Accio: Variable-Amount, Optimized-Unlinkable and NIZK-Free Off-Chain Payments via Hubs

Zhonghui Ge
Shanghai Jiao Tong University
Shanghai, China
zhonghui.ge@sjtu.edu.cn

Jiayuan Gu
Shanghai Jiao Tong University
Shanghai, China
sjtu_gjy@sjtu.edu.cn

Chenke Wang
Shanghai Jiao Tong University
Shanghai, China
w_chenke@sjtu.edu.cn

Yu Long*
Shanghai Jiao Tong University
Shanghai, China
longyu@sjtu.edu.cn

Xian Xu*
East China University of Science and Technology
Shanghai, China
xuxian@ecust.edu.cn

Dawu Gu*
Shanghai Jiao Tong University
Shanghai, China
dwgu@sjtu.edu.cn

ABSTRACT

Payment channel hubs (PCHs) serve as a promising solution to achieving quick off-chain payments between pairs of users. They work by using an untrusted tumbler to relay the payments between the payer and payee and enjoy the advantages of low cost and high scalability. However, the most recent privacy-preserving payment channel hub solution that supports variable payment amounts suffers from limited unlinkability, e.g., being vulnerable to the abort attack. Moreover, this solution utilizes non-interactive zero-knowledge proofs, which bring huge costs on both computation time and communication overhead. Therefore, how to design PCHs that support variable amount payments and unlinkability, but reduce the use of huge-cost cryptographic tools as much as possible, is significant for the large-scale practical applications of off-chain payments.

In this paper, we propose Accio, a variable amount payment channel hub solution with optimized unlinkability, by deepening research on unlinkability and constructing a new cryptographic tool. We provide the detailed Accio protocol and formally prove its security and privacy under the Universally Composable framework. Our prototype demonstrates its feasibility and the evaluation shows that Accio outperforms the other state-of-the-art works in both communication and computation costs.

KEYWORDS

Payment Channel Hub, Variable Amount, Unlinkability, NIZK-free

1 INTRODUCTION

As a new tool to build trust in the digital world, blockchain has been widely utilized to provide an open medium of exchange. People have seen more and more developments on blockchain since the introduction of *smart contracts*. However, the blockchain suffers from low scalability due to the realization of the secure consensus. For example, in Bitcoin [20], only 7 payments are achieved per second and the payment confirmation time is as long as 1 hour, which is far from satisfactory.

To improve blockchains' scalability, a promising method is the *payment channel* (e.g., [1, 21]). Roughly, two blockchain users can open one payment channel and perform *off-chain payments* by updating the channel state. Since the off-chain payments only need confirmation from both channel users instead of the underlying chain, they enjoy the advantages of high payment throughput and low confirmation delay. For users of different channels, payments could be off-chain performed via the relay of intermediaries.

The two most typical relaying methods are payment channel networks and payment channel hubs. Compared with the former solution [10, 11, 17–19, 21, 24], payment channel hubs (PCHs) provide a lower-cost way to facilitate payments between each pair of users. Particularly, by using an untrusted *tumbler* as the intermediary (also called the *hub*), each user only needs to set up one payment channel with the tumbler, which in turn facilitates the payment between each pair of users. In this way, each PCH payment consists of two sub-payments, the payer pays the tumbler, and the tumbler pays the payee, in two channels, respectively.

Many efforts have been devoted to the PCH construction. The Lightning Network (LN) [21] achieves the security of PCH payments by introducing the conditional payment, which guarantees that sub-payments in both channels succeed or fail (called payment *atomicity*). In addition to security, the privacy problem in PCHs also needs to be considered. Although the tumbler facilitates payments among PCH users, it may be aware of and abuse the payment relationships between payers and payees. For instance, in the tumbler's view in LN, all PCH users have absolutely no privacy since the tumbler can directly link each payer and payee pair via the two sub-payments conducted with them. To achieve secure and unlinkable PCHs, TumbleBit [16] introduces the *puzzle-promising and puzzle-solving paradigm*, which breaks the link between two sub-payments and is adopted by all known subsequent works on *unlinkable* PCHs. A^2L [27] extends the idea of TumbleBit and provides a more efficient and universal solution. Recent work [13] points out the flaw in A^2L's security model and proposes A^2L^+ and A^2L^{UC} as the amendment at the cost of more group operations. However, all these privacy-preserving schemes only support fixed-amount payments since the tumbler is aware of the states of both the payer-tumbler and tumbler-payee channels, which are updated using the same amount. When the amount is *variable*, the tumbler could directly link the payer and payee by their channel states.

Bolt [14] constructs unlinkable PCHs supporting variable payment amounts. But it works only for anonymous blockchains such

*Corresponding authors.

Table 1: Comparison of PCH constructions.

Schemes	Variable Amount	Unlinkability	Anonymity Set Size *	Abort Attack Resistance	Griefing Attack Resistance	NIZK-free	Bitcoin Compatibility★	Bi-directional Channel
TumbleBit[16]	○	●	n	○	○	●†	●	○
A^2L[27]	○	●	n	○	●	○	●	●
A^2L^+, A^2L^{UC}[13]	○	●	n	○	●	○	●	●
LN [21]	●	○	1	○	○	●	●	●
Bolt[14]	●	●	N	○	○	○	○	●
Perun[9]	●	●	$n^‡$	●	●	●	○	●
BlindHub [22]	●	●	n	○	●	○	●	●
Accio	●	●	N	●	●	●†	○	○

● Support ○ No support

* For comparison, we only consider the maximum anonymity set in the optimistic case. Especially, n denotes the number of payees during one epoch, and N denotes the number of total payees connecting to the tumbler.

† NIZK is required in the setup phase / UC-proof.

★ Bolt requires the underlying blockchain to be anonymous, while Perun and ours require the underlying blockchain to be Turing-complete.

‡ For the PCH of n users, it is necessary to establish $O(n^2)$ additional virtual channels to support privacy-preserving payments between each pair of users.

as [3]. Perun [9] cancels this limitation by introducing virtual channels, within which each pair of users can perform payments without the help of the tumbler. Nevertheless, Perun comes at a significant channel management cost. It requires $O(n^2)$ virtual channels to facilitate the payments between n users, in contrast to the mentioned $O(n)$ solutions for fixed-amount payments.

Most recently, a new privacy-preserving PCH construction supporting variable amount, BlindHub [22], was proposed. It works on non-confidential blockchains (e.g., Bitcoin) and enables the unlinkability between two sub-payments with an equal but variable amount. Technically, BlindHub inherits the puzzle-promising and puzzle-solving paradigm and brings the payment amount into the puzzle construction. The tumbler updates channel states in a secure and privacy-preserving way, i.e., the tumbler is not aware of the plaintext channel state and the amount, and the correctness of the update is guaranteed by the non-interactive zero-knowledge (NIZK) proofs. However, it still suffers from the following drawbacks,

- The unlinkability in BlindHub is limited. To utilize the puzzle-promising and puzzle-solving paradigm, all the unlinkable PCH schemes (including TumbleBit, A^2L versions, and BlindHub) work by dividing time into epochs and hiding the payer / payee of each payment among all payers / payees executing payments in the *same* epoch. However, the unlinkability derived from this method is unsatisfactory for two reasons. First, the vulnerability to *abort attack*, i.e., the tumbler can derive the payment relation by deliberately aborting the payments with users. To our knowledge, how to avoid the abort attack of unlinkable PCHs remains open [22]. Second, the anonymous set is limited by the PCH users active in the same epoch. The size of the anonymity set may be too small when there are only a few participants in one epoch. Ideally, enrolling all users connected to the tumble into the anonymous set can optimize unlinkability.
- Using NIZK techniques (esp., in each sub-payment) brings huge communication overhead. Even with optimizations in NIZK techniques, the overall cost is 87860 KB [22]. Compared with A^2L, the cost is enormous since the latter costs only around 10 KB.

Moreover, like A^2L^+, BlindHub has *not* been proven secure under the Universally Composable (UC) model [6, 7]. The main reason is the lacking of a trapdoor mechanism in the simulation. Therefore, the simulator cannot connect one puzzle to its variants and fails to link the adversarial sessions when the link is necessary for the simulation [13].

From the above analysis, a natural question arises.

Is it possible to design a secure PCH scheme that supports unlinkable payments under variable amounts, achieves the optimized unlinkability, is constructed without NIZK *techniques, and can be proven UC-secure?*

Contributions and roadmap. This work provides the affirmative answer to this question by exploring the nature of PCH and canceling the puzzle-promising and puzzle-solving paradigm. We call the resulting PCH protocol Accio. Our contributions are as follows.

- *Variable-amount, optimized-unlinkable, and NIZK-free PCH construction.* We replace the puzzle-promising and puzzle-solving paradigm with a novel method and achieve optimized unlinkability without NIZK. Specifically, we only hide the state of the tumbler-payee channel from the tumbler (i.e., the state of the payer-tumbler channel is kept in plaintext) and enable the tumbler to securely update the hidden state without knowing its plaintext content. In each payment, the payer sends the payer-tumbler channel state to the tumbler, together with the hidden tumbler-payee channel state. Both states are updated by the tumbler, using the same (but variable) amount (Section 3). As analyzed in Section 3.5, this method achieves both payment atomicity and optimized unlinkability without NIZK. The comparison with previous works is shown in Table 1.
  Especially, the *updatable* hidden state is constructed using the primitive called *randomizable signature of updatable commitments* (RSUC) (Section 4).
- *Formal security definition and proof.* We formally model Accio leveraging the UC framework and prove that our construction achieves balance security and the variable-amount payment unlinkability (Section 5, 6). It is remarkable that the proof of UC-security requires NIZK, even though Accio is NIZK-free.

- *Implementation and evaluation.* We implement Accio, including the on-chain and off-chain operations, and evaluate its performance. We also compare its off-chain performance with the state-of-the-art fixed-amount and variable-amount PCH constructions. The results show that Accio outperforms both in terms of computation and communication overhead. Especially, the communication overhead of Accio is 4 orders of magnitude less than BlindHub (Section 7).

2 PRELIMINARY

2.1 Blockchain and Smart Contracts

Blockchain is a distributed ledger maintained by distrusted miners. The consensus mechanism of blockchain guarantees that a valid transaction takes at most Δ time to be recorded on the ledger [12, 20]. The time delay Δ is also known as the *blockchain delay*.

Our scheme is built upon Turing-complete platforms, on which smart contracts are enabled. Specifically, the smart contract is the code deployed upon the blockchain, executed by miners when invoked by contract calls from parties.

2.2 Payment Channels

There are 3 procedures during the lifetime of one payment channel, *open*, *update*, and *close*.

Open. A channel is opened by its two users, denoted as Alice and Bob, escrowing on-chain funds to it. The deposits serve as their initial channel balances and are updated along with the channel payments, as described below.

Update. Channel users complete payments by *updating* the channel state. Concretely, the channel state consists of the *channel balance allocation* between the two channel users, and the *channel identifier* (e.g., the address of the channel contract can serve as the channel identifier). The former changes with each payment, while the latter uniquely specifies the channel and keeps unchanged throughout.

To perform one payment, one user, suppose Alice, generates the new state based on the latest channel state, where the balance allocation is updated according to the payment. Then Alice authenticates the new state and sends it along with the authentication to Bob, who checks the correctness concerning the following items. (1) *Latestness identification*. The new state comes from the updating of the latest channel state. (2) *Balance-sufficiency attestation*. In the latest state, Alice has sufficient coins to afford this payment. (3) *Authentication-validness verification*. Alice's authentication on the new state is valid. If all checks pass, Bob replies with his authentication on the new state. At that time, either user has collected the other one's authentication on the new state, which becomes the latest state, and the update completes. If Alice receives no reply from Bob, it is not sure whether the update will take effect. Then Alice initiates the channel close, and the update result can be derived from the state accepted by the blockchain, as described below.

Close. Both channel users could initiate the channel close to refund their channel balances to the blockchain. The cashouts are based on the latest channel state, which is provided by channel users. The channel operating mechanism guarantees that the honest users' cashouts are not less than their latest channel balances.

Bidirectional vs. unidirectional channels. There are two types of channels: bidirectional [21] and unidirectional [26], where the former allows either channel user to pay the other, and the latter enforces only one user to pay the other one, denoted as the sender and receiver, respectively. Therefore, for the unidirectional channel, only the sender must deposit coins to the channel to support further payments in the update phase. Two unidirectional payment channels supporting opposite payment directions can implement the functionality of one bidirectional channel.

The unidirectional channel has one advantage regarding the channel close. The state submitted by the receiver can be regarded as the latest since the receiver keeps receiving coins from the sender and benefits most from the latest state. In case the state submitted by the receiver is invalid (e.g., not authenticated by the sender), the receiver would be punished, and all the channel funds would be refunded to the sender.

Thus, in the unidirectional channel, enforcing the receiver's submission of a close state can guarantee that the honest user would not lose coins. Therefore, the channel could be closed in two ways: either the sender initiates the channel close and the receiver responds timely with the channel state, or the receiver initiates the channel close with the channel state. In case the sender initiates the channel close and the receiver fails to respond timely, the sender triggers the timeout and gets all the channel funds.

2.3 Payment Channel Hubs

Based on payment channels, payment channel hubs (PCHs) enable any pair of users to perform off-chain payments via the same intermediary, called the *tumbler*. Each user opens a payment channel with the tumbler, which links any pair of users and thus helps them perform the off-chain payment.

Concretely, the *payment* between the payer and payee could be divided into two *sub-payments*: the payer pays the tumbler in the payer-tumbler channel, and the tumbler pays the payee in the tumbler-payee channel. Both sub-payments are of the same amount. Via the relay of the tumbler, the payer pays, the payee collects, and the tumbler keeps the sum of its balances in two channels unchanged, achieving the payment. The tumbler may charge fees for facilitating the payment. For the simplification of the presentation, fees are ignored until the last section of this paper.

During each payment, the honest user must be guaranteed not to lose coins, even when others collude. The property could be defined as *atomicity*, i.e., both sub-payments either succeed or fail.

3 ACCIO OVERVIEW

In this section, we give an overview of Accio. We first clarify the goals, including the security guarantees, desired privacy properties, and efficiency requirements. Then by recalling the traditional puzzle-promising and puzzle-solving paradigm used in unlinkable PCHs, we point out why it cannot meet all these requirements, in order to motivate Accio. We demonstrate that Accio can achieve all the goals by hiding the state on only one side with a new cryptographic tool (here 'side' means the payer-tumbler channel or the tumbler-payee channel). Based on the hidden state, we describe the payment procedure of Accio, along with the channel open and close procedures.

3.1 Goals to Be Achieved

We aim to construct secure and unlinkable PCHs which support variable-amount payments without involving any costly NIZK proof. Specifically, each payment is conducted via the tumbler by performing two sub-payments of the same (but variable) amount in the payer-tumbler and tumbler-payee channels, respectively. During this process, the following properties need to be satisfied.

- *Balance security.* The system cannot be exploited to "print" new money or steal money from honest parties, even when others collude.
- *(Optimized) Payment unlinkability.* The tumbler (without colluding with others) cannot link the payer to the payee of one variable-amount payment. Specifically, the payer / payee relation hides among a set of relations, and unlinkability can be optimized by enlarging the set size.
- *NIZK-free.* Remove the dependence on NIZK-proofs, which tend to bring in high communication and computation costs, to enhance the usability of unlinkable PCHs.

3.2 Necessity and Method in Replacing the Puzzle-Promising and Puzzle-Solving Paradigm

Starting from TumbleBit, all existing solutions of unlinkable PCHs, including A^{2}L versions and BlindHub, utilize the *puzzle-promising and puzzle-solving paradigm* pattern. More concretely, in the *puzzle-promising* phase, the tumbler starts this procedure by generating a puzzle for the payee and promises that coins will be delivered to the payee if the payee solves the puzzle. In the*puzzle-solving* phase, to obtain the solution without being linked, the payer randomizes this puzzle and pays the tumbler (the same amount of coins as the tumbler promises to pay the payee) in exchange for the solution of the randomized puzzle. Later, the payer can use the tumbler's answer to recover the solution of the original puzzle and then complete the payment between the tumbler and the payee. Since the tumbler could not link the puzzle to its randomization, the payer-payee relation is hidden among all pairs performing successful payments during the two phases.

Although the paradigm can achieve transaction unlinkability in PCHs, it has the following drawbacks.

- *Huge costs in variable-amount situations.* In this paradigm, the tumbler has access to the current states of the payer-tumbler and the tumbler-payee channels. It can link a pair of payer-payee directly by comparing their transferred coin amounts if they are in plaintext. To make the paradigm work in the variable-amount PCHs, both the payment amount and channel states must be hidden from the tumbler.
  Natural as the idea appears, hiding the channel state would actually make it hard to guarantee the balance security since the tumbler needs to ensure its channel state has been correctly updated, while both the channel state and the payment amount are invisible. To solve this issue, people utilize NIZK proofs in *each* sub-payment request to prove to the tumbler that the channel state is well constructed and updated, i.e., the state is updated with a consistent payment amount. This, however, turns out to be quite time and bandwidth consuming due to the use of NIZK.
  Even worse, countermeasures against the *griefing attack*, which is inherent in this paradigm, aggravate PCHs' costs. Griefing is the DoS attack against the tumbler. Concretely, the payee starts the puzzle-promising phase with the tumbler but with no puzzle-solving phase following. Consequently, the tumbler's coins in the tumbler-payee channel are locked in vain. Currently, people use an additional registration phase to solve this problem, where the payer locks coins in advance, and the payee proves that there is a payer who has locked coins when initiating the puzzle-promising phase.
- *Limited unlinkability.* This paradigm divides time into epochs. Both payers and payees interact with the tumbler, and each of them hides within the payer / payee set, acting in one epoch. Firstly, the anonymity set is limited to the number of pairs performing payments successfully in the same epoch, not all users connected to the tumbler. Besides, this gives rise to the tumbler's *abort attack* of the unlinkability. Concretely, by aborting the payment with one payer, the payment with the payee would also fail, and the tumbler can derive the payer / payee relationship from this event.

Due to the problems with the traditional puzzle-promising and puzzle-solving paradigm, we propose a new PCH workflow to improve over previous works. Considering the goals mentioned in Section 3.1, the workflow needs to have the following features.

