Starfish: Rebalancing Multi-Party Off-Chain Payment Channels

Minghui Xu, *Member, IEEE*, Wenxuan Yu, Guangyong Shang, Guangpeng Qi, Dongliang Duan, Shan Wang, Kun Li, Yue Zhang, and Xiuzhen Cheng, *Fellow, IEEE*

Abstract—**Blockchain technology has revolutionized the way transactions are executed, but scalability remains a major challenge. Payment Channel Network (PCN), as a Layer-2 scaling solution, has been proposed to address this issue. However, skewed payments can deplete the balance of one party within a channel, restricting the ability of PCNs to transact through a path and subsequently reducing the transaction success rate. To address this issue, the technology of rebalancing has been proposed. However, existing rebalancing strategies in PCNs are limited in their capacity and efficiency. Cycle-based approaches only address rebalancing within groups of nodes that form a cycle network, while non-cycle-based approaches face high complexity of on-chain operations and limitations on rebalancing capacity. In this study, we propose Starfish, a rebalancing approach that captures the star-shaped network structure to provide high rebalancing efficiency and large channel capacity. Starfish requires only N-time on-chain operations to connect independent channels and aggregate the total budget of all channels. To demonstrate the correctness and advantages of our method, we provide a formal security proof of the Starfish protocol and conduct comparative experiments with existing rebalancing techniques.**

Index Terms—**Blockchain, payment channel network, rebalancing.**

I. Introduction

OVER the past decade, blockchain technology [1] has experienced a rapid development as a trusted and decentralized means of executing transactions. However, its scalability remains a significant challenge. As a response, the research community and industry have proposed a number of Layer-2 scaling solutions to handle transactions off the Mainnet (Layer 1) [2], [3], [4], [5]. Payment channel is an innovative layer-2 scaling technique that enables transaction processing off-chain and final settlement on-chain. The collaboration of interconnected payment channels forms a payment channel network (PCN) [6], [7], [8], [9], [10], [11], [12]. However, the channel depletion problem, which refers to the scenario where a payment channel within the network runs out of funds, has a strong impact on the scalability of a PCN [13], [14], [15]. Such a phenomenon occurs when a significant number of transactions flow through a particular channel, exhausting its available balance. These transactions, known as skewed ones, render the channel temporarily unusable for further transactions until it is replenished with funds.

Minghui Xu, Wenxuan Yu, Dongliang Duan, Kun Li, Yue Zhang, and Xiuzhen Cheng are wih the Schoole of Computer Science and Technology, Shandong University, Qingdao 266237, China. (e-mail: mhxu@sdu.edu.cn, haitengseat@gmail.com, 2201546691@qq.com, kunli@sdu.edu.cn, zyuein-fosec@gmail.com, xzcheng@sdu.edu.cn)

Guangyong Shang and Guangpeng Qi are with Inspur Yunzhou Industrial Internet Co., Ltd, Jinan 250101, China (e-mail: shangguangyong@inspur.com; qigp@inspur.com).

Shan Wang is with the Department of Computing, The Hong Kong Polytechnic University, Hong Kong, China (e-mail: shanwangsec@gmail.com).

Corresponding author: Wenxuan Yu

Researchers have proposed two types of approaches to address the issue of channel depletion: routing-based and rebalancing-based. Routing-based solutions [16], [17], [18], [19], [20], [21] aim to balance and facilitate transactions within a payment channel network by selecting appropriate routing strategies. Rebalancing-based solutions [22], [23], [24], [25], on the other hand, realize the goal of rebalancing channels with depleted funds by transferring balances between channels. These two techniques complement each other, with the former helping to choose appropriate payment paths, while the latter reviving payment routes that were previously unusable. We focus on rebalancing-based methods in this paper, and address their limitations with a live example first.

Fig. 1: (a) Visualization of the Bitcoin lightning network by displaying nodes with a degree up to 32. Color nodes according to their degrees: white for 1, green for 2, orange for 3-4, blue for 5-8, purple for 9-16, and red for 17-32. Each gray edge represents a payment channel. (b) The distribution of the node degree.

Fig. 1(a) presents a visual representation of the real-world Bitcoin lightning network[1], exhibiting its mesh topology. Each gray edge represents a payment channel. Fig. 1(b) illustrates the dense nature of the Bitcoin lightning network, wherein approximately 70% of the intermediary nodes possess a degree exceeding two. Nodes characterized by high degrees may experience heightened transactional throughput, thereby elevating the risk of channel depletion in the presence of skewed transactions. However, such nodes also benefit from enhanced opportunities to effectively rebalance channels, consequently mitigating depletion risks. Current solutions are inadequately tailored for rebalancing within such a dense network topology, impeding their ability to achieve optimal rebalancing efficiency and accommodate large channel capacities.

