{cm}_1^{new}, \mathtt{cm}_2^{new}, v_{pub}, h_{sig}, h_1, h_2, k_1, k_2, timestamp)$.

Afterwards, $\mathcal{C}$ does the same for Q'.

– Answering `Insert` queries. In this case, $Q = (\mathtt{Insert}, tx)$ and $Q = (\mathtt{Insert}, tx')$. The answer to each query proceeds as in the `L-IND` experiment.

In each of the above cases, the response to $\mathcal{A}$ is computed independently of the bit b. Thus, when $\mathcal{A}$ outputs a guess b', it must be the case that $Pr[b = b'] = 1/2$, i.e., $\mathcal{A}$'s advantage in $\ni_{\mathbf{sim}}$ is 0.

B.2 Indistinguishability from Real Experiment

We construct a sequence of hybrid experiments ($\ni_{\mathbf{real}}, \ni_1, \ni_2, \ni_3, \ni_{\mathbf{sim}}$), in each of these experiments a challenger $\mathcal{C}$ conducts a different modification of the `L-IND` experiment. We define $\ni_{\mathbf{real}}$ to be the original `L-IND` experiment, and $\ni_{\mathbf{sim}}$ to be the simulation described above. Given experiment $\ni$, we define $\mathbf{Adv}^{\ni}$ to be the absolute value of the difference between the `L-IND` advantage of $\mathcal{A}$ in $\ni$ and that in $\ni_{\mathbf{real}}$. Also, let

– $q_{\mathbf{CA}}$ be the number of `CreateAddress` queries issued by $\mathcal{A}$,
– $q_{\mathbf{CP}}$ be the number of `CreatePKCM` queries issued by $\mathcal{A}$.
– $q_{\mathbf{P}}$ be the number of `Pour` queries issued by $\mathcal{A}$,
– $q_{\mathbf{M}}$ be the number of `Mint` queries issued by $\mathcal{A}$,

Finally, define $\mathbf{Adv}^{\mathbf{Enc}}$ to be $\mathcal{A}$'s advantage in **Enc**'s IND-CCA and IK-CCA experiments, $\mathbf{Adv}^{\mathtt{PRF}}$ to be $\mathcal{A}$'s advantage in distinguishing the pseudorandom function `PRF` from a random one, and $\mathbf{Adv}^{\mathtt{COMM}}$ to be $\mathcal{A}$'s advantage against the hiding property of `COMM`.

We now describe each of the hybrid experiments.

– Experiment $\ni_1$. The experiment $\ni_1$ modifies $\ni_{\mathbf{real}}$ by simulating the zk-SNARKs. More precisely, we modify $\ni_{\mathbf{real}}$ so that $\mathcal{C}$ simulates each zk-SNARK proof, as follows. At the beginning of the experiment, instead of invoking $\mathtt{KeyGen}(1^\lambda, C_{\mathtt{POUR}})$, $\mathcal{C}$ invokes $\mathtt{Sim}(1^\lambda, C_{\mathtt{POUR}})$ and obtains $(\mathtt{pk}_{\text{POUR}}, \mathtt{vk}_{\text{POUR}}, \mathtt{trap})$. At each subsequent invocation of the `Pour` algorithm, $\mathcal{C}$ computes $\pi_{\text{POUR}} \leftarrow \mathtt{Sim}(\mathtt{trap}, x)$, without using any witnesses, instead of using Prove. Since the zk-SNARK system is perfect zero knowledge, the distribution of the simulated π_{POUR} is identical to that of the proofs computed in $\ni_{\mathbf{real}}$. Hence $\mathbf{Adv}^{\ni_1} = 0$.
– Experiment $\ni_2$. The experiment $\ni_2$ modifies $\ni_1$ by replacing the ciphertexts in a pour transaction by encryptions of random strings. Each time $\mathcal{A}$ issues a `Pour` query where one of $(addr_{pk,1}^{new}, addr_{pk,2}^{new})$ is in `ADDR`, the ciphertexts $\mathbf{C}_1^{new}$, $\mathbf{C}_2^{new}$ are generated as follows:

1. $(\mathtt{pk}_{enc}^{new}, \mathtt{sk}_{enc}^{new}) \leftarrow \mathcal{K}_{enc}(\mathbf{pp}_{enc})$;
2. for each $j \in \{1,2\}$, $\mathbf{C}_j^{new} := \mathcal{E}_{enc}(\mathtt{pk}_{enc}^{new}, j, r)$ where r is a message randomly and uniformly sampled from plaintext space.

By Lemma 1, $\left|\mathbf{Adv}^{\ni_2} - \mathbf{Adv}^{\ni_1}\right| \leq 4 \cdot q_{\mathbf{P}} \cdot \mathbf{Adv}^{\mathbf{Enc}}$.

– Experiment $\ni_3$. The experiment $\ni_3$ modifies $\ni_2$ by replacing all PRF-generated values with random strings:
  - each time $\mathcal{A}$ issues a `CreateAddress` query, the value a_{pk} within the returned addr_{pk} is substituted with a random string of the same length;
  - each time $\mathcal{A}$ issues a `Pour` query, each of the serial numbers $\mathtt{sn}_1^{old}$, $\mathtt{sn}_2^{old}$ in tx_{Pour} is substituted with a random string of the same length, and h_1 and h_2 with random strings of the same length.

  By Lemma 2, $\left|\mathbf{Adv}^{\ni_3} - \mathbf{Adv}^{\ni_2}\right| \leq q_{\mathbf{CA}} \cdot \mathbf{Adv}^{\mathtt{PRF}}$
– Experiment $\ni_{\mathbf{sim}}$. The experiment $\ni_{\mathbf{sim}}$ is already described above. For comparison, we explain how it differs from $\ni_3$: all the commitments are replaced with commitments to random inputs:
  - each time $\mathcal{A}$ issues a `CreatePKCM` query, the commitment `pkcm` is substituted with a random string of suitable length; and
  - each time $\mathcal{A}$ issues a `Mint` query, the coin commitment `cm` in tx_{Mint} is substituted with a commitment to a random input; and
  - each time $\mathcal{A}$ issues a `Pour` query, for each $j \in \{1,2\}$, if the output address $addr_{pk,j}^{new}$ is in `ADDR`, $\mathtt{cm}_j^{new}$ is substituted with a commitment to a random input.

  By Lemma 3, $\left|\mathbf{Adv}^{\ni_{\mathbf{sim}}} - \mathbf{Adv}^{\ni_3}\right| \leq (q_{\mathbf{M}} + 4 \cdot q_{\mathbf{P}} + q_{\mathbf{CP}}) \cdot \mathbf{Adv}^{\mathtt{COMM}}$

By summing over $\mathcal{A}$'s advantages in the hybrid experiments, we can bound $\mathcal{A}$'s advantage in $\ni_{\mathbf{real}}$ by

$$\mathbf{Adv}^{\text{L-IND}}_{\Pi,\mathcal{A}}(\lambda) \leq 4 \cdot q_{\mathbf{P}} \cdot \mathbf{Adv}^{\mathbf{Enc}} + q_{\mathbf{CA}} \cdot \mathbf{Adv}^{\mathtt{PRF}} + (q_{\mathbf{M}} + 4 \cdot q_{\mathbf{P}} + q_{\mathbf{CP}}) \cdot \mathbf{Adv}^{\mathtt{COMM}}$$

which is negligible in λ. This concludes the proof of ledger indistinguishability.

Below, we sketch proofs for the lemmas used above.

Lemma 1. *Let AdvEnc be the maximum of: $\mathcal{A}$'s advantage in the IND-CCA experiment against the encryption scheme* **Enc**, *and $\mathcal{A}$'s advantage in the IK-CCA experiment against the encryption scheme* **Enc**. *Then after $q_{\mathbf{P}}$ `Pour` queries, $\left|\mathbf{Adv}^{\ni_2} - \mathbf{Adv}^{\ni_1}\right| \leq 4 \cdot q_{\mathbf{P}} \cdot \mathbf{Adv}^{\mathbf{Enc}}$.*

The proof of Lemma 1 is exactly the same to the proof of Lemma D.1 in [20], so we omit it here.

Lemma 2. *Let $\mathbf{Adv}^{PRF}$ be $\mathcal{A}$'s advantage in distinguishing the pseudorandom function PRF from a random function. Then, after $q_{\mathbf{CA}}$ `CreateAddress` queries, $\left|\mathbf{Adv}^{\ni_3} - \mathbf{Adv}^{\ni_2}\right| \leq q_{\mathbf{CA}} \cdot \mathbf{Adv}^{PRF}$.*

Proof sketch. We first constuct a hybrid $\mathbf{H}$, intermediate between $\ni_2$ and $\ni_3$, in which we replace all values computed by the first oracle-generated key a_{sk} with random strings. On receiving $\mathcal{A}$'s first `CreateAddress` query, replace the public address $\text{addr}_{\text{pk}} = (a_{pk}, \mathtt{pk}_{enc})$ with $\text{addr}_{\text{pk}} = (\tau, \mathtt{pk}_{enc})$ where τ is a random string of the appropriate length. On each subsequent `Pour` query tx_{Pour}, for each $i \in 1, 2$, if $addr_{pk,i}^{old} = \text{addr}_{\text{pk}}$ then:

1. replace $\mathtt{sn}_i^{old}$ with a random string of appropriate length;
2. replace each of h_1, h_2 with a random string of appropriate length;
3. simulate the zk-SNARK proof π_{POUR}.

We now argue that $\mathcal{A}$'s advantage in $\mathbf{H}$ is at most $\mathbf{Adv}^{\mathtt{PRF}}$ more than in $\ni_2$. Let a_{sk} be the secret key generated by the oracle in the first `CreateAddress` query. In $\ni_2$ (as in $\ni_{\mathbf{real}}$):

- $a_{pk} := \mathtt{PRF}^{addr}_{a_{sk}}(0)$;
- for each $i \in \{1,2\}$, $\mathtt{sn}_i := \mathtt{PRF}^{sn}_{a_{sk}}\rho)$ for a random ρ;
- for each $i \in \{1,2\}$, $h_i := \mathtt{PRF}^{pk}_{a_{sk}}(i \| h_{sig})$ and h_{sig} is unique.

Now let $\mathcal{O}$ be an oracle that implements either $\mathtt{PRF}_{a_{sk}}$ or a random function. We show that if $\mathcal{A}$ distinguishes $\mathbf{H}$ from $\ni_2$ with probability ϵ, we can construct a distinguisher for the two implementations of $\mathcal{O}$. In fact, when $\mathcal{O}$ implements $\mathtt{PRF}_{a_{sk}}$, the distribution of the experiment is identical to that of $\ni_2$; when $\mathcal{O}$ is a random function, the distribution is identical to $\mathbf{H}$. Therefore, $\mathcal{A}$'s advantage is at most $\mathbf{Adv}^{\mathtt{PRF}}$.

Finally, by the hybrid argument, we extend to all $q_{\mathbf{CA}}$ oracle-generated addresses; then, $\mathcal{A}$'s advantage gain from $\ni_2$ to $\ni_3$ is at most $q_{\mathbf{CA}} \cdot \mathbf{Adv}^{\mathtt{PRF}}$. The final hybrid is equal to $\ni_3$, we obtain that $\left|\mathbf{Adv}^{\ni_3} - \mathbf{Adv}^{\ni_2}\right| \leq q_{\mathbf{CA}} \cdot \mathbf{Adv}^{\mathtt{PRF}}$.

Lemma 3. *Let $\mathbf{Adv}^{\mathtt{COMM}}$ be $\mathcal{A}$'s advantage against the hiding property of `COMM`. After $q_{\mathbf{M}}$ `Mint` queries, $q_{\mathbf{P}}$ `Pour` queries and $q_{\mathbf{CP}}$ `CreatePKCM` queries, $\left|\mathbf{Adv}^{\ni_{\mathbf{sim}}} - \mathbf{Adv}^{\ni_3}\right| \leq (q_{\mathbf{M}} + 4 \cdot q_{\mathbf{P}} + q_{\mathbf{CP}}) \cdot \mathbf{Adv}^{\mathtt{COMM}}$.*

Proof sketch. We only provide a short sketch, because the structure of the argument is similar to the one used to prove Lemma 2 above.

For the first `Mint` or `Pour` query, replace the "internal" commitment $k := \mathtt{COMM}_s($ $H \parallel m)$ with a $\mathtt{COMM}_s(\tau)$ where τ is a random string of appropriate length. Since ρ is random, $\mathcal{A}$'s advantage in distinguishing this modified experiment from $\ni_2$ is at most $\mathbf{Adv}^{\mathtt{COMM}}$. Then, if we modify all $q_{\mathbf{M}}$ `Mint` queries and all $q_{\mathbf{P}}$ `Pour` queries, by replacing the $q_{\mathbf{M}} + 2 \cdot q_{\mathbf{P}}$ internal commitments with random strings, we can bound $\mathcal{A}$'s advantage by $(q_{\mathbf{M}} + 2 \cdot q_{\mathbf{P}}) \cdot \mathbf{Adv}^{\mathtt{COMM}}$.

Next, similarly, replace the coin commitment in the first `Pour` with a commitment to a random value, then $\mathcal{A}$'s advantage in distinguishing this modified experiment from the above one is at most $\mathbf{Adv}^{\mathtt{COMM}}$. Then, we modify all $q_{\mathbf{P}}$ `Pour` queries, by replacing the $2 \cdot q_{\mathbf{P}}$ output coin commitments with random strings, we can update the bound to $\mathcal{A}$'s advantage to $(q_{\mathbf{M}} + 2 \cdot q_{\mathbf{P}}) \cdot \mathbf{Adv}^{\mathtt{COMM}}$.

Finally, we modify the $q_{\mathbf{CP}}$ `CreatePKCM` commitments to replace the resulting $q_{\mathbf{CP}}$ public key commitments by a random string of appropriate length, we obtain the experiment $\ni_{\mathbf{sim}}$ and get that $\left|\mathbf{Adv}^{\ni_{\mathbf{sim}}} - \mathbf{Adv}^{\ni_3}\right| \leq (q_{\mathbf{M}} + 4 \cdot q_{\mathbf{P}} + q_{\mathbf{CP}}) \cdot \mathbf{Adv}^{\mathtt{COMM}}$.

References

1. Andresen, G.: Blocksize economics. bitcoinfoundation.org (2014)
2. Ben-Sasson, E., Chiesa, A., Tromer, E., Virza, M.: Succinct non-interactive arguments for a von neumann architecture. IACR Cryptology ePrint Archive 2013, 879 (2013)
3. Bonneau, J., Narayanan, A., Miller, A., Clark, J., Kroll, J.A., Felten, E.W.: Mixcoin: Anonymity for bitcoin with accountable mixes. In: International Conference on Financial Cryptography and Data Security. pp. 486–504. Springer (2014)
4. Danezis, G., Fournet, C., Kohlweiss, M., Parno, B.: Pinocchio coin: building zerocoin from a succinct pairing-based proof system. In: Proceedings of the First ACM workshop on Language support for privacy-enhancing technologies. pp. 27–30. ACM (2013)
5. Eyal, I., Gencer, A.E., Sirer, E.G., Van Renesse, R.: Bitcoin-ng: A scalable blockchain protocol. In: 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16). pp. 45–59. USENIX Association (2016)
6. Garay, J., Kiayias, A., Leonardos, N.: The bitcoin backbone protocol: Analysis and applications. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 281–310. Springer (2015)
7. Garman, C., Green, M., Miers, I.: Accountable privacy for decentralized anonymous payments (2016)
8. Gennaro, R., Jarecki, S., Krawczyk, H., Rabin, T.: Secure distributed key generation for discrete-log based cryptosystems. In: International Conference on the Theory and Applications of Cryptographic Techniques. pp. 295–310. Springer (1999)
9. Heilman, E., Baldimtsi, F., Alshenibr, L., Scafuro, A., Goldberg, S.: Tumblebit: An untrusted tumbler for bitcoin-compatible anonymous payments. IACR Cryptology ePrint Archive 2016, 575 (2016)
10. King, S., Nadal, S.: Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. self-published paper, August 19 (2012)
11. Kosba, A., Miller, A., Shi, E., Wen, Z., Papamanthou, C.: Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In: Security and Privacy (SP), 2016 IEEE Symposium on. pp. 839–858. IEEE (2016)
12. Kroll, J.A., Davey, I.C., Felten, E.W.: The economics of bitcoin mining, or bitcoin in the presence of adversaries. In: Proceedings of WEIS. vol. 2013. Citeseer (2013)
13. Maxwell, G.: Coinjoin: Bitcoin privacy for the real world. In: Post on Bitcoin Forum (2013)
14. Maxwell, G.: Coinswap: Transaction graph disjoint trustless trading. CoinSwap: Transaction-graphdisjointtrustlesstrading (2013)
15. Miers, I., Garman, C., Green, M., Rubin, A.D.: Zerocoin: Anonymous distributed e-cash from bitcoin. In: Security and Privacy (SP), 2013 IEEE Symposium on. pp. 397–411. IEEE (2013)
16. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008)
17. Poon, J., Dryja, T.: The bitcoin lightning network: Scalable off-chain instant payments (2016)
18. Reid, F., Harrigan, M.: An analysis of anonymity in the bitcoin system. In: Security and privacy in social networks, pp. 197–223. Springer (2013)
19. Ruffing, T., Moreno-Sanchez, P., Kate, A.: Coinshuffle: Practical decentralized coin mixing for bitcoin. In: European Symposium on Research in Computer Security. pp. 345–364. Springer (2014)
20. Sasson, E.B., Chiesa, A., Garman, C., Green, M., Miers, I., Tromer, E., Virza, M.: Zerocash: Decentralized anonymous payments from bitcoin. In: IEEE Symposium on Security and Privacy. pp. 459–474 (2014)
21. Sompolinsky, Y., Zohar, A.: Accelerating bitcoin's transaction processing. fast money grows on trees, not chains. IACR Cryptology ePrint Archive 2013(881) (2013)
22. Valenta, L., Rowan, B.: Blindcoin: Blinded, accountable mixes for bitcoin. In: International Conference on Financial Cryptography and Data Security. pp. 112–126. Springer (2015)
23. Wood, G.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum Project Yellow Paper 151 (2014)
24. Ziegeldorf, J.H., Grossmann, F., Henze, M., Inden, N., Wehrle, K.: Coinparty: Secure multi-party mixing of bitcoins. In: Proceedings of the 5th ACM Conference on Data and Application Security and Privacy. pp. 75–86. ACM (2015)