- *Hiding channel state on only one side.* To perform the payment, state updates on both sides (i.e., the payer-tumbler side and the tumbler-payee side) need to be approved by the tumbler. As analyzed above, it is impossible to achieve unlinkability in the variable amount payment if the tumbler knows the plaintext states of *both* sides, and hiding *both* states from the tumbler would result in NIZK to guarantee balance security with high costs. We solve this problem by hiding *only one* side's state (named the *hidden-state* side) from the tumbler, while the other side (named the *plaintext-state* side) together with the payment amount remain in plaintext. In this way, we cancel the need for NIZK on the plaintext-state side.
- *Removing interaction on the hidden-state side.* Further, we eliminate all the interactions on the hidden-state side to achieve unlinkability without NIZK (here we focus on after the channel is opened and before the channel is closed). As a result, the plaintext-state side is supposed to take care of the update for both sides by interacting with the tumbler, and the hidden-state should be updatable in the NIZK-free way. To ensure atomicity, the correct output of the tumbler is the confirmation that both sides' states have been updated with the same payment amount. In this way, unlinkability can be guaranteed since the tumbler only communicates with the plaintext side.

To be certain, the *only-one-side-interaction* workflow with the features above can resist both the griefing attack and the abort attack, and moreover, induces an enlarged anonymity set (See Section 3.5 for detailed explanation).

3.3 Our Approach for Hiding the Channel State without NIZK

The hidden state introduced in the last section must satisfy the requirements for ordinary channels and unlinkable payments. We summarize these requirements into five rules as below, where **R1**-**R3** are derived from the ordinary channels and **R4**-**R5** are necessary for the unlinkable payments supporting variable amounts.

- **(R1)** *Latestness identifiable*. Both channel users can efficiently identify the latest hidden state.
- **(R2)** *Balance-sufficiency attestable*. On the hidden-state side, both the tumbler and the user can confirm that the other has sufficient balances for each update. Thus, one successful payment via the tumbler means that the payer and the tumbler have sufficient deposits to support this procedure.
- **(R3)** *Authentication-validness verifiable*. Each hidden state generated by the update operation can be publicly verified that it has been authenticated by the tumbler.
- **(R4)** *Updatable*. The hidden state is updatable, which means that each user's balance has been changed according to the payment amount. The updated state is still in the hidden format, and can be publicly verified.
- **(R5)** *Content-binding hidden*. The tumbler can neither derive information from the hidden state nor link it to previous hidden states. Besides, each hidden state is bound to a unique channel and the channel's balance allocation.

Approaches to meeting R1-R2. At the first impression, generating the proof that a hidden state is the latest one is hard to achieve since both the channel identity and the channel balance are invisible to the tumbler. To resolve this dilemma, we utilize the fact that PCH users always want to protect their own benefits. Concretely, we design a way to ensure that each PCH user always provides the latest states to avoid the loss of its benefit. In other words, all PCH users benefit more from the latest state than the expired ones. We emphasize that this is quite *different* from the rational player assumption since the PCH system remains secure even when some irrational PCH users intentionally submit expired states, and only the irrational one suffers.

We can achieve this target when the tumbler / PCH-user only plays the sender / receiver role in a unidirectional channel. So, the tumbler-payee channel must be unidirectional and in the hidden form. Since the interaction with the tumbler is removed (refer to removing interaction on the hidden-state side in Section 3.2), a new state can only be provided by the payee, who always benefits more from a new state than an older one, satisfying the requirement **R1**. For simplicity, we also denote the payer-tumbler channel to be unidirectional in the rest of this paper.

This unidirectional tumbler-payee channel setting also facilitates the satisfaction of requirement **R2** since the payee suffers from benefits loss by submitting a state in which the tumbler's rest deposits are insufficient to pay the payee.

Approaches to meeting R3-R5. We propose a new cryptographic primitive called *randomizable signature of updatable commitments*, abbreviated as RSUC. Informally, it has the following features.

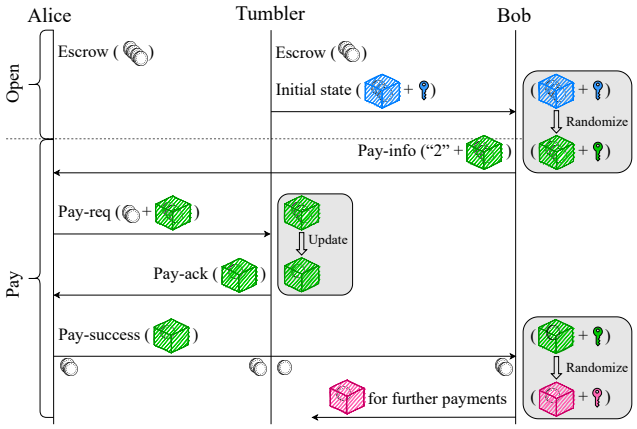

Figure 1: Overview of the channel open procedure and execution of one payment in Accio. In this figure, each box presents one hidden state. The key with the same color as the box presents the randomness used in the hidden state, and boxes of the same color have the same key.

- It allows the creator to hide a value and authenticate the hidden result, which outputs the *authorized hidden result*. The authentication can be publicly verified.
- Anyone can randomize the authorized hidden result without changing the value or violating the authentication of the creator. The most significant point is that even the creator cannot link the original authorized hidden result to any randomized one. Thus, the creator cannot identify the value of a given randomized result.
- Only the creator of the original authorized hidden result can update the value hidden in the randomized ones, i.e., adding / subtracting an extra value, and authenticate the update. It is worth noting that the creator achieves this publicly verifiable update without identifying the updated value in the randomized authorized hidden results.
- Each authorized hidden result, being randomized / updated or not, can be opened to only one value.

The operations of randomization and update in RSUC can be performed in an interweaving way. As a result, **R3-R5** are satisfied in the following ways.

- Encoding the tumbler-payee channel state into a value in RSUC allows the tumbler to hide and authenticate the initial state. The hidden state is then sent to the payee, which is used in the following updates. Later, the tumbler can update the (randomized) hidden state, with publicly verifiable results (satisfying the requirement **R3**).
- Given the update request from a payer, the tumbler can update the hidden state to a new one and authenticate it (satisfying the requirement **R4**).
- The tumbler cannot link one hidden state from its randomization. Since the hidden state can only be opened to one value, it uniquely identifies the channel and the channel balance allocation (satisfying the requirement **R5**).

3.4 Workflow with the Hidden State

We introduce Accio's workflow based on the introduction and analysis of the one-side hidden state in the last sections. Here we use the process of one payment from Alice to Bob via the Tumbler as an example. To illustrate the whole lifetime of the hidden state, we also introduce the channel open and close procedures. Figure 1 shows the open procedure and execution of one payment.

Open. To startup Accio, Alice and the Tumbler escrow coins to the Alice-Tumbler and Tumbler-Bob channels, respectively, in the same way as the ordinary PCHs. Then the Tumbler runs RSUC to convert the initial Tumbler-Bob channel's state (e.g., Tumbler has 3 coins in this channel initially) into the initial hidden state with Tumbler's authentication and sends the result with the hidden state's opening to Bob. After checking the authorization and correctness of the initial hidden state, Bob randomizes it to get a new hidden version. The open procedure runs only once to support all the subsequent update procedures. It is worth noting that the payer of one opened channel can perform payments with multiple payees, and vice versa. For example, the Alice-Tumbler channel can trade not only with the Tumbler-Bob channel, but also with any other Tumbler-payee channel. Below we use the payment with Bob as an example.

Payment. To perform the payment, Bob sends the payment requirement, including the payment amount (e.g., 2 coins) and the current authorized hidden state of the Tumbler-Bob channel to Alice in private. To start the payment, Alice sends the update request to the Tumbler, including the amount (e.g., 2 coins) that Alice would pay and the hidden state from Bob. Upon this update request, Tumbler gets two coins from Alice on the condition that Tumbler provides the updated (and also authorized) hidden state, in which the Tumbler-Bob channel state has updated consistently, i.e., Tumbler's balance is decreased by 2 and Bob's balance is increased by 2. Obtaining the correctly updated hidden state, Alice forwards it to Bob, who randomizes it again. Then Bob can withdraw coins from the Tumbler-Bob channel or launch the next payment.

It is worth noting that, during the payment, each hidden state (including the original version, randomized versions, and updated versions) is publicly verifiable with the Tumbler's authentication. If the Tumbler responds to Alice incorrectly (or refuses Alice's request), all the balances remain the same, which fails the payment.

Close. Both the Alice-Tumbler and the Tumbler-Bob channels can be closed in the same way as discussed in Section 2.2. To maximize its profit, Bob always launches the channel close by submitting the latest hidden state and revealing the plaintext of the hidden state to cash out all of its coins in the channel.

3.5 Analysis on the Achievement of Goals

In this section, we provide the intuition of the proofs of Accio's properties. The formal definition and proof are given in the UC model in Section 5 and 6.

Balance security. During each payment, the payer pays the tumbler and gets the updated hidden state, which the payee uses to get the equal amount from the tumbler-payee channel. Conversely, the tumbler can get the coins from the payer-tumbler channel only if providing the correct updated states to the payer. In this way, both sub-payments succeed or fail, guaranteeing atomicity. Furthermore, as ensured by RSUC, each hidden state can be opened to a single value, meaning that the payee could get the same amount of coins as the payers paid.

Variable-amount payment unlinkability. The critical reason for achieving unlinkability is that each payee randomizes every received hidden state, including the initial and the updated ones. From the view of the tumbler, all the randomized results generated by each payee are distributed the same. Thus, the tumbler cannot link one payer's update request to any initial hidden state or the previous hidden states, even knowing the payment amount.

It is worth noting that the unlinkability problem becomes more involved when the tumbler-payee channel is closed. In this case, the tumbler gets knowledge of the payees' cashout values. With the already known payment amounts from the payers, the tumbler may break the unlinkability. In Section 3.6, we claim that this kind of privacy leakage is unavoidable when using non-confidential payment amounts. More to this point, we propose the precondition for the definition of unlinkability in non-confidential blockchains.

Optimized-unlinkability. Accio resists the abort attack and enjoys the enlarged anonymous set size.

- For abort attacks, even if the tumbler refuses to update the state, the payer and payee cannot be linked since the tumbler only interacts with the payer and has no idea about users' identities on the side of payees.
- By removing epochs, each payer/payee relation hides among all payers/payees pairs participating in the mixing instead of only those within one epoch (such as [13, 16, 22, 27]). Therefore, the anonymity set has been enlarged.

NIZK-free. We realize RSUC with the state-of-the-art tools without NIZK and thus the unlinkable PCH construction. It is worth noting that NIZK is required to accomplish the UC-security proof. See Section 6 for more details.

Griefing attack resistance. Furthermore, the griefing attack is avoided. That is because, during the whole payment process, the tumbler does not need to lock coins in advance to facilitate the payment. As a result, no additional registration phase is necessary.

3.6 Unlinkability Precondition

Working on top of the non-confidential blockchains, the tumbler gets the following information straightforwardly.

- The users' initial in-channel balances are exposed to the blockchain, and the channel close always reveals the final allocation of balance within each channel. Thus, the tumbler knows the total amount transferred to the payee in each closed tumbler-payee channel. Besides, the tumbler in the PCH construction is aware of the received payment requests. All these common information leaks are unavoidable due to the non-confidential nature of all the current PCH constructions [13, 16, 22, 27].
- In Accio, the variable payment amount is also visible to the tumbler, and the amount itself may leak information. For example, suppose there are only two payees with the tumbler's escrowed in-channel funds being 1M coins and 1 coin respectively. Then for the payment request of more than 1

coin, its payee will be immediately identified. We emphasize that this kind of leakage cannot be avoided completely, as we have discussed in Section 3.2.

As a result, the discussion of unlinkability in PCHs only makes sense on the condition that the above-mentioned "natural leakage" will not ruin the unlinkability trivially. We refer to this condition as the *unlinkability precondition*, and provide a comprehensive description in Appendix B.

4 RANDOMIZABLE SIGNATURE OF UPDATABLE COMMITMENTS AND ITS APPLICATION TO ACCIO

In this section, we introduce the definition of RSUC and its constructions. Applying the primitive, we provide a more detailed description of the procedures of Accio.

4.1 Definition of Randomizable Signature of Updatable Commitments

Definition 1 (Randomizable Signature of Updatable Commitments). *A randomizable signature of updatable commitments scheme* RSUC *consists of a tuple of algorithms* (Setup, KeyGen, AuthCom, VfCom, VfAuth, RdmAC, UpdAC, VfUpd)*, the syntax of which is described as follows:*

- $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$: *A* PPT *algorithm that on input the security parameter λ, outputs the public parameters* pp*, which is the implicit input to the following algorithms.*
- $(\widehat{\mathsf{sk}}, \widehat{\mathsf{vk}}) \leftarrow \mathsf{KeyGen}(\mathsf{pp})$: *A* PPT *algorithm run by the tumbler. It outputs the authentication-verification key pair* $(\widehat{\mathsf{sk}}, \widehat{\mathsf{vk}})$.
- $(\mathsf{cm}, \widehat{\sigma}) \leftarrow \mathsf{AuthCom}(v, \widehat{\mathsf{sk}}; r)$: *A* PPT *algorithm run by the tumbler. On input the value v, the authentication key* $\widehat{\mathsf{sk}}$*, using the randomness r in the commitment, it outputs the commitment* cm *and the authentication $\widehat{\sigma}$ on the commitment. We call a* $(\mathsf{cm}, \widehat{\sigma})$ *pair the authenticated commitment.*
- $1/0 \leftarrow \mathsf{VfCom}(\mathsf{cm}, v, r)$: *A* DPT *algorithm that on input the commitment* cm*, the value v, and the randomness r, outputs 1 if the v is the committed value and 0 otherwise.*
- $1/0 \leftarrow \mathsf{VfAuth}(\mathsf{cm}, \widehat{\sigma}, \widehat{\mathsf{vk}})$: *A* DPT *algorithm that on input the authenticated commitment (*cm*, $\widehat{\sigma}$), and the verification key* $\widehat{\mathsf{vk}}$*, outputs 1 if the authentication is valid and 0 otherwise.*
- $(\mathsf{cm}', \widehat{\sigma}') \leftarrow \mathsf{RdmAC}(\mathsf{cm}, \widehat{\sigma}; r')$: *A* PPT *algorithm that on input the authenticated commitment* $(\mathsf{cm}, \widehat{\sigma})$*, using the randomization factor r', outputs the randomized result (*$\mathsf{cm}', \widehat{\sigma}'$*). It is worth noting that* cm *and* cm' *are the commitments of the same value v with different randomnesses. In Accio, it is run by the payee.*
- $(\mathsf{cm}_{\mathsf{new}}, \widehat{\sigma}_{\mathsf{new}}) \leftarrow \mathsf{UpdAC}(\mathsf{cm}, a, \widehat{\mathsf{sk}})$: *A* PPT *algorithm run by the tumbler. On input the commitment* cm*, the update value a, and the authentication key* $\widehat{\mathsf{sk}}$*, it outputs the new authenticated commitment* $(\mathsf{cm}_{\mathsf{new}}, \widehat{\sigma}_{\mathsf{new}})$*. In this way, the committed value v in* cm *is updated to $v + a$, committed in* $\mathsf{cm}_{\mathsf{new}}$*, and* $\widehat{\sigma}_{\mathsf{new}}$ *is the valid authentication on* $\mathsf{cm}_{\mathsf{new}}$.
- $1/0 \leftarrow \mathsf{VfUpd}(\mathsf{cm}, a, \mathsf{cm}_{\mathsf{new}}, \widehat{\sigma}_{\mathsf{new}}, \widehat{\mathsf{vk}})$: *A* DPT *algorithm that on input the commitment* cm*, the update value a, the new authenticated commitment (*$\mathsf{cm}_{\mathsf{new}}$, $\widehat{\sigma}_{\mathsf{new}}$*), and the verification key* $\widehat{\mathsf{vk}}$*, outputs 1 if the updated result is correct and 0 otherwise.*

Definition 2 (Secure RSUC). *A* RSUC *scheme is secure if it satisfies correctness, unforgeability of the authenticated hidden value, and randomization properties.*

Next, we explain the 3 properties of RSUC, and the formal definition of these properties can be found in Appendix C.

Correctness. It requires that all authenticated commitments generated from AuthCom, RdmAC, or UpdAC pass all the verifications, including VfAuth, VfCom, and VfUpd. Particularly, commitments in RdmAC's input and output share the same value, but different random factors, while commitments in VfUpd's input and output have the same random factor but different values. In other words, RdmAC (resp. VfUpd) changes AuthCom's output's random factor (resp. committed value).

Unforgeability of the authenticated hidden value. It requires that the adversary can forge neither a correct authenticated commitment without the knowledge of $\widehat{\mathsf{sk}}$ nor a different hidden value. (1) *Forge the authentication.* The adversary cannot forge a commitment whose "equivalence class" has not been authenticated without being detected. For the equivalence class of commitment cm, denoted as [cm], we mean all randomizations of cm, i.e., $[\mathsf{cm}] := \{\mathsf{cm}' | \exists r', (\mathsf{cm}', \widehat{\sigma}') \leftarrow \mathsf{RdmAC}(\mathsf{cm}, \widehat{\sigma}; r')\}$. (2) *Forge the hidden value.* For an authenticated commitment, the adversary cannot open it to another value.

Randomization. It requires that neither the randomization nor the update results of an authenticated commitment can be linked to the original version, even by the creator of the authenticated commitment. Note that the randomization property implies the value hidden property, i.e., the committed value cannot be derived from the commitment.

4.2 Realization of Randomizable Signature of Updatable Commitments

The concrete realization of RSUC is partly inspired by the primitive *signatures on randomizable ciphertexts* (SoRC), which is first introduced in [4] and improved in [2].

It is noteworthy that independently of us, BlindHub essentially uses the SoRC primitive to link one puzzle with the payment amount. Unlike BlindHub, our scheme completely abandons the puzzle-promising and puzzle-solving method, and uses a SoRC variant (i.e., RSUC) that exploits the additive homomorphic commitment. We refer readers to [2] for more details about SoRC and provide the concrete RSUC construction with proof in Appendix C.

4.3 Applying RSUC to Accio

To bootstrap the PCH service, the tumbler runs KeyGen to get the $(\widehat{\mathsf{sk}}, \widehat{\mathsf{vk}})$ pair. Below are the three detailed procedures in Accio, including channel open, payment, and channel close. For the formal definition and protocol description, we refer the readers to Section 5 and Section 6, respectively.

4.3.1 Open. To open a channel in the non-confidential blockchain, the sender of the channel, i.e., payer in the payer-tumbler channel and tumbler in the tumbler-payee channel, escrows coins to the channel. After the escrow transaction has been recorded on the blockchain, the channel is opened.

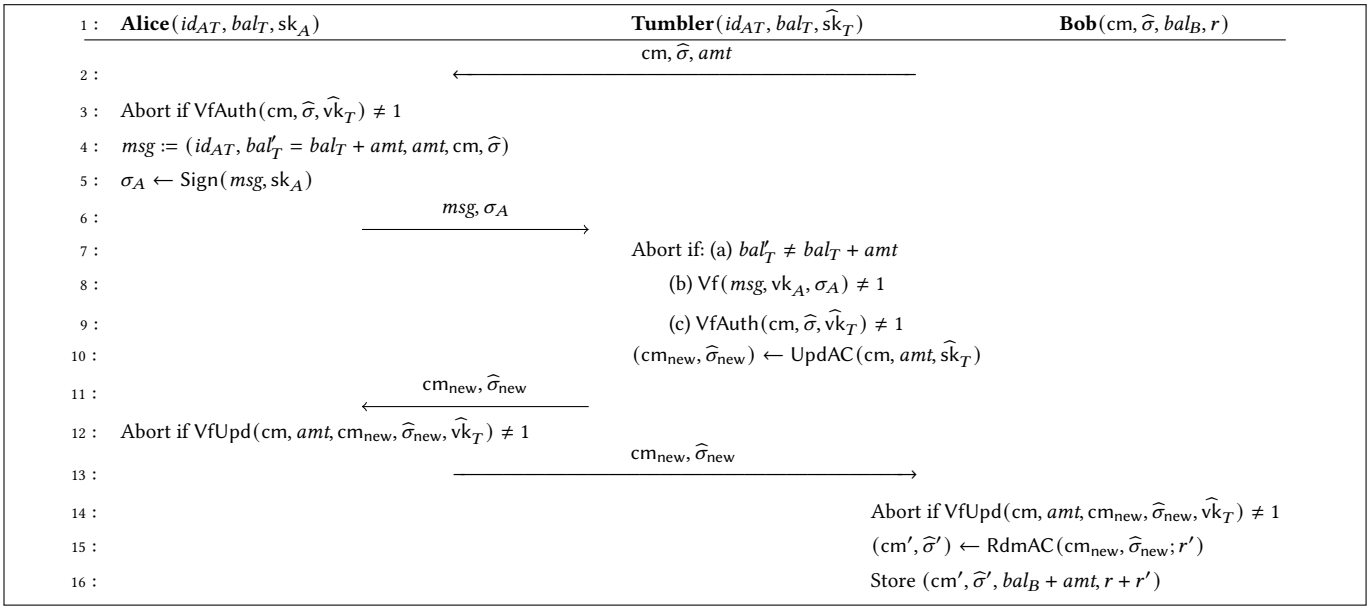

Figure 2: Payment procedure in Accio. In the figure, the payment of value amt is performed between payer Alice and payee Bob via Tumbler. $\widehat{\mathsf{vk}}_T$ is Tumbler's verification key used in RSUC, and vk_A is Alice's verification key used in the underlying blockchain. For the Alice-Tumbler channel, id_{AT} is the identifier of the Alice-Tumbler channel, and bal_T is Tumbler's balance in this channel, which both Alice and Tumbler know. For the Tumbler-Bob channel, $(\mathsf{cm}, \widehat{\sigma})$ is the current authorized hidden state, bal_B is Bob's balance and r is the randomness of cm, which are only known by Bob.

Concerning the tumbler-payee channel, the tumbler runs AuthCom with its $\widehat{\mathsf{sk}}$ to turn the initial plaintext channel state into the hidden channel state and authenticate it. Then the tumbler sends the authenticated commitment $(\mathsf{cm}, \widehat{\sigma})$ and the corresponding value and random factor, i.e., (v, r), to the payee. After checking $(\mathsf{cm}, \widehat{\sigma})$'s validity with VfCom and VfAuth, the payee uses RdmAC to get the first randomized hidden state.

To deploy the RSUC scheme, the plaintext channel state needs to be encoded into a value v in AuthCom. Since the plaintext channel state consists of *the channel identity id* and *the channel balance allocation* between the tumbler and the payee, both need to be contained by v. In addition, the channel identity should not be changed when updating the channel state, i.e., changing the channel balance allocation. Considering these two requirements, we encode the state as below:

$$v = \mathsf{H}(id)||bal,$$

where H is a secure (i.e., collision-resistant) hash function such that $\mathsf{H}(id)$ uniquely identifies the channel, and bal is the number of coins belonging to the payee. Since there are only two users in the channel, the tumbler's balance can be derived by subtracting the payee's balance bal from the channel's total deposits. So bal represents the channel balance allocation.

The encoding also helps in saving the cost for proving that the balance in the hidden state lies in the correct range. Roughly, we utilize the fact that the payee's balance keeps increasing during the protocol execution and the upper limit of the balance space is unattainable. Concretely, assume that v is n-bits long, with both $\mathsf{H}(id)$ and bal are $n/2$ bits long. We also assume that the off-chain payment value amt is m bits long (i.e., the maximum payment amount in each payment is $2^m - 1$). Then at least $2^{\frac{n}{2}-m}$ times of payments are required to *overflow* the balance space. For example, when $n = 256$ and $m = 32$, the user needs to perform at least 2^{96} times of payments, which is impractical.

Considering the case that once the payee's balance contained in the state exceeds the channel deposit (Note that this is not the overflow case as above), the state would not be accepted by the contract when the channel is closed, and therefore the payee would not get coins refunded. Thus, this encoding method expresses the channel state securely and precisely.

4.3.2 Payment. With the overview in Figure 1 in mind, we now detail the payment procedure in Figure 2. The core idea to achieve payment atomicity is that the payer pays the tumbler coins in exchange for the tumbler's update on the hidden state of the tumbler-payee channel. After getting the payment amount amt and the authorized commitment $(\mathsf{cm}, \widehat{\sigma})$ from the payee secretly (Line 2, Fig. 2), the payer checks the hidden state's validity with VfAuth (Line 3, Fig. 2). If the check passes, the payer sends a payment request to the tumbler to (1) pay the tumbler amt coins and (2) get the updated hidden state for the payee. In detail, the payer re-allocates the payer-tumbler channel's balances by adding amt more coins to the tumbler and signing the request with the signing key sk_A.

Specifically, the payer sends the updating request containing the authorized commitment (Line 4-6, Fig. 2). Then the tumbler checks the following (Lines 7-9, Fig. 2).

- The updating request contains the correct balance re-allocation result of the payer-tumbler channel.
- The updating request can be verified with the payer's verification key vk_A.

- The received authorized commitment can be verified with the tumbler's verification key $\widehat{\mathsf{vk}}_T$.

If all checks pass, the tumbler runs UpdAC to get the updated authorized commitment $(\mathsf{cm}_{\mathsf{new}}, \widehat{\sigma}_{\mathsf{new}})$, which is returned to the payer (Lines 10-11, Fig. 2).

By running VfUpd, the payer can check the validity of the updated authorized commitment (which contains the updated state in the tumbler-payee channel) and send it to the payee if the check passes (Lines 12-13, Fig. 2).

Upon receiving an updated authorized commitment, the payee uses VfUpd to check its validity. Note that the payee can always open the valid updated authorized commitment with the same random factor as cm. Regardless of starting the next payment, the payee runs RdmAC to avoid being linked by the tumbler. The payee records the new balance and random factor (Lines 14-16, Fig. 2).

In the payment procedure, there are several cases that need to be clarified about the behavior of the tumbler.

- The case that the tumbler does not reply or replies incorrectly to the payer. In this case, the payer initiates the channel *close*, and the update result can be derived from the response by the tumbler. If the updated state is submitted by the tumbler, the payment is completed and the payer forwards the updated hidden state to the payee. Therefore, the payer could get the update result in at most 2Δ rounds, Δ rounds for the payer to initiate the channel close and Δ rounds for the tumbler to submit the channel state. From the view of the payee, the waiting time for the payment result is at most $4 + 2\Delta$ rounds, where 2Δ rounds are spent on the payer getting the result via closing channel and 4 rounds are spent on the communications between the three parties, as shown in Figure 2. If the payment fails, the tumbler does not get the coins from the payer. At that time, the payee randomizes the current state for further payments or channel close.
- The case that the tumbler has insufficient balance in the tumbler-payee channel. Recall the encoding format introduced above and that each update derives a value $\mathsf{H}(id)||bal+amt$. If $bal+amt$ exceeds the total channel deposits, the state would not be accepted by contract when closing the channel. At that time, the payee gets no refunds and the payee's benefits would be damaged.

4.3.3 Close. The latest channel state is supposed to be submitted to the blockchain when the channel is closed. Both the tumbler-payee channel state and the payer-tumbler channel state contain the authorized commitment, as described below.

In the tumbler-payee channel, the payee with channel identifier id reveals the latest authenticated hidden state with its value and random factor, i.e., $(\mathsf{cm}, \widehat{\sigma}; bal, r)$, to launch channel close. After checking that cm can be opened with randomness r and message $\mathsf{H}(id)||bal$, and the state is valid with VfCom and VfAuth, both the tumbler and the payee can be refunded from the blockchain.

In the payer-tumbler channel, the state is in plaintext. As pointed out in the payment procedure, the updated authorized commitment is also contained in the state in case the payer needs to obtain it via the channel close. Hence, the state has the form $(msg, \sigma, \mathsf{cm}_{\mathsf{new}}, \widehat{\sigma}_{\mathsf{new}})$, where (msg, σ) is the payer's payment request, indicating that the state has been confirmed by the payer, and $(\mathsf{cm}_{\mathsf{new}}, \widehat{\sigma}_{\mathsf{new}})$ is the updated authorized state, verifiable by VfUpd.

To ensure the close is carried out safely, we should also deal with other situations. For example, the tumbler initiates close in the tumbler-payee channel while the payee's current payment has not finished. We provide detailed solutions in Section 6.

5 FORMAL DEFINITION OF ACCIO

In this section, we give the formal definition of Accio, including the security model and the ideal functionality. We assume that the system is used by a fixed set of parties $\mathcal{P} = \{P_1, P_2, ..., P_n\}$, and define a payment channel β as an attribute tuple (β.id, β.type, β.sender, β.receiver, β.fund, β.balance). Among the attributes, $\beta.\mathsf{id} \in \{0,1\}^*$ is the unique channel identifier, $\beta.\mathsf{type} \in \{\mathsf{payer}, \mathsf{payee}\}$ (the former means a payer-tumbler channel while the latter a tumbler-payee channel), β.sender and β.receiver denote the channel's sender and receiver respectively, β.fund denotes the funds escrowed to the channel by the channel sender, $\beta.\mathsf{balance} : \{\beta.\mathsf{sender}, \beta.\mathsf{receiver}\} \rightarrow \mathbb{R}^{\geq 0}$ maps the channel user to its balance in the channel. Furthermore, β.users denotes the set of both the sender and receiver.

5.1 Security Model

We use the UC framework [6], more concretely, the Global UC version [7], to model the security guarantees formally. The security model is defined with respect to the global ledger $\mathcal{L}(\Delta)$, where Δ is the blockchain delay. In the security model, the real world and the ideal world are defined. In the former, parties and the adversary $\mathcal{A}$ execute the protocol π and interact with the smart contract, which is modeled as the ideal functionality $\mathcal{C}$. In the latter world, the ideal functionality $\mathcal{F}$, which can be regarded as a trusted third party, receives inputs from parties, and the simulator $\mathcal{S}$, which plays the role of adversary and returns the output. The environment $\mathcal{Z}$ provides inputs to parties and collects their outputs for both worlds.

Security is proved using the indistinguishability between the real and ideal worlds. In a nutshell, since the outputs of $\mathcal{Z}$ running in the real world and the ideal world are computationally indistinguishable, and the security properties are naturally achieved in the ideal world due to the existence of an assumed and trusted functionality, the protocol then also achieves the desired security properties, which completes the proof.

5.1.1 Adversary Model. We assume that the adversary $\mathcal{A}$ can corrupt arbitrary parties at the beginning of the protocol. For corruption, we mean that $\mathcal{A}$ gets all messages sent to the corrupted parties and controls their behaviors.

5.1.2 Communication Network. We consider a synchronous communication network, where communication proceeds in discrete rounds, and each party is aware of the given round. We assume that parties communicate via the authenticated communication channel, and the message delivery requires 1 round. Using the channel, the source and integrity of the received message can be verified. For example, if party P sends a message to Q in round t, Q would receive it at the beginning of round $t + 1$, and Q is sure that the message is from P. The adversary can see the message content and change the order of messages sent in the same round, but it cannot modify, delay, or drop the messages between parties or insert a new

message. For other communications, such as those between one party and the ideal functionality, take 0 rounds. For simplicity, the computation takes 0 rounds.

Same as [13, 16, 22, 27], we assume that the payer and payee of one payment communicate using the anonymous communication channel, which means that the adversary cannot notice the communication. It is a natural assumption. Otherwise, the payment unlinkability is easily broken by observing the communication between parties. Besides, as inspired by [8–10], we use the presentation "reply to message M with message M'" to replace the session and sub-session identifiers to improve the readability.

5.1.3 Ledger and Contract Functionalities. Analogous to [8–10], we formalize the underlying blockchain as a global functionality $\mathcal{L}$, as shown below.

Ledger Functionality $\mathcal{L}$

Ledger initialization: Upon receiving $(x_1, ..., x_n) \in \mathbb{R}^n_{\geq 0}$ from the environment $\mathcal{Z}$, store the tuple.
Adding coins: Upon receiving (add, P_i, y), if $P_i \in \mathcal{P}$ and $y \in \mathbb{R}_{\geq 0}$ then set $x_i := x_i + y$, else do nothing.
Removing coins: Upon receiving (remove, P_i, y), if $P_i \in \mathcal{P}$ and $y \in \mathbb{R}_{\geq 0}$ and $x_i \geq y$ then set $x_i := x_i - y$, else reply with (nofunds) and stop.

The value $x_i (i = 1, ..., n)$ in tuple $(x_1, ..., x_n)$ recorded in the ledger indicates P_i's balance and can be accessed by all parties, the adversary, and the environment. The "add" and "remove" operations are performed by the ideal functionality $\mathcal{F}$ in the ideal world, and the contract functionality C in the real world. To model the non-immediate property of the cryptocurrency updates, functionalities impose a time delay on messages sent to the ledger. The exact number of rounds to send the command is determined by the adversary $\mathcal{A}$. We denote a ledger functionality $\mathcal{L}$ with maximal delay $\Delta \in \mathbb{N}$ as $\mathcal{L}(\Delta)$.

The contract functionality formalizes the smart contracts, leading to C-hybrid real world. Each contract is identified by its corresponding channel and $C(\beta.\mathsf{id})$ denotes the contract of channel β. Receiving messages from parties, the functionality updates parties' on-chain funds according to predefined rules.

5.1.4 Security and Privacy Goals. Our security goal is to keep the honest parties from losing coins, and the privacy goal is to prevent the tumbler from linking the payer and payee of one payment. We emphasize again that both goals are achieved under variable payment amounts. Formally, our goals are listed as follows.

Balance security. The system should not be exploited to print new money or steal existing money from honest parties, even when parties collude.

Variable-amount payment unlinkability. When the unlinkability precondition is satisfied (see Section 3.6), with no collusion with others, the tumbler should not learn information that allows it to associate the payer and the payee of one payment.

5.1.5 Security Definition. Denote λ as the security parameter. With respect to the global ledger $\mathcal{L}(\Delta)$, let $\mathsf{EXEC}^{\mathcal{L}(\Delta),C}_{\pi,\mathcal{Z},\mathcal{A}}(\lambda)$ be the output of the environment $\mathcal{Z}$ when interacting with the adversary $\mathcal{A}$ and the protocol π in the C-hybrid world, and $\mathsf{EXEC}^{\mathcal{L}(\Delta)}_{\mathcal{F},\mathcal{Z},\mathcal{S}}(\lambda)$ be the output when interacting with the ideal functionality $\mathcal{F}$ and the simulator $\mathcal{S}$. Security is defined as follows.

Definition 3. *A protocol π running in the C-hybrid world UC-realizes the ideal functionality $\mathcal{F}$ with respect to a global ledger $\mathcal{L}(\Delta)$, if for any* PPT *adversary $\mathcal{A}$ there exists a simulator $\mathcal{S}$ such that*

$$\mathsf{EXEC}^{\mathcal{L}(\Delta),C}_{\pi,\mathcal{Z},\mathcal{A}}(\lambda) \approx \mathsf{EXEC}^{\mathcal{L}(\Delta)}_{\mathcal{F},\mathcal{Z},\mathcal{S}}(\lambda),$$

where $\approx$ denotes that the two distributions are computationally indistinguishable.

5.2 Ideal Functionality of Accio

The ideal functionality $\mathcal{F}$ uses a set Γ to record channels, and maintains the balance history for each channel. There are 3 procedures, *(A) Open*, *(B) Payment*, and *(C) Close* in the functionality, as shown in Figure 3. Among the 3 procedures, the open procedure starts when invoked by the sender of the channel, the payment procedure handles the payment between the payer and payee via the tumbler, and the close procedure refunds users' balances in the channel to the blockchain when invoked by either channel user. Considering the privacy goal, information leaked to $\mathcal{S}$ is specially defined. Furthermore, when we say that the ideal functionality operates within a time limit, we mean that the ideal functionality waits for the instruction from the simulator within that time limit and then starts the operation. This modeling states that the exact time when a message arrives at the ledger is determined by the adversary $\mathcal{A}$.

We denote the tumbler as T. To simplify the presentation, we assume that there is at most one channel between a user and the tumbler. It means that the user uniquely identifies the channel. The assumption is only for the simplification of the presentation and can be eliminated by identifying one channel by its identifier. Now we are ready to analyze the achievement of goals.

Ideal Functionality $\mathcal{F}$

(A) Open

Upon receiving (open, η) from η.sender: Ignore if η is not in the correct format or η.id is not unique or η.sender has insufficient coins on $\mathcal{L}(\Delta)$ for η.fund. If η.sender is in the open procedure of multiple channels at the same time, we require η.sender has enough funds on $\mathcal{L}(\Delta)$ for all channels being created. Otherwise, send (open, η) to $\mathcal{S}$, within Δ rounds send (remove, η.sender, η.fund) to $\mathcal{L}(\Delta)$, output (opened, η) to η.sender, and set $\Gamma(\eta.\mathsf{id}) := \eta$. If η.type = payee, η.sender is corrupt, and receive the reply (no-state, η.id) from $\mathcal{S}$, ignore the following messages sent to η.receiver concerning β. Otherwise, output (opened, η) to η.receiver.
/* The (no-state) message indicates that the corrupt tumbler does not send the correct initial channel state to the payee. */

(B) Payment

1. Upon receiving (pay-payee, A, amt) from B, retrieve γ from Γ where $B = \gamma$.receiver, $T = \gamma$.sender, and γ.type = payee, and proceed as follows,
   1) When B is honest, ignore if $\gamma = \perp$ or $\gamma.\mathsf{balance}(T) < amt$ or B is in γ's other procedures.
   2) When B is corrupt and T is honest and γ exists, if receive (update-basis, B, bal_B) from $\mathcal{S}$ and bal_B is in γ's balance history, update γ's state upon bal_B in this payment, else, abort.

/* The (update-basis) message indicates that the corrupt payee performs the payment upon a previous channel state. */

3) If A is corrupt, send (pay-payee, A, amt, B) to $\mathcal{S}$, else, send (pay-payee, $\perp$, $\perp$, $\perp$) to $\mathcal{S}$.

4) If receive (fake-payee, A, amt, B, B') from $\mathcal{S}$ and both B and B' are corrupt, store (A, amt, B') and within $4+2\Delta$ rounds wait for messages in Step 2. Otherwise, store (A, amt, B) and within $4+2\Delta$ rounds wait for messages in Step 2.

/* The (fake-payee) message indicates that the corrupt payee performs the payment on behalf of another corrupt payee. */

2. Upon receiving (pay-payer, B, amt) from A, retrieve β from Γ where $A = \beta$.sender, $T = \beta$.receiver, and β.type = payer, and proceed as follows,

1) When A or T is honest, ignore if $\beta = \perp$ or β.balance(A) $< amt$ or A (or T) is in β's other procedures.

2) If A is corrupt and B is honest, set $\tau := 1+2\Delta$. Otherwise, set $\tau := 0$. If (a) receive (fake-payer, A, A', amt, B) from $\mathcal{S}$ and both A and A' are corrupt and have stored (A', amt, B) within τ rounds, or (b) have stored (A, amt, B) within τ rounds, proceed. Otherwise, abort.

/* The (fake-payer) message indicates that the corrupt payer performs the payment on behalf of another corrupt payer. */

3) If B is corrupt, send (pay-payer, A, amt, B) to $\mathcal{S}$. Else, send (pay-payer, A, amt, $\perp$) to $\mathcal{S}$.

4) If T is corrupt, wait for $\mathcal{S}$'s reply within $1+\Delta$ rounds and go to Step 3 / 4. Else, set β.balance(A) := β.balance(A) $-amt$, β.balance(T) := β.balance(T)+ amt, output (paid, A, amt) to T and (paid) to A.

5) Abort if $\gamma = \perp$, or A is corrupt and not receive the reply (send-update) from $\mathcal{S}$ within $3+2\Delta$ rounds after receiving B's request.

/* The (send-update) message indicates that the corrupt payer forwards the payment result to the payee. */

6) If B is honest, set γ.balance(T) := γ.balance(T) $-$ amt, γ.balance(B) := γ.balance(B) + amt, output (paid) to B, and stop. Otherwise, denote bal_B as the received updated basis, add (bal_B + amt) to γ's balance history, and stop.

3. Upon receiving the reply (pay-not-ack, A, bal_T, T) from $\mathcal{S}$: If bal_T is in β's balance history and $0 \le bal_T \le \beta$.fund, set $rfd := bal_T$, else, set $rfd := \beta$.fund. Within Δ rounds, send (add, A, rfd) and (add, T, β.fund $-rfd$) to $\mathcal{L}(\Delta)$, and remove β from Γ.

4. Upon receiving the reply (pay-ack, A, amt, T) from $\mathcal{S}$:

1) Set β.balance(A) := β.balance(A) $-$ amt, β.balance(T) := β.balance(T) + amt, add β.balance(T) to β's balance history, output (paid) to A. If receive the reply (chan-close, β.id) from $\mathcal{S}$, send (add, A, β.balance(A)) and (add, T, β.balance(T)) to $\mathcal{L}(\Delta)$ within Δ rounds, and remove β from Γ.

2) Abort if $\gamma = \perp$, or when A is corrupt, not receive the reply (send-update) from $\mathcal{S}$ within $3+2\Delta$ rounds after receiving B's request.

3) If B is honest, set γ.balance(T) := γ.balance(T) $-$ amt, γ.balance(B) := γ.balance(B) + amt, output (paid) to B, and stop. Otherwise, denote bal_B as the received update basis, add (bal_B + amt) to γ's balance history, and stop.

(C) Close

Upon receiving (close, id) from P: Set $\eta := \Gamma(id)$ and ignore if no such η exists or $P \notin \eta$.users or η is in the close procedure. Send (close, id, P) to $\mathcal{S}$. If P is η.receiver, set $tw := \Delta$. Else, if η.type = payer, set $tw := 2\Delta$, else, set $tw := 4+4\Delta$. Distinguish the following cases,

1) Case both η.users are corrupt: If receive (close-bal, id, bal) from $\mathcal{S}$ and $0 \le bal \le \eta$.fund, set $rfd := bal$, else, set $rfd := 0$. Within Δ rounds, send (add, η.sender, η.fund$-$ rfd) and (add, η.receiver, rfd) to $\mathcal{L}(\Delta)$, remove η from Γ, and stop.

2) Case η.receiver is honest: Finish the potential ongoing payment. Send (close-bal, id, η.balance(η.receiver)) to $\mathcal{S}$, and within tw rounds, send (add, η.sender, η.balance(η.sender)) and (add, η.receiver, η.balance(η.receiver)) to $\mathcal{L}(\Delta)$, output (closed, id) to η.users, remove η from Γ, and stop.

3) Case η.receiver is corrupt: If receive (close-bal, id, bal) from $\mathcal{S}$, where bal is in η's balance history and $0 \le bal \le \eta$.fund, set $rfd := bal$. Otherwise, set $rfd := 0$. Send (add, η.sender, η.fund $-$ rfd) and (add, η.receiver, rfd) to $\mathcal{L}(\Delta)$ within tw rounds, output (closed, id) to η.users, remove η from Γ, and stop.

Figure 3: Ideal functionality $\mathcal{F}$

Balance security. In the open procedure, coins escrowed to the channel serve as the sender's initial channel balance. When performing one payment, the states of both the payer-tumbler and the tumbler-payee channels are updated according to the same payment amount. When the payee is corrupt, the balance update may be processed based on a historical state. Due to the unidirectional nature of the tumbler-payee channel, the tumbler would get more coins and the payee loses coins. When the payer is corrupt, a remarkable case is that only the payer-tumbler channel state is updated while the tumbler-payee channel is not. This corresponds to the case in the real world where the corrupt payer obtains the updated state but does not forward it to the payee. In $\mathcal{F}$, it is modeled by not receiving (send-update) from the simulator. At that time, only the corrupt payer loses coins, and honest parties do not.

Besides, there are cases where a corrupt payee performs the payment on behalf of another corrupt payee, and a corrupt payer on behalf of another corrupt payer. Although the protocol execution is not affected from the view of honest parties, these behaviors affect the balance allocation in related channels. These cases are considered in $\mathcal{F}$ by (fake-payer/payee) messages.

When a channel is closing, the on-chain refunds are the latest channel balances, indicating that no honest parties lose coins. Especially, if the channel receiver is corrupt, the refunded balance is selected from the channel balance history, and the sender may get more coins than deserved, resulting in the loss of the receiver itself. Reflecting in the real world, it corresponds to the channel receiver sending expired or invalid states to the contract for the channel close procedure. Furthermore, each user's refund is in the correct range, and the summary of the two users' refunds is precisely the total channel funds, indicating no printing money.

Variable-amount payment unlinkability. During one payment, the tumbler is only aware of the payer. Especially, when the tumbler is corrupt, and both the payer and payee are honest, i.e., the tumbler has no collusion with payers/payees, $\mathcal{F}$ leaks no information about the payee to $\mathcal{S}$, meaning the corrupt tumbler would not know the identity of the payee. Therefore, unlinkability is achieved.

6 DETAILED DESCRIPTION OF ACCIO

This section provides the detailed description and the security theorem about Accio, with the formal protocol shown in Figure 4 and the contract functionality in Figure 5.

6.1 Procedures of Accio

Setup. Before the protocol execution, the tumbler generates the authentication-verification key pair and broadcasts the authentication key to start the service. At the same time, the tumbler sends the proof of knowledge of the authentication key to guaranteeing the provision of the hub service. Users enroll in the system after verifying the proof.

Open. To open one channel, the channel sender submits the channel open request to the contract, which checks the request and sends (opened) to users if checks pass (Procedure (A), Fig.5). Here the expression that "the contract sends messages to users" is the modeling of users checking the blockchain for transactions.

Especially, if the channel is the tumbler-payee channel, after it is opened, the tumbler sends the initial state to the payee (Step 2, Fig.4(A)). From the view of the payee, if the correct initial state is not received from the tumbler within 1 round after the channel is opened, the payee ignores the following messages concerning this channel (Step 3, Fig.4(A)).

Payment. The payment workflow is the same as described in Section 3.4 and 4.3. Differently, for achieving the UC-security proof (which would be analyzed in Section 6.2), the payee needs to provide the NIZK proof of the knowledge of the hidden state's opening. The payer checks the correctness of the proof before executing the payment. Furthermore, the payer sends the proof along with the payment request to the tumbler, who also checks the correctness of the proof before updating the channel state.

From the payee's view, the waiting time bound for one payment should be specified to avoid a long waiting time. In the worst case, the payer needs to obtain the update result by closing the channel and observing the state submitted by the tumbler (Step 6, Fig.4(B)). It costs at most 2Δ time, where the channel close initiation needs Δ time, and the tumbler's response needs the other Δ time. Therefore, from the view of the payee, at most $4 + 2\Delta$ time is required to get the updated hidden state, where 4 rounds are spent on the communications between the three parties, as shown in Figure 2. One payment is regarded as failed if no or incorrect response is received before the time limit (Step 2, Fig.4(B)).

Close. To close the channel, the channel receiver submits the channel state (or responds to the close request from the sender) to the contract, which checks its correctness and refunds users accordingly (Procedure (B), Fig. 5).

Considering that before the state submission, the receiver of the tumbler-payee channel needs to finish the possible ongoing payment, which costs at most $4 + 2\Delta$ time. Therefore, we extend the waiting time for the payee's response in the contract. Furthermore, to prevent the tumbler from identifying the payee of one payment via the channel close, we enforce all payees to respond to the close request after $4+2\Delta$ rounds (Step 1(2), Fig.4(C)). Concretely, when the tumbler receives the payment request from the payer, the tumbler refuses the update, which causes the corresponding payee to wait for more time for the update result. To identify the real payee, the tumbler can close channels with all payees and see which payee responds to the close request after the payer-tumbler channel is closed. With the countermeasures, the difference in payees' responding time is eliminated, therefore resisting the attack.

(A) Open

The channel sender, denoted as P, proceed as follows,

1. Upon receiving $(\texttt{open}, \eta)$ from $\mathcal{Z}$: Ignore if η has incorrect format, or η.id is not unique, or P is not η.sender, or has less than η.fund coins on $\mathcal{L}(\Delta)$. Else, send $(\texttt{open}, \eta)$ to C.
2. If receive $(\texttt{opened}, \eta)$ from C within Δ rounds: Set $\Gamma^P(\eta.\text{id}) := \eta$, and output $(\texttt{opened}, \eta)$. If η.type = payee, generate $(\text{cm}, \widehat{\sigma}) \leftarrow \text{AuthCom}(\text{H}(\eta.\text{id})||0, \widehat{\text{sk}}_P; r)$, and send $(\texttt{chan-state}, \eta.\text{id}, \text{cm}, r, \widehat{\sigma})$ to η.receiver. Stop.

The channel receiver, denoted as Q, proceed as follows,

3. Upon receiving $(\texttt{opened}, \eta)$ from C: When β.type = payee, if receive $(\texttt{chan-state}, \eta.\text{id}, \text{cm}, r, \widehat{\sigma})$ from η.sender where $\text{VfCom}(\text{cm}, \text{H}(\eta.\text{id})||0, r) \wedge \text{VfAuth}(\text{cm}, \widehat{\sigma}, \widehat{\text{vk}}) = 1$, set $(\text{cm}', \widehat{\sigma}') := \text{RdmAC}(\text{cm}, \widehat{\sigma}; r')$, η.state $:= (\text{cm}', \widehat{\sigma}', 0, r + r')$. Else, abort.
4. Set $\Gamma^Q(\eta.\text{id}) := \eta$, output $(\texttt{opened}, \eta)$, and stop.

(B) Payment

The receiver B proceeds as follows,

1. Upon receiving $(\texttt{pay-payee}, A, amt)$ from $\mathcal{Z}$: Retrieve γ from Γ^B where γ.type = payee, γ.sender = T, γ.receiver = B, and $amt \le \gamma.\text{balance}(T)$. Ignore if $\gamma = \perp$ or γ is in other procedures. Else, let $(\text{cm}, \widehat{\sigma}, bal_B, r) := \gamma$.state, generate the proof π of the knowledge of the opening of cm, send $(\texttt{pay-info}, \text{cm}, \widehat{\sigma}, amt, \pi)$ to A.
2. Within $4 + 2\Delta$ rounds, if receive the reply $(\texttt{pay-success}, \text{cm}_{\text{new}}, \widehat{\sigma}_{\text{new}})$ from A where $\text{VfUpd}(\text{cm}, amt, \text{cm}_{\text{new}}, \widehat{\sigma}_{\text{new}}, \widehat{\text{vk}}_T) = 1$, set $(\text{cm}', \widehat{\sigma}') \leftarrow \text{CRdm}(\text{cm}_{\text{new}}, \widehat{\sigma}_{\text{new}}; r')$, γ.state $:= (\text{cm}', \widehat{\sigma}', bal_B + amt, r + r')$, $\gamma.\text{balance}(T) := \gamma.\text{balance}(T) - amt$, and output $(\texttt{paid})$. Otherwise, set $(\text{cm}', \widehat{\sigma}') \leftarrow \text{CRdm}(\text{cm}, \widehat{\sigma}; r')$, γ.state $:= (\text{cm}', \widehat{\sigma}', bal_B, r + r')$, and stop.

The sender A proceeds as follows,

3. Upon receiving $(\texttt{pay-payer}, B, amt)$ from $\mathcal{Z}$: Retrieve β from Γ^A where β.type = payer, β, sender = A, β.receiver = T, and $amt \le \beta.\text{balance}(A)$. Ignore if $\beta = \perp$ or β is in other procedures.
4. Within 1 round, if receive $(\texttt{pay-info}, \text{cm}, \widehat{\sigma}, amt, \pi)$ from B where $\text{VfAuth}(\text{cm}, \widehat{\sigma}, \widehat{\text{vk}}_T) = 1$ and π is correct, set $bal_T := \beta.\text{balance}(T) + amt$, $msg = (\beta.\text{id}, bal_T, amt, \text{cm}, \widehat{\sigma})$, $\sigma_A \leftarrow \text{Sign}(\text{sk}_A, msg)$, send $(\texttt{pay-req}, msg, \sigma_A, \pi)$ to T. Otherwise, abort.
5. Within 1 round, if receive the reply $(\texttt{pay-ack}, \text{cm}_{\text{new}}, \widehat{\sigma}_{\text{new}})$ from T with $\text{VfUpd}(\text{cm}, amt, \text{cm}_{\text{new}}, \widehat{\sigma}_{\text{new}}, \widehat{\text{vk}}_T) = 1$, set $\beta.\text{balance}(A) := \beta.\text{balance}(A) - amt$, $\beta.\text{balance}(T) := \beta.\text{balance}(T) + amt$, reply $(\texttt{pay-success}, \text{cm}_{\text{new}}, \widehat{\sigma}_{\text{new}})$ to B, output $(\texttt{paid})$, and stop.
6. Otherwise, initiate β's close. If β is closed with $(\text{cm}_{\text{new}}, \widehat{\sigma}_{\text{new}})$ where $\text{VfUpd}(\text{cm}, amt, \text{cm}_{\text{new}}, \widehat{\sigma}_{\text{new}}, \widehat{\text{vk}}_T) = 1$, reply $(\texttt{pay-success}, \text{cm}_{\text{new}}, \widehat{\sigma}_{\text{new}})$ to B, and output $(\texttt{paid})$. Remove β from Γ^A and stop.

The tumbler T proceeds as follows,

7. Upon receiving $(\texttt{pay-req}, msg = (id, bal_T, amt, \text{cm}, \widehat{\sigma}), \sigma_A, \pi)$ from A: Set $\beta := \Gamma^T(id)$. Ignore if $\beta = \perp$ or β is in other procedures or β.type $\neq$ payer or β.sender $\neq A$ or β.receiver $\neq T$ or $\beta.\text{balance}(A) < amt$ or $\beta.\text{balance}(T) + amt \neq bal_T$ or $\text{Vf}(msg, \sigma_A, \text{vk}_A) \wedge \text{VfAuth}(\text{cm}, \widehat{\sigma}, \widehat{\text{vk}}_T) \neq 1$ or π is incorrect. Otherwise, generate $(\text{cm}_{\text{new}}, \widehat{\sigma}_{\text{new}})$ via $\text{UpdAC}(\text{cm}, amt, \widehat{\text{sk}})$, set β.state $:= (msg, \sigma_A, \text{cm}_{\text{new}}, \widehat{\sigma}_{\text{new}})$, $\beta.\text{balance}(A) := \beta.\text{balance}(A) - amt$, $\beta.\text{balance}(T) := \beta.\text{balance}(T) + amt$, reply $(\texttt{pay-ack}, \text{cm}_{\text{new}}, \widehat{\sigma}_{\text{new}})$ to A, output $(\texttt{paid}, A, amt)$, and stop.

(C) Close

The close initiator P proceeds as follows,

1. Upon receiving $(\texttt{close}, id)$ from $\mathcal{Z}$: Set $\eta := \Gamma^P(id)$. Ignore if $\eta = \perp$ or η is in the close procedure. Finish the ongoing payment and distinguish the following cases,

1) Case $P = \eta$.sender: Send (close, $\perp$) to $C(id)$. Set $tw = 0$ if η.type = payer and $tw = 4 + 2\Delta$ otherwise. If not receive the reply (closed) from $C(id)$ within $2\Delta + tw$ rounds, send (timeout) to $C(id)$, output (closed, id) upon receiving it from $C(id)$, set $\Gamma^P(id) := \perp$, and stop.

2) Case $P = \eta$.receiver: If η.type = payee, wait until $4 + 2\Delta$ rounds. Send (close, η.state) to $C(id)$, output (closed, id) upon receiving it from $C(id)$, set $\Gamma^P(id) := \perp$, and stop.

The close responder Q proceeds as follows,

2. Upon receiving (closing) from $C(id)$: Set $\eta := \Gamma^Q(id)$. Finish the potential ongoing payment. If η.type = payee, wait until $4 + 2\Delta$ rounds. Send (close-resp, η.state) to $C(id)$, output (closed, id) upon receiving it from $C(id)$, set $\Gamma^Q(id) := \perp$, and stop.

3. Upon receiving (closed) from $C(id)$: output (closed, id), set $\Gamma^Q(id) := \perp$, and stop.

Figure 4: Procedures (A) Open, (B) Payment, and (C) Close.

Contract Functionality C

(A) The contract for channel open

Upon receiving (open, η) from η.sender: Ignore if η is not in the correct format or η.id is not unique or η.sender has insufficient coins on $\mathcal{L}(\Delta)$ for the deposit. Otherwise, within Δ rounds, send (remove, η.sender, η.fund) to $\mathcal{L}(\Delta)$. If the operation fails due to insufficient funds, then stop. Else, send (opened, η) to η.users, and go to point (B).

(B) The contract execution for $C(\eta.\text{id})$

Upon receiving (close, ST) from P: Ignore the message if $P \notin \eta$.users. Otherwise, distinguish the following cases,

/*The definition of the channel state ST varies according to the channel type. See Section 4.3 for the state phrasing.*/

1. Case P is η.receiver: If ST is a valid state, extract bal_r from ST, else, set $bal_r := 0$, $ST := \perp$. Within Δ rounds, send (add, η.sender, η.fund $- bal_r$) and (add, η.receiver, bal_r) to $\mathcal{L}(\Delta)$, send (closed, ST) to η.users. Close the contract instance.

2. Case P is η.sender: Within Δ rounds, send (closing) to η.receiver, set $tw := 0$ if η.type = payer and $tw := 4 + 2\Delta$ otherwise. Wait for one of the following two messages,

1) (close-resp, ST) from η.receiver within $\Delta + tw$ rounds: If ST is a valid state, extract bal_r from ST, else, set $bal_r := 0$, $ST := \perp$. Within Δ rounds, send (add, η.sender, η.fund $- bal_r$)) and (add, η.receiver, bal_r) to $\mathcal{L}(\Delta)$, send (closed, ST) to η.users. Close the contract instance.

2) (timeout) after $2\Delta + tw$ rounds: Within Δ rounds, send (add, η.sender, η.fund) to $\mathcal{L}(\Delta)$, send (closed, $\perp$) to η.users, and close the contract instance.

Figure 5: Contract functionality C.

6.2 Theorem

Theorem 1. *The protocol running in the C-hybrid world UC realizes the ideal functionality $\mathcal{F}$ with respect to the global ledger $\mathcal{L}(\Delta)$.*

Proof (Sketch). The complete proof is in Appendix D and we provide the proof sketch here. The main difficulty of the simulation lies in simulating the hidden state, which is two-fold.

- Firstly, in a payment involving a corrupted payee, the simulator must extract the channel that the provided hidden state corresponds to. Similar to the claim in [13], proving UC-security requires a "trapdoor mechanism" for the simulator to link adversarial sessions during simulation. The lack of the "trapdoor mechanism" flaws [27]'s security proof.
- Secondly, facing an honest payee, the simulator needs to provide the state of the tumbler-payee channel in cases of the payment or channel close, which is in the hidden format. However, due to the unlinkability, the simulator is unaware of all payments that the payee has been involved in and, therefore, of the current state at that time.

To solve the first difficulty, we add a NIZK proof when the payee provides the hidden channel state. Concretely, the payee proves the knowledge of the hidden state's opening. The payer performs the payment after verifying the correctness of the proof and forwards the proof to the tumbler, who also checks the proof's correctness before executing the payment. From the proof, the simulator could extract the committed value and therefore the corresponding channel identifier and balance. Following this, the extraction is achieved.

To solve the second difficulty, the simulator generates the hidden state as follows. Inspired by A^2L^{UC} where the simulator gets the tumbler's puzzle-solving key from a UC-secure two-party computation protocol, we let the simulator obtain the tumbler's authentication key used in RSUC. Concretely, when the system starts, the tumbler provides users a UC-secure NIZK proof on the authentication key. From this proof, the simulator recovers the authentication key and can generate the hidden state itself.

During the payment process, the simulator generates a hidden state with a randomly selected value and signs on it. The environment cannot distinguish the simulated state from the actual state used in the real world. Otherwise, it can be invoked to violate the randomization property of RSUC. When closing the channel, the simulator generates the hidden state using the channel identifier and the revealed channel balance and signs on it.

We emphasis that all the above NIZK proofs are only for the UC-security reason and can be eliminated if the construction is only required to be game-based or property-based secure. We provide the security proof in the game-based setting in Appendix E. □

7 PERFORMANCE EVALUATION

We evaluate the performance of Accio, including its on-chain and off-chain costs. The PCH parties pay on-chain costs when interacting with the blockchain, including the open and close procedures, which are conducted only once during the lifetime of each channel. The off-chain part includes the communication and computation cost of the one-time generating of the initial hidden state and the payment cost. To measure the effectiveness of our scheme, we also compare the on-chain / off-chain payment cost with the state-of-the-art fixed-amount and variable-amount PCH constructions.

7.1 On-chain Cost Evaluation and Comparison

Implementation. Using the Solidity language, we implement the contract functionality in Figure 5 upon Ethereum. The instantiation of RSUC is based on the curve *BN-128*. This curve supports our instantiation in Appendix C, and Ethereum introduces precompiled contracts for operations on this curve [23].

Approach to evaluation. The on-chain cost is measured by fees paid to the blockchain miners for the contract calls. In Ethereum,

the cost is calculated by *gas*, which depends on the data volume and the computation complexity of the contract calls. We set the gas price as 4 Gwei and the exchange rate to 1200 USD per Ether (as of Jan. 2023). The contract deployment costs 2.04M gas ($9.79). The deployment cost can be reduced by the technique in [9], and we focus on the protocol running costs below. Since channels in Accio are unidirectional and the channel users' operations are different according to their roles, i.e., the sender or the receiver, we measure their on-chain costs on each procedure, respectively. Furthermore, the cost varies with the channel type and mainly lies in verifying the channel state provided by the receiver in cases of initiating the channel close or responding to the close initiated by the sender. Therefore, the costs of receivers in responding to the close are further distinguished by the channel type. See results in Table 2.

Gas costs. From the results, it can be seen that the channel open costs the sender 57k gas ($0.27). For channel close, it costs the sender around 51k gas ($0.25) to initiate the close. As for the receiver, the cost for submitting the channel state is below 622k ($2.99), similar for both channel types. It can also be seen that during the channel's lifetime, the receiver costs more than the sender.

Gas cost comparison. We compare Accio's on-chain cost with other privacy-preserving PCH schemes. We note that Bolt only runs on anonymous chains and is not comparable to the schemes mentioned below.

- Built upon Ethereum, Perun costs 756k / 210k gas in the pessimistic / optimistic case respectively over the channel lifetime with 1 virtual channel [9].
- For the Bitcoin-compatible privacy-preserving PCH constructions, such as A^2L and BlindHub, they require 4 on-chain transactions during the channel lifetime, including the funding transaction, the commit transaction, the split transaction, and the adaptor execution delivery transaction or the timeout transaction. To compare these constructions with Accio, we measure their costs using the normal Ethereum transaction cost, which costs 21,000 gas for each transaction. So these schemes cost 84k gas.

Therefore, Accio's channel operation cost is less than Perun in the pessimistic case, but larger than the Bitcoin-compatible solutions.

7.2 Off-chain Cost Evaluation and Comparison

Evaluation details. The code is written in C programming language and uses the MCL [25] library to accomplish operations on curve *BN-128*. The performance is evaluated in a LAN network, using a machine with a 3.20GHz AMD 5800H processor with 8 cores, and 40GB RAM, running Ubuntu 22.04.1 LTS. For comparison, we also run the A^2L code in the same setting and distinguish the costs by users' roles. Especially, the running time is averaged from 100 runs. Results of performing one payment are shown in Table 3. Besides, in the tumbler-payee channel, the tumbler spends 0.69KB and negligible time to generate the initial hidden state, and then the payee costs 0.004s to verify it. We emphasize that this procedure runs only once during the lifetime of each tumbler-payee channel.

Table 2: On-chain gas costs of the open and close procedures. The receiver's cost in channel close is presented in the format of cost in the payer-tumbler / tumbler-payee channel.

	Open	Sender-initiated Close		Receiver-initiated Close
		Responded	Timeout	
Sender	57234	51316	90008	0
Receiver	0	608138 / 621607	0	605961 / 619430

Communication overhead and comparison. The result is shown in Figure 3, with results of A^2L versions and BlindHub also shown for comparison. Specifically,

- In Accio, the payer's communication message size is 2.69KB, around 2 times the overhead of the tumbler and payee. It is because the most communication overhead is on delivering the authenticated hidden state, and the payer needs to send the authenticated hidden state received from the payee to the tumbler and forward the updated one received from the tumbler to the payee.
- Compared with A^2L versions, all three types of participants' overhead in Accio are 45%-82% less, and the total overhead is 73% less. We emphasize again that the payment amount in A^2L versions is of fixed denomination while it is variable in Accio. Thus, to achieve a desired amount off-chain payment, the cost of A^2L grows linearly/logarithmically with the increase of the payment amount, but Accio's cost is independent of it.
- BlindHub costs 87860KB in total (See Table 2 in [22]), which is 4 orders of magnitude larger than Accio's communication message size (which is 2.69KB).

Communication round comparison. To finish one payment, the solutions based on the puzzle-promising and puzzle-solving paradigm require at least 6 rounds of off-chain communication between participants. Additionally, 3 more rounds are necessary for the payer's registration to resist the griefing attack. In total, one payment in A^2L or BlindHub requires 9 rounds of communication. By contrast, Accio only requires 4 rounds of communication and suffers less from network latency.

Computation overhead comparison. In Accio, the off-chain computation for the payment process costs 0.015s in total. Specifically, the respective running time of the tumbler and the payee is 0.004s, and the payer's running time is the most, which is around 0.007s. The reason is that the most costly step is to verify the tumbler's authentication on the hidden state. During one payment, the payer executes two times of verification: one is on the hidden state provided by the payee, and the other one is on the updated hidden state received from the tumbler. Besides, we test the off-chain running time of A^2L versions on the same platform. The result shows that one payment in A^2L versions costs more than 0.223 seconds, and Accio obtains at least 93% less than A^2L versions in the total payment time. Overall, BlindHub with the optimized implementation performs comparably to A^2L (See Section 10 in [22]), and Accio is around 10 times faster than A^2L.

Table 3: Communication overhead of one off-chain payment procedure in A^2L versions, BlindHub, and Accio, respectively.

Scheme	Payer (KB)	Tumbler (KB)	Payee (KB)	Payment (KB)
A^2L versions	$\geq$ 4.85	$\geq$ 7.59	$\geq$ 7.39	$\geq$ 9.92
BlindHub	55843	87855	32017	87860
Accio	2.69	1.38	1.31	2.69

8 FURTHER DISCUSSION

Fee integration. Fees are necessary to motivate the tumbler to participate in the payments. In Accio, the fee mechanism can be integrated as in LN, since the payment amounts are in plaintext. For example, the payer pays the tumbler (amt+fee) coins, and in turn, the tumbler updates the hidden tumbler-payee channel state with (amt-fee). This fee integration method should also be implemented in the contract when verifying the channel state.

From Uni-directional channel to Bi-directional channel. As clarified in Section 3.3, to guarantee users' balance security, Accio requires the tumbler-payee channel to be uni-directional. To enable the bi-directional unlinkable payment between each pair of PCH users, the direct solution is by establishing both the payer-tumbler and the tumbler-payee channels for each user. We leave the NIZK-free and privacy-preserving PCH constructions with bi-directional channel support as the future work.

9 CONCLUSION

In this paper, we have proposed a variable amount and optimized unlinkable off-chain payment channel hub scheme without using NIZK. The proposed scheme, Accio, allows off-chain users to perform non-fixed amounts and unlinkable off-chain payments via an untrusted tumbler at a low cost. We formalize Accio in the universally composable model and prove that this protocol satisfies both security and unlinkability. Experimental evaluation shows that Accio outperforms the state-of-the-art unlinkable PCH constructions in reducing the costs for both communication and computation.

REFERENCES

[1] Lukas Aumayr, Oguzhan Ersoy, Andreas Erwig, Sebastian Faust, Kristina Hostáková, Matteo Maffei, Pedro Moreno-Sanchez, and Siavash Riahi. 2021. Generalized Channels from Limited Blockchain Scripts and Adaptor Signatures. In *Advances in Cryptology , ASIACRYPT*, Vol. 13091. Springer, 635–664.

[2] Balthazar Bauer and Georg Fuchsbauer. 2020. Efficient Signatures on Randomizable Ciphertexts. In *Security and Cryptography for Networks, SCN*, Vol. 12238. Springer, 359–381.

[3] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. 2014. Zerocash: Decentralized Anonymous Payments from Bitcoin. In *IEEE Symposium on Security and Privacy, SP*. IEEE Computer Society, 459–474.

[4] Olivier Blazy, Georg Fuchsbauer, David Pointcheval, and Damien Vergnaud. 2011. Signatures on Randomizable Ciphertexts. In *International Conference on Practice and Theory in Public Key Cryptography, PKC*, Vol. 6571. Springer, 403–422.

[5] Jan Camenisch, Stephan Krenn, and Victor Shoup. 2011. A Framework for Practical Universally Composable Zero-Knowledge Protocols. In *Advances in Cryptology, ASIACRYPT*, Vol. 7073. Springer, 449–467.

[6] Ran Canetti. 2001. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *Annual Symposium on Foundations of Computer Science, FOCS*. IEEE Computer Society, 136–145.

[7] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. 2007. Universally Composable Security with Global Setup. In *Theory of Cryptography, TCC*, Vol. 4392. Springer, 61–85.

[8] Stefan Dziembowski, Lisa Eckey, Sebastian Faust, Julia Hesse, and Kristina Hostáková. 2019. Multi-party Virtual State Channels. In *Advances in Cryptology, EUROCRYPT*, Vol. 11476. Springer, 625–656.

[9] Stefan Dziembowski, Lisa Eckey, Sebastian Faust, and Daniel Malinowski. 2019. Perun: Virtual Payment Hubs over Cryptocurrencies. In *IEEE Symposium on Security and Privacy, SP*. IEEE, 106–123.

[10] Stefan Dziembowski, Sebastian Faust, and Kristina Hostáková. 2018. General State Channel Networks. In *Conference on Computer and Communications Security, CCS*. ACM, 949–966.

[11] Christoph Egger, Pedro Moreno-Sanchez, and Matteo Maffei. 2019. Atomic Multi-Channel Updates with Constant Collateral in Bitcoin-Compatible Payment-Channel Networks. In *Conference on Computer and Communications Security, CCS*. ACM, 801–815.

[12] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. 2015. The Bitcoin Backbone Protocol: Analysis and Applications. In *Advances in Cryptology, EUROCRYPT*, Vol. 9057. Springer, 281–310.

[13] Noemi Glaeser, Matteo Maffei, Giulio Malavolta, Pedro Moreno-Sanchez, Erkan Tairi, and Sri Aravinda Krishnan Thyagarajan. 2022. Foundations of Coin Mixing Services. In *Conference on Computer and Communications Security, CCS*, Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi (Eds.). ACM, 1259–1273.

[14] Matthew Green and Ian Miers. 2017. Bolt: Anonymous Payment Channels for Decentralized Currencies. In *Conference on Computer and Communications Security, CCS*. ACM, 473–489.

[15] Jens Groth, Rafail Ostrovsky, and Amit Sahai. 2006. Perfect Non-interactive Zero Knowledge for NP. In *Advances in Cryptology, EUROCRYPT*, Vol. 4004. Springer, 339–358.

[16] Ethan Heilman, Leen Alshenibr, Foteini Baldimtsi, Alessandra Scafuro, and Sharon Goldberg. 2017. TumbleBit: An Untrusted Bitcoin-Compatible Anonymous Payment Hub. In *Annual Network and Distributed System Security Symposium, NDSS*. The Internet Society.

[17] Giulio Malavolta, Pedro Moreno-Sanchez, Aniket Kate, and Matteo Maffei. 2017. SilentWhispers: Enforcing Security and Privacy in Decentralized Credit Networks. In *Annual Network and Distributed System Security Symposium, NDSS*. The Internet Society.

[18] Giulio Malavolta, Pedro Moreno-Sanchez, Aniket Kate, Matteo Maffei, and Srivatsan Ravi. 2017. Concurrency and Privacy with Payment-Channel Networks. In *Conference on Computer and Communications Security, CCS*. ACM, 455–471.

[19] Andrew Miller, Iddo Bentov, Surya Bakshi, Ranjit Kumaresan, and Patrick McCorry. 2019. Sprites and State Channels: Payment Networks that Go Faster Than Lightning. In *Financial Cryptography and Data Security, FC*, Vol. 11598. Springer, 508–526.

[20] Satoshi Nakamoto. 2019. *Bitcoin: A peer-to-peer electronic cash system.* Technical Report. Manubot.

[21] Joseph Poon and Thaddeus Dryja. 2016. The bitcoin lightning network: Scalable off-chain instant payments.

[22] Xianrui Qin, Shimin Pan, Arash Mirzaei, Zhimei Sui, Oguzhan Ersoy, Amin Sakzad, Muhammed F. Esgin, Joseph K. Liu, Jiangshan Yu, and Tsz Hon Yuen. 2023. BlindHub: Bitcoin-Compatible Privacy-Preserving Payment Channel Hubs Supporting Variable Amounts. In *IEEE Symposium on Security and Privacy, SP*.

[23] Christian Reitwiessner. 2017. EIP-196: Precompiled contracts for addition and scalar multiplication on the elliptic curve alt_bn128. (2017).

[24] Stefanie Roos, Pedro Moreno-Sanchez, Aniket Kate, and Ian Goldberg. 2018. Settling Payments Fast and Private: Efficient Decentralized Routing for Path-Based Transactions. In *Annual Network and Distributed System Security Symposium, NDSS*. The Internet Society.

[25] Mitsunari Shigeo. 2020. MCL: a portable and fast pairing-based cryptography library. (2020). https://github.com/herumi/mcl

[26] Jeremy Spilman. 2013. Anti dos for tx replacement. *bitcoin-dev mailing list* (2013).

[27] Erkan Tairi, Pedro Moreno-Sanchez, and Matteo Maffei. 2021. A^2L: Anonymous Atomic Locks for Scalability in Payment Channel Hubs. In *IEEE Symposium on Security and Privacy, SP*. IEEE, 1834–1851.

A CRYPTOGRAPHIC TOOLS

Here we present the cryptographic tools used in the paper.

Hash Function. A hash function H maps strings of arbitrary length to fixed-length values. A cryptographic hash function is required to be *collision-resistant*, i.e., finding two inputs that hash to the same value is difficult.

Digital Signature. A digital signature scheme $(\mathsf{KGen}, \mathsf{Sign}, \mathsf{Vf})$ generates the signing-verification key pair $(\mathsf{sk}, \mathsf{vk})$ via $\mathsf{KGen}(1^\lambda)$. Sign generates signature σ on the message m and the signing key sk, and Vf verifies the validity of the signature σ, based on the inputs message m and the verification key vk. We require the digital signature scheme to satisfy the adaptively existential unforgeability under chosen message attack.

Signature of Randomizable Ciphertexts. Roughly, the signature of randomizable ciphertexts (SoRC) [2, 4] lets one adapt a signature on a ciphertext to a randomization of the latter. The randomized ciphertext encrypts the same message as the original one. Furthermore, the randomization of this ciphertext, together with an adapted signature, looks like a random ciphertext with a fresh signature, indicating the unlinkability between the randomized version and the original.

Homomorphic Commitment. A commitment scheme allows the prover to commit a message (using the commit algorithm) without leaking any information about it, and open the commitment with the message later (checked using the open algorithm). Two properties, *binding* and *hiding*, are guaranteed. The former states that one commitment can only be opened to one message, and the latter implies that the committed message could not be derived from the commitment. We also require the scheme to be *homomorphic*, i.e., given the commitment of message m_1 and the commitment of message m_2, the commitment of message $m_1 + m_2$ could be derived.

In our work, we use the ElGamal public key encryption as the commitment scheme. Concretely, we fix the encryption key and regard it as a system parameter. To generate the commitment, the prover runs the encryption algorithm and the ciphertext acts as the commitment. To open the commitment, the prover reveals the message and randomness used in the ciphertext, using which the verifier repeats the encryption process and checks if the result is the same as the commitment. The binding and hiding property directly follows the correctness and CPA-security of ElGamal. Moreover, the homomorphic property is inherited.

Non-Interactive Zero-Knowledge. Let $\mathcal{R}$ be an NP relation and $\mathcal{L} := \{x | \exists w, s.t. \mathcal{R}(x, w) = 1\}$ be the corresponding language. A non-interactive zero-knowledge proof system $\mathsf{NIZK} := (\mathsf{Setup}, \mathsf{P}, \mathsf{V})$ allows the prover to prove the correctness of a statement without leaking any further information. Concretely, Setup generates the system parameter crs, a prover with a statement x and witness w gets the proof π via $\mathsf{P}(\mathsf{crs}, x, w)$, and the proof could be verified using $\mathsf{V}(\mathsf{crs}, x, \pi)$.

The NIZK scheme has 3 properties, *completeness*, *soundness*, and *zero-knowledge*. Intuitively, the completeness states that an honest prover could always generate a proof that is accepted by the verifier. The soundness implies that the prover could not generate an accepted proof for an incorrect statement. The zero-knowledge states that the verifier could not know further information except that the statement is correct. Refer readers to [5, 15] for details.

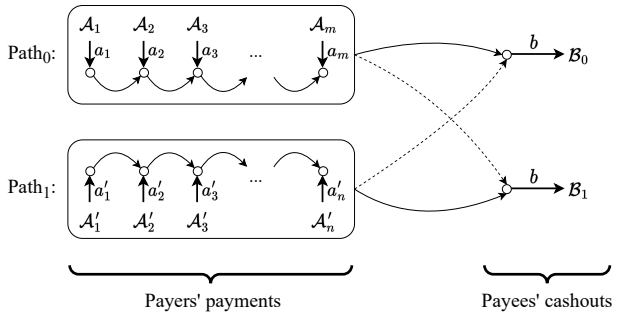

Figure 6: Unlinkability precondition clarification using the payment-cashout graph. In this figure, there are 2 paths of payments with $\Sigma_{i=1}^{m} a_i = \Sigma_{i=1}^{n} a'_i = b$. Each of the two payees, $\mathcal{B}_0$ and $\mathcal{B}_1$, gets b cashes out of the channel. From the natural leakage, the tumbler cannot uniquely link either path to $\mathcal{B}_0$ or $\mathcal{B}_1$. In this way, none of the payers, belonging to either path, can be linked to the payee.

B DISCUSSION ON INFORMATION LEAKAGE

Inspired by TumbleBit [16], we capture the unlinkability precondition using the payment-cashout graph. Specifically, for a comprehensive description of the unlinkability adversary's capabilities, we consider the worst case. That is, all the tumbler-payee channels are closed. In this case, the tumbler obtains the maximum information about the payment results.

We provide an instance in Figure 6 to help the illustration. Concretely, suppose the PCH works for a set of (greater than or equal to 2) payers and 2 payees (B_0 and B_1). After $m + n$ successful payments, the cashout of each payee is b coins. Denoting the path as a sequence of payments, these payments compose 2 (disjoint) paths, Path_0 and Path_1, each of which has the amount summary b. Given this precondition, the natural leakage cannot help the tumbler to link the payer to the payee, since both Path_0 and Path_1 have the same probability to compose the cashout of B_0 and B_1. Each payment's payee cannot be identified from B_0 and B_1.

Now we further characterize the unlinkability precondition, by extending the above detailed instance in to the general case, where there are m payees instead of two, and all the tumbler-payee channels are closed (thus the tumbler gets knowledge of each payee's cashout values).

Intuitively, since the tumbler can see the payment amount of each payment and the cashout result (i.e., the total refunded amount) of each payee, two conditions need to be satisfied to avoid the natural leakage's damage to unlinkability:

- **Non-unique cashout path.** To prevent the payee of any payment from being uniquely identified, its amount should not be uniquely required to compose this payee's cashout. It implies that, for each payee, at least 2 non-intersect compositions of the payment amounts lead to this cashout, which we denoted as the (cashout) paths.
- **Non-unique payment amount belonging.** Any payment amount received from payers could be used to composite the cashout paths of at least 2 payees.

More formally, supposing the set of all payees is $\{B_1, B_2, ..., B_m\}$, B_i's total refunded amount is δ_i. The sequence of payment amounts handled by the tumbler is denoted as AMT $:=< a_1, a_2, ..., a_n >$[1]. We also define the cashout path Ω as any sub-sequence of AMT. The path value $V(\Omega) := \sum_{a_j \in \Omega} a_j$ is the sum of payment amounts in path Ω. For a given value b, we define $\mathsf{Paths}(b) := \{\Omega | V(\Omega) = b\}$ as all possible cashout paths that the path value is b. For each a_j belonging to AMT, we use the set $\mathsf{Payee}(a_j)$ to collect a_j's potential payees. That is, $\mathsf{Payee}(a_j) = \{B_k | \exists \Omega \in \mathsf{Paths}(\delta_k), a_j \in \Omega\}$. We describe the two pre-conditions to define unlinkability below.

(1) $\forall B_i, 1 \le i \le m \Rightarrow \exists \Omega, \Omega' \in \mathsf{Paths}(\delta_i) \wedge \Omega \cap \Omega' = \emptyset$,
(2) $\forall a_j, 1 \le j \le n \Rightarrow |\mathsf{Payee}(a_j)| \ge 2$.

It is worth noting that PCH is used in off-chain cases where payments are usually of small amounts and high frequency, which means that the two requirements could be easily satisfied, and thus, the unlinkability makes sense in practice. Furthermore, measures can be exploited to reduce privacy leakage. For example, we can restrict the tumbler to initiate the close of the tumbler-payee channel only after a predefined time. This way, sufficient service-providing time for the payees is enabled, and more potential payment paths could be composed.

C DEFINITIONS AND CONSTRUCTION OF RSUC

C.1 Definitions of RSUC

Correctness. We use the experiment in Figure 7 to define correctness. A RSUC is correct if for all PPT adversary $\mathcal{A}$ it holds that $\Pr[\mathbf{Exp}^{\mathsf{CORR}}_{\mathcal{A},\mathsf{RSUC}}(\lambda) = 1] = 1$.

Unforgeability of the authenticated hidden value. We use the experiment in Figure 8 to define the unforgeability. Formally, a RSUC scheme satisfies the unforgeability of the authenticated hidden value if for all PPT adversary $\mathcal{A}$, it holds that $\Pr[\mathbf{Exp}^{\mathsf{EUF-AHV}}_{\mathcal{A},\mathsf{RSUC}}(\lambda) = 1] \le \mathsf{negl}(\lambda)$.

Randomization. We use the experiment in Figure 9 to define the randomization. A RSUC scheme is randomizable if for all PPT adversary $\mathcal{A}$ it holds that $|\Pr[\mathbf{Exp}^{\mathsf{RAND}}_{\mathcal{A},\mathsf{RSUC}}(\lambda) = 1] - 1/2| \le \mathsf{negl}(\lambda)$.

C.2 Construction of RSUC

Below we give the construction of RSUC. Although the detailed construction comes from SoRC [2], we extract its additive homomorphism property and utilize it to achieve the updatable property, which is critical in our scheme.

- $\mathsf{Setup}(1^\lambda)$: Return $\mathsf{pp} = (p, \mathbb{G}, G, P, \widehat{\mathbb{G}}, \widehat{G}, \mathbb{G}_T, e)$.
- $\mathsf{KeyGen}(1^\lambda)$: $\widehat{\mathsf{sk}} := (x_0, x_1) \xleftarrow{\$} (\mathbb{Z}_p^*)^2$, $\widehat{\mathsf{vk}} := (\widehat{X}_0 = x_0\widehat{G}, \widehat{X}_1 = x_1\widehat{G})$, return $(\widehat{\mathsf{sk}}, \widehat{\mathsf{vk}})$.
- $\mathsf{AuthCom}(v, \widehat{\mathsf{sk}} = (x_0, x_1); r)$: $\mathsf{cm} := (C_0 = rG, C_1 = vG + rP)$, $s \xleftarrow{\$} \mathbb{Z}_p^*$, return $(\mathsf{cm}, \widehat{\sigma} := (Z, S, \widehat{S}, T))$ with (1) $Z := \frac{1}{s}(G + x_0C_0 + x_1C_1)$, (2) $S := sG$, (3) $\widehat{S} := s\widehat{G}$, (4) $T := \frac{1}{s}(x_0G + x_1P)$.
- $\mathsf{VfCom}(\mathsf{cm} = (C_0, C_1), v, r)$: Return 1 if $C_0 = rG$ and $C_1 = vG + rP$, and return 0 otherwise.

[1]Here we use sequence instead of a set because the i-th payment's amount a_i could be non-unique.

$\mathbf{Exp}^{\mathsf{CORR}}_{\mathcal{A},\mathsf{RSUC}}(\lambda)$

$\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$
$(\widehat{\mathsf{sk}}, \widehat{\mathsf{vk}}) \leftarrow \mathsf{KeyGen}(\mathsf{pp})$
$v \leftarrow \mathcal{A}(\widehat{\mathsf{vk}})$
$(\mathsf{cm}, \widehat{\sigma}) \leftarrow \mathsf{AuthCom}(v, \widehat{\mathsf{sk}}; r)$
$\mathsf{Pair} := (\mathsf{cm}, \widehat{\sigma}, v, r)$, $\mathsf{Update} := \perp$, $\mathsf{Signal} := \mathsf{Continue}$
$\mathsf{Signal} \leftarrow \mathcal{A}^{O^{\mathsf{Update}}, O^{\mathsf{Random}}}(\mathsf{cm}, \widehat{\sigma}, r)$
if Signal = Stop **then**
  parse Pairas $(\mathsf{cm}^*, \widehat{\sigma}^*, v^*, r^*)$
  $b_0 := \mathsf{VfAuth}(\mathsf{cm}^*, \widehat{\sigma}^*, \widehat{\mathsf{vk}}) \wedge \mathsf{VfCom}(\mathsf{cm}^*, v^*, r^*)$
  if Update $\neq \perp$ **then**
    parse Update as $(\mathsf{cm}^\#, a)$
    $b_1 := \mathsf{VfUpd}(\mathsf{cm}^\#, a, \mathsf{cm}^*, \widehat{\sigma}^*, \widehat{\mathsf{vk}})$
  else $b_1 = 1$
  return $b_0 \wedge b_1$

Oracle $O^{\mathsf{Update}}_{(\widehat{\mathsf{vk}}, \widehat{\mathsf{sk}})}(a)$

parse Pair as $(\mathsf{cm}, \widehat{\sigma}, v, r)$
$(\mathsf{cm}_{\mathsf{new}}, \widehat{\sigma}_{\mathsf{new}}) \leftarrow \mathsf{UpdAC}(\mathsf{cm}, a, \widehat{\mathsf{sk}})$
$\mathsf{Pair} := (\mathsf{cm}_{\mathsf{new}}, \widehat{\sigma}_{\mathsf{new}}, v + a, r)$
$\mathsf{Update} := (\mathsf{cm}, a)$

Oracle $O^{\mathsf{Random}}(r')$

parse Pair as $(\mathsf{cm}, \widehat{\sigma}, v, r)$
$(\mathsf{cm}', \widehat{\sigma}') \leftarrow \mathsf{RdmAC}(\mathsf{cm}, \widehat{\sigma}; r')$
$\mathsf{Pair} := (\mathsf{cm}', \widehat{\sigma}', v, r + r')$
$\mathsf{Update} := \perp$

Figure 7: Experiment for RSUC's correctness.

- $\mathsf{VfAuth}(\mathsf{cm} = (C_0, C_1), \widehat{\sigma} = (Z, S, \widehat{S}, T), \widehat{\mathsf{vk}} = (\widehat{X}_0, \widehat{X}_1))$: Return 0 if $S = 0$. Return 1 if the following equations hold and 0 otherwise. (1) $e(Z, \widehat{S}) = e(G, \widehat{G})e(C_0, \widehat{X}_0)e(C_1, \widehat{X}_1)$, (2) $e(G, \widehat{S}) = e(S, \widehat{G})$, (3) $e(T, \widehat{S}) = e(G, \widehat{X}_0)e(P, \widehat{X}_1)$.
- $\mathsf{RdmAC}(\mathsf{cm} = (C_0, C_1), \widehat{\sigma} = (Z, S, \widehat{S}, T); r')$: $s' \xleftarrow{\$} \mathbb{Z}_p^*$, return $(\mathsf{cm}', \widehat{\sigma}' := (Z', S', \widehat{S}', T'))$ with (1) $\mathsf{cm}' := (C_0 + r'G, C_1 + r'P)$, (2) $Z' := \frac{1}{s'}(Z + r'T)$, (3) $S' := s'S$, (4) $\widehat{S}' := s'\widehat{S}$, (5) $T' := \frac{1}{s'}T$.
- $\mathsf{UpdAC}(\mathsf{cm} = (C_0, C_1), a, \widehat{\mathsf{sk}} = (x_0, x_1))$: $\mathsf{cm}_{\mathsf{new}} := (C_0, C_1 + aG)$, $s \xleftarrow{\$} \mathbb{Z}_p^*$, return $(\mathsf{cm}_{\mathsf{new}}, \widehat{\sigma}_{\mathsf{new}} := (Z, S, \widehat{S}, T))$ with (1) $Z := \frac{1}{s}(G + x_0C_0 + x_1(C_1 + aG))$, (2) $S := sG$, (3) $\widehat{S} := s\widehat{G}$, (4) $T := \frac{1}{s}(x_0G + x_1P)$.
- $\mathsf{VfUpd}(\mathsf{cm} = (C_0, C_1), a, \mathsf{cm}_{\mathsf{new}} = (C_0', C_1'), \widehat{\sigma}_{\mathsf{new}} = (Z, S, \widehat{S}, T), \widehat{\mathsf{vk}} = (\widehat{X}_0, \widehat{X}_1))$: Return 0 if $S = 0$. Return 1 if $C_0' = C_0$, $C_1' = C_1 + aG$, and the following equations hold, and return 0 otherwise. (1) $e(Z, \widehat{S}) = e(G, \widehat{G})e(C_0', \widehat{X}_0)e(C_1', \widehat{X}_1)$, (2) $e(G, \widehat{S}) = e(S, \widehat{G})$, (3) $e(T, \widehat{S}) = e(G, \widehat{X}_0)e(P, \widehat{X}_1)$.

$\mathbf{Exp}^{\mathsf{EUF-AHV}}_{\mathcal{A},\mathsf{RSUC}}(\lambda)$

$Q := \emptyset$
$\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$
$(\widehat{\mathsf{sk}}, \widehat{\mathsf{vk}}) \leftarrow \mathsf{KeyGen}(\mathsf{pp})$
$(\mathsf{cm}, \widehat{\sigma}, v, r, v', r') \leftarrow \mathcal{A}^{O^{\mathsf{AuthCom}}, O^{\mathsf{UpdAC}}}(\widehat{\mathsf{vk}})$
$b := \mathsf{VfAuth}(\mathsf{cm}, \widehat{\sigma}, \widehat{\mathsf{vk}})$
$c_0 := \mathsf{cm} \notin Q$
$c_1 := \mathsf{VfCom}(\mathsf{cm}, v, r) \wedge \mathsf{VfCom}(\mathsf{cm}, v', r')$
return $b \wedge (c_0 \vee c_1)$

Oracle $O^{\mathsf{AuthCom}}_{(\widehat{\mathsf{vk}},\widehat{\mathsf{sk}})}(v)$

$(\mathsf{cm}, \widehat{\sigma}) \leftarrow \mathsf{AuthCom}(v, \widehat{\mathsf{sk}}; r)$
$Q := Q \cup [\mathsf{cm}]$
return $(\mathsf{cm}, \widehat{\sigma}, r)$

Oracle $O^{\mathsf{UpdAC}}_{(\widehat{\mathsf{vk}},\widehat{\mathsf{sk}})}(\mathsf{cm}, \widehat{\sigma}, a)$

return $\perp$ **if** $\mathsf{VfAuth}(\mathsf{cm}, \widehat{\sigma}, \widehat{\mathsf{vk}}) = 0$
$(\mathsf{cm}_{\mathsf{new}}, \widehat{\sigma}_{\mathsf{new}}) \leftarrow \mathsf{UpdAC}(\mathsf{cm}, a, \widehat{\mathsf{sk}})$
$Q := Q \cup [\mathsf{cm}_{\mathsf{new}}]$
return $(\mathsf{cm}_{\mathsf{new}}, \widehat{\sigma}_{\mathsf{new}})$

Figure 8: Experiment for RSUC's unforgeability of the authenticated hidden value.

$\mathbf{Exp}^{\mathsf{RAND}}_{\mathcal{A},\mathsf{RSUC}}(\lambda)$

$\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$
$(\mathsf{cm}_0, \widehat{\sigma}_0, \mathsf{cm}_1, \widehat{\sigma}_1, \widehat{\mathsf{vk}}) \leftarrow \mathcal{A}(\mathsf{pp})$
$b \xleftarrow{\$} \{0, 1\}$
$(\mathsf{cm}', \widehat{\sigma}') \leftarrow \mathsf{RdmAC}(\mathsf{cm}_b, \widehat{\sigma}_b; r)$
$b' \leftarrow \mathcal{A}(\mathsf{cm}', \widehat{\sigma}')$
return $\mathsf{VfAuth}(\mathsf{cm}_0, \widehat{\sigma}_0, \widehat{\mathsf{vk}}) \wedge \mathsf{VfAuth}(\mathsf{cm}_1, \widehat{\sigma}_1, \widehat{\mathsf{vk}}) \wedge (b' = b)$

Figure 9: Experiment for RSUC's randomization.

Theorem 2. *The construction is a secure* RSUC *scheme.*

Proof. The proof of this theorem consists of the following three splitting lemmas. □

Lemma 1. *The construction satisfies the correctness property.*

Proof. The correctness follows directly. □

Lemma 2. *The construction satisfies the unforgeability of the authenticated hidden value property.*

Proof. The proof could be deduced to the unforgeability of SoRC (see Definition 6 in [2]) via fixing the encryption key, and the binding property of the commitment scheme. □

Lemma 3. *The construction satisfies the randomization property.*

Proof. From [2], for an adversary that creates a signature verification key as well as ciphertext and a signature on it, randomization of this ciphertext together with an adapted signature looks like a random ciphertext with a fresh signature on it, completing the proof. □

D PROOF OF THEOREM 1

In the beginning, $\mathcal{S}$ corrupts the parties that $\mathcal{A}$ corrupts. $\mathcal{S}$ generates the public parameters used in RSUC. If the tumbler is honest, $\mathcal{S}$ generates $(\widehat{\mathsf{sk}}_T, \widehat{\mathsf{vk}}_T)$ via KeyGen and the NIZK proof π, and sends the public key with the proof to the adversary. If the tumbler is corrupted, $\mathcal{S}$ extracts its authentication key via the provided proof.

Figure 10 shows the simulation of the three procedures. Especially, when we say $\mathcal{S}$ extracts the channel identifier and balance out of a hidden state received from the adversary, we mean $\mathcal{S}$ extracts $\mathsf{H}(id)||bal$ from the provided NIZK proof, and then derives id by computing the hash of current channel identifiers and comparing the result with $\mathsf{H}(id)$. Furthermore, when the simulator provides the hidden state for the honest payee, we mean the simulator selects a random value and generates the commitment accordingly as the hidden state, and then generates the randomizable signature on it using the generated or extracted tumbler's signing key. It should be emphasized that the security and privacy achievements of the protocol, denoted by π, rely on the security of the underlying RSUC, and we prove it using the hybrid argument.

Hybrid-0: Replace calls to the NIZK schemes of the protocol, with calls to the NIZK functionality $\mathcal{F}_{\mathsf{NIZK}}$.

Hybrid-1: In this game, the ideal functionality gives the simulator $\mathcal{S}'$ the honest payee's identity in the payment phase. The simulator $\mathcal{S}'$ acts as described in Figure 10, except that when simulating the payment and channel closing phase when the payee is honest, $\mathcal{S}'$ simulates its hidden state as the honest payee in the real protocol.

Lemma 4. *For all* PPT *distinguisher* $\mathcal{Z}$,

$$\mathsf{EXEC}^{\mathcal{L}(\Delta),C}_{\pi,\mathcal{Z},\mathcal{A}}(\lambda) \approx \mathsf{Hybrid\text{-}0}^{\mathcal{L}(\Delta),C,\mathcal{F}_{NIZK}}_{\pi,\mathcal{Z},\mathcal{A}}(\lambda)$$

Proof. It follows from the security of the NIZK scheme. □

Lemma 5. *For all* PPT *distinguisher* $\mathcal{Z}$,

$$\mathsf{Hybrid\text{-}0}^{\mathcal{L}(\Delta),C,\mathcal{F}_{NIZK}}_{\pi,\mathcal{Z},\mathcal{A}}(\lambda) \approx \mathsf{Hybrid\text{-}1}^{\mathcal{L}(\Delta)}_{\mathcal{F},\mathcal{Z},\mathcal{S}'}(\lambda).$$

Proof. The difference between the two executions lies on how to react on the following event. In the payment or channel closing phase, when the tumbler is honest, the adversary provides a hidden state whose contained channel identifier belongs to the corrupted payee's channels, and the contained balance is not the channel's current or previous balances. In Hybrid-0, when the event occurs, the adversary performs the payment upon this balance, or gets the balance (if it is in the right range) from the ledger when closing the channel. In Hybrid-1, the payment is aborted, or the adversary gets no balance from the ledger.

The indistinguishability follows the hidden state unforgeability of RSUC and the collision resistance of the hash function. When the event happens, it indicates that either the hidden state forge

event occurs, or the hash collision is found. Due to the hash function's collision resistance property, the latter case is avoided. As for forging the hidden state, we give a reduction to the unforgeability experiment below.

The reduction takes the verification key received from the experiment as its verification key. The reduction sends the verification key and the proof of knowledge of the corresponding authentication key, which is generated by the NIZK simulator, to the adversary. During the execution, when receiving the hidden state generation request, the reduction queries the experiment's AuthCom oracle, returns the result, and records the hidden state and its plaintext in table $\mathcal{M}$. When receiving the hidden state update request, the reduction queries the experiment's UpdAC oracle, returns the result, and records the updated hidden state and its plaintext in table $\mathcal{M}$. At any time, if the reduction receives a valid hidden state whose plaintext is not in the table, then the reduction obtains a winning instance of the unforgeability experiment. Since winning the experiment is of negligible probability, the abort event occurs with negligible probability, and the two executions are indistinguishable. □

Lemma 6. *For all* PPT *distinguisher* $\mathcal{Z}$,

$$\text{Hybrid-1}^{\mathcal{L}(\Delta)}_{\mathcal{F},\mathcal{Z},\mathcal{S}'}(\lambda) \approx \text{EXEC}^{\mathcal{L}(\Delta)}_{\mathcal{F},\mathcal{Z},\mathcal{S}}(\lambda)$$

Proof. The difference between the two executions is on the hidden states provided by the honest payees. In the payment phase, the simulator $\mathcal{S}'$ provides the honest payees' actual hidden states, while states provided by $\mathcal{S}$ is generated by selecting random values and signing on the generated commitments using the tumbler's signing key. In the channel closing phase, the simulator $\mathcal{S}'$ also provides the real hidden state for the honest payee, and $\mathcal{S}$ generates the state according to the revealed balance. The two hidden states have the same value but different randomness.

The indistinguishability follows the randomization property of RSUC. If the adversary can distinguish the two executions, it can be reduced to the advantage in the LR-formulation of the randomization experiment of RSUC. Concretely, in the payment phase, the reduction sends the actual and simulated honest payees' channel states to the experiment. Then the reduction uses the received states in the following steps. In the channel closing phase, the reduction generates the hidden states according to revealed balances. In the end, according to $\mathcal{Z}$'s distinguishment result, the reduction returns output in the randomization experiment. Since the probability of winning the randomization experiment is negligible, the two executions are indistinguishable. □

(A) Open

Distinguish the following cases,

1. Upon receiving $(\texttt{open}, \eta)$ from $\mathcal{F}$: Send $(\texttt{opened}, \eta)$ to η.users on behalf of C within Δ rounds, set $\Gamma(\eta.\text{id}) := \eta$. If η.type = payee, set $(\text{cm}, \widehat{\sigma}) := \text{AuthCom}\ (\text{H}(\eta.\text{id})||0, \widehat{\text{sk}}; r)$, and send $(\texttt{chan-state}, \eta.\text{id}, \text{cm}, r, \widehat{\sigma})$ to η.receiver on behalf of η.sender. (**Case** η.sender **is honest**)

2. Upon P sending $(\texttt{open}, \eta)$ to C: Send $(\texttt{open}, \eta)$ to $\mathcal{F}$ on behalf of P and if receive the reply $(\texttt{open}, \eta)$ from $\mathcal{F}$, send $(\texttt{opened}, \eta)$ to η.users on behalf of C within Δ rounds, and set $\Gamma(\eta.\text{id}) := \eta$. When η.type = payee, if η.receiver is corrupted, stop. Otherwise, if P sends $(\texttt{chan-state}, \eta.\text{id}, \text{cm}, r, \widehat{\sigma})$ to η.receiver where $\text{VfAuth}(\text{cm}, \widehat{\sigma}, \widehat{\text{vk}}) \wedge \text{VfCom}(\text{cm}, \text{H}(\eta.\text{id})||0, r) = 1$, stop. Else, reply with $(\texttt{no-state}, \eta.\text{id})$ to $\mathcal{F}$ on behalf of P. (**Case** η.sender **is corrupt**)

(B) Payment

Distinguish the following cases,

1. Upon receiving $(\texttt{pay-payer}, A, amt, \perp)$ from $\mathcal{F}$, let $\beta \in \Gamma$ where β.sender = A, β.receiver = T, and β.type = payer. Generate a $(\text{cm}, \widehat{\sigma})$ pair such that $\text{VfAuth}\ (\text{cm}, \widehat{\sigma}, \widehat{\text{vk}}_T) = 1$ and a NIZK proof π of the knowledge of cm's opening, set $msg := (\beta.\text{id}, \beta.\text{balance}(T) + amt, amt, \text{cm}, \widehat{\sigma})$, $\sigma_A \leftarrow \text{Sign}(msg, \text{sk}_A)$, and send $(\texttt{pay-req}, msg, \sigma_A, \pi)$ to T on behalf of A. Execute the sub-procedure in β and stop. (**Case** A **and** B **are honest,** T **is honest or corrupt**)

2. Upon A sending $(\texttt{pay-req}, msg = (id, bal, amt, \text{cm}, \widehat{\sigma}), \sigma_A, \pi)$ to T: Ignore if $\beta := \Gamma(id) = \perp$ or $A \neq \beta$.sender or $T \neq \beta$.receiver or β.type $\neq$ payer or $bal \neq \beta.\text{balance}(T) + amt$ or $\text{VfAuth}(\text{cm}, \widehat{\sigma}, \widehat{\text{vk}}_T) \wedge \text{Vf}(msg, \sigma_A, \text{vk}_A) \neq 1$ or π is invalid. Otherwise, extract γ and bal from π. If $\gamma \notin \Gamma$ or γ.receiver is honest, set $B := \perp$. Else, set $B := \gamma$.receiver, send $(\texttt{pay-payee}, A, amt)$ to $\mathcal{F}$ on behalf of B, and send $(\texttt{update-basis}, B, bal)$ to $\mathcal{F}$. Send $(\texttt{pay-payer}, B, amt)$ to $\mathcal{F}$ on behalf of A. Generate $(\text{cm}_{\text{new}}, \widehat{\sigma}_{\text{new}}) \leftarrow \text{UpdAC}(\text{cm}, amt, \widehat{\text{sk}}_T)$, set $\beta.\text{balance}(T) := bal$, $\beta.\text{balance}(A) := \beta.\text{fund} - bal$, $\beta.\text{state} := (msg, \sigma_A, \widehat{\sigma}, \text{cm}_{\text{new}}, \widehat{\sigma}_{\text{new}})$, and reply to A with $(\texttt{pay-ack}, \text{cm}_{\text{new}}, \widehat{\sigma}_{\text{new}})$ on behalf of T. If $B \neq \perp$, reply to $\mathcal{F}$ with $(\texttt{send-update})$. (**Case** A **and** B **are corrupt,** T **is honest**)

3. Upon B sending $(\texttt{pay-info}, \text{cm}, \widehat{\sigma}, amt, \pi)$ to A: Ignore if $\text{VfAuth}(\text{cm}, \widehat{\sigma}, \widehat{\text{vk}}_T) \neq 1$ or π is invalid. Otherwise, extract γ and bal from π. If $\gamma \notin \Gamma$ or γ.receiver is honest, set $B' := \perp$. Else, set $B' := \gamma$.receiver, and send $(\texttt{pay-payee}, A, amt)$ to $\mathcal{F}$ on behalf of B', send $(\texttt{update-basis}, B', bal)$ to $\mathcal{F}$, and send $(\texttt{fake-payee}, A, amt, B', B)$ if $B \neq B'$. Upon receiving $(\texttt{pay-payer}, A, amt, B)$ from $\mathcal{F}$, retrieve β from Γ where β.sender = A, β.receiver = T, and β.type = payer, generate $msg := (\beta.\text{id}, \beta.\text{balance}(T) + amt, amt, \text{cm}, \widehat{\sigma})$, $\sigma_A \leftarrow \text{Sign}(msg, \text{sk}_A)$, and send $(\texttt{pay-req}, msg, \sigma_A, \pi)$ to T on behalf of A. In channel β, execute the sub-procedure for the payment. If the new hidden state is obtained by A, forward it to B on behalf of A. (**Case** A **is honest and** B **is corrupt,** T **is honest or corrupt**)

4. Upon receiving $(\texttt{pay-payee}, A, amt, B)$ from $\mathcal{F}$: Generate a $(\text{cm}, \widehat{\sigma})$ pair such that $\text{VfAuth}(\text{cm}, \widehat{\sigma}, \widehat{\text{vk}}_T) = 1$ and a NIZK proof π of the knowledge of cm's opening, send $(\texttt{pay-info}, \text{cm}, \widehat{\sigma}, amt, \pi)$ to A on behalf of B. Distinguish the following cases, (**Case** A **is corrupt and** B **is honest,** T **is honest or corrupt)**

1) Case T is corrupt: Send $(\texttt{pay-payer}, B, amt)$ to $\mathcal{F}$ on behalf of A, send $(\texttt{pay-ack}, A, amt, T)$ to $\mathcal{F}$. If A replies with $(\texttt{pay-success}, \text{cm}_{\text{new}}, \widehat{\sigma}_{\text{new}})$ to B within $3 + 2\Delta$ rounds where $\text{VfUpd}(\text{cm}, amt, \text{cm}_{\text{new}}, \widehat{\sigma}_{\text{new}}, \widehat{\text{vk}}_T) = 1$, reply to $\mathcal{F}$ with $(\texttt{send-update})$.

2) Case T is honest: If there exists corrupt A' sends $(\texttt{pay-req}, msg = (id, bal, amt, \text{cm}, \widehat{\sigma}), \sigma_{A'}, \pi)$ to T where $\beta = \Gamma(id) \neq \perp$ and $A' = \beta$.sender and $T = \beta$.receiver and β.type = payer, send $(\texttt{pay-payer}, B, amt)$ to $\mathcal{F}$ on behalf of A', send $(\texttt{fake-payer}, A', A, amt, B)$ to $\mathcal{F}$ if $A' \neq A$, and execute the sub-procedure in β. If A replies with $(\texttt{pay-success}, \text{cm}_{\text{new}}, \widehat{\sigma}_{\text{new}})$ to B within $3 + 2\Delta$ rounds where $\text{VfUpd}(\text{cm}, amt, \text{cm}_{\text{new}}, \widehat{\sigma}_{\text{new}}, \widehat{\text{vk}}_T) = 1$, reply to $\mathcal{F}$ with $(\texttt{send-update})$.

Sub-procedure for payment in channel β

Denote A sends $(msg = (id, bal, amt, \text{cm}, \widehat{\sigma}), \sigma_A, \pi)$ to T. Distinguish the following cases,

1. Case T is honest: Generate $(\text{cm}_{\text{new}}, \widehat{\sigma}_{\text{new}}) \leftarrow \text{UpdAC}\ (\text{cm}, amt, \widehat{\text{sk}}_T)$, reply with $(\texttt{pay-ack}, \text{cm}_{\text{new}}, \widehat{\sigma}_{\text{new}})$ to A on behalf of T, set $\beta.\text{balance}(T) := bal$, $\beta.\text{balance}(A) := \beta.\text{fund} - bal$, $\beta.\text{state} := (msg, \sigma_A, \widehat{\sigma}, \text{cm}_{\text{new}}, \widehat{\sigma}_{\text{new}})$.

2. Case T is corrupt: If T replies to A with $(\texttt{pay-ack}, \mathsf{cm}_{\mathsf{new}}, \widehat{\sigma}_{\mathsf{new}})$ where $\mathsf{VfUpd}(\mathsf{cm}, amt, \mathsf{cm}_{\mathsf{new}}, \widehat{\sigma}_{\mathsf{new}}, \widehat{\mathsf{vk}}_T) = 1$, send $(\texttt{pay-ack}, A, amt, T)$ to $\mathcal{F}$. Otherwise, within Δ rounds send $(\texttt{closing})$ to T on behalf of $C(id)$ and proceed as follows,

1) If T sends $(\texttt{close-resp}, ST)$ to $C(id)$, (a) If $ST = (msg, \sigma_A, \mathsf{cm}_{\mathsf{new}}, \widehat{\sigma}_{\mathsf{new}})$ where $\mathsf{VfAuth}(\mathsf{cm}, amt, \mathsf{cm}_{\mathsf{new}}, \widehat{\sigma}_{\mathsf{new}}, \widehat{\mathsf{vk}}_T) = 1$, send $(\texttt{pay-ack}, A, amt, T)$ and $(\texttt{chan-close}, id)$ to $\mathcal{F}$. (b) Else, if ST is verified valid, extract bal' from it, send $(\texttt{pay-not-ack}, A, bal', T)$ to $\mathcal{F}$. (c) Otherwise, set $ST := \perp$, and send $(\texttt{pay-not-ack}, A, 0, T)$ to $\mathcal{F}$.

2) Send $(\texttt{closed}, ST)$ to T on behalf of $C(id)$, and set $\Gamma(id) := \perp$.

(C) Close

Let $\eta := \Gamma(id)$, $P, Q \in \eta$.users as the close initiator and responder, respectively. Set $tw := 4 + 2\Delta$ if η.type = payee and set $tw := 0$ if η.type = payer. Distinguish the following cases,

1. Case P is η.receiver. If P is honest, upon receiving $(\texttt{close}, id, P)$ from $\mathcal{F}$, execute procedure (C.1) to generate η.state, within Δ rounds send $(\texttt{closed}, \eta.\text{state})$ to η.users on behalf of $C(id)$, set $\Gamma(id) := \perp$, and stop. Else, upon P sends $(\texttt{close-resp}, ST)$ to $C(id)$, send $(\texttt{close}, id)$ to $\mathcal{F}$ on behalf of P, execute procedure (C.2) to handle the state ST, set $\Gamma(id) := \perp$, and stop.

2. Case P is η.sender. If P is honest, upon receiving $(\texttt{close}, id, P)$ from $\mathcal{F}$, within Δ rounds send $(\texttt{closing})$ to Q on behalf of $C(id)$. Else, upon P sending $(\texttt{close})$ to $C(id)$, send $(\texttt{close}, id)$ to $\mathcal{F}$ on behalf of P, within Δ rounds send $(\texttt{closing})$ to Q on behalf of $C(id)$. Distinguish the following cases,

1) Case Q is honest. Execute procedure (C.1) to generate η.state, within $2\Delta + tw$ rounds send $(\texttt{closed}, \eta.\text{state})$ to η.users on behalf of $C(id)$, set $\Gamma(id) := \perp$, and stop.

2) Case Q is corrupt. If Q sends $(\texttt{close-resp}, ST)$ to $C(id)$ within $\Delta + tw$ rounds, execute procedure (C.2) to handle ST. Otherwise, after $2\Delta + tw$ rounds send $(\texttt{close-bal}, id, 0)$ to $\mathcal{F}$, within $3\Delta + tw$ rounds send $(\texttt{closed}, \perp)$ to η.users on behalf of $C(id)$, set $\Gamma(id) := \perp$, and stop.

(C.1) Generate the channel state. If η.type = payer, retrieve the stored η.state. Else, upon receiving balance bal from $\mathcal{F}$, set $(\mathsf{cm}, \widehat{\sigma}) \leftarrow \mathsf{AuthCom}(\mathsf{H}(id)||bal, \widehat{\mathsf{sk}}_T; r)$, $\eta.\text{state} := (\mathsf{cm}, \widehat{\sigma}, bal, r)$.

(C.2) Handle the state ST. If ST is valid, extract bal from it. Else, set $bal := 0$ and $ST := \perp$. Send $(\texttt{close-bal}, id, bal)$ to $\mathcal{F}$, and within Δ rounds send $(\texttt{closed}, ST)$ to η.users on behalf of $C(id)$.

Figure 10: The simulation.

E GAME-BASED SECURITY PROOF OF ACCIO

In this section, we prove the gamed-based security of Accio, which requires no NIZK proof in the protocol execution. The Accio security consists of two parts, the users' balance security and the payment unlinkability.

The balance security implies that any role of user (including the payer, the tumbler and the payee) would not lose coins. More specifically, when one channel is closed, its users could always get coins no less than the deserved in-channel balance. Here we only consider cases that users' balance security is hampered by the channel states. It is because the channel operating mechanism guarantees that the closing can be completed in $O(\Delta)$ time according to the state provided by the channel receiver, and cases that no state is provided are also specially treated. Furthermore, we focus on the tumbler and payee balance security in their channels, since the payer-tumbler channel state is in plaintext and known by both channel users. Thus the payer and tumbler balance security in their channels would not be compromised.

The payment unlinkability means that the tumbler cannot link the payer with the payee of one payment. Below we focus on cases that the tumbler-payee channels are not closed. When these channels are closed, the payment unlinkability holds with the unlinkability pre-condition in Section 3.6 is satisfied.

Definition 4 (Secure Accio). *Accio is secure if it satisfies the tumbler balance security, payee balance security, and payment unlinkability properties.*

Tumbler balance security. Give our protocol, the tumbler balance security means the tumbler would not suffer from the coin loss, i.e. coins received from payers are no less than coins sent to the payees. We use the experiment in Figure 11 to define the tumbler balance security. Accio satisfies the tumbler balance security if for all PPT adversary $\mathcal{A}$, it holds that $\Pr[\mathbf{Exp}^{\mathsf{Tumbler-Security}}_{\mathcal{A},\mathsf{Accio}}(\lambda)=1] \leq \mathsf{negl}(\lambda)$.

Payee balance security. Give our protocol, the payee balance security means the payee could always get deserved coins when the channel is closed. We use the experiment in Figure 12 to define the payee balance security. Accio satisfies the payee balance security if for all PPT adversary $\mathcal{A}$, it holds that $\Pr[\mathbf{Exp}^{\mathsf{Payee-Security}}_{\mathcal{A},\mathsf{Accio}}(\lambda)=1] \leq \mathsf{negl}(\lambda)$.

Payment unlinkability. Give our protocol, the payment unlinkability means that when receiving the payment request from one payer, the tumbler could not link it to the corresponding payee, nor link the transaction to previous transactions. We use the experiment in Figure 13 to define the payment unlinkability. Accio satisfies the payment unlinkability if for all PPT adversary $\mathcal{A}$, it holds that $|\Pr[\mathbf{Exp}^{\mathsf{Payment-Unlinkability}}_{\mathcal{A},\mathsf{Accio}}(\lambda)=1] - 1/2| \leq \mathsf{negl}(\lambda)$.

Theorem 3. *Assuming* RSUC *is secure and the hash function* H *is collision resistant, Accio is secure.*

Proof. The proof of this theorem consists of the following three splitting lemmas. □

Lemma 7. *Accio achieves tumbler balance security.*

Proof. For each payment, the tumbler receives coins from the payer and increases the hidden payee's balance in the received hidden state using the same amount, which means the tumbler sends the same amount of coins to the payee.

The event that the adversary wins the experiment implies the adversary obtains more coins than deserved ones from the hidden channel states. Further, it means at least one of the following two sub-events happened: (i) the adversary generates a hidden state and forges the tumbler's randomizable signature on it, (ii) the adversary opens a hidden state to another balance allocation of the same channel, or opens it to another channel. Due to the collision resistance property of the hash function and the perfect binding property of the commitment scheme, the latter case is avoided and we focus on the first sub-event below.

We construct the reduction that once the adversary wins tumbler balance security experiment, the reduction could use the adversary to win the unforgeability of the authenticated hidden value game

of RSUC. The reduction acts as follows. Receiving the randomizable signature verification key from the challenger, the reduction forwards it to the adversary. When the adversary queries the open oracle, the reduction obtains the authenticated commitment via querying the AuthCom oracle, and forwards the result to the adversary. When the adversary queries the update oracle, the reduction queries the UpdAC oracle to obtain the updated commitment along with the authentication, and replies the result to the adversary. Furthermore, the reduction stores the queried authenticated commitment. When the adversary queries the close oracle, the reduction stores the queried authenticated commitment.

When the adversary wins the game, the reduction uniformly selects one authenticated commitment from the storage and returns it to the challenger. If the adversary can win the game with a non-negligible probability p, the reduction can win with probability p/N, where N is the size of storage. Thus the reduction can win with non-negligible probability, which contradicts the unforgeability of the authenticated hidden value property of RSUC, completing the proof. □

Lemma 8. *Accio achieves payee balance security.*

Proof. The only event that leads the payee to lose coins is the payee cannot open the updated or randomized hidden state using the channel identifier, the current balance allocation and stored randomness. Since the event cannot happen, the payee balance security is guaranteed. □

Lemma 9. *Accio achieves payment unlinkability.*

Proof. It follows the randomization property of RSUC and the reduction acts as follows. Receiving the public parameters from the challenger, the reduction forwards them to the adversary. In the challenge phase, the reduction sends the stored two states to the challenger, forwards the result to the adversary, and replies to the challenger with the adversary's distinguishment result. If the adversary can distinguish the two states, the reduction can win the randomization game. Thus, the proof is completed. □

F DISCUSSION ON THE ENCODING OF THE CHANNEL STATE

Recall that the payee's channel state is encoded into the input of RSUC in the way of $\mathsf{H}(id)||bal$, where bal is the payee's balance in the channel. If one payee's received payment amounts overflow the balance space, the former part, $\mathsf{H}(id)$, is changed and the balance is reset. It means that the tumbler has received these coins from payers, while the payee cannot claim the same amount of coins when closing the channel.

$\mathbf{Exp}^{\mathsf{Tumbler-Security}}_{\mathcal{A},\mathsf{Accio}}(\lambda)$

$H := \emptyset, C := \emptyset$/ H and C record identifiers of channels with honest and corrupt payees
$F_s := 0, F_r := 0$/ F_s and F_r are the number of coins sent and received by the tumbler
$\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$, $(\widehat{\mathsf{sk}}, \widehat{\mathsf{vk}}) \leftarrow \mathsf{KeyGen}(\mathsf{pp})$
$\mathsf{Signal} \leftarrow \mathcal{A}^{O^{\mathsf{Open}}, O^{\mathsf{Update}}, O^{\mathsf{Close}}}(\widehat{\mathsf{vk}})$
if Signal = Stop:
  if $H = \emptyset \wedge C = \emptyset \wedge F_s > F_r$ **then return** 1
  else return 0

Oracle $O^{\mathsf{Open}}_{(\widehat{\mathsf{vk}},\widehat{\mathsf{sk}})}$ (type = honest / corrupt, id, f)

return $\perp$ **if** $id \in H \cup C$ or $f < 0$
store (channel, id, f)
$(\mathsf{cm}, \widehat{\sigma}) \leftarrow \mathsf{AuthCom}(\mathsf{H}(id)||0, \widehat{\mathsf{sk}}; r)$
if type = honest **then** add id to H and store(state, cm, $\widehat{\sigma}$, id, 0, r)
else add id to C
return $(\mathsf{cm}, \widehat{\sigma}, r)$

Oracle $O^{\mathsf{Update}}_{(\widehat{\mathsf{vk}},\widehat{\mathsf{sk}})}$((type = honest, id, amt)/(type = corrupt, id, cm, $\widehat{\sigma}$, amt))

if type = honest $\wedge$ $id \in H$:
  retrieve stored (channel, id, f) and (state, cm, $\widehat{\sigma}$, id, bal, r)
  abort if $f - bal < amt$
  $(\mathsf{cm}', \widehat{\sigma}') \leftarrow \mathsf{RdmAC}(\mathsf{cm}, \widehat{\sigma}; r')$
  $(\mathsf{cm}_{\mathsf{new}}, \widehat{\sigma}_{\mathsf{new}}) \leftarrow \mathsf{UpdAC}(\mathsf{cm}', amt, \widehat{\mathsf{sk}})$
  change (state, cm, $\widehat{\sigma}$, id, bal, r) to (state, $\mathsf{cm}_{\mathsf{new}}, \widehat{\sigma}_{\mathsf{new}}$, id, $bal + amt, r + r'$)
  $F_r = F_r + amt$
  return $(\mathsf{cm}', \widehat{\sigma}', \mathsf{cm}_{\mathsf{new}}, \widehat{\sigma}_{\mathsf{new}})$
if type = corrupt $\wedge$ $id \in C$:
  abort if $\mathsf{VfAuth}(\mathsf{cm}, \widehat{\sigma}, \widehat{\mathsf{vk}}) = 0$
  $(\mathsf{cm}_{\mathsf{new}}, \widehat{\sigma}_{\mathsf{new}}) \leftarrow \mathsf{UpdAC}(\mathsf{cm}, amt, \widehat{\mathsf{sk}})$
  $F_r = F_r + amt$
  return $(\mathsf{cm}_{\mathsf{new}}, \widehat{\sigma}_{\mathsf{new}})$
else abort

Oracle $O^{\mathsf{Close}}_{(\widehat{\mathsf{vk}},\widehat{\mathsf{sk}})}$((type = honest, id)/(type = corrupt, id, cm, $\widehat{\sigma}$, bal, r))

if type = honest $\wedge$ $id \in H$:
  retrieved stored (channel, id, f) and (state, id, cm, $\widehat{\sigma}$, bal, r)
  $(\mathsf{cm}', \widehat{\sigma}') \leftarrow \mathsf{RdmAC}(\mathsf{cm}, \widehat{\sigma}; r')$
  $F_s = F_s + f - bal$ and delete id from H
  return $(\mathsf{cm}', \widehat{\sigma}', bal, r + r')$
if type = corrupt $\wedge$ $id \in C$:
  retrieve stored (channel, id, f)
  if $\mathsf{VfCom}(\mathsf{cm}, \mathsf{H}(id)||bal, r) = 1 \wedge 0 \leq bal \leq f$ **then**
    $F_s = F_s + f - bal$ and delete id from C
    return bal
  else abort
else abort

Figure 11: Experiment for Accio's tumbler balance security.

$\mathbf{Exp}^{\text{Payee-Security}}_{\mathcal{A},\text{Accio}}(\lambda)$

$H := \emptyset$ / H records identifiers of honest payees' channels
$\text{pp} \leftarrow \text{Setup}(1^\lambda)$
$\widehat{\text{vk}} \leftarrow \mathcal{A}(\text{pp})$
$id \leftarrow \mathcal{A}^{O^{\text{Open}}, O^{\text{Update}}, O^{\text{Close}}}()$
abort if $id \notin H$
retrieve stored $(\text{state}, \text{cm}, \widehat{\sigma}, id, bal, r)$
if $\text{VfCom}(\text{cm}, \text{H}(id)||bal, r) = 0$ **return** 1
else return 0

Oracle $O^{\text{Open}}(id, f, \text{cm}, \widehat{\sigma}, r)$

$c := \text{VfAuth}(\text{cm}, \widehat{\sigma}, \widehat{\text{vk}}) \wedge \text{VfCom}(\text{cm}, \text{H}(id)||0, r)$
abort if $id \in H$ or $f < 0$ or $c = 0$
$(\text{cm}', \widehat{\sigma}') \leftarrow \text{RdmAC}(\text{cm}, \widehat{\sigma}; r')$
store $(\text{channel}, id, f)$ and $(\text{state}, \text{cm}', \widehat{\sigma}', id, 0, r + r')$

Oracle $O^{\text{Update}}(id, amt)$

abort if $id \notin H$
retrieve stored $(\text{channel}, id, f)$ and $(\text{state}, \text{cm}, \widehat{\sigma}, id, bal, r)$
abort if $f - bal < amt$
send $(\text{cm}, \widehat{\sigma})$ to $\mathcal{A}$
if receive $(\text{cm}_{\text{new}}, \widehat{\sigma}_{\text{new}})$ and $\text{VfUpd}(\text{cm}, amt, \text{cm}_{\text{new}}, \widehat{\sigma}_{\text{new}}, \widehat{\text{vk}}) = 1$
  $(\text{cm}', \widehat{\sigma}') \leftarrow \text{RdmAC}(\text{cm}_{\text{new}}, \widehat{\sigma}_{\text{new}}; r')$
  $bal_{\text{new}} = bal + amt, r_{\text{new}} = r + r'$
else
  $(\text{cm}', \widehat{\sigma}') \leftarrow \text{RdmAC}(\text{cm}, \widehat{\sigma}; r')$
  $bal_{\text{new}} = bal, r_{\text{new}} = r + r'$
change $(\text{state}, \text{cm}, \widehat{\sigma}, id, bal, r)$ to $(\text{state}, \text{cm}', \widehat{\sigma}', id, bal_{\text{new}}, r_{\text{new}})$

Oracle $O^{\text{Close}}(id)$

abort if $id \notin H$
retrieved stored $(\text{channel}, id, f)$ and $(\text{state}, id, \text{cm}, \widehat{\sigma}, bal, r)$
send $(\text{cm}, \widehat{\sigma}, bal, r)$ to $\mathcal{A}$
delete id from H

Figure 12: Experiment for Accio's payee balance security.

$\mathbf{Exp}^{\text{Payment-Unlinkability}}_{\mathcal{A},\text{Accio}}(\lambda)$

$\text{pp} \leftarrow \text{Setup}(1^\lambda)$
$(\widehat{\text{vk}}, \text{cm}_0, \widehat{\sigma}_0, id_0, f_0, r_0, \text{cm}_1, \widehat{\sigma}_1, id_1, f_1, r_1) \leftarrow \mathcal{A}(\text{pp})$
$c_0 := \text{VfAuth}(\text{cm}_0, \widehat{\sigma}_0, \widehat{\text{vk}}) \wedge \text{VfAuth}(\text{cm}_1, \widehat{\sigma}_1, \widehat{\text{vk}})$
$c_1 := \text{VfCom}(\text{cm}_0, \text{H}(id_0)||0, r_0) \wedge \text{VfCom}(\text{cm}_1, \text{H}(id_1)||0, r_1)$
abort if $c_0 \wedge c_1 = 0$
store $(\text{channel}, id_0, f_0)$ and $(\text{channel}, id_1, f_1)$
$(\text{cm}'_0, \widehat{\sigma}'_0) \leftarrow \text{RdmAC}(\text{cm}_0, \widehat{\sigma}_0; r'_0)$, store $(\text{state}, \text{cm}'_0, \widehat{\sigma}'_0, id_0, 0, r_0 + r'_0)$
$(\text{cm}'_1, \widehat{\sigma}'_1) \leftarrow \text{RdmAC}(\text{cm}_1, \widehat{\sigma}_1; r'_1)$, store $(\text{state}, \text{cm}'_1, \widehat{\sigma}'_1, id_1, 0, r_1 + r'_1)$
$\text{Signal} \leftarrow \mathcal{A}^{O^{\text{Update}}}()$
if Signal = Stop:
  $b \leftarrow \{0, 1\}$
  retrieve stored $(\text{state}, cm_b, \widehat{\sigma}_b, id_b, bal_b, r_b)$
  $b^* \leftarrow \mathcal{A}(\text{cm}_b, \widehat{\sigma}_b)$
  if $b^* = b$ **then return** 1
  else return 0

Oracle $O^{\text{Update}}(id, amt)$

abort if $id \neq id_0 \wedge id \neq id_1$
retrieve stored $(\text{channel}, id, f)$ and $(\text{state}, \text{cm}, \widehat{\sigma}, id, bal, r)$
abort if $f - bal < amt$
send $(\text{cm}, \widehat{\sigma})$ to $\mathcal{A}$
if receive $(\text{cm}_{\text{new}}, \widehat{\sigma}_{\text{new}})$ and $\text{VfUpd}(\text{cm}, amt, \text{cm}_{\text{new}}, \widehat{\sigma}_{\text{new}}, \widehat{\text{vk}}) = 1$
  $(\text{cm}', \widehat{\sigma}') \leftarrow \text{RdmAC}(\text{cm}_{\text{new}}, \widehat{\sigma}_{\text{new}}; r')$
  $bal_{\text{new}} = bal + amt, r_{\text{new}} = r + r'$
else
  $(\text{cm}', \widehat{\sigma}') \leftarrow \text{RdmAC}(\text{cm}, \widehat{\sigma}; r')$
  $bal_{\text{new}} = bal, r_{\text{new}} = r + r'$
change $(\text{state}, \text{cm}, \widehat{\sigma}, id, bal, r)$ to $(\text{state}, \text{cm}', \widehat{\sigma}', id, bal_{\text{new}}, r_{\text{new}})$

Figure 13: Experiment for Accio's payment unlinkability.