Two categories of rebalancing approaches exist: cycle-based [22], [23], [24] and non-cycle-based [6], [26], [25]. The cycle-based approach, exemplified by the Revive protocol

---

[1]We get the snapshot of the lightning network topology on 2021-03-31, which contains 10,529 nodes and 38,910 channels.

[22], leverages nodes organized in a cyclic network topology for rebalancing. Cycle [23] further augments the efficacy of Revive. However, cycles take resources (time and bandwidth) to identify, and PCNs frequently feature non-cyclic structures, e.g., star-shaped ones, rendering such methods inefficient and ineffective. Close-Open [6] and LOOP [26] are two non-cycle-based solutions that achieve channel balancing by reopening a new channel. These solutions are simple but require frequent on-chain operations, resulting in significant time and financial expenses. Shaduf [25], as a non-cycle-based rebalancing approach, rebalances channels by binding (each costs $2\times$ on-chain operations) adjacent channels and enabling balance transferring between them. When using the common "All Bind" or "All to One" binding strategies, the required number of on-chain operations to bind N channels are respectively $N(N-1)$ or $2(N-1)$. Additionally, the channel capacity in Shaduf is limited by the highest capacity among the channels bound together.

In this study, we present Starfish, a rebalancing technique that improves rebalancing efficiency and channel capacity compared to existing approaches. Starfish is applicable to any node as it takes only the one-hop local topology of the node (the node is called a hub in the Starfish protocol to differentiate it from other nodes), a starfish-style structure centered at the hub, and rebalances among the channels associated with the hub to overcome the balance depletion of any channel associated with the hub. Moreover, all the nodes in a PCN can run Starfish independently and simultaneously, as each node rebalances only the amounts it allocates to its associated channels. In a nutshell, major contributions of this paper can be summarized as follows:

- **High rebalancing efficiency.** Starfish offers an effective solution for rebalancing multiple channels associated with a node. We use a unique merge contract that requires only N on-chain operations to connect several independent channels, resulting in a cost-effective rebalancing process. The traditional approach balances only two adjacent channels at a time. In contrast, our Starfish protocol can simultaneously merge and rebalance multiple channels based on their specific needs and channel connection conditions, thereby enhancing the efficiency of multi-channel rebalancing.
- **Large channel capacity.** Starfish improves channel scalability and efficiency by combining the capacities of multiple merged channels. This allows for the largest rebalancing capacity by utilizing the total budget of all channels. Additionally, Starfish promotes liquidity among merged channels, which is more effective than the traditional approach of assigning a fixed budget to each channel. These features enable the system to handle a significant volume of off-chain transactions with a high rate of success.
- **Formal security proof and comparison study.** In order to validate the safety and liveness properties of Starfish when facing potential malicious channel participants, we formalize the protocol and use the universally composable framework to prove that Starfish satisfies all the necessary

security requirements for rebalancing. By experiments, Starfish has shown superior performance in various typical conditions when compared to other similar protocols.

II. RELATED WORK

In a payment channel network, channel rebalancing refers to the process of adjusting the distribution of funds within a payment channel to ensure optimal liquidity and functionality. Table I presents a comprehensive comparative analysis on Starfish and other prevailing methods for rebalancing.

Close-Open [6] rebalances a channel by reopening it through two on-chain operations, while LOOP [26] reduces two to one. This implies that for a n-time refunding, $2n$ and n on-chain transactions are respectively required. Note that these two approaches do not depend on cycles, and their on-chain consensus waiting time is $O(\Delta)$, where Δ is the latency of transaction confirmation on a blockchain. Revive [22]'s rebalancing process does not require on-chain operations but relies on a cycle formed by payment channels. It sends a series of off-chain transactions with equal amounts in a cycle to rebalance channels that have been depleted. A drawback of Revive is that when rebalancing through transactions, normal transactions on the cycle should be stopped. Cycle [23] breaks this limitation, allowing the rebalancing process to be performed simultaneously with normal transactions. Cycle categorizes the rebalancing states into global and local ones, and relies on a smart contract to record the global state and arbitrate conflicts. For a n-time refunding, Cycle requires one on-chain operation and waits for $O(\Delta)$ on-chain consensus time.

Shaduf [25], through one on-chain binding operation, after waiting for $O(\Delta)$ on-chain consensus time, allows two channels to perform refunding n times through off-chain signatures without relying on any cycle. For a mutual binding of multiple channels, Shaduf takes different binding strategies to determine the number of Bind/Unbind operations on-chain. Particularly, three typical binding strategies are adopted: (1)"High to Low": users bind the channel with the highest balance to the one with the lowest balance, followed by the second-highest balance to the second-lowest balance, and so on, resulting in $N/2$ Bind/Unbind operations for N channels and thus involving N on-chain operations; (2) "All to One": all channels are bound to the same channel, resulting in $N-1$ Bind/Unbind operations and $2(N-1)$ on-chain operations; (3) "All Bind": each channel is bound to every other channel, resulting in $N(N-1)/2$ Bind/Unbind operations and $N(N-1)$ on-chain operations. Note that Close-Open, LOOP, Revive, and Cycle cannot rebalance multiple channels while Shaduf supports such a feature. With different binding strategies, each channel's capacity varies. However, the average capacity that can be used by each channel for any strategy cannot exceed $\mathcal{C}_{max}$. In comparison, Starfish consistently elevates the average transaction capacity per channel to $\sum \mathcal{C}_i$ and requires only N on-chain operations.

III. PRELIMINARIES

Payment Channel. A payment channel is a type of off-chain solution that allows two parties to establish a temporary

TABLE I: Comparison of Starfish with Other Approaches

	#OPTs for n-Time Refunding	Wait Time for n-Time Refunding	Usability of C/NC Cases	#OPTs for N-channel Bind (Merge)	Capacity Bound
Close-Open [6]	$2n$	$\mathcal{O}(\Delta)$	●	-	$\mathcal{C}_{max}$
LOOP [26]	n	$\mathcal{O}(\Delta)$	●	-	$\mathcal{C}_{max}$
Revive [22]	0	$\mathcal{O}(1)$	◑	-	$\mathcal{C}_{max}$
Cycle [23]	1	$\mathcal{O}(\Delta)$	◑	-	$\mathcal{C}_{max}$
Shaduf [25]	1	$\mathcal{O}(1)$	●	$2(N-1)^\dagger$	$\mathcal{C}_{max}$
Starfish	1	$\mathcal{O}(1)$	●	N	$\sum \mathcal{C}_i$

#OPT The number of required on-chain operations

C/NC Cycle-based / Non-cycle-based

Δ The latency of transaction confirmation on a blockchain

$\dagger$ "All to One" binding strategy. Note that the "All Bind" binding strategy leads to $N(N-1)$ OPTs; while the "High to Low" binding strategy leads to N OPTs but cannot rebalance every pair of channels.

$\mathcal{C}_{max}$ The highest channel capacity.

channel for exchanging balances, conduct transactions within the channel, and record the final balances on the underlying blockchain. Such a channel does not require recording each transaction on the blockchain, and its lifecycle consists of three operations.

- **Open.** The process of opening a payment channel involves each party creating a smart contract that locks a certain amount of fund as its initial balance on the blockchain.
- **Update.** Once the channel is open, the participants can transact off-chain by signing transactions between themselves.
- **Close.** To close the channel, the participants must publish the final balances on the blockchain. This allows each party to receive the leftover of the fund that has been locked in the smart contract.

Payment Channel Networks. A payment channel network is composed of interconnected payment channels, allowing two nodes in the network to complete transactions off-chain through a path formed by multiple adjacent channels. In such a network, nodes along a payment path can be categorized into two types: endpoint nodes and routing nodes. The two endpoint nodes act as the initiator and the receiver of a transaction, while routing nodes are responsible for facilitating transaction routing and may charge fees for this service. The network often exhibits a mesh topology, with many nodes having high degrees. Such nodes typically serve as routing nodes without actively initiating transactions. As a result, they are referred to as payment hubs. Payment hubs may frequently face channel depletion issues due to the presence of numerous skewed payments in a payment channel network.

IV. THE DESGIN OF STARFISH

We demonstrate the design of Starfish protocol via a four-channel example illustrated in Fig. 2. Assume a node H (also called hub in Starfish) has established four off-chain channels with users A, B, C, and D, denoted as (H, A), (H, B), (H, C), and (H, D), respectively. The node H has deposited 0, 5, 10, and 21 units respectively in these channels. There

exist two issues in this payment network: (1) H is unable to transfer money to A because H has depleted its deposit in (H, A); (2) even though there is a total balance of 36 owned by H in the payment network, H cannot initiate a transaction of 36 because the balances in these channels are not shared. Starfish aims to rebalance these channels to allow the node H to transact freely with any of the parties A, B, C, and D using its total balance of 36. It primarily involves four procedures: Open Merge, Update Edge, Update Merge, and Close Merge, along with three procedures related to channels: Open Channel, Update Channel and Close Channel.

A. Open Merge: Creating Starfish from Scratch

Open Merge initiates to merge the balances of all channels associated with a single node (i.e., hub), into a single funding pool whose capacity is defined to be the sum of all the involved channel balances. Such a merge operates over a starfish-like structure as depicted in Fig. 2 (Open Merge). It can rebalance the channels to avoid channel depletion and increase liquidity to lift the restrictions on fund transfer. The merge is performed through a smart contract called MERGE, and the collective capacity of the starfish structure after merge remains constant during off-chain trading. For example, in Fig. 2 (Open Merge), to merge the channels, H initiates a merge procedure to pledge all its balances in (H, A), (H, B), (H, C), and (H, D) to the MERGE contract. As all balances are merged, the hub H possesses a collective capacity of 36, which remains constant during the subsequent off-chain trading. By traditional approaches, merging N channels with the "All to One" strategy necessitates $N-1$ instances of binding, resulting in $2(N-1)$ on-chain operations since each binding operates two channels. The issue with this method is that the hub is redundantly operated $N-2$ times. In Starfish, the merge contract only operates one time on each channel to extract its balance.

After collecting all balances to the MERGE contract, off-chain trading can be enabled. Nevertheless, two prominent challenges commonly emerge in this context. (1) Excessive payment: It is necessary to ensure that each newly issued

Fig. 2: Procedures of Starfish.

transaction does not exceed the remaining balance of the payment account. Such a scenario can occur due to network delay or errors in balance counting. Each member of the group must confirm to this requirement. In this case, a smart contract should be able to determine which transaction to reject and finish final settlements safely. (2) Double spending: Starfish should avoid double-spending attacks, especially when the hub colludes with end-users. For example, if the hub H transacts with A and D for amounts of 18 and 19, respectively, the sum is more than the total balance it possesses (i.e., 36), a double-spending attack occurs. A commonly used approach to counter such an attack is to apply consensus algorithms. However, if we adopt a consensus protocol to ensure the full order of all off-chain transactions, it would lead to high communication overhead and the hub that orderly deals with transactions can be a bottleneck.

To solve these challenges, we partition the overall capacity of the funding pool into smaller allocations during the open merge procedure. As depicted in Fig. 2 (Open Merge), the total capacity of 36 is apportioned into 21, 5, 6, and 4 units, with each being assigned to a specific edge, facilitating the connection between the hub and an individual end user. For instance, the capacity of 21 is allocated to the edge identified as ϵ_{H-A}, thereby establishing the initial merge balances of H and A on ϵ_{H-A} as 21 and 0 units, respectively. The allocation of capacity to individual edges is mainly determined by the preceding transaction volumes. In cases where the transaction volume between entities H and A is substantial, the capacity of the ϵ_{H-A} edge should be considerable. After this capacity partitioning, each edge in the starfish structure is allocated with certain amount of balance (the result of rebalancing), namely the capacity of the edge, with which the associated users can trade off-chain. Furthermore, to counter the double-spending attack, each edge is associated with a version number termed versionE.

B. Update Edge: Treatment to Double Spending

The update edge, functioning as an off-chain process, is responsible for modifying the merge balances of the designated users. As depicted in Fig. 2 (Update Edge), H transfers 2, 1, and 3 units to A, B, and C on ϵ_{H-A}, ϵ_{H-B}, and ϵ_{H-C}, respectively. Following three update edge operations, the versionE associated with each edge is incremented. The transactional flow is capable of being bidirectional, with the

stipulation that the capacity of each edge remains conserved during off-chain transactions. Both parties involved in a transaction are required to sign it. These signatures, along with the corresponding versionE values, are leveraged to establish a full order of transactions on the edge and provide evidence to smart contracts to carry out final settlements correctly. Consequently, the Starfish protocol effectively mitigates internal conflicts arising from excessive payments or double-spending attacks.

Moreover, such a design prioritizes efficiency by organizing multiple users into two-party groups based on edges. This strategic partitioning mitigates the need for cumbersome multi-party consensus mechanisms in addressing aforementioned double-spending attacks, while also enabling swift transaction processing in parallel across all edges through the update merge process.

C. Update Merge and Close Merge: Rebalancing Capacity for Improved Availability

In the Starfish protocol, the facilitation of capacity transfers occurs seamlessly through an off-chain update merge operation that is straightforward to implement. This operation is initiated by a request, prompting two edges to transfer their capacities. Subsequently, this transfer is recorded in an off-chain transaction necessitating the signatures of the hub and the two involved end users. Upon broadcast, the transaction requires global consensus for approval. To ensure the sequential integrity of all update merge transactions, an atomic broadcast involving all users is employed. Upon successful completion of the atomic broadcast, indicating unanimous consensus, the update merge transaction is validated and assigned a version number denoted as versionM. This methodology ensures the security of the update merge process while circumventing costly on-chain operations. Given the expediency of atomic broadcasts and the infrequent occurrences of update merge transactions compared to standard off-chain transactions, the associated overhead remains acceptable. For instance, as depicted in Fig. 2 (Update Merge and Close Merge), ϵ_{H-A} transfer the capacity of 4 to ϵ_{H-B} by a update merge operation.

Additionally, the close merge operation safeguards against the potential disruption of the nodes or the dissolution of merge structures upon the departure of end users. It involves disentangling merged channels and is initiated by a request for unmerging, requiring the involvement of the end-users of the affected edges. Throughout the close merge procedure, users

within the edge verify the update edge and merge states by inspecting versionE and versionM, addressing any inconsistencies through challenges. The combined utilization of versionE and versionM ensures the full ordering of transactions. In Fig. 2-(Update Merge and Close Merge), D exits the merge, decreasing the total capacity from 36 to 32. The close merged users retain ownership of the off-chain channels. Furthermore, initiating the closure of channels is within the purview of any user and involves the termination of a channel. The closure process can present complexities, especially when channels are integrated into a merge structure, requiring prior close merge procedure.

V. FORMAL MODELING OF STARFISH

This section outlines the formal model used to analyze the Starfish protocol, covering the network assumptions, security objectives, and the specific notations for channels, merges, and edges.

A. Model

The network has a fixed set of parties $\mathcal{P} = \{P_1, ..., P_n\}$, where all parties are rational non-myopic players [27]. A message (m, σ_P) is deemed valid if σ_P constitutes a valid signature of party P on m. Similar to Perun [3], we consider a synchronous network where messages are exchanged in a round-based manner. Each message between parties is delivered within one round. For instance, a message sent by A in round r can be received by B before the beginning of round $r + 1$. Communications between the parties and the environment $\mathcal{E}$ as well as the ideal functionality $\mathcal{F}$ are instantaneous, with messages sent through secure channels that prevent tampering. The adversary $\mathcal{A}$ can see the messages from honest parties but cannot alter them. We model a static adversary $\mathcal{A}$ that corrupts arbitrary parties before the protocol begins. Corruption means that $\mathcal{A}$ gets all internal states of the corrupted parties and takes full control of them. We denote the transaction confirmation delay as Δ. We use τ to represent the time points related to on-chain contract operations, and t to denote the time points of off-chain operations.

B. Security Goals

Our primary security goal is to ensure the protection of the balances of honest parties, guaranteeing that their funds remain uncompromised. More specifically, our protocol requires a unanimous consent from both parties when opening and updating channels; a collective agreement from all involved parties during the open merge; mutual consent from the parties involved in the update edge and update merge; the processes of closing merge and channel should be completed within a reasonable time, and it is imperative to ensure that the coin shift is conducted based on the latest state of the honest parties. The detailed properties are delineated below:

- **Consensus on Open Channel and Update Channel:** Within a channel, both opening and updating require mutual agreements from the involved parties. The opening of a channel can be completed within $O(\Delta)$ time, whereas updating the channel's state is achieved in a constant time.

- **Consensus on Open Merge and Update Edge:** The open merge process involves multiple channels associated with an intermediate party, requiring the consent of all parties involved. The open merge process requires $O(\Delta)$ time to complete. For the update edge, both parties must confirm each merge state, which takes constant time to complete.

- **Consensus on Update Merge:** The update merge process entails the transfer of capacity between two edges, necessitating the confirmation of state updates among all honest parties. The entire update merge operation should take constant time.

- **Guaranteed Close Merge and Close Channel:** A party associated with an edge can request to close merge and conduct a coin shift for the channel. The close merge process takes $O(\Delta)$ time. Any party in the channel can request to close the channel, which also takes $O(\Delta)$ time.

- **Guaranteed balance payout for users:** The process to close merge and close a channel must settle accounts according to the latest state involving the honest party.

C. Notations

This section defines the core components and operations within our payment channel network model. We describe the properties of individual channels, merge operations that involve multiple connections, and the edges within those merges.

1) Payment Channel (β): A fundamental element is the payment channel, denoted by β, which facilitates transactions directly between two parties. Each channel β is characterized by:

- Unique identifier: $\beta.id \in \{0,1\}^*$. A unique string that distinguishes this channel from all others in the network.
- Participating parties: $\beta.\text{users} = (A, B)$. The ordered pair identifying the two parties authorized to transact using this channel.
- Channel balances: $\beta.\text{balanceC} : \beta.\text{users} \rightarrow \mathbb{R}^{\geq 0}$. This function maps each party (A or B) to their current non-negative balance within the channel. The total funds remain constant within the channel during internal payments.
- Channel Payment (θ): A payment operation within the channel is represented by a function $\theta : \beta.\text{users} \rightarrow \mathbb{R}$. This function specifies the amount transferred, satisfying the conservation property $\theta(A) + \theta(B) = 0$. Applying the payment θ updates the balances such that the new balance for each party $X \in \{A, B\}$ becomes $\beta.\text{balanceC}(X) + \theta(X)$. We denote this update collectively as $\beta.\text{balanceC} \leftarrow \beta.\text{balanceC} + \theta$.
- Channel version number: $\beta.\text{versionC}$. An integer that increments with each update to the channel's state (e.g., after a payment θ), used for state synchronization and conflict resolution.
- Merge set: $\beta.\text{mergeSet}$. This set contains all the merge contracts associated with the channel β.

2) Merge (φ): A merge, denoted by φ, represents a higher-level construct, potentially involving multiple users connected

through a central point or hub via individual edges. A merge is defined by:

- Unique identifier: $\varphi.\tilde{id} \in \{0,1\}^*$. A unique string identifying this specific merge instance.
- Merged Users: $\varphi.\text{users}$. The set containing all users participating in this merge, typically connected to a common hub.
- Merged edges: $\varphi.\text{edges} = (\epsilon_1, \ldots, \epsilon_n)$. A tuple listing all the individual edge connections (*defined below*) that constitute this merge contract.
- Merge version number: $\varphi.\text{versionM}$. An integer that increments when the merge state itself is updated (e.g., through a re-merge or capacity reallocation), tracking the evolution of the merged structure.

3) Edge within a Merge (ϵ): Each edge ϵ within a merge's edge list ($\epsilon \in \varphi.\text{edges}$) represents a specific bilateral relationship, usually between a hub and one user within the merge context. An edge has the following attributes:

- Participating users: $\epsilon.\text{users} = (\text{hub}, \text{user})$. An ordered pair identifying the two endpoints of this edge, typically the central hub and a specific user from $\varphi.\text{users}$.
- Edge capacity: $\epsilon.\text{capacity} \in \mathbb{R}^{\geq 0}$. A non-negative value representing the total capacity locked into this specific edge connection.
- Edge balances: $\epsilon.\text{balanceE} : \epsilon.\text{users} \rightarrow \mathbb{R}^{\geq 0}$. This function maps the hub and the user to their respective non-negative balances within this edge. The sum of these balances must always equal the edge's capacity: $\epsilon.\text{balanceE}(\text{hub}) + \epsilon.\text{balanceE}(\text{user}) = \epsilon.\text{capacity}$.
- Edge Payment ($\tilde{\theta}$): A payment along the edge is defined by a function $\tilde{\theta} : \epsilon.\text{users} \rightarrow \mathbb{R}$, satisfying $\tilde{\theta}(\text{hub}) + \tilde{\theta}(\text{user}) = 0$. Applying $\tilde{\theta}$ updates the edge balances: $\epsilon.\text{balanceE} \leftarrow \epsilon.\text{balanceE} + \tilde{\theta}$.
- Edge version number: $\epsilon.\text{versionE}$. An integer that increments after each update specific to this edge (e.g., after an edge payment $\tilde{\theta}$), tracking the state progression of the individual edge.

4) Merge Update ($\hat{\theta}$): An update merge operation, denoted $\hat{\theta}$, facilitates the reallocation of capacity between two edges, say ϵ_1 and ϵ_2, that are part of the same merge φ. This operation is defined as a function: $\hat{\theta} : \varphi.\text{edges} \rightarrow \mathbb{R}$. This function specifies the change in capacity for edges within the merge. For a reallocation between ϵ_1 and ϵ_2, it must satisfy: $\hat{\theta}(\epsilon_1) + \hat{\theta}(\epsilon_2) = 0$ and $\hat{\theta}(\epsilon_j) = 0$ for all other edges $\epsilon_j \in \varphi.\text{edges} \setminus \{\epsilon_1, \epsilon_2\}$. Applying $\hat{\theta}$ updates the capacities of the involved edges as follows: $\epsilon_i.\text{capacity} \leftarrow \epsilon_i.\text{capacity} + \hat{\theta}(\epsilon_i)$, for $i \in \{1, 2\}$. For brevity, we may denote this collective capacity update for the affected edges $\epsilon \in \{\epsilon_1, \epsilon_2\}$ simply as $\epsilon.\text{capacity} + \hat{\theta}$.

VI. DETAILED STARFISH PROTOCOL

This section provides a detailed description of the Starfish protocol's implementation, including the functioning of the channel and merge smart contracts, the definition of the ideal functionality for security analysis, and the step-by-step execution of the real-world protocol.

A. Channel Contract and Merge Contract

We have used the UC security model [28] to demonstrate the security of our protocol. This model defines two worlds: the ideal world and the real world. In the real world, parties execute the protocol Π while facing an adversary $\mathcal{A}$ and interacting with the contract functionality $\mathcal{C}$. In the ideal world, the idealized protocol, known as ideal functionality $\mathcal{F}$, is executed through interactions with parties and a simulator $\mathcal{S}$, which simulates the behaviors of the adversary. All parties receive inputs from and send outputs to the environment $\mathcal{E}$. The protocol Π is considered UC-secure if the environment $\mathcal{E}$ cannot computationally distinguish whether it is interacting with the protocol in the real world or the one in the ideal world. We define λ as the security parameter, and $\text{EXEC}_{\mathcal{E},\mathcal{A}}^{\Pi,\mathcal{C}}(\lambda)$ denotes the output of environment $\mathcal{E}$ executing the real world protocol Π with adversary $\mathcal{A}$ in the $\mathcal{C}$-hybrid world. $\text{IDEAL}_{\mathcal{E},\mathcal{S}}^{\mathcal{F}}(\lambda)$ denotes the output of environment $\mathcal{E}$ executing the ideal functionality $\mathcal{F}$ with simulator $\mathcal{S}$ in the ideal world. The formal security definition is as follows:

Definition 1. *Protocol Π executing in the $\mathcal{C}$-hybrid world UC-realizes the ideal functionality $\mathcal{F}$ with respect to the global ledger $\mathcal{L}$ and with blockchain delay Δ, if for any PPT adversary $\mathcal{A}$ there exists a simulator $\mathcal{S}$ such that*

$$\text{EXEC}_{\mathcal{E},\mathcal{A}}^{\Pi,\mathcal{C}}(\lambda) \approx \text{IDEAL}_{\mathcal{E},\mathcal{S}}^{\mathcal{F}}(\lambda) \tag{1}$$

where $\approx$ denotes the computational indistinguishability.

Ledger Functionality $\mathcal{L}$

Initialization: The ledger functionality is initialized by a message $(x_1, \ldots, x_n) \in \mathbb{R}_n^{\geq 0}$ from the environment $\mathcal{E}$, where x_i denotes the coins of party P_i in $\mathcal{L}$; the tuple is stored in the ledger.
Adding coins: Upon receiving a message (add, P_i, y) (where $P_i \in \mathcal{P}$ and $y \in \mathbb{R}^{\geq 0}$), the ledger functionality updates $x_i := x_i + y$.
Removing coins: Upon receiving a message (remove, P_i, y) (where $P_i \in \mathcal{P}$ and $y \in \mathbb{R}^{\geq 0}$), the ledger functionality updates $x_i := x_i - y$ if $x_i \geq y$; otherwise, ignores the message and stops.

Fig. 3: Ledger Functionality

Ledger and contract functionalities: The ledger functionality $\mathcal{L}$ is designed as a foundational, transparent, and immutable record of the balances x_i for each party P_i in the system. The core design principle is to establish a single, publicly observable source of truth for asset ownership. A key design choice is the indirect manipulation of balances: parties cannot directly alter the ledger. Instead, all updates (adding or removing coins) are exclusively triggered by the contract functionality $\mathcal{C}$ (in the real world) or its ideal counterpart $\mathcal{F}$ (in the ideal world). This ensures that all ledger modifications adhere to the predefined logic embedded within smart contracts, enhancing security and control. The ledger itself maintains a simple state as a tuple of balances $(x_1, \ldots, x_n)$ and supports two basic operations: add (always successful for non-negative amounts) and remove (only successful if sufficient balance exists), thereby preventing negative balances at the base layer. The

ledger operates passively, responding only to commands from other functionalities.

Contract Functionality $\mathcal{C}$

(A) The contract execution of channel β

Wait for the following messages:

1) Upon receiving $(\mathtt{open}, \beta)$ from A in τ, send $(\mathtt{remove}, A, \beta.\mathsf{balanceC}(A))$ to ledger $\mathcal{L}$ and send $(\mathtt{opening}, \beta)$ to B in τ. Wait for one of the following messages:
   a) Upon receiving $(\mathtt{open}, \beta)$ from B within $\tau_1 \leq \tau + \Delta$, send $(\mathtt{remove}, B, \beta.\mathsf{balanceC}(B))$ to $\mathcal{L}$ in τ_1, output $(\mathtt{opened})$ to $\beta.\mathsf{users}$.
   b) Otherwise, send $(\mathtt{add}, A, \beta.\mathsf{balanceC}(A))$ to $\mathcal{L}$ after $\tau_2 > \tau + \Delta$ and close the contract. Output $(\mathtt{not\text{-}opened})$ to A.

2) Upon receiving $(\mathtt{chan\text{-}merge}, \varphi)$ from $\mathcal{C}(\varphi.\tilde{id})$ in τ, add φ to $\beta.\mathsf{mergeSet}$, set $\beta.\mathsf{balanceC}(\mathsf{hub}) := \beta.\mathsf{balanceC}(\mathsf{hub}) - \epsilon.\mathsf{capacity}$ and stop.

3) Upon receiving $(\mathtt{chan\text{-}closeM}, \varphi, \mathsf{msgE})$ from $\mathcal{C}(\varphi.\tilde{id})$ in τ, remove φ from $\beta.\mathsf{mergeSet}$. Set $\beta.\mathsf{balanceC} := \beta.\mathsf{balanceC} + \epsilon.\mathsf{balanceE}$, and stop.

4) Upon receiving $(\mathtt{closeC}, id, \mathsf{msgC})$ from A in τ, if msgC is valid, store msgC. Send $(\mathtt{closeC}, id)$ to B. Upon receiving $(\mathtt{closeC}, id, \mathsf{msgC})$ from B within $\tau_1 \leq \tau + 4\Delta$, if msgC of B is valid and $\beta^{(B)}.\mathsf{versionC} > \beta^{(A)}.\mathsf{versionC}$, store B's msgC and discard the old one. Send $(\mathtt{add}, A, \beta.\mathsf{balanceC}(A))$ and $(\mathtt{add}, B, \beta.\mathsf{balanceC}(B))$ to $\mathcal{L}$, output $(\mathtt{closedC})$ to $\beta.\mathsf{users}$.

(B) The contract execution of merge φ

Wait for the following messages:

1) Upon receiving $(\mathtt{merge}, \varphi, \Sigma)$ from hub in τ, let β be the underlying channel for each edge $\epsilon \in \varphi.\mathsf{edges}$. If $\beta.\mathsf{balanceC}(\mathsf{hub}) \geq \epsilon.\mathsf{capacity}$, set $\epsilon.\mathsf{balanceE}(\mathsf{hub}) := \epsilon.\mathsf{capacity}$. Send $(\mathtt{chan\text{-}merge}, \varphi)$ to $\mathcal{C}(\beta.id)$ in τ, output $(\mathtt{merged})$ to all users and stop. Otherwise, output $(\mathtt{not\text{-}merged})$ to all users and stop.

2) Upon receiving $(\mathtt{closeM}, \tilde{id}, \epsilon, \mathsf{msgM}, \mathsf{msgE})$ from hub in τ, if msgM and msgE are valid states, store them, and send $(\mathtt{closingM}, \tilde{id}, \epsilon)$ to user. Wait for the following messages:
   a) Upon receiving $(\mathtt{closeM}, \tilde{id}, \epsilon, \mathsf{msgM}, \mathsf{msgE})$ from user in $\tau_1 \leq \tau + \Delta$, if msgM and msgE of user are valid, $\varphi^{(\mathsf{user})}.\mathsf{versionM} > \varphi^{(\mathsf{hub})}.\mathsf{versionM}$ and $\epsilon^{(\mathsf{user})}.\mathsf{versionE} > \epsilon^{(\mathsf{hub})}.\mathsf{versionE}$, store user's msgM and msgE and discard old ones. Send $(\mathtt{closeM\text{-}check}, \tilde{id}, \mathsf{msgM})$ to users.
   b) Upon receiving $(\mathtt{timeout}, \tilde{id})$ after $\tau_2 > \tau + \Delta$ from hub, send $(\mathtt{closeM\text{-}check}, \tilde{id}, \mathsf{msgM})$ to users.
   c) Upon receiving $(\mathtt{closeM\text{-}challenge}, \tilde{id}, \mathsf{msgM})$ from R within $\tau_3 \leq \tau + 2\Delta$, if R's msgM is a valid state and $\varphi^{(R)}.\mathsf{versionM}$ is the highest, keep R's msgM, discard the old one, set $\epsilon.\mathsf{balanceE}$ of msgE to be equal to $\epsilon.\mathsf{capacity}$ of msgM.

   Remove user from $\varphi.\mathsf{users}$. Remove ϵ from $\varphi.\mathsf{edges}$. Send $(\mathtt{chan\text{-}closeM}, \varphi, \mathsf{msgE})$ to $\mathcal{C}(\beta.id)$. Output $(\mathtt{closedM}, \tilde{id}, \epsilon)$ to all users and stop.

Fig. 4: Contract functionality $\mathcal{C}$: (A) The contract execution of channel β; (B) The contract execution of merge φ.

The essence of $\mathcal{C}$'s design lies in its role as a neutral intermediary ensuring the consistent state transitions of channels and merges by interacting with the underlying ledger $\mathcal{L}$ and the involved parties. For channel establishment, it enforces mutual agreement within a defined timeframe. For merges, it facilitates the linking and unlinking of separate contracts, adjusting balances accordingly. During channel closure, it guarantees that both parties agree on the final state before the funds are released back to them, providing a secure and reliable mechanism for managing these collaborative agreements.

The contract functionality $\mathcal{C}$ is designed to manage the lifecycle of two primary types of agreements: channels and merges. For a channel β, the contract oversees its opening by coordinating the removal of initial balances of participants A and B from the ledger $\mathcal{L}$ upon receiving respective $\mathtt{open}$ requests. If both parties agree within a time bound Δ, the channel is marked as $\mathtt{opened}$; otherwise, A's initial balance is returned. During the channel's operation, $\mathcal{C}$ handles merge requests ($\mathtt{chan\text{-}merge}$) by adding a merge contract φ to β's set and decreasing the hub's balