A Universally Composable Non-interactive Aggregate Cash System

Yanxue Jia[1], Shi-Feng Sun[1(✉)], Hong-Sheng Zhou[2], and Dawu Gu[1(✉)]

[1] Shanghai Jiao Tong University, Shanghai, China
{jiayanxue,shifeng.sun,dwgu}@sjtu.edu.cn
[2] Virginia Commonwealth University, Richmond, USA
hszhou@vcu.edu

Abstract. Mimblewimble is a privacy-preserving cryptocurrency, providing the functionality of transaction aggregation. Once certain coins have been spent in Mimblewimble, they can be deleted from the UTXO set. This is desirable: now storage can be saved and computation cost can be reduced. Fuchsbauer et al. (EUROCRYPT 2019) abstracted Mimblewimble as an Aggregate Cash System (ACS) and provided security analysis via game-based definitions.

In this paper, we revisit the ACS, and focus on *Non-interactive* ACS, denoted as NiACS. We for the first time propose a simulation-based security definition and formalize an ideal functionality for NiACS. Then, we construct a NiACS protocol in a hybrid model which can securely realize the ideal NiACS functionality in the Universal Composition (UC) framework. In addition, we propose a building block, which is a variant of the ElGamal encryption scheme that may be of independent interest. Finally, we show how to instantiate our protocol, and obtain the first NiACS system with UC security.

1 Introduction

Decentralized cryptocurrencies like Bitcoin have attracted huge attention in the past decade. While these cryptocurrencies have multiple advantages over the traditional electronic payment systems, we must note that these benefits are at the expense of transaction-privacy or user-anonymity [6,32]: users' transaction data in the distributed ledgers of the cryptocurrency systems are public, and thus can be traced. Many strategies have been taken to improve the privacy (e.g., using a fresh pseudonymous address for each payment). Unfortunately, it has been demonstrated that the expected user-anonymity can still be lost: an attacker could deanonymize the transactions on the ledger by clustering and analyzing the transaction graph [36,40].

Motivated by these security concerns, extensive efforts have been devoted to develop privacy-preserving techniques for improving the amount confidentiality and user anonymity of cryptocurrencies. Typically, homomorphic commitments are used to ensure the confidentiality of the amounts in transactions. To enable user anonymity, there exist two design paradigms:

S. Agrawal and D. Lin (Eds.): ASIACRYPT 2022, LNCS 13791, pp. 745–773, 2022.
https://doi.org/10.1007/978-3-031-22963-3_25

(i) First, anonymity sets are introduced to hide the identities of the users. Good examples include Monero [37], Zcash [8] and Quisquis [19]. Unfortunately, as mentioned in [19], the information that which coins have been spent in Monero/Zcash is not allowed to be revealed, and thus the already spent coins cannot be eliminated from the cryptocurrency systems; as a consequence, the size of UTXO sets in these systems will grow constantly. Then Quisquis [19] was proposed to solve this problem but at the price of introducing a complicated mechanism: the users have to have their wallets to "watch on" the blockchain so that certain information (e.g., wallet address or public key) in their wallets can be properly updated. Moreover, Quisquis suffers from the so-called "front-running" attacks as pointed out in [11].

(ii) To achieve anonymity, coins of a transaction can be mixed with those of other transactions; this is called CoinJoin. Now, the coins that have been spent can be deleted from the UTXO set; In this way, the size of UTXO set will be significantly reduced. We note that, many systems (e.g., CoinShuffle [41] and Mixcoin [10]) under this design paradigm focus on anonymity but not considering the confidentiality of the amounts in transactions. It is worth mentioning that the anonymity of cryptocurrencies under this design paradigm may be weakened, when some parties are designated for receiving and mixing transactions. Still, it has attracted much attention thanks to its potential for high performance.

Mimblewimble. A new cryptocurrency dubbed Mimblewimble, which follows the second paradigm and considers confidentiality, was proposed by an anonymous author in [26] and further improved by Poelstra [39]. A nice feature Mimblewimble additionally enjoys is that, when multiple transactions are aggregated and the corresponding coins are mixed, it allows cut-through[1] while maintaining the verifiability of the aggregate transaction. This feature can reduce storage and benefit new users to verify the system. To formally analyze the security of Mimblewimble, Fuchsbauer et al. [21] abstracted it as an Aggregate Cash System (ACS) and formalized its security via a series of games. Specifically, they proposed three game-based security properties: *inflation-resistance* ensures that coins can only be supplied by legitimate ways (e.g., coinbase transactions), *theft-resistance* guarantees that no one can spend coins without the corresponding spending keys, and *transaction indistinguishability* requires that the amount should be hidden and change coins and output coins be indistinguishable. The work [21] by Fuchsbauer et al. is significant for the formal security analysis of Mimblewimble, but their definition is still subject to the following limitations:

– First, we point out that the security games proposed in [21] are not "succinct" enough. For example the theft-resistance property definition is strongly correlated with their construction. More concretely, to define theft-resistance

[1] A basic property of the UTXO model is that a sequence of two transactions, the first one spending an output out_1 and creating out_2, followed by the second one spending out_2 and creating out_3, is equivalent to a single cut-through transaction spending out_1 and creating out_3.

property, a notion of *pre-transaction* has been introduced. However, this *pre-transaction* is part of their protocol construction. This indicates that their theft-resistance property definition is not general enough: their definition only allows *pre-transaction* dependent protocols, and does not allow other natural constructions.

- Second, Fuchsbauer et al. [21] did not define the unlinkability of inputs and outputs, which is an important security property of ACS. More specifically, this property means that, when a transaction is mixed with others, anyone *not involved in these transactions* cannot identify which inputs and/or outputs belong to the same transaction. Note that if a party can obtain the individual transactions before aggregation, then she/he must be aware of the linkability of the inputs and outputs; we thus do not consider the unlinkability[2] against the parties who are continuously monitoring the network, or parties who are responsible for aggregating transactions.
- While game-based definitions are useful for capturing the security properties of ACS as in [21], it is more desirable to investigate the security properties in the real/ideal simulation paradigm: First, following the game-based definition approach, typically we are not clear if the list of security properties that we formalized are sufficient; often certain natural security properties are missed. Second, following the real/ideal simulation paradigm, "The security guarantees achieved are easily understood (because the ideal model is easily understood) [34]." In addition, simulation-based definitions allow sequential or even universal composability, enabling modular design and analysis. Please see Lindell's tutorials [34,35] for a more careful elaboration.

In addition, a transaction in ACS has to be generated jointly by the sender and receiver (see Sect. 2.2 for more details). In practice, however, it is not easy to guarantee that both the sender and receiver are always online at the same time. For example, it may be difficult for an online retailer to keep his wallet online all the time to receive irregular payments. A better way is that the retailer publishes an account for receiving payments on the sales website, and the buyers can complete payments at any time without the retailer's cooperation. Beam [1] and MWC [3], the two representative projects based on ACS, have made efforts to mitigate the problem to some extent, but do not solve it completely.

1.1 Contributions

In this work, we focus on mitigating the above limitations by proposing an Aggregate Cash System supporting non-interactive payments, denoted as NiACS, and defining its security in the real/ideal simulation paradigm. Our contributions are summarized below:

[2] In practice, Grin and Beam enhance unlinkability by leveraging Dandelion relay protocol [18] that aggregates transactions during the propagation. However, their approach still cannot realize *complete* unlinkability, since someone on the network will always be able to see an unaggregated transaction.

We first define an ideal functionality $\mathcal{F}_{\mathsf{NiACS}}$, which captures the core features of NiACS but does not depend on the concrete design of NiACS. Our ideal functionality $\mathcal{F}_{\mathsf{NiACS}}$ captures the *inflation-resistance*, *theft-resistance*, and *transaction indistinguishability* properties in [21]; in addition our functionality $\mathcal{F}_{\mathsf{NiACS}}$ captures *unlinkability*, which is a very important security property for privacy-preserving cryptocurrencies. We remark that, the *unlinkability* has not been formalized in [21].

Our ideal functionality $\mathcal{F}_{\mathsf{NiACS}}$ is *not* introduced for formalizing security properties for *interactive* ACS. However, to capture interactive payments, we can redefine the ideal functionality; for example, we can let the functionality inform the receiver before dealing with a transaction, and this is missing in the current functionality $\mathcal{F}_{\mathsf{NiACS}}$.

Furthermore, we propose a NiACS protocol Π_{NiACS} that UC-realizes $\mathcal{F}_{\mathsf{NiACS}}$ in a hybrid model. In contrast to Mimblewimble, our design can support non-interactive payments. That is, the sender is able to generate a valid transaction by himself, and the receiver can directly obtain private information of output coins from the transaction without out-of-band communication over a private channel. Particularly, to avoid the out-of-band communication, we propose a new variant of ElGamal encryption, the ciphertext of which includes a Pedersen commitment and its openings (i.e., randomness and value) can be obtained readily by the holder of the decryption key. Moreover, we present a concrete instantiation of our NiACS protocol, thus obtaining the first NiACS with UC security.

1.2 Related Work

Over the past decade, extensive efforts have been made to achieve provably secure privacy-preserving cryptocurrencies [8,10,19,26,37]. For example, Ring Confidential Transaction (RingCT), the core protocol of Monero, was first formally analyzed by Sun et al. [42], which was further refined by subsequent works [17,33,45]. In addition, Zcash was proposed along with formal security properties by Ben-Sasson et al. [8]. However, Garman et al. [23] pointed out that the security properties defined in [8] is incomplete and complex, and adversary can leverage these weaknesses to break the security. Moreover, Garman et al. [23] gave a simulation-based definition to avoid the weaknesses. More recently, more and more works have focused on the simulation-based definitions for Blockchain protocols. Badertscher et al. abstracted Bitcoin as a ledger functionality in [7]. Kerber et al. [28] gave a private ledger functionality and designed a privacy-preserving proof-of-stake (PoS) blockchain protocol that can securely realize the private ledger functionality in the UC setting.

Mimblewimble was first proposed by [26] and then improved further by Poelstra [39]. Mimblewimble is simple to implement and has been used in three open-source cryptocurrency projects, i.e., Beam [1], Grin [2], and MWC [3]. However, no formal security analysis has been given for these works until the work by Fuchsbauer et al. [21]. In particular, Fuchsbauer et al. [21] abstracted Mimblewimble as ACS and defined its security properties for the first time.

However, these security properties are incomplete and complex. In particular, unlinkability is an important property of Mimblewimble, but not formally defined in [21]. Besides, they proposed a new one-round interactive ACS that has been implemented in MWC [3]. Later, Yu [44] leveraged one-time addresses to achieve non-interactive payments for Mimblewimble, but without formal analysis.

In a concurrent and independent work, Fuchsbauer et al. [22] pointed out and fixed the flaws in the proposal by [44], and formally analyzed their own modified scheme based on game-based definitions. Fuchsbauer et al. [22] and Yu [44] share the same initial idea of achieving non-interactive transactions with us, but there are many differences between their construction and ours. In particular, Fuchsbauer et al. [22] added a new type of excess $X := \prod \hat{R}_i / \prod D_i$, called "stealth excess", where $\hat{R}_i$ is used to "transmit" the secret key of a one-time address and D_i is a one-time doubling key used to prevent feed-me attack. For a coin, when it is in an output list, it will be associated with $\hat{R}_i$; when it is in an input list, it will be associated with D_i. Since $\hat{R}_i \neq D_i$, if the coin is cut through, the stealth excess cannot be verified. Therefore, their scheme cannot support *cut-through*. In contrast, our transactions only include one type of excess, namely the original excess in Mimblewimble, and thus our construction still supports cut-through. Note that cut-through is an important feature of Mimblewimble as it can save the on-chain storage cost. In addition, we for the first time define a simulation-based security model for NiACS, while Fuchsbauer et al. [22] still follows a game-based security model, which is not suitable for complex execution environments.

2 Technical Overview

To overcome the security and practicality issues mentioned before, we first define an ideal functionality for ACS supporting non-interactive payments, denoted by $\mathcal{F}_{\mathsf{NiACS}}$. Compared to the game-based security definition proposed in [21], $\mathcal{F}_{\mathsf{NiACS}}$ is more general and comprehensive. Furthermore, we propose a new non-interactive payment system, dubbed Π_{NiACS}, that securely realizes $\mathcal{F}_{\mathsf{NiACS}}$. Before showing the high-level idea of our design, we first briefly introduce how to define an ideal functionality that captures the desirable security of NiACS.

2.1 Non-interactive Aggregate Cash System Functionality

As ACS is essentially a privacy-preserving ledger, we attempt to define its ideal functionality with the abstraction $\mathcal{F}_{\mathsf{Ledger}}$ of the most basic ledger Bitcoin [7,29] as the starting point. At a high level, the ledger functionality defined in [29] is the same as that defined in [7]. More concretely, anyone can submit a transaction to $\mathcal{F}_{\mathsf{Ledger}}$, then $\mathcal{F}_{\mathsf{Ledger}}$ will validate the transaction by a predicate $\mathsf{Validate}$. If the transaction is valid, it will be added into a buffer. Periodically, the transactions in the buffer will be moved to state in the form of a block, where the state refers to the ledger state and the transactions in the state cannot be changed. Moreover, anyone is allowed to read the content of state. In a nutshell, $\mathcal{F}_{\mathsf{Ledger}}$ defines

the basic interfaces of a ledger, including submitting transactions, maintaining ledger, and reading ledger.

Compared to the basic ledger, ACS additionally protects *privacy* (payment amount confidentiality, sender anonymity and receiver anonymity) and allows *aggregation* of transactions before packing. Therefore, to define the ideal functionality of ACS, we need to further specify the content of a transaction and add interfaces for aggregation. In addition, in this work, we focus on ACS supporting non-interactive payments, and thus we denote the ideal functionality as $\mathcal{F}_{\mathsf{NiACS}}$.

To preserve privacy, a transaction cannot contain the identifiers of relevant parties and the amount of each coin, but a transaction needs to specify which coins are spent or created. Therefore, for each coin, we define an identifier cid that is pseudorandom and does not reveal its amount and owner, and for each party, $\mathcal{F}_{\mathsf{NiACS}}$ maintains a list of coins that are possessed by the party. When a party called user wants to transfer some coins, he inputs $(\{\mathsf{cid}_i\}, \{\hat{P}_j, \hat{v}_j\})$ to $\mathcal{F}_{\mathsf{NiACS}}$ where $\{\mathsf{cid}_i\}$ is an identifier list of the coins to be spent, $\hat{P}_j$ is a receiver who will receive a coin of amount $\hat{v}_j$. If all the coins identified by $\{\mathsf{cid}_i\}$ are owned by the party, and the sum of the amounts is enough, $\mathcal{F}_{\mathsf{NiACS}}$ will notify the adversary (namely, simulator) $\mathcal{S}$ to generate a coin identifier $\hat{\mathsf{cid}}_j$ for each output coin whose owner is $\hat{P}_j$ and amount is $\hat{v}_j$. At this point, $\mathcal{F}_{\mathsf{NiACS}}$ generates a transaction $\mathsf{TX} := (\{\mathsf{cid}_i\}, \{\hat{\mathsf{cid}}_j\})$ according to the party's payment request. We can see that $\mathcal{F}_{\mathsf{NiACS}}$ generates the transaction without the participation of any receiver, which means that the functionality captures the *non-interactive payments*. Moreover, it is the transaction that will be added to state, not the payment request. Therefore, when other parties obtain state by reading the ledger, regarding the transaction, they can only see the identifiers in $\{\mathsf{cid}_i\}$ and $\{\hat{\mathsf{cid}}_j\}$ but learn nothing about the owners and the amounts. However, if parties can get the individual transaction, they can learn the linkability of the coins in it, which will weaken the sender anonymity. The aggregation explained below helps to break the linkability.

As mentioned before, unlike in Bitcoin, a transaction in NiACS will be aggregated with other transactions before being packed into a block. Thus, the coins of a transaction are mixed with coins in other transactions such that the linkability of input and output coins is broken. More specifically, in $\mathcal{F}_{\mathsf{NiACS}}$, we add a role called aggregator who is responsible for aggregating transactions. Once a transaction TX is generated, $\mathcal{F}_{\mathsf{NiACS}}$ sends it to the parties who act as aggregators. Then, an aggregator will aggregate the transactions to a "large" transaction, which will be added to buffer and eventually moved to state. For example, given two transactions $\mathsf{TX}_1 := (\{\mathsf{cid}_i^1\}, \{\hat{\mathsf{cid}}_j^1\})$ and $\mathsf{TX}_2 := (\{\mathsf{cid}_i^2\}, \{\hat{\mathsf{cid}}_j^2\})$, an aggregator can generate an aggregate transaction $\mathsf{TX}_1 + \mathsf{TX}_2 := (\{\mathsf{cid}_i^1\} \cup \{\mathsf{cid}_i^2\}, \{\hat{\mathsf{cid}}_j^1\} \cup \{\hat{\mathsf{cid}}_j^2\})$ and submit it to $\mathcal{F}_{\mathsf{NiACS}}$. When others get the aggregate transaction, they cannot identify which coins belong to a transaction as the coin identifiers are independent and pseudorandom. In addition, a user is allowed to spend the output coins of an unconfirmed transaction with an elevated fee, as described in [4]. Therefore, cut-through can occur when the transactions are aggregated.

To sum up, we define $\mathcal{F}_{\mathsf{NiACS}}$ by adding privacy protection and aggregation features to the basic ledger functionality $\mathcal{F}_{\mathsf{Ledger}}$. In $\mathcal{F}_{\mathsf{NiACS}}$, a transaction only

contains pseudorandom and independent coin identifiers, and thus anyone cannot learn the amounts and participants. In addition, there are aggregators responsible for aggregating transactions and submitting the aggregate transactions to the ledger. Through aggregation, some coins can be cut through, which can reduce the storage of the ledger. Moreover, a transaction is stored in the ledger after being mixed with other transactions such that the linkability of the input coins and output coins is hidden, which further enhances sender anonymity.

Next we proceed to introduce the high-level idea of our design Π_{NiACS}. Since our design is inspired by Mimblewimble [21,39], before continuing we first briefly recall the main idea of Mimblewimble.

2.2 Recall Mimblewimble

Mimblewimble [21,39] is an interactive payment system with support for transaction aggregation. Briefly, a coin in Mimblewimble is a Pedersen commitment $\mathsf{cm} := g^r h^v$, where v is the amount of this coin, and it is spent with only the randomness r (usually called spending key)[3]. A transaction here consists of an input list, an output list, and a kernel, as shown in Fig. 1; the concept of kernel is firstly introduced in Mimblewimble, which plays a crucial role in guaranteeing the validity of the transaction. Particularly, the input (resp. output) list includes the spent (resp. newly created) coins, and the kernel contains the contents used for proving the balance of the transaction and the ownership of the spent coins. To illustrate how the validity of transactions is guaranteed, we take a concrete example as below.

Let TX_1 be a transaction including 2 input coins $\{\mathsf{cm}_1^1, \mathsf{cm}_2^1\}$ and 3 output coins $\{\hat{\mathsf{cm}}_1^1, \hat{\mathsf{cm}}_2^1, \hat{\mathsf{cm}}_3^1\}$[4]. Besides, an item called *excess* $E_1 := \prod_{j=1}^{3} \hat{\mathsf{cm}}_j^1 / \prod_{i=1}^{2} \mathsf{cm}_i^1$ is contained in its kernel. Obviously, if the transaction is balanced, E_1 is a commitment to 0. To show the balance of this transaction, the sender generates a proof that E_1 is a commitment to 0 as shown in Fig. 1; essentially, this is realized by invoking a zero-knowledge ideal functionality $\mathcal{F}_{\mathsf{NIZK}}^{\mathsf{zero}}$, and the witness is the randomness of E_1. However, the randomness of E_1 is derived from the randomnesses of both the input commitments $\{\mathsf{cm}_i^1\}_{i\in[2]}$ and output commitments $\{\hat{\mathsf{cm}}_j^1\}_{j\in[3]}$, and the randomnesses of output commitments are only known to the receiver, so the sender has to generate the proof interactively with the receiver. Moreover, since the ownership of the coin in Mimblewimble is equivalent to the knowledge of the opening of the commitment, the proof also implies that the input coins are indeed spent by the owner.

Further to break the linkability of inputs and outputs in a transaction, Mimblewimble adopts the idea of CoinJoin, that is, to aggregate different transactions into a "large" one. As indicated in [21], however, it is not hard to find out the input and output coins of a transaction from the aggregate transaction by solving

[3] In contrast, the coin in other cryptocurrencies like Zcash is spent with the opening of the commitment and a secret key associated with the address recording the coin.

[4] Note that for each output coin $\hat{\mathsf{cm}}_i^1$, there is also a range proof to guarantee that the committed value is valid (i.e., $v \in [0, v_{\mathsf{max}}]$), but we ignore it here for simplicity.

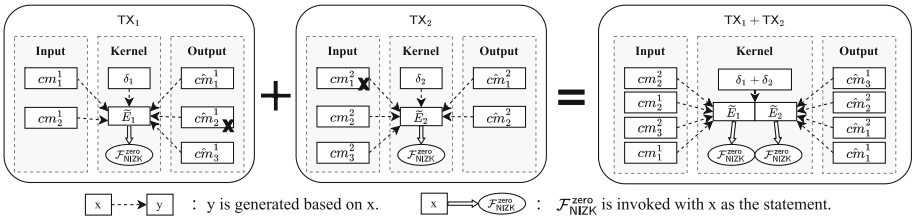

Fig. 1. The transaction and aggregation process of Mimblewimble ($\hat{\mathsf{cm}}_2^1 = \mathsf{cm}_1^2$).

a simple subset sum problem based on the excess. To solve this problem, an offset is added to blind the excess, e.g., the final excess in the kernel of TX_1 becomes $\tilde{E}_1 := E_1 \cdot g^{\delta}$. Next we take the example given in Fig. 1 to explain the aggregation process: The initial input (resp. output) list of the aggregate transaction $\mathsf{TX}_1 + \mathsf{TX}_2$ is the union of input (resp. output) lists of TX_1 and TX_2 in random order. If some coins in TX_2's input list are also in TX_1's output list, then the coins (together with the associated range proofs) are removed from the input and output lists of $\mathsf{TX}_1 + \mathsf{TX}_2$, which is the so-called *cut-through*. More concretely, $\hat{\mathsf{cm}}_2^1$ in TX_1 is equal to cm_1^2 in TX_2, so in $\mathsf{TX}_1 + \mathsf{TX}_2$, cm_1^2 and $\hat{\mathsf{cm}}_2^1$ are removed from input list and output list of the aggregate transaction, respectively. The kernel of $\mathsf{TX}_1 + \mathsf{TX}_2$ is the union of the TX_1's kernel and TX_2's kernel, except that the offset of $\mathsf{TX}_1 + \mathsf{TX}_2$ is $\delta_1 + \delta_2$.

From the above, we can see that proving the excess being a commitment to 0 is the main reason of making Mimblewimble interactive. Next, we show how to surround this obstacle and design a NiACS that securely realizes the proposed ideal functionality $\mathcal{F}_{\mathsf{NiACS}}$.

2.3 Our Non-interactive Aggregate Cash System Π_{NiACS}

Recall that in Mimblewimble each coin is spent with the randomness of its commitment as the spending key, and the output coins of each transaction have to be created by the receiver. Therefore, the sender knows nothing about the randomness of each output coin, and he has to interact with the receiver for proving that the excess is a commitment to 0. To realize non-interactive payments, a natural choice is to let the sender create the output coins. Using this approach, the sender can generate the proof of balance by himself, but the associated randomness can never be used as the spending key of the coin. Hence, our essential idea is to verify the balance of the transaction and the ownership of the input coins separately. To this end, we introduce an address for each coin in our Π_{NiACS}, then the secret key corresponding to the address is used to spend the associated coin while the randomness of the associated commitment is used only to prove balance. Following this way, the interaction between the sender and receiver can be avoided, but *the first challenging task we face is to bind a coin and an address.*

In fact, the combination of commitments and addresses have been employed previously to achieve privacy-preserving cryptocurrencies (e.g., Monero [37] and

Zcash [8]). Among them, a signature for each transaction is usually generated for preventing it from being tampered with. In our design, however, distinct transactions will be mixed or aggregated to hide the relations of input coins and output coins (namely, linkability). Then when verifying the aggregate transaction, the verifiers need to pick out the individual transaction to verify the signature, which will break the unlinkability. Therefore, it is not easy to bind a coin and an address while supporting transaction aggregation.

Bind an Address and a Coin. As discussed before, binding a coin to an address through a signature on the whole transaction will break the unlinkability. Therefore, a natural idea is to bind each coin and the associated address through a separate signature. However,the question is how to generate such a signature?

Fortunately, we observe that in our Π_{NiACS}, the randomnesses of commitments do not act as the spending keys anymore, and the sender knows the randomnesses of all output coins as they are generated by the sender himself rather than the receiver. Moreover, the proof of the excess being a commitment to 0 leaks nothing about the randomnesses of the coins, due to the zero-knowledge property. Hence, our idea is to use the randomness of each commitment as the signing key to sign the corresponding address, which can be achieved by leveraging the primitive named signature of knowledge [15].

More specifically, signature of knowledge extends the traditional notion of digital signature to the notion that allows one to issue signatures on behalf of any NP statement, which can be interpreted as follows: "A person in possession of a witness w to the statement $x \in \mathcal{L}$ has signed message m." An instance of signature of knowledge can be related to a language. In the confidential transaction, for each commitment, a range proof is generated to prove that the committed value is within a specific range $[0, v_{\mathsf{max}}]$. Therefore, we can define the language $\mathcal{L}_{\mathsf{range}} := \{\mathsf{cm} \mid \exists\ r, v \text{ s.t. } \mathsf{cm} = g^r h^v \wedge v \in [0, v_{\mathsf{max}}]\}$, and only the one knowing the opening (r, v) of cm can sign an address by invoking $\mathcal{F}_{\mathsf{SoK}}^{\mathsf{range}}$.

To guarantee the validity of transactions, *the second challenge is to prove the balance of each transaction as well as the knowledge of spending keys (i.e., the ownership of input coins).* Regarding the former, it can be proved in the same way as Mimblewimble. Therefore, the main challenge is to prove the ownership of the input coins.

Prove the Ownership of Input Coins. A natural solution is to provide a zero-knowledge proof that the sender knows the corresponding secret key of the address. However, an independent proof can be stolen and used in other transactions. A common approach for avoiding this problem is to bind the address and the transaction through a signature of knowledge (i.e., sign the transaction using the secret key), but as discussed before, signing the whole transaction will break the unlinkability. Therefore, what we essentially need is to bind the address to an "abstract" of the transaction that does not reveal the relation of the inputs and outputs. We observe that an excess in Mimblewimble is abstracted from all input and output coins of a transaction, and that it reveals nothing about the relation between the inputs and outputs, due to the added offset. Thus, we bind the address of each coin to the transaction via signing its excess with the associated spending key. Then the same excess will be signed n times if the

transaction includes n input coins. In this case, when the transaction is aggregated with others, any party can learn that these input coins belong to the same transaction, which may reveal partial information about the linkability. To avoid this leakage, our key idea is to randomly split the excess into n parts, and then to sign each part with a separate spending key. Using this approach, the sender can prove the knowledge of spending keys while preventing the proofs from being stolen and used in other transactions. Similarly, this can be realized through the signature of knowledge functionality, where the witness is the spending key and the message is a part of the excess.

Following the above way, the sender can generate a valid transaction in a non-interactive way, but the receiver cannot spend the received coins as she does not know the private information (i.e., openings of commitments). Therefore, *the third challenge is how to send the private information to the receiver.*

Send Private Information to Receiver. A natural approach is to send the value and randomness to the receiver through a private communication channel. In this way, the sender and receiver must interact at least once per transaction, which defeats our purpose of achieving non-interactive payments. Another way is to encrypt the private information with the receiver's public key and send the ciphertext along with each output coin. This will avoid the interaction between the sender and receiver, but results in a significant increase of transaction size. Inspired by the recent work due to Chen et al. [16], we propose a novel way of encrypting the private information while mitigating the transaction expansion.

In particular, Chen et al. [16] proposed a twisted ElGamal encryption to transfer values privately (from sender to receiver) as follows. Roughly, the sender encrypts a value v into a ciphertext in the form of $(pk^r, g^r h^v)$, where $pk = g^{sk}$ and sk is known by the receiver and r is randomly chosen by the sender, and includes the ciphertext in the transaction; as $g^r h^v$ is in fact a Pedersen commitment, we write the ciphertext as (X, cm) for simplicity. After receiving the ciphertext, the receiver can then recover v by computing $\mathsf{cm}/X^{\frac{1}{sk}}$. Note that the value is in a certain range, and thus the receiver can get v from h^v. Unfortunately, the receiver cannot get the randomness r, so she cannot spend the coin cm. To overcome this problem, they proposed to spend the coin in an alternative way. More specifically, after recovering the amount v from $C := (X, \mathsf{cm})$, the receiver generates a new coin $C' := (X', \mathsf{cm}')$ with the equivalent amount as C. Further, the receiver provides a proof through a Σ-protocol to prove that the messages in cm and cm' are identical.

At the first glance, their approach works in our design as well. Unfortunately, we find it does not support cut-through. Particularly, we assume that a coin cm created in transaction TX_c is spent in transaction TX_s through a new coin cm' with the equivalent amount. Note that in TX_c, it is cm that is used to generate the excess $E_c := \mathsf{cm} \cdot E_c^* \cdot g^{\delta_c}$, where E_c^* denotes the excess of other coins excluding cm and δ_c denotes the offset. In contrast, it is cm' that is used to generate the excess $E_s := E_s^*/\mathsf{cm}' \cdot g^{\delta_s}$ of TX_s, where E_s^* is the excess of other commitments than cm' and δ_s is the offset. Now we can see that if the two transactions are aggregated and the coin is cut through, then the excess of the aggregate transaction should be $E_s^* \cdot E_c^* \cdot g^{\delta_s + \delta_c}$ according to our design.

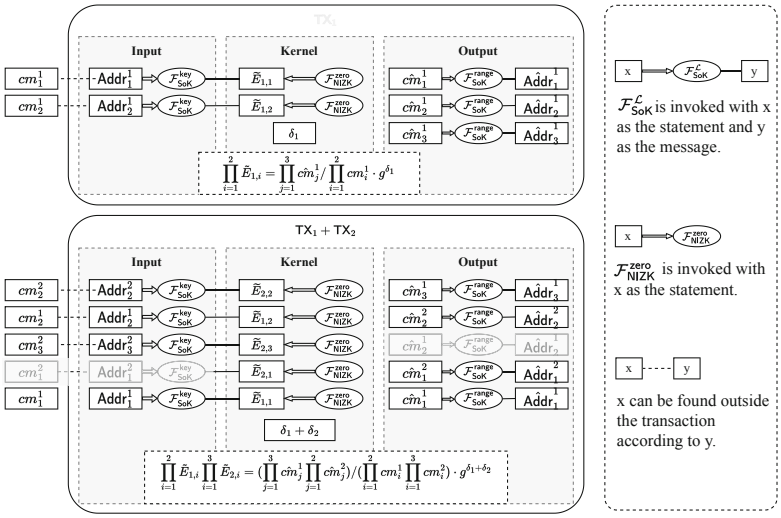

Fig. 2. The transaction and aggregation process of Π_{NiACS} ($\hat{\mathsf{cm}}_2^1 = \mathsf{cm}_1^2$).

Following the above approach, however, $E_s \cdot E_c = E_s^* \cdot E_c^* \cdot \frac{\mathsf{cm}}{\mathsf{cm}'} \cdot g^{\delta_s+\delta_c}$, which is not equal to $E_s^* \cdot E_c^* \cdot g^{\delta_s+\delta_c}$ as $\frac{\mathsf{cm}}{\mathsf{cm}'} \neq 1$. Therefore, the approach by Chen et al. fails to work in our design.

To tackle the above problem, our essential idea is to enable the receiver to recover both the value and the randomness directly from the coin cm. To this end, we propose a new variant of ElGamal by generating the ciphertext as $(pk^r, g^{H(g^r)}h^v)$, where the randomness of the commitment is chosen through a random oracle $H(\cdot)$. In this way, the receiver holding sk can easily recover g^r and thus get $H(g^r)$.

To this point, we obtain our NiACS protocol Π_{NiACS}. Following the above ideas, the transaction and aggregation process in our design are as shown in Fig. 2. More details are shown in Sect. 4.

3 Simulation-Based Security for NiACS

In this section, we propose a simulation-based security definition for NiACS through an ideal functionality $\mathcal{F}_{\mathsf{NiACS}}$.

As $\mathcal{F}_{\mathsf{NiACS}}$ is essentially a ledger that records aggregate transactions rather than individual ones. Therefore, we define $\mathcal{F}_{\mathsf{NiACS}}$ starting from the basic ledger

Algorithm 1. State Update

```
procedure ExtendState(, , buffer, T, counter)
    Send ⟨ClockRead, sid⟩ to G_Clock and receive ⟨ClockRead, sid, τ⟩ from G_Clock;
    if |τ − T · counter| > T then
        := ||Blockify(τ, buffer);
        buffer := ε;
        counter := counter + 1;
        Send ⟨ClockUpdate, sid⟩ to G_Clock.
```

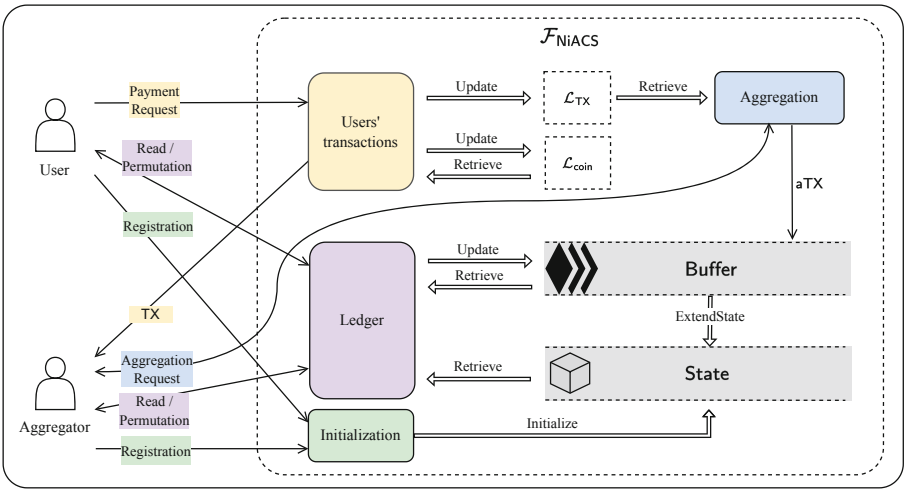

Fig. 3. Overview of ideal functionality $\mathcal{F}_{\mathsf{NiACS}}$.

functionality $\mathcal{F}_{\mathsf{Ledger}}$ formalized by Kiayias et al. [29] and follow their parameters. Prior to introducing our functionality $\mathcal{F}_{\mathsf{NiACS}}$, we first recall the parameters used later: buffer is to contain the valid transactions that have not been confirmed, state is the state of ledger containing confirmed transactions, the constant T denotes the time interval between generating blocks, and counter is to keep track of the number of state updates. In addition, the state is updated by EXTEND-STATE shown in Algorithm 1, where the function $\mathsf{Blockify}(\tau, \mathsf{buffer})$ is used to organize the transactions in buffer into a block and τ is a time obtained from a global clock ideal functionality $\mathcal{G}_{\mathsf{Clock}}$.

Now we proceed to introduce our functionality $\mathcal{F}_{\mathsf{NiACS}}$ as shown in Fig. 3. Roughly speaking, $\mathcal{F}_{\mathsf{NiACS}}$ consists of four parts: *Initialization, Users' transactions, Aggregation, and Ledger*. In initialization part, parties register on the ledger while stating their roles (user/aggregator), and users need to state their initial amounts. The registration information will be recorded into the genesis block. After the system is bootstrapped, the parties can also register. Here, for simplicity, we assume that parties only register at the beginning. In users' transactions part, users can submit payment requests to $\mathcal{F}_{\mathsf{NiACS}}$, and if a payment request is valid, $\mathcal{F}_{\mathsf{NiACS}}$ will generate the corresponding transaction and send it to the aggregators. In aggregation part, aggregators can send aggregation requests to $\mathcal{F}_{\mathsf{NiACS}}$, and if an aggregation request is valid, $\mathcal{F}_{\mathsf{NiACS}}$ will aggregate corresponding transactions and return the aggregate transaction to the aggregator. In ledger part, any party can read the ledger. Moreover, the adversary can (1) directly submit transactions without being aggregated to ledger, and (2) permute the buffer, which reflects his influence on the delivery. Next, we introduce each part of $\mathcal{F}_{\mathsf{NiACS}}$ in detail.

Initialization. In Fig. 4, we describe the initialization process. $\mathcal{F}_{\mathsf{NiACS}}$ first initializes state $:= \epsilon$, buffer $:= \epsilon$ and counter $:= 0$. Besides, $\mathcal{F}_{\mathsf{NiACS}}$ initializes

Functionality $\mathcal{F}_{\mathsf{NiACS}}$: Initialization

The functionality is parameterized with variables $\mathcal{L}_{\mathsf{coin}}$, $\mathcal{L}_{\mathsf{TX}}$, $\mathcal{L}_{\mathcal{A}}$, state, buffer, counter and a constant T. Initially, $\mathcal{L}_{\mathsf{coin}} := \emptyset$, $\mathcal{L}_{\mathsf{TX}} := \emptyset$, $\mathcal{L}_{\mathcal{A}} := \emptyset$, state $:= \varepsilon$, buffer $:= \varepsilon$ and counter $:= 0$. Upon receiving a message $\langle \textsc{Corrupt}, \mathsf{sid}, P_t \rangle$ from adversary $\mathcal{S}$, add P_t to $\mathcal{L}_{\mathcal{A}}$.

Initialization: Upon receiving a message $\langle \textsc{Registration}, \mathsf{sid}, \mathsf{Infor}_{\mathsf{reg}}, P_k \rangle$ from P_k, do:

1: **if** There is $(P_k, \cdot, \cdot)$ in buffer **then** Ignore the message;
2: **else**
3: Parse $\mathsf{Infor}_{\mathsf{reg}} := (\mathsf{role}, v_k)$ where $\mathsf{role} \in \{\text{user}, \text{aggregator}\}$;
4: **if** $\mathsf{role} = \text{user} \wedge v_k \in [0, v_{\max}]$ **then**
5: Send $\langle \textsc{Registration}, \mathsf{sid}, \text{user}, v_k, P_k \rangle$ to adversary $\mathcal{S}$;
6: **if** Receive message $\langle \textsc{Registrated}, \mathsf{sid}, (P_k, \text{user}), \mathsf{cid}_k \rangle$ from $\mathcal{S}$ **then**
7: Add $(\mathsf{cid}_k, v_k, P_k, P_k)$ to $\mathcal{L}_{\mathsf{coin}}$;
8: buffer := buffer$||(P_k, \text{user}, (\mathsf{cid}_k, v_k))$;
9: **else if** $\mathsf{role} = \text{aggregator} \wedge v_k = \perp$ **then**
10: Send $\langle \textsc{Registration}, \mathsf{sid}, \text{aggregator}, \perp, P_k \rangle$ to adversary $\mathcal{S}$;
11: **if** Receive message $\langle \textsc{Registrated}, \mathsf{sid}, (P_k, \text{aggregator}), \perp \rangle$ from $\mathcal{S}$ **then**
12: buffer := buffer$||(P_k, \text{aggregator}, \perp)$;
13: Upon receiving $\langle \textsc{Initialized}, \mathsf{sid} \rangle$ from $\mathcal{S}$, (state, buffer, counter) := $\textsc{ExtendState}(\mathsf{state}, \mathsf{buffer}, \mathsf{T}, \mathsf{counter})$, send $\langle \textsc{Initialized}, \mathsf{sid}, \mathsf{cid}_k / \perp \rangle$ to P_k.

Fig. 4. Ideal functionality of NiACS (Initialization).

three lists, $\mathcal{L}_{\mathsf{coin}}$, $\mathcal{L}_{\mathsf{TX}}$ and $\mathcal{L}_{\mathcal{A}}$ for recording coins, transactions and corrupted parties, respectively. At the beginning, $\mathcal{F}_{\mathsf{NiACS}}$ will be notified which parties are corrupted by receiving the message $\langle \textsc{Corrupt}, \mathsf{sid}, P_t \rangle$ from adversary $\mathcal{S}$. Then P_t will be recorded in $\mathcal{L}_{\mathcal{A}}$. For registration, a party P_k will send the registration information to $\mathcal{F}_{\mathsf{NiACS}}$. In the registration information, P_k needs to declare his role. If P_k acts as a user, he needs to declare his initial amount. Once receiving a registration message $\langle \textsc{Registration}, \mathsf{sid}, \mathsf{Infor}_{\mathsf{reg}}, P_k \rangle$ from P_k, $\mathcal{F}_{\mathsf{NiACS}}$ first checks if the party has registered before. If so, $\mathcal{F}_{\mathsf{NiACS}}$ ignores the message. Note that a party can only register as one role. If the party wants to register as a user and the corresponding initial amount v_k is in a valid range, $\mathcal{F}_{\mathsf{NiACS}}$ sends $\langle \textsc{Registration}, \mathsf{sid}, \text{user}, v_k, P_k \rangle$ to adversary $\mathcal{S}$. Then $\mathcal{S}$ will generate a coin identifier cid_k for the initial coin, and send a message $\langle \textsc{Registrated}, \mathsf{sid}, (P_k, \text{user}), \mathsf{cid}_k \rangle$ to $\mathcal{F}_{\mathsf{NiACS}}$. At this point, the initial coin of P_k is created, and $\mathcal{F}_{\mathsf{NiACS}}$ will add the coin $(\mathsf{cid}_k, v_k, P_k, P_k)$ into $\mathcal{L}_{\mathsf{coin}}$ in the format of (identifier, value, owner, creator) and adds $(P_k, \text{user}, (\mathsf{cid}_k, v_k))$ into buffer. If P_k is registered as an aggregator, after receiving the agreement from $\mathcal{S}$, $\mathcal{F}_{\mathsf{NiACS}}$ adds $(P_k, \text{aggregator}, \perp)$ into buffer. We allow the adversary to decide when the registration is finished, and thus once receiving $\langle \textsc{Initialized}, \mathsf{sid} \rangle$ from $\mathcal{S}$, $\mathcal{F}_{\mathsf{NiACS}}$ executes ExtendState to get the genesis state. $\mathcal{F}_{\mathsf{NiACS}}$ returns $\langle \textsc{Initialized}, \mathsf{sid}, \mathsf{cid}_k \rangle$ to inform P_k of successful registration.

Users' Transactions. In Fig. 5, we describe the process of a user submitting a transaction. A user P_k forwards a payment request $\langle \textsc{Submit}, \mathsf{sid}, P_k, \mathcal{L}_{\mathsf{Agg}}, \{\mathsf{cid}_i\}, \{(\hat{P}_j, \hat{v}_j)\} \rangle$ to $\mathcal{F}_{\mathsf{NiACS}}$ to initiate the process of generating a transaction. In the payment request, $\mathcal{L}_{\mathsf{Agg}}$[5] is a list of aggregators who

[5] Environment $\mathcal{Z}$ can abstract the situation that a party can send a transaction to different aggregator sets at different times, by assigning different aggregator lists for a transaction.

Functionality $\mathcal{F}_{\mathsf{NiACS}}$: Users' transactions

Upon receiving a message $\langle \textsc{Submit}, \mathsf{sid}, P_k, \mathcal{L}_{\mathsf{Agg}}, \{\mathsf{cid}_i\}, \{(\hat{P}_j, \hat{v}_j)\}\rangle$ from P_k, do the following:

1: $(\mathsf{state}, \mathsf{buffer}, \mathsf{counter}) := \textsc{ExtendState}(\mathsf{state}, \mathsf{buffer}, \mathsf{T}, \mathsf{counter})$;
2: Set $\mathsf{Info} := (\{\mathsf{cid}_i\}, \{(\hat{P}_j, \hat{v}_j)\})$, and retrieve $(\mathsf{TX}, \mathcal{L}^*_{\mathsf{Agg}}, \mathsf{Info})$ from $\mathcal{L}_{\mathsf{TX}}$ according to Info;
3: **if** there is no $(\mathsf{TX}, \mathcal{L}^*_{\mathsf{Agg}}, \mathsf{Info})$ in $\mathcal{L}_{\mathsf{TX}}$ **then**
4: **for** each cid_i **do**
5: Retrieve $(\mathsf{cid}_i, v_i, P_i, \cdot)$ from $\mathcal{L}_{\mathsf{coin}}$;
6: **if** $P_i \neq \perp$ **then**
7: **if** $P_k \neq P_i \vee \mathsf{cid}_i \notin \mathsf{state}||\mathsf{buffer}$ **then** Ignore the message;
8: **else**
9: **if** $P_k \notin \mathcal{L}_{\mathcal{A}} \vee \mathsf{cid}_i \notin \mathsf{state}||\mathsf{buffer}$ **then** Ignore the message;
10: **if** $\sum_i v_i \neq \sum_j \hat{v}_j \vee \exists j, s.t., \hat{v}_j \notin [0, v_{\mathsf{max}}]$ **then** Ignore the message;
11: **for** each j **do**
12: **if** $P_k \in \mathcal{L}_{\mathcal{A}} \vee \hat{P}_j \in \mathcal{L}_{\mathcal{A}}$ **then** Send $\langle \textsc{RequestID}, \mathsf{sid}, (\hat{P}_j, \hat{v}_j)\rangle$ to $\mathcal{S}$;
13: **else** Send $\langle \textsc{RequestID}, \mathsf{sid}, \perp\rangle$ to $\mathcal{S}$;
14: Receive $\langle \textsc{ResponseID}, \mathsf{sid}, \hat{\mathsf{cid}}_j\rangle$ from $\mathcal{S}$;
15: Add $(\hat{\mathsf{cid}}_j, \hat{v}_j, \hat{P}_j, P_k)$ to $\mathcal{L}_{\mathsf{coin}}$;
16: Send $\langle \textsc{ResponseID}, \mathsf{sid}, \{\hat{\mathsf{cid}}_j\}\rangle$ to P_k;
17: Generate $\mathsf{TX} := (\{\mathsf{cid}_i\}, \{\hat{\mathsf{cid}}_j\})$;
18: $\mathcal{L}^*_{\mathsf{Agg}} := \emptyset$;
19: **for** each $\mathsf{Agg}_t \in \mathcal{L}_{\mathsf{Agg}}$ **do**
20: **if** Agg_t is an aggregator **then**
21: **if** $P_k \in \mathcal{L}_{\mathcal{A}}$ **then**
22: Send $\langle \textsc{SendTX}, \mathsf{sid}, \mathsf{TX}, P_k, \mathsf{Agg}_t\rangle$ to $\mathcal{S}$;
23: **else**
24: Send $\langle \textsc{SendTX}, \mathsf{sid}, \perp, P_k, \mathsf{Agg}_t\rangle$ to $\mathcal{S}$;
25: **if** Receive $\langle \textsc{SendTX}, \mathsf{sid}, P_k, \mathsf{Agg}_t, \mathrm{OK}\rangle$ from $\mathcal{S}$ **then**
26: Send $\langle \textsc{ReceivedTX}, \mathsf{sid}, \mathsf{TX}\rangle$ to Agg_t;
27: Add Agg_t to $\mathcal{L}^*_{\mathsf{Agg}}$;
28: Write $(\mathsf{TX}, \mathcal{L}^*_{\mathsf{Agg}}, \mathsf{Info})$ to $\mathcal{L}_{\mathsf{TX}}$;
29: **else**
30: **for** each $\mathsf{Agg}_t \in \mathcal{L}_{\mathsf{Agg}}/\mathcal{L}^*_{\mathsf{Agg}}$ **do**
31: **if** Agg_t is an aggregator **then**
32: **if** $P_k \in \mathcal{L}_{\mathcal{A}}$ **then**
33: Send $\langle \textsc{SendTX}, \mathsf{sid}, \mathsf{TX}, P_k, \mathsf{Agg}_t\rangle$ to $\mathcal{S}$;
34: **else**
35: Send $\langle \textsc{SendTX}, \mathsf{sid}, \perp, P_k, \mathsf{Agg}_t\rangle$ to $\mathcal{S}$;
36: **if** Receive $\langle \textsc{SendTX}, \mathsf{sid}, P_k, \mathsf{Agg}_t, \mathrm{OK}\rangle$ from $\mathcal{S}$ **then**
37: Send $\langle \textsc{ReceivedTX}, \mathsf{sid}, \mathsf{TX}\rangle$ to Agg_t;
38: Add Agg_t to $\mathcal{L}^*_{\mathsf{Agg}}$;
39: Rewrite $(\mathsf{TX}, \mathcal{L}^*_{\mathsf{Agg}}, \mathsf{Info})$ to $\mathcal{L}_{\mathsf{TX}}$;

Fig. 5. Ideal functionality of NiACS (Users' transactions).

can receive the transaction and aggregate it, $\{\mathsf{cid}_i\}$ is the set of coins to be spent, and $\{(\hat{P}_j, \hat{v}_j)\}$ is the set of receivers and the corresponding values. Upon receiving a payment request, $\mathcal{F}_{\mathsf{NiACS}}$ will perform the following three steps:

– *Transaction validation*: $\mathcal{F}_{\mathsf{NiACS}}$ first needs to get the current state and check if P_k can spend all the coins in $\{\mathsf{cid}_i\}$. More specifically, for each cid_i, $\mathcal{F}_{\mathsf{NiACS}}$ retrieves $(\mathsf{cid}_i, v_i, P_i, \cdot)$ from $\mathcal{L}_{\mathsf{coin}}$ and checks if $P_i = P_k$. If so, P_k can spend this coin. Note that it is possible that $P_i = \perp$, which means that the coin is generated by a corrupted party and has no designated owner. In this case, any corrupted party in $\mathcal{L}_{\mathcal{A}}$ can spend this coin. For all i, if P_k can indeed spend coin cid_i and all these coins are in state, $\mathcal{F}_{\mathsf{NiACS}}$ further checks if all the output values are valid (i.e., $\hat{v}_j \in [0, v_{\mathsf{max}}]$ for all j) and the transaction

is balanced (i.e., $\sum_i v_i = \sum_j \hat{v}_j$). If all above verifications pass, $\mathcal{F}_{\mathsf{NiACS}}$ will generate new coins for the receivers as below.

- *Creating new coins*: $\mathcal{F}_{\mathsf{NiACS}}$ sends a message to request $\mathcal{S}$ to generate the identifiers of new coins. Since the adversary $\mathcal{S}$ knows the values/amounts of coins generated or received by corrupted parties, $\mathcal{F}_{\mathsf{NiACS}}$ will send $\langle \textsc{RequestID}, \mathsf{sid}, (\hat{P}_j, \hat{v}_j)\rangle$ to the adversary if the sender P_k or a receiver $\hat{P}_j$ is corrupted. Otherwise, $\mathcal{F}_{\mathsf{NiACS}}$ will send $\langle \textsc{RequestID}, \mathsf{sid}, \perp\rangle$ to capture that both the receiver and the amount of the coin are hidden from $\mathcal{S}$. Upon receiving the response $\langle \textsc{ResponseID}, \mathsf{sid}, \hat{\mathsf{cid}}_j\rangle$ from $\mathcal{S}$, where $\hat{\mathsf{cid}}_j$ is the identifier of the new coin, $\mathcal{F}_{\mathsf{NiACS}}$ records $(\hat{\mathsf{cid}}_j, \hat{v}_j, \hat{P}_j, P_k)$ in $\mathcal{L}_{\mathsf{coin}}$. After getting the identifiers of all the new coins, $\mathcal{F}_{\mathsf{NiACS}}$ represents the transaction as $\mathsf{TX} := (\{\mathsf{cid}_i\}, \{\hat{\mathsf{cid}}_j\})$, which will be sent to the designated aggregators in $\mathcal{L}_{\mathsf{Agg}}$ as below.
- *Sending* TX *to aggregators*: Although an aggregator list $\mathcal{L}_{\mathsf{Agg}}$ for a transaction TX is assigned in the input message, whether or not an aggregator can receive the transaction TX is eventually determined by the adversary $\mathcal{S}$, so the set of aggregators who indeed receive the transaction is a subset of the assigned aggregators. When receiving a transaction for the first time, $\mathcal{F}_{\mathsf{NiACS}}$ initializes a list $\mathcal{L}^*_{\mathsf{Agg}}$ to record the aggregators who will finally receive the transaction. For each aggregator Agg_t in $\mathcal{L}_{\mathsf{Agg}}$, $\mathcal{F}_{\mathsf{NiACS}}$ will ask $\mathcal{S}$ if Agg_t can receive the transaction through a message $\langle \textsc{SendTX}, \mathsf{sid}, \mathsf{TX}, P_k, \mathsf{Agg}_t\rangle$. Note that if the sender P_k is corrupted, $\mathcal{S}$ can know which transaction is required to be sent to an aggregator, so $\mathsf{TX} \neq \perp$. But when the sender is honest, $\mathcal{S}$ cannot know the information about the transaction, and thus $\mathsf{TX} = \perp$. Upon receiving $\langle \textsc{SendTX}, \mathsf{sid}, P_k, \mathsf{Agg}_t, \mathrm{OK}\rangle$, $\mathcal{F}_{\mathsf{NiACS}}$ will send $\langle \textsc{ReceivedTX}, \mathsf{sid}, \mathsf{TX}\rangle$ to Agg_t and add Agg_t to $\mathcal{L}^*_{\mathsf{Agg}}$. After sending the transaction to the allowed aggregators, $\mathcal{F}_{\mathsf{NiACS}}$ needs to record the transaction into $\mathcal{L}_{\mathsf{TX}}$, including its transaction identifier TX, aggregators receiving the transaction $\mathcal{L}^*_{\mathsf{Agg}}$ and its details $(\{\mathsf{cid}_i\}, \{(\hat{P}_j, \hat{v}_j)\})$. In addition, P_k can repeatedly input the same transaction, but with different aggregator lists. In this case, $\mathcal{F}_{\mathsf{NiACS}}$ only sends the transaction to the new aggregators in $\mathcal{L}_{\mathsf{Agg}}/\mathcal{L}^*_{\mathsf{Agg}}$.

Aggregation. In Fig. 6, we show how $\mathcal{F}_{\mathsf{NiACS}}$ aggregates certain transactions and puts the aggregate transactions into buffer. An aggregator Agg_k can ask $\mathcal{F}_{\mathsf{NiACS}}$ to aggregate the transactions in $\{\mathsf{TX}_t\}$ and put the aggregate transaction into the ledger by submitting an aggregation request $\langle \textsc{Aggregate}, \mathsf{sid}, \{\mathsf{TX}_t\}\rangle$. Upon receiving the request, $\mathcal{F}_{\mathsf{NiACS}}$ initializes three empty lists $\mathcal{L}_{\mathsf{inp}}$, $\mathcal{L}_{\mathsf{outp}}$ and $\mathcal{L}_{\mathsf{cut}}$. $\mathcal{L}_{\mathsf{inp}}$ and $\mathcal{L}_{\mathsf{outp}}$ are used to record the identifiers of spent coins and created coins in the aggregate transaction, respectively. For each transaction TX_t in $\{\mathsf{TX}_t\}$, $\mathcal{F}_{\mathsf{NiACS}}$ first checks if the aggregator Agg_k indeed received the transaction according to the records in $\mathcal{L}_{\mathsf{TX}}$. If not, $\mathcal{F}_{\mathsf{NiACS}}$ will ignore the transaction TX_t, otherwise parses TX_t as $(\{\mathsf{cid}_i\}, \{\hat{\mathsf{cid}}_j\})$ and adds all identifiers in $\{\mathsf{cid}_i\}$ and $\{\hat{\mathsf{cid}}_j\}$ to $\mathcal{L}_{\mathsf{inp}}$ and $\mathcal{L}_{\mathsf{outp}}$ respectively. At this point, the transactions in $\{\mathsf{TX}_t\}$ received by the aggregator Agg_k have been aggregated into $(\mathcal{L}_{\mathsf{inp}}, \mathcal{L}_{\mathsf{outp}})$. Then cut-through proceeds as follows: For each spent coin $\mathsf{cid}_i \in \mathcal{L}_{\mathsf{inp}}$, $\mathcal{F}_{\mathsf{NiACS}}$ checks if it belongs to $\mathcal{L}_{\mathsf{outp}}$, if so, cid_i will be removed from $\mathcal{L}_{\mathsf{inp}}$ and $\mathcal{L}_{\mathsf{outp}}$. Obviously, if a cut happens on a coin that is created or received by a corrupted party, the adversary $\mathcal{S}$ will

Functionality $\mathcal{F}_{\mathsf{NiACS}}$: Aggregation

Upon receiving a message $\langle \textsc{Aggregate}, \mathsf{sid}, \{\mathsf{TX}_t\}\rangle$ from an aggregator Agg_k, do:

1: $(\mathsf{state}, \mathsf{buffer}, \mathsf{counter}) := \textsc{ExtendState}(\mathsf{state}, \mathsf{buffer}, \mathsf{T}, \mathsf{counter})$;
2: Initialize $\mathcal{L}_{\mathsf{inp}} := \emptyset$, $\mathcal{L}_{\mathsf{outp}} := \emptyset$, $\mathcal{L}_{\mathsf{cut}} := \emptyset$, $c := 0$;
3: **for** each TX_t **do**
4: Retrieve $(\mathsf{TX}_t, \mathcal{L}^*_{\mathsf{Agg},t}, \mathsf{Info}_t)$ from $\mathcal{L}_{\mathsf{TX}}$;
5: **if** $\mathsf{Agg}_k \in \mathcal{L}^*_{\mathsf{Agg},t}$ **then**
6: Parse $\mathsf{TX}_t := (\{\mathsf{cid}_i\}, \{\hat{\mathsf{cid}}_j\})$;
7: Add all $\{\hat{\mathsf{cid}}_j\}$ to $\mathcal{L}_{\mathsf{outp}}$;
8: **for** each cid_i **do**
9: **if** $\mathsf{cid}_i \notin \mathcal{L}_{\mathsf{outp}}$ **then** Add cid_i to $\mathcal{L}_{\mathsf{inp}}$;
10: **else**
11: Remove cid_i from $\mathcal{L}_{\mathsf{outp}}$;
12: $c := c + 1$;
13: Retrieve $(\mathsf{cid}_i, v_i, P_i, P_k)$ from $\mathcal{L}_{\mathsf{coin}}$;
14: **if** $P_i \in \mathcal{L}_{\mathcal{A}} \vee P_k \in \mathcal{L}_{\mathcal{A}}$ **then** Add cid_i into $\mathcal{L}_{\mathsf{cut}}$;
15: $\mathsf{aTX} := (\mathcal{L}_{\mathsf{inp}}, \mathcal{L}_{\mathsf{outp}})$;
16: **if** all $\mathsf{cid}_i \in \mathcal{L}_{\mathsf{inp}}$ are in $\mathsf{state}||\mathsf{buffer}$ **then**
17: $\mathsf{buffer} := \mathsf{buffer}||\mathsf{aTX}$;
18: Send $\langle \textsc{AggTX}, \mathsf{sid}, \mathsf{aTX}\rangle$ to Agg_k and $\langle \textsc{AggTX}, \mathsf{sid}, \mathsf{Agg}_k, \mathsf{aTX}, c, \mathcal{L}_{\mathsf{cut}}\rangle$ to $\mathcal{S}$;

Fig. 6. Ideal functionality of NiACS (Aggregation).

Functionality $\mathcal{F}_{\mathsf{NiACS}}$: Ledger

Upon receiving a message $\langle \textsc{SubmitToLedger}, \mathsf{sid}, \{\mathsf{cid}_i\}, \{(\hat{\mathsf{cid}}_j, \hat{P}_j, \hat{v}_j)\}\rangle$ from adversary $\mathcal{S}$ (on behalf of a corrupted user or a corrupted aggregator), do the following:

1: $(\mathsf{state}, \mathsf{buffer}, \mathsf{counter}) := \textsc{ExtendState}(\mathsf{state}, \mathsf{buffer}, \mathsf{T}, \mathsf{counter})$;
2: **for** each cid_i **do**
3: Retrieve $(\mathsf{cid}_i, v_i, P_i, \cdot)$ from $\mathcal{L}_{\mathsf{coin}}$;
4: **if** $(P_i \neq \perp \wedge P_i \notin \mathcal{L}_{\mathcal{A}}) \vee \mathsf{cid}_i \notin \mathsf{state}||\mathsf{buffer}$ **then** Ignore the message;
5: **if** $\sum_i v_i \neq \sum_j \hat{v}_j \vee \exists j, s.t., \hat{v}_j \notin [0, v_{\mathsf{max}}]$ **then** Ignore the message;
6: **for** each j **do**
7: Add $(\hat{\mathsf{cid}}_j, \hat{v}_j, \hat{P}_j, \perp)$ to $\mathcal{L}_{\mathsf{coin}}$;
8: Generate $\mathsf{TX} := (\{\mathsf{cid}_i\}, \{\hat{\mathsf{cid}}_j\})$;
9: $\mathsf{buffer} := \mathsf{buffer}||\mathsf{TX}$;

Upon receiving a message $\langle \textsc{Read}, \mathsf{sid}\rangle$ from a party P_k or adversary $\mathcal{S}$, do the following:

1: $(\mathsf{state}, \mathsf{buffer}, \mathsf{counter}) := \textsc{ExtendState}(\mathsf{state}, \mathsf{buffer}, \mathsf{T}, \mathsf{counter})$;
2: **if** $P_k \in \mathcal{L}_{\mathcal{A}}$ or the requester is $\mathcal{S}$ **then**
3: Send $\langle \textsc{Read}, \mathsf{sid}, (\mathsf{state}, \mathsf{buffer})\rangle$ to the requestor;
4: **else**
5: Send $\langle \textsc{Read}, \mathsf{sid}, \mathsf{state}\rangle$ to the requestor;

Upon receiving a message $\langle \textsc{Permute}, \mathsf{sid}, \pi\rangle$ from adversary $\mathcal{S}$, apply the permutation π on the elements of buffer;

Fig. 7. Ideal functionality of NiACS (Ledger).

be aware of it. Thus, $\mathcal{F}_{\mathsf{NiACS}}$ uses $\mathcal{L}_{\mathsf{cut}}$ to record these coins. We allow $\mathcal{S}$ to know how many coins (related to honest parties and corrupted parties) are cut, and denotes the number as a variable c. After executing the above process, $\mathcal{L}_{\mathsf{inp}}$ and $\mathcal{L}_{\mathsf{outp}}$ constitute the aggregate transaction aTX. Finally, if all the input coins of aTX are in state, $\mathcal{F}_{\mathsf{NiACS}}$ adds aTX into buffer and sends $\langle \textsc{AggTX}, \mathsf{sid}, \mathsf{aTX}\rangle$ and $\langle \textsc{AggTX}, \mathsf{sid}, \mathsf{Agg}_k, \mathsf{aTX}, c, \mathcal{L}_{\mathsf{cut}}\rangle$ to Agg_k and $\mathcal{S}$, respectively.

Ledger. For the basic interfaces of a ledger, we follow the ledger functionality defined in [29]. To be self-contained, we show the ledger functionality in Fig. 7. More specifically, we follow their definition in the following three aspects: (1) The abstraction of consensus layer: $\mathcal{F}_{\mathsf{NiACS}}$ executes the procedure ExtendState shown in Algorithm 1 to extend the state, which is an abstract of consensus layer; (2) The way of parties and adversary reading the ledger: for an honest party, $\mathcal{F}_{\mathsf{NiACS}}$ only provides state, but for a corrupted party, $\mathcal{F}_{\mathsf{NiACS}}$ give state and buffer; (3) Allowing adversary to permute buffer: to abstract the case where adversary can delay the delivery of transactions in the network, $\mathcal{F}_{\mathsf{NiACS}}$ receives a permutation π from the adversary, and apply the permutation on buffer.

In addition, we allow the adversary $\mathcal{S}$ to directly submit transactions to the ledger. Note that in the ledger functionality defined in [29], both honest and corrupted parties can directly submit transactions to the ledger. Whereas, in our NiACS, each honest sender's transaction first needs to be aggregated and then submitted to the ledger by the designated aggregators. Therefore, honest parties who intend to protect privacy will not directly submit transactions to the ledger.

Security Properties Captured By our Definition. Informally, our ideal functionality $\mathcal{F}_{\mathsf{NiACS}}$ captures the following security properties: *inflation-resistance, theft-resistance, transaction indistinguishability, and unlinkability.* More specifically, $\mathcal{F}_{\mathsf{NiACS}}$ requires parties to register the initial amounts and spend the coins with enough value, which implies inflation-resistance. For each coin, $\mathcal{F}_{\mathsf{NiACS}}$ records its owner and only allows the owner to spend the coin, which means theft-resistance. A transaction consists of an input list and an output list, each containing multiple coin identifiers cid. Since the coin identifiers are pseudorandom, the transaction amount is hidden and change coins and output coins are indistinguishable obviously, which provides transaction indistinguishability. Moreover, the transactions contained in the state of the ledger is in an aggregate form, so the irrelevant parties cannot learn which coins belong to the same transaction, i.e., unlinkability.

4 Our Non-interactive Aggregate Cash System

In this section, we present the details of our protocol Π_{NiACS}. First, we propose a new variant of ElGamal encryption that is important for realizing non-interactive payments. Then, we introduce the ideal functionalities and auxiliary algorithms used through our design. At last, we present our protocol based on these functionalities, auxiliary algorithms, and the variant of ElGamal encryption.

4.1 New Variant of ElGamal Encryption

Inspired by the twisted ElGamal encryption [16], we propose a new variant of ElGamal encryption scheme shown in Fig. 8. The ciphertext is of the form $\left(pk^r, g^{H(g^r)}h^m\right)$ for a message $m \in \mathbb{Z}_p$. Therefore, the receiver can recover both the randomness $H(g^r)$ and the message m by using the secret key sk, which

plays a crucial role for our design. We remark that, as shown in [11,16,19], the encrypted message m can be efficiently recovered from h^m when the message space is small, e.g., by brute-force enumeration as in [11,19], or Shanks's algorithm as in [16]. We show the security of our new variant of ElGamal encryption by Theorem 1, and please find the proof in the full version.

$\underline{\mathsf{Setup}(1^\lambda)}$: runs $(\mathbb{G}, g, p) \leftarrow \mathsf{GroupGen}(1^\lambda)$, picks $h \xleftarrow{\$} \mathbb{G}^*$, chooses a hash function $H : \mathbb{G} \mapsto \mathbb{Z}_p$, and outputs $pp := (\mathbb{G}, g, h, p, H)$ as the global public parameters.
$\underline{\mathsf{KeyGen}(pp)}$: chooses $sk \xleftarrow{\$} \mathbb{Z}_p$, computes $pk := g^{sk}$, and outputs a key pair (sk, pk).
$\underline{\mathsf{Enc}(pk, m)}$: chooses $r \xleftarrow{\$} Z_p$, computes $X := pk^r$, $\alpha := H(g^r)$, $Y := g^\alpha h^m$, and outputs the ciphertext $C := (X, Y)$.
$\underline{\mathsf{Dec}(sk, C)}$: parses $C := (X, Y)$, computes $\alpha := H(X^{\frac{1}{sk}})$, $h^m := \frac{Y}{g^\alpha}$, recovers m from h^m.

Fig. 8. New variant of ElGamal.

Theorem 1. *Assuming that the Divisible Computational Diffie-Hellman (DCDH) problem is hard*[6]*, the proposed encryption scheme is IND-CPA secure in the random oracle model.*

4.2 Ideal Functionalities and Auxiliary Algorithms

We design our protocol in a hybrid model. To ease the understanding of our protocol, we first recall the subroutine ideal functionalities invoked in our design and represent some specific processes as auxiliary algorithms. We give the details of these functionalities and auxiliary algorithms in the full version.

Ideal Functionalities. The ideal functionalities used throughout our design can be divided into two categories; the first is used for transaction layer, while the second is for consensus layer. For the former, it is summarized in Table 1. For the latter, we note that the functionality $\mathcal{F}_{\mathsf{NiACS}}$ defined in Sect. 3 is a private ledger, and thus can be seen as a "private" version of $\mathcal{F}_{\mathsf{Ledger}}$ defined in [29]. Therefore, we focus on designing privacy-preserving transaction layer while assuming there is a secure consensus layer. In this work, we design our protocol Π_{NiACS} by leveraging the functionality $\mathcal{F}_{\mathsf{Ledger}}$.

Auxiliary Algorithms. For the auxiliary algorithms, we divide them into two categories: one-time addresses and construction of transactions.

One-Time Addresses. In our system, each user has a permanent address $\mathsf{Addr} := g^{\mathsf{Key}}$, where $\mathsf{Key} \xleftarrow{\$} \mathbb{Z}_p$ is the associated secret key and g is a generator of the cyclic group $\mathbb{G}$ with order p. We use one-time address to hide the identity of a user in the real world. One-time addresses/secret keys are generated as follows:

[6] Informally, the DCDH assumption means that, given a tuple (g, g^a, g^b), where g is a generator of a cyclic group $\mathbb{G}$ with prime order p and $a, b \xleftarrow{\$} \mathbb{Z}_p$, the probability of computing $g^{a/b}$ is negligible.

– GenOTAddr is to generate a one-time address and its auxiliary string for a user with permanent address Addr. It takes Addr as input, and outputs a one-time address pk and corresponding auxiliary string R.
– GenOTKey is to generate a one-time spending key. It takes a permanent address/key pair $(\mathsf{Addr}, \mathsf{Key})$ and a one-time address/auxiliary string (pk, R) as input, if (pk, R) is derived from $(\mathsf{Addr}, \mathsf{Key})$, outputs the one-time secret key sk, otherwise outputs $\perp$.

Table 1. Ideal functionalities and descriptions

$\mathcal{F}_{\mathsf{NIZK}}^{\mathsf{addr}}$	Non-interactive zero-knowledge for language $\mathcal{L}_{\mathsf{addr}} := \{(pk, R) \mid \exists\ (\mathsf{Addr}, r)$, s.t. $pk = \mathsf{Addr}^{H(\mathsf{Addr}^r)}, R = g^r\}$, which is used to prove that the one-time address pk and its auxiliary string R are correctly generated
$\mathcal{F}_{\mathsf{NIZK}}^{\mathsf{zero}}$	Non-interactive zero-knowledge for language $\mathcal{L}_{\mathsf{zero}} := \{\mathsf{cm} \mid \exists\ r$, s.t. $\mathsf{cm} = g^r h^0\}$, which is used to prove that each excess part is a commitment to 0
$\mathcal{F}_{\mathsf{NIZK}}^{\mathsf{enc}}$	Non-interactive zero-knowledge for language $\mathcal{L}_{\mathsf{enc}} := \{(pk, (X, \mathsf{cm})) \mid \exists\ (r, v)$, s.t. $X = pk^r, \mathsf{cm} = g^{H(g^r)} h^v\}$, which is used to prove that the amount v is correctly encrypted
$\mathcal{F}_{\mathsf{SoK}}^{\mathsf{range}}$	Signature of knowledge for language $\mathcal{L}_{\mathsf{range}} := \{\mathsf{cm} \mid \exists\ (r, v)$, s.t. $\mathsf{cm} = g^r h^v \wedge v \in [0, v_{\mathsf{max}}]\}$, which is used to prove that the amount of each coin is within a valid range and to sign an address
$\mathcal{F}_{\mathsf{SoK}}^{\mathsf{key}}$	Signature of knowledge for language $\mathcal{L}_{\mathsf{key}} := \{pk \mid \exists\ sk$, s.t. $pk = g^{sk}\}$, which is used to prove the knowledge of a spending key and to sign an excess part
$\mathcal{F}_{\mathsf{SMT}}$	Secure message transmission is for users to send transactions to aggregators

Construction of Transactions. The remaining algorithms are used to generate and verify transactions as follows:

– GenExcess is to generate the excess and offset. It takes as input all the commitments and their openings in both the input and output lists (i.e., $\{\mathsf{cm}_i, (\alpha_i, v_i)\}$ and $\{\hat{\mathsf{cm}}_j, (\hat{\alpha}_j, \hat{v}_j)\}$), then chooses an offset δ and computes the final excess $\tilde{E}$ and its randomness $\tilde{e}$. After that, it splits the excess $\tilde{E}$ into n parts, s.t., $\tilde{E} = \tilde{E}_1 \cdot \tilde{E}_2 \cdots \tilde{E}_n$, and outputs $(\{\tilde{E}_i, \tilde{e}_i\}, \delta)$.
– GenOutput and VerOutput are to generate and verify the output coins, respectively. For each output, GenOutput takes $(\hat{v}_j, \hat{\mathsf{Addr}}_j)$ as input, then generates and outputs a one-time address $\hat{\mathsf{cid}}_j := (\hat{pk}_j, \hat{R}_j)$, a proof $\hat{\pi}_j^{\mathsf{addr}}$ that the one-time address is correctly generated, a ciphertext $(\hat{X}_j, \hat{\mathsf{cm}}_j)$ of v_j, a proof $\hat{\pi}_j^{\mathsf{enc}}$ that $(\hat{X}_j, \hat{\mathsf{cm}}_j)$ is correctly generated, a signature $\hat{\sigma}_j^{\mathsf{range}}$ on $(\hat{pk}_j, \hat{R}_j)$ and the randomness $\hat{\alpha}_j$ of $\hat{\mathsf{cm}}_j$. VerOutput takes $(\hat{\mathsf{cid}}_j, \hat{\pi}_j^{\mathsf{addr}}, (\hat{X}_j, \hat{\mathsf{cm}}_j), \hat{\pi}_j^{\mathsf{enc}}, \hat{\sigma}_j^{\mathsf{range}})$ as input, then outputs 1 if $\hat{\pi}_j^{\mathsf{addr}}$, $\hat{\pi}_j^{\mathsf{enc}}$ and $\hat{\sigma}_j^{\mathsf{range}}$ are valid, and 0 otherwise.

– GenInput and VerInput are used to generate and verify the proof of spending key for each input coin, respectively. GenInput takes as inputs a partial excess $\tilde{E}_i$ and the associated randomness $\tilde{e}_i$, a coin identifier cid_i (i.e., one-time address) and the corresponding one-time spending key sk_i, then outputs a proof $\tilde{\pi}_i^{\mathsf{zero}}$ that $\tilde{E}_i$ is a commitment to 0 and a signature σ_i^{key} that proves the knowledge of sk_i and binds the input coin to $\tilde{E}_i$. VerInput takes $(\mathsf{cid}_i, (\tilde{E}_i, \tilde{\pi}_i^{\mathsf{zero}}), \sigma_i^{\mathsf{key}})$ as input, and outputs 1 if both $\tilde{\pi}_i^{\mathsf{zero}}$ and σ_i^{key} are valid, otherwise returns 0.
– Aggregate is to aggregate a valid individual transaction TX with an (aggregate) transaction $(\mathcal{L}_I, \mathcal{L}_O, \mathcal{L}_K, \Delta)$. It takes as input a transaction TX, an input list $\mathcal{L}_I$, an output list $\mathcal{L}_O$, a kernel list $\mathcal{L}_K$ and Δ, then outputs a new aggregate transaction $(\mathcal{L}_I, \mathcal{L}_O, \mathcal{L}_K, \Delta)$.

4.3 Description of Π_{NiACS}

Given the above ideal functionalities and auxiliary algorithms, we show the specification of our Π_{NiACS} in Fig. 9, Fig. 10, Fig. 11 and Fig. 12. Please refer to the full version for the detailed description.

Protocol Π_{NiACS}: Initialization

A party P_k, upon receiving message $\langle \text{Registration}, \mathsf{sid}, \mathsf{Infor}_{\mathsf{reg}}, P_k \rangle$ from $\mathcal{Z}$, does the following:

1: Parse $\mathsf{Infor}_{\mathsf{reg}} := (\mathsf{role}, v_k)$;
2: **if** $\mathsf{role} = \text{user} \wedge v_k \in [0, v_{\mathsf{max}}]$ **then**
3: $\quad \mathsf{Key}_k \xleftarrow{\$} \mathbb{Z}_p$, $\mathsf{Addr}_k := g^{\mathsf{Key}_k}$;
4: $\quad (pk_k, R_k) :=$GenOTAddr$(\mathsf{sid}, \mathsf{ssid}, \mathsf{Addr}_k)$, and set $\mathsf{cid}_k := (pk_k, R_k)$;
5: $\quad$ Randomly choose $\alpha_k \xleftarrow{\$} \mathbb{Z}_p$, and compute $\mathsf{cm}_k := g^{\alpha_k} h^{v_k}$;
6: $\quad \mathsf{Initcoin} := (\mathsf{Addr}_k, (\mathsf{cid}_k, \mathsf{cm}_k), (\alpha_k, v_k))$;
7: $\quad$ Invoke $\mathcal{F}_{\mathsf{Ledger}}$ with $\langle \text{Registration}, \mathsf{sid}, (P_k, \text{user}, \mathsf{Initcoin}) \rangle$;
8: $\quad$ Upon receiving $\langle \text{Initialized}, \mathsf{sid} \rangle$ from $\mathcal{F}_{\mathsf{Ledger}}$, output $\langle \text{Initialized}, \mathsf{sid}, \mathsf{cid}_k \rangle$ to $\mathcal{Z}$;
9: **else if** $\mathsf{role} = \text{aggregator} \wedge v_k = \perp$ **then**
10: $\quad$ Invoke $\mathcal{F}_{\mathsf{Ledger}}$ with $\langle \text{Registration}, \mathsf{sid}, (P_k, \text{aggregator}, \perp) \rangle$;
11: $\quad$ Upon receiving $\langle \text{Initialized}, \mathsf{sid} \rangle$ from $\mathcal{F}_{\mathsf{Ledger}}$, output $\langle \text{Initialized}, \mathsf{sid} \rangle$ to $\mathcal{Z}$;
12: **else** Ignore the message;

Fig. 9. Our Π_{NiACS} supporting non-interactive payments (Initialization).

Protocol Π_{NiACS}: Users' transactions

A party P_k with permanent address and key $(\mathsf{Addr}, \mathsf{Key})$, upon receiving a message $\langle \textsc{Submit}, \mathsf{sid}, P_k, \mathcal{L}_{\mathsf{Agg}}, \{\mathsf{cid}_i\}, \{(\hat{P}_j, \hat{v}_j)\}\rangle$ from $\mathcal{Z}$, does the following:

1: Invoke $\mathcal{F}_{\mathsf{Ledger}}$ with $\langle \textsc{Read}, \mathsf{sid}\rangle$, get $\langle \textsc{Read}, \mathsf{sid}, \mathsf{state}\rangle$ (A corrupted P_k will also get buffer);
2: Initialize $\mathsf{in} := 0$, $\mathsf{out} := \sum_j \hat{v}_j$, $\mathcal{I} := \emptyset$, $\mathcal{I}' := \emptyset$;
3: **for** each cid_i **do**
4: **if** There is no cid_i in state **then** Ignore the message;
5: **else**
6: Retrieve $(\mathsf{cid}_i, (X_i, \mathsf{cm}_i))$ from state;
7: $sk_i := \textsc{GenOTKey}(\mathsf{sid}, \mathsf{ssid}, \mathsf{Addr}, \mathsf{Key}, \mathsf{cid}_i)$;
8: **if** $sk_i = \perp$ **then** Ignore the message;
9: **else**
10: $(\alpha_i, v_i) := \mathsf{Dec}(sk_i; (X_i, \mathsf{cm}_i))$;
11: **if** $v_i \notin [0, v_{\mathsf{max}}]$ **then** Ignore the message;
12: **else** $\mathsf{in} := \mathsf{in} + v_i$;
13: **if** $\mathsf{in} = \mathsf{out}$ **then**
14: **for** each j **do**
15: Retrieve $(\hat{P}_j, \hat{\mathsf{Addr}}_j)$ from state;
16: $((\hat{\mathsf{cid}}_j, \hat{\pi}_j^{\mathsf{addr}}, (\hat{X}_j, \hat{\mathsf{cm}}_j), \hat{\pi}_j^{\mathsf{enc}}, \hat{\sigma}_j^{\mathsf{range}}), \hat{\alpha}_j) := \textsc{GenOutput}(\mathsf{sid}, \mathsf{ssid}, (\hat{v}_j, \hat{\mathsf{Addr}}_j))$;
17: $(\{\tilde{E}_i, \tilde{e}_i\}, \delta) := \textsc{GenExcess}(\{\mathsf{cm}_i, \alpha_i, v_i\}, \{\hat{\mathsf{cm}}_j, \hat{\alpha}_j, \hat{v}_j\})$;
18: **for** each i **do**
19: $(\tilde{\pi}_i^{\mathsf{zero}}, \sigma_i^{\mathsf{key}}) := \textsc{GenInput}(\mathsf{sid}, \mathsf{ssid}, sk_i, \mathsf{cid}_i, \tilde{E}_i, \tilde{e}_i)$;
20: Set $\mathsf{TX} := (\{(\mathsf{cid}_i, \sigma_i^{\mathsf{key}})\}, \{(\hat{\mathsf{cid}}_j, \hat{\pi}_j^{\mathsf{addr}}, (\hat{X}_j, \hat{\mathsf{cm}}_j), \hat{\pi}_j^{\mathsf{enc}}, \hat{\sigma}_j^{\mathsf{range}})\}, \{(\tilde{E}_i, \tilde{\pi}_i^{\mathsf{zero}})\}, \delta)$;
21: **for** each $\mathsf{Agg}_t \in \mathcal{L}_{\mathsf{Agg}}$ **do**
22: Invoke $\mathcal{F}_{\mathsf{SMT}}$ with $\langle \textsc{Send}, \mathsf{sid}, \mathsf{ssid}, \mathsf{TX}, P_k, \mathsf{Agg}_t\rangle$;
23: Output $\langle \textsc{ResponseID}, \mathsf{sid}, \{\hat{\mathsf{cid}}_j\}\rangle$ to $\mathcal{Z}$;
24: **else** Ignore the message.

Fig. 10. Our Π_{NiACS} supporting non-interactive payments (Users' transactions).

Initialization. Figure 9 shows the initialization process of Π_{NiACS}. Steps 1–6 show the initialization process of a user, including generating permanent address/key and initial coin, and submitting the initial information to $\mathcal{F}_{\mathsf{Ledger}}$. Steps 7–8 show the initialization process of an aggregator, i.e., registering the role to $\mathcal{F}_{\mathsf{Ledger}}$.

Users' Transactions. In Fig. 10, we describe how a user constructs a transfer transaction. In steps 3–13, the user checks if the transfer request from $\mathcal{Z}$ is valid. If so, the user generates the transaction by invoking GenOutput, GenExcess and GenInput as shown in steps 14–20. At last, the user sends the transaction to the designated aggregators through $\mathcal{F}_{\mathsf{SMT}}$ as shown in steps 21–22.

Aggregation. Figure 11 shows the process of aggregation. In steps 2–14, the aggregator checks if the transactions received from $\mathcal{F}_{\mathsf{SMT}}$ are valid by verifying the kernel and invoking VerInput and VerOutput, and records the valid transactions. Then the aggregator aggregates the valid transactions specified by $\mathcal{Z}$ through executing Aggregate and submits the aggregate transaction to $\mathcal{F}_{\mathsf{Ledger}}$, as shown in steps 16–21. Note that the aggregator just outputs the coin identifiers in (aggregate) transactions to $\mathcal{Z}$, rather than the real-world (aggregate) transactions. Therefore, the aggregator executes Clean to extract the coin identifiers from (aggregate) transactions.

Protocol Π_{NiACS}: Aggregation

An aggregator Agg_t, upon receiving $\langle \textsc{Sent}, \mathsf{sid}, \mathsf{ssid}, \mathsf{TX}, P_k, \mathsf{Agg}_t \rangle$ from $\mathcal{F}_{\mathsf{SMT}}$ where $\mathsf{TX} := (\{(\mathsf{cid}_i, \sigma_i^{\mathsf{key}})\}, \{(\hat{\mathsf{cid}}_j, \hat{\pi}_j^{\mathsf{addr}}, (\hat{X}_j, \hat{\mathsf{cm}}_j), \hat{\pi}_j^{\mathsf{enc}}, \hat{\sigma}_j^{\mathsf{range}})\}, \{(\tilde{E}_i, \tilde{\pi}_i^{\mathsf{zero}})\}, \delta)$, does the following:

1: Invoke $\mathcal{F}_{\mathsf{Ledger}}$ with $\langle \textsc{Read}, \mathsf{sid} \rangle$, get $\langle \textsc{Read}, \mathsf{sid}, \mathsf{state} \rangle$;
2: **for** each cid_i **do**
3: **if** cid_i is in state **then** Retrieve $(\mathsf{cid}_i, (X_i, \mathsf{cm}_i))$ from state;
4: **else** Ignore the message;
5: **if** $\prod_j \hat{\mathsf{cm}}_j / \prod_i \mathsf{cm}_i \cdot g^{\delta} = \prod_i \tilde{E}_i$ **then**
6: **for** each $\tilde{E}_i$ **do**
7: **if** $\textsc{VerInput}(\mathsf{sid}, \mathsf{ssid}, (\mathsf{cid}_i, (\tilde{E}_i, \tilde{\pi}_i^{\mathsf{zero}}), \sigma_i^{\mathsf{key}}))=0$ **then**
8: Ignore the message;
9: **for** each $\hat{\mathsf{cm}}_j$ **do**
10: **if** $\textsc{VerOutput}(\mathsf{sid}, \mathsf{ssid}, (\hat{\mathsf{cid}}_j, \hat{\pi}_j^{\mathsf{addr}}, (\hat{X}_j, \hat{\mathsf{cm}}_j), \hat{\pi}_j^{\mathsf{enc}}, \hat{\sigma}_j^{\mathsf{range}}))=0$ **then**
11: Ignore the message;
12: **else** Ignore the message;
13: Generate $\mathsf{TX}^* := (\{\mathsf{cid}_i\}, \{\hat{\mathsf{cid}}_j\})$;
14: Add $(\mathsf{TX}^*, \mathsf{TX})$ into $\mathcal{L}_{\mathsf{TX}}$;
15: Output $\langle \textsc{ReceivedTX}, \mathsf{sid}, \mathsf{TX}^* \rangle$ to $\mathcal{Z}$;

Upon receiving a message $\langle \textsc{Aggregate}, \mathsf{sid}, \{\mathsf{TX}_t^*\} \rangle$ from $\mathcal{Z}$, do the following:

16: Initialize $\mathcal{L}_I := \emptyset$, $\mathcal{L}_O := \emptyset$, $\mathcal{L}_K := \emptyset$ and $\Delta := 0$;
17: **for** each TX_t^* **do**
18: Retrieve the transaction TX_t from $\mathcal{L}_{\mathsf{TX}}$;
19: $(\mathcal{L}_I, \mathcal{L}_O, \mathcal{L}_K, \Delta) := \textsc{Aggregate}(\mathsf{TX}_t, \mathcal{L}_I, \mathcal{L}_O, \mathcal{L}_K, \Delta)$;
20: Set $\mathsf{aTX} := (\mathcal{L}_I, \mathcal{L}_O, \mathcal{L}_K, \Delta)$, and $\mathsf{aTX}^* := \textsc{Clean}(\mathsf{aTX})$;
21: Invoke $\mathcal{F}_{\mathsf{Ledger}}$ with $\langle \textsc{Submit}, \mathsf{sid}, \mathsf{aTX} \rangle$;
22: Output $\langle \textsc{AggTX}, \mathsf{sid}, \mathsf{aTX}^* \rangle$ to $\mathcal{Z}$;

Fig. 11. Our Π_{NiACS} supporting non-interactive payments (Aggregation).

Protocol Π_{NiACS}: Ledger

A party P_k, upon receiving a message $\langle \textsc{Read}, \mathsf{sid} \rangle$ from $\mathcal{Z}$, do the following:

1: Invoke $\mathcal{F}_{\mathsf{Ledger}}$ with $\langle \textsc{Read}, \mathsf{sid} \rangle$;
2: **if** P_k is uncorrupted **then**
3: Receive $\langle \textsc{Read}, \mathsf{sid}, \mathsf{state} \rangle$ from $\mathcal{F}_{\mathsf{Ledger}}$;
4: Set $\mathsf{state}^* := \textsc{Clean}(\mathsf{state})$;
5: Output $\langle \textsc{Read}, \mathsf{sid}, \mathsf{state}^* \rangle$ to $\mathcal{Z}$;
6: **else**
7: Receive $\langle \textsc{Read}, \mathsf{sid}, (\mathsf{state}, \mathsf{buffer}) \rangle$ from $\mathcal{F}_{\mathsf{Ledger}}$;
8: Set $\mathsf{state}^* := \textsc{Clean}(\mathsf{state})$, $\mathsf{buffer}^* := \textsc{Clean}(\mathsf{buffer})$;
9: Output $\langle \textsc{Read}, \mathsf{sid}, (\mathsf{state}^*, \mathsf{buffer}^*) \rangle$ to $\mathcal{Z}$;

Adversary $\mathcal{A}$, upon receiving a message $\langle \textsc{Permute}, \mathsf{sid}, \pi \rangle$ from $\mathcal{Z}$, do the following:

10: Forward $\langle \textsc{Permute}, \mathsf{sid}, \pi \rangle$ to $\mathcal{F}_{\mathsf{Ledger}}$.

Fig. 12. Our Π_{NiACS} supporting non-interactive payments (Ledger).

Ledger. In Fig. 12, we describe the part related to reading and maintaining ledger. Steps 2–5 show how the honest party obtains state. Besides state, a corrupted party can also obtain buffer and permute it as shown in steps 7–10. Likewise, the party just outputs the coin identifiers in state or buffer to $\mathcal{Z}$. Therefore, the aggregator executes Clean to extract the coin identifiers from state or buffer.

4.4 Security

Next we show the security of Π_{NiACS} against the static adversaries by Theorem 2. Please refer to the full version for the proof.

Theorem 2. *Assuming that DCDH problem is hard, the protocol Π_{NiACS} UC-realizes $\mathcal{F}_{\mathsf{NiACS}}$ in the $\{\mathcal{F}_{\mathsf{NIZK}}, \mathcal{F}_{\mathsf{SoK}}, \mathcal{F}_{\mathsf{SMT}}, \mathcal{F}_{\mathsf{Ledger}}, \mathcal{F}_{\mathsf{RO}}\}$-hybrid model, in the presence of static malicious adversaries.*

5 Instantiations

In the previous section, we describe our protocol Π_{NiACS} and prove it can UC-realize $\mathcal{F}_{\mathsf{NiACS}}$ in a hybrid model. In this section, we describe how to realize the subroutine ideal functionalities $\mathcal{F}_{\mathsf{NIZK}}$ and $\mathcal{F}_{\mathsf{SoK}}$ used in our Π_{NiACS}, which dominate the cost of our protocol. Next, we will describe the sub-protocols to achieve $\mathcal{F}_{\mathsf{NIZK}}$ and $\mathcal{F}_{\mathsf{SoK}}$ in the stand-alone setting and the UC setting.

Stand-Alone Setting. Recall our Π_{NiACS}, $\mathcal{F}_{\mathsf{NIZK}}^{\mathsf{zero}}$ is used to prove that an excess part $\tilde{E}_i$ is a commitment to 0, namely $E_i := g^{e_i}$, and $\mathcal{F}_{\mathsf{SoK}}^{\mathsf{key}}$ is used to prove the knowledge of secret key sk to a public key $pk := g^{sk}$ while signing an address. We can see that the languages in the two functionalities can be summarized as $\mathcal{L}_{\mathsf{DLOG}} := \{X \mid \exists\ x,\ \text{s.t. } X = g^x\}$. Therefore, $\mathcal{F}_{\mathsf{NIZK}}^{\mathsf{zero}}$ and $\mathcal{F}_{\mathsf{SoK}}^{\mathsf{key}}$ can be securely realized based on the Σ-protocol for proving knowledge of a discrete logarithm shown in Fig. 13. More specifically, for $\mathcal{F}_{\mathsf{NIZK}}^{\mathsf{zero}}$, X is the excess part, and x is the corresponding discrete logarithm. By using Fiat-Shamir transform [20], the interactive protocol in Fig. 13 can be converted into a non-interactive one where the challenge c is generated by a random oracle with (X, R) as the input. In practice, the random oracle will be instantiated by a hash function. Obviously, when X is pk and x is sk, the protocol in Fig. 13 can be used to prove the knowledge of a spending key, and can also be transformed to a non-interactive protocol by using Fiat-Shamir transform. At this point, we obtain a protocol for zero-knowledge proof of spending key. Next, we need to transform it into a protocol for signature of knowledge. Much work (e.g., [5,14]) has proved that the Fiat-Shamir transform can also be used to convert a public-coin proof of

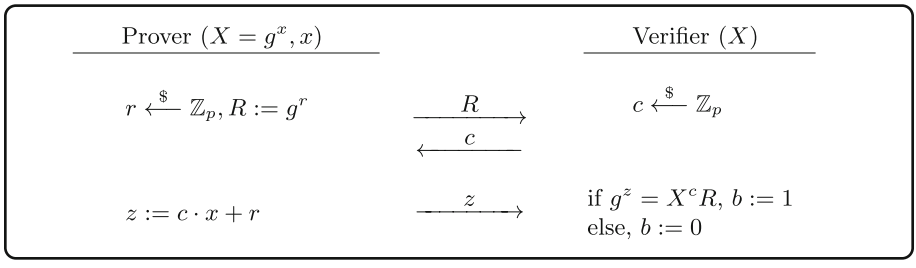

Fig. 13. Interactive Zero-knowledge proof of a discrete logarithm.

knowledge into a signature scheme by taking the message to be signed as the part of input to random oracle. More concretely, in our protocol, the excess part $\tilde{E}_i$ will be input to the random oracle along with (X, R). Therefore, we obtain the protocol that can securely realize $\mathcal{F}_{\mathsf{SoK}}^{\mathsf{key}}$.

Like other privacy-preserving cryptocurrencies, we also leverage Bulletproof [12] to generate range proofs. As mentioned in [12], Bulletproof is a public-coin proof of knowledge, and thus Bulletproof can also be converted into a non-interactive scheme by using Fiat-Shamir transform. Similarly, based on Bulletproof, we can obtain the protocol to securely realize $\mathcal{F}_{\mathsf{SoK}}^{\mathsf{range}}$ by taking the message to be signed (i.e., the address of each output coin in our Π_{NiACS}) as the part of input to random oracle.

The new variant of ElGamal proposed in this work can allow the receiver to obtain the value and randomness of a commitment by decryption, but the ciphertext needs to be generated using a hash function. Likewise, the one-time address and its auxiliary string are generated by using hash function. Hence, we cannot use Σ-protocol to realize $\mathcal{F}_{\mathsf{NIZK}}^{\mathsf{enc}}$ and $\mathcal{F}_{\mathsf{NIZK}}^{\mathsf{addr}}$. We need to use the general-purpose zk-SNARK [9,25,38,43].

UC Setting. The above protocols only securely realize the corresponding ideal functionalities in the stand-alone setting. Next, we discuss how to transform the above protocols to achieve UC security.

As for the Σ-protocol, we can use the compiler proposed by Camenisch et al. [13] to transform them to realize UC-security. The known practical instantiations (e.g., [9,25,38,43]) for SNARK also do not UC-realize $\mathcal{F}_{\mathsf{NIZK}}$ as they cannot satisfy Black-Box Simulation Extractability. Like other works, e.g. Hawk [31], Gyges [27], Ouroboros Crypsinous [28], we can also leverage the C∅C∅ framework proposed by Kosba et al. [30] to achieve Black-Box Simulation Extractability (namely, SSE-NIZK) in the standard CRS model.

6 Performance Analysis

In this section, we first give a performance estimation of our Π_{NiACS} where $\mathcal{F}_{\mathsf{NIZK}}$ and $\mathcal{F}_{\mathsf{SoK}}$ are achieved in the stand-alone setting. Then, we compare our Π_{NiACS} with Mimblewimble [39] and the work by Fuchsbauer et al. [22].

Table 2. Performance estimation and comparison.

	Spending time	Verifying time	TX size
[39]	$(2m+2)\cdot \mathsf{exp} + \mathsf{H} + m \cdot T^{\mathsf{P}}_{\mathsf{range}}$	$3\cdot \mathsf{exp} + \mathsf{H} + m \cdot T^{\mathsf{V}}_{\mathsf{range}}$	$(m+n+2)\cdot \lvert\mathbb{G}\rvert + 2\cdot\lvert\mathbb{Z}_p\rvert$ $+m\cdot\lvert\pi^{\mathsf{range}}\rvert$
[22]	$(4n+6m+3)\cdot \mathsf{exp}$ $+(2n+3m+1)\cdot \mathsf{H}+$ $m\cdot(T^{\mathsf{P}}_{\mathsf{range}}+ T^{\mathsf{P}}_{\mathsf{addr}} + T^{\mathsf{P}}_{\mathsf{enc}})$	$(3n+5)\cdot \mathsf{exp} + (n+1)\cdot \mathsf{H}+$ $m\cdot(T^{\mathsf{V}}_{\mathsf{range}}+ T^{\mathsf{V}}_{\mathsf{addr}} + T^{\mathsf{V}}_{\mathsf{enc}})$	$(3n+4m+3)\cdot \lvert\mathbb{G}\rvert$ $+(n+m+3)\cdot\lvert\mathbb{Z}_p\rvert$ $+m\cdot(\lvert\pi^{\mathsf{range}}\rvert+ \lvert\pi^{\mathsf{addr}}\rvert + \lvert\pi^{\mathsf{enc}}\rvert)$
Π_{NiACS}	$(4n+7m+1)\cdot \mathsf{exp}$ $+(3n+2m)\cdot \mathsf{H}+$ $m\cdot(T^{\mathsf{P}}_{\mathsf{range}}+ T^{\mathsf{P}}_{\mathsf{addr}} + T^{\mathsf{P}}_{\mathsf{enc}})$	$(4n+1)\cdot \mathsf{exp} + 2n\cdot \mathsf{H}+$ $m\cdot(T^{\mathsf{V}}_{\mathsf{range}}+ T^{\mathsf{V}}_{\mathsf{addr}} + T^{\mathsf{V}}_{\mathsf{enc}})$	$(5n+4m)\cdot \lvert\mathbb{G}\rvert$ $+(2n+1)\cdot\lvert\mathbb{Z}_p\rvert$ $+m\cdot(\lvert\pi^{\mathsf{range}}\rvert+ \lvert\pi^{\mathsf{addr}}\rvert + \lvert\pi^{\mathsf{enc}}\rvert)$

n: the number of input coins; m: the number of output coins; exp: an exponentiation operation in group $\mathbb{G}$ with prime order p; H: a hash function; $|\mathbb{G}|$: the length of element in group $\mathbb{G}$; $|\mathbb{Z}_p|$: the length of element in $\mathbb{Z}_p$; $T^{\mathsf{P/V}}_{\mathsf{x}}$: the time to generate/verify a proof for language $\mathcal{L}_{\mathsf{x}}$ ($\mathsf{x} \in \{\mathsf{range}, \mathsf{addr}, \mathsf{enc}\}$); $|\pi^{\mathsf{x}}|$: the length of proof for language $\mathcal{L}_{\mathsf{x}}$ ($\mathsf{x} \in \{\mathsf{range}, \mathsf{addr}, \mathsf{enc}\}$); The costs marked in blue are not necessary against rational adversaries; The costs marked in gray are not actually mentioned in [22], but they are necessary against malicious adversaries.

6.1 Performance Estimation

According to the instantiations described above, we give a performance estimation in Table 2. More specifically, according to the results shown in [12], the proving time $T^{\mathsf{P}}_{\mathsf{range}}$ for range $[0, 2^{64}]$ is 29ms while the verification time $T^{\mathsf{V}}_{\mathsf{range}}$ is 3.9ms. The range proof size π^{range} is 675 bytes. For the languages $\mathcal{L}_{\mathsf{addr}} := \{(pk, R) \mid \exists\ (\mathsf{Addr}, r)$ and $\mathcal{L}_{\mathsf{enc}} := \{(pk, (X, \mathsf{cm})) \mid \exists\ (r, v),$ s.t. $X = pk^r, \mathsf{cm} = g^{H(g^r)}h^v\}$, we use the scheme in [25], a general-purpose zk-SNARK, to generate the proofs, and thus the proof sizes $|\pi_{\mathsf{addr}}|$ and $|\pi_{\mathsf{enc}}|$ are both $2\mathbb{G}_1 + \mathbb{G}_2$. The corresponding proving time ($T^{\mathsf{P}}_{\mathsf{addr}}$ and $T^{\mathsf{P}}_{\mathsf{enc}}$) and verification time ($T^{\mathsf{V}}_{\mathsf{addr}}$ and $T^{\mathsf{V}}_{\mathsf{enc}}$) mainly depend on the number of constraints. Concretely, we implement the hash function in $\mathcal{L}_{\mathsf{addr}}$ and $\mathcal{L}_{\mathsf{enc}}$ by using MiMCHash-256. The number of constraints required by $\mathcal{L}_{\mathsf{addr}}$ and $\mathcal{L}_{\mathsf{enc}}$ is 11,742 and 14,799, respectively[7]. Moreover, in practice, the proofs for $\mathcal{L}_{\mathsf{addr}}$ and $\mathcal{L}_{\mathsf{enc}}$ are not necessary as explained below, and we mark the corresponding costs in blue.

In the security analysis in Sect. 4.4, we assume that the adversary will have malicious behaviors arbitrarily. However, it is reasonable to assume that the adversary is rational in practice. As mentioned in [24], a rational adversary is expected to act in a utility-maximizing way. The costs marked in blue are related to $\mathcal{F}^{\mathsf{addr}}_{\mathsf{NIZK}}$ and $\mathcal{F}^{\mathsf{enc}}_{\mathsf{NIZK}}$, which are used to ensure that the one-time address and ciphertext are correctly generated for the receiver, respectively. The receiver in practice can identify if the one-time address and ciphertext are valid without the proofs, and if not, the receiver can abort the deal (e.g., refuse to send the goods), and thus can not be harmed. Moreover, the sender cannot benefit from it. Therefore, a rational adversary will not carry out this malicious behavior. Obviously, the receiver can identify if a one-time address is valid by invoking

[7] If zk-SNARK is transformed to SSE-NIZK by using the framework in [30], the number of constraints required by $\mathcal{L}_{\mathsf{addr}}$ and $\mathcal{L}_{\mathsf{enc}}$ will increase to about 71,742 and 74,799, respectively.

GenOTKey. Next, we will explain how the receiver identifies whether a ciphertext is generated correctly.

If a transaction containing ciphertext (X^*, Y) can be confirmed, Y must be a valid commitment due to our design, i.e., $Y = g^{\alpha}h^{v}$. If X^* associated with Y is generated correctly, denoted as X, the receiver can recover (α, v) by computing $\alpha := H(X^{\frac{1}{sk}}), h^{v} := Y/g^{\alpha}$ and recovering v from h^{v} where $v \in [0, 2^{64}]$. Otherwise, i.e., $X^* = X' \neq X$, the receiver will obtain $\alpha' = H(X'^{\frac{1}{sk}}) \neq \alpha$ and $h^{v'} = Y/g^{\alpha'} = g^{\alpha-\alpha'}h^{v}$. Due to the random oracle, $h^{v'}$ is randomly distributed and so the probability of $v' \in [0, 2^{64}]$ is $\frac{2^{64}}{2^{256}}$, which is negligible. Therefore, the receiver can recognize the invalid ciphertext by checking if $v \in [0, 2^{64}]$.

6.2 Comparison

We also give the performance estimations of Mimblewimble [39] and the non-interactive solution proposed independently and concurrently by Fuchsbauer et al. [22] in Table 2. It can be seen that both our work and Fuchsbauer et al. [22] degrade performance to achieve non-interaction, and the performances of the two non-interactive solutions are comparable. As for our work, besides introducing addresses and related proofs, the main reason leading to a higher cost is that our protocol needs to split the excess into multiple parts. Similarly, Fuchsbauer et al. [22] also introduce addresses and related proofs. Although they do not split the excess, they add a doubling key for each one-time address, thus resulting in the comparable additional cost. In a nutshell, the two non-interactive solutions are more suitable for the scenarios where non-interaction is strongly desirable. Nevertheless, designing a non-interaction version without degrading performance is still a challenging problem.

Acknowledgements. We would like to thank the anonymous reviewers for their insightful suggestions and comments. This work was supported in part by the National Key Research and Development Project 2020YFA0712300 and the National Natural Science Foundation of China (Grant No. 62272294). Hong-Sheng Zhou acknowledges support by NSF grant CNS-1801470, a Google Faculty Research Award and a research gift from Ergo Platform.

References

1. Beam documentation. https://documentation.beam.mw/
2. Grin documentation. https://docs.grin.mw/wiki/transactions/contracts/
3. Mwc documentation. https://www.mwc.mw/docs
4. Solving unconfirmed bitcoin transactions in electrum. https://data-dive.com/unconfirmed-bitcoin-transactions-electrum
5. Abdalla, M., An, J.H., Bellare, M., Namprempre, C.: From identification to signatures via the fiat-shamir transform: minimizing assumptions for security and forward-security. In: Knudsen, L.R. (ed.) EUROCRYPT 2002. LNCS, vol. 2332, pp. 418–433. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-46035-7_28

6. Androulaki, E., Karame, G.O., Roeschlin, M., Scherer, T., Capkun, S.: Evaluating user privacy in bitcoin. In: Sadeghi, A.-R. (ed.) FC 2013. LNCS, vol. 7859, pp. 34–51. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39884-1_4
7. Badertscher, C., Maurer, U., Tschudi, D., Zikas, V.: Bitcoin as a transaction ledger: a composable treatment. In: Katz, J., Shacham, H. (eds.) CRYPTO 2017. LNCS, vol. 10401, pp. 324–356. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63688-7_11
8. Ben-Sasson, E., et al.: Zerocash: decentralized anonymous payments from bitcoin. In: 2014 IEEE Symposium on Security and Privacy, pp. 459–474. IEEE Computer Society Press (2014). https://doi.org/10.1109/SP.2014.36
9. Ben-Sasson, E., Chiesa, A., Genkin, D., Tromer, E., Virza, M.: SNARKs for C: verifying program executions succinctly and in zero knowledge. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013. LNCS, vol. 8043, pp. 90–108. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40084-1_6
10. Bonneau, J., Narayanan, A., Miller, A., Clark, J., Kroll, J.A., Felten, E.W.: Mixcoin: anonymity for bitcoin with accountable mixes. In: Christin, N., Safavi-Naini, R. (eds.) FC 2014. LNCS, vol. 8437, pp. 486–504. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-45472-5_31
11. Bünz, B., Agrawal, S., Zamani, M., Boneh, D.: Zether: towards privacy in a smart contract world. In: Bonneau, J., Heninger, N. (eds.) FC 2020. LNCS, vol. 12059, pp. 423–443. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-51280-4_23
12. Bünz, B., Bootle, J., Boneh, D., Poelstra, A., Wuille, P., Maxwell, G.: Bulletproofs: short proofs for confidential transactions and more. In: 2018 IEEE Symposium on Security and Privacy, pp. 315–334. IEEE Computer Society Press (2018). https://doi.org/10.1109/SP.2018.00020
13. Camenisch, J., Krenn, S., Shoup, V.: A framework for practical universally composable zero-knowledge protocols. In: Lee, D.H., Wang, X. (eds.) ASIACRYPT 2011. LNCS, vol. 7073, pp. 449–467. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25385-0_24
14. Camenisch, J., Stadler, M.: Efficient group signature schemes for large groups. In: Kaliski, B.S. (ed.) CRYPTO 1997. LNCS, vol. 1294, pp. 410–424. Springer, Heidelberg (1997). https://doi.org/10.1007/BFb0052252
15. Chase, M., Lysyanskaya, A.: On signatures of knowledge. In: Dwork, C. (ed.) CRYPTO 2006. LNCS, vol. 4117, pp. 78–96. Springer, Heidelberg (2006). https://doi.org/10.1007/11818175_5
16. Chen, Yu., Ma, X., Tang, C., Au, M.H.: PGC: decentralized confidential payment system with auditability. In: Chen, L., Li, N., Liang, K., Schneider, S. (eds.) ESORICS 2020. LNCS, vol. 12308, pp. 591–610. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-58951-6_29
17. Esgin, M.F., Zhao, R.K., Steinfeld, R., Liu, J.K., Liu, D.: MatRiCT: efficient, scalable and post-quantum blockchain confidential transactions protocol. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) ACM CCS 2019, pp. 567–584. ACM Press (2019). https://doi.org/10.1145/3319535.3354200
18. Fanti, G., et al.: Dandelion++: lightweight cryptocurrency networking with formal anonymity guarantees (2018)
19. Fauzi, P., Meiklejohn, S., Mercer, R., Orlandi, C.: Quisquis: a new design for anonymous cryptocurrencies. In: Galbraith, S.D., Moriai, S. (eds.) ASIACRYPT 2019. LNCS, vol. 11921, pp. 649–678. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-34578-5_23

20. Fiat, A., Shamir, A.: How to prove yourself: practical solutions to identification and signature problems. In: Odlyzko, A.M. (ed.) CRYPTO 1986. LNCS, vol. 263, pp. 186–194. Springer, Heidelberg (1987). https://doi.org/10.1007/3-540-47721-7_12
21. Fuchsbauer, G., Orrù, M., Seurin, Y.: Aggregate cash systems: a cryptographic investigation of mimblewimble. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019. LNCS, vol. 11476, pp. 657–689. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17653-2_22
22. Fuchsbauer, G., Orrù, M.: Non-interactive mimblewimble transactions, revisited. Cryptology ePrint Archive, Paper 2022/265 (2022). https://eprint.iacr.org/2022/265
23. Garman, C., Green, M., Miers, I.: Accountable privacy for decentralized anonymous payments. In: Grossklags, J., Preneel, B. (eds.) FC 2016. LNCS, vol. 9603, pp. 81–98. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54970-4_5
24. Groce, A., Katz, J., Thiruvengadam, A., Zikas, V.: Byzantine agreement with a rational adversary. In: Czumaj, A., Mehlhorn, K., Pitts, A., Wattenhofer, R. (eds.) ICALP 2012. LNCS, vol. 7392, pp. 561–572. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31585-5_50
25. Groth, J.: On the size of pairing-based non-interactive arguments. In: Fischlin, M., Coron, J.-S. (eds.) EUROCRYPT 2016. LNCS, vol. 9666, pp. 305–326. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49896-5_11
26. Jedusor, T.E.: Mimblewimble (2016). https://download.wpsoftware.net/bitcoin/wizardry/mimblewimble.txt
27. Juels, A., Kosba, A.E., Shi, E.: The ring of Gyges: investigating the future of criminal smart contracts. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) ACM CCS 2016, pp. 283–295. ACM Press (2016). https://doi.org/10.1145/2976749.2978362
28. Kerber, T., Kiayias, A., Kohlweiss, M., Zikas, V.: Ouroboros crypsinous: privacy-preserving proof-of-stake. In: 2019 IEEE Symposium on Security and Privacy, pp. 157–174. IEEE Computer Society Press (2019). https://doi.org/10.1109/SP.2019.00063
29. Kiayias, A., Zhou, H.-S., Zikas, V.: Fair and robust multi-party computation using a global transaction ledger. In: Fischlin, M., Coron, J.-S. (eds.) EUROCRYPT 2016. LNCS, vol. 9666, pp. 705–734. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49896-5_25
30. Kosba, A., et al.: How to use SNARKs in universally composable protocols. Cryptology ePrint Archive, Report 2015/1093 (2015). http://eprint.iacr.org/2015/1093
31. Kosba, A.E., Miller, A., Shi, E., Wen, Z., Papamanthou, C.: Hawk: the blockchain model of cryptography and privacy-preserving smart contracts. In: 2016 IEEE Symposium on Security and Privacy, pp. 839–858. IEEE Computer Society Press (2016). https://doi.org/10.1109/SP.2016.55
32. Koshy, P., Koshy, D., McDaniel, P.: An analysis of anonymity in bitcoin using P2P network traffic. In: Christin, N., Safavi-Naini, R. (eds.) FC 2014. LNCS, vol. 8437, pp. 469–485. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-45472-5_30
33. Lai, R.W.F., Ronge, V., Ruffing, T., Schröder, D., Thyagarajan, S.A.K., Wang, J.: Omniring: scaling private payments without trusted setup. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) ACM CCS 2019, pp. 31–48. ACM Press (2019). https://doi.org/10.1145/3319535.3345655
34. Lindell, Y.: Tutorial on secure multi-party computation (2003). http://www.cs.biu.ac.il/lindell/research-statements/tutorial-secure-computation.ppt

35. Lindell, Y.: Survey: secure composition of multiparty protocols (2005). http://www.cs.biu.ac.il/lindell/research-statements/survey-composition-40-min-05.ppt
36. Meiklejohn, S., et al.: A fistful of bitcoins: characterizing payments among men with no names. Commun. ACM **59**(4), 86–93 (2016). https://doi.org/10.1145/2896384, https://doi.org/10.1145/2896384
37. Noether, S.: Ring signature confidential transactions for monero. Cryptology ePrint Archive, Report 2015/1098 (2015). http://eprint.iacr.org/2015/1098
38. Parno, B., Howell, J., Gentry, C., Raykova, M.: Pinocchio: nearly practical verifiable computation. In: 2013 IEEE Symposium on Security and Privacy, pp. 238–252. IEEE Computer Society Press (2013). https://doi.org/10.1109/SP.2013.47
39. Poelstra, A.: Mimblewimble (2016). https://download.wpsoftware.net/bitcoin/wizardry/mimblewimble.pdf
40. Ron, D., Shamir, A.: Quantitative analysis of the full bitcoin transaction graph. In: Sadeghi, A.-R. (ed.) FC 2013. LNCS, vol. 7859, pp. 6–24. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39884-1_2
41. Ruffing, T., Moreno-Sanchez, P., Kate, A.: CoinShuffle: practical decentralized coin mixing for bitcoin. In: Kutyłowski, M., Vaidya, J. (eds.) ESORICS 2014. LNCS, vol. 8713, pp. 345–364. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11212-1_20
42. Sun, S.-F., Au, M.H., Liu, J.K., Yuen, T.H.: RingCT 2.0: a compact accumulator-based (linkable ring signature) protocol for blockchain cryptocurrency monero. In: Foley, S.N., Gollmann, D., Snekkenes, E. (eds.) ESORICS 2017. LNCS, vol. 10493, pp. 456–474. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66399-9_25
43. Wahby, R.S., Setty, S.T.V., Ren, Z., Blumberg, A.J., Walfish, M.: Efficient RAM and control flow in verifiable outsourced computation. In: NDSS 2015. The Internet Society (2015)
44. Yu, G.: Mimblewimble non-interactive transaction scheme. Cryptology ePrint Archive, Paper 2020/1064 (2020). https://eprint.iacr.org/2020/1064
45. Yuen, T.H., et al.: RingCT 3.0 for blockchain confidential transaction: shorter size and stronger security. In: Bonneau, J., Heninger, N. (eds.) FC 2020. LNCS, vol. 12059, pp. 464–483. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-51280-4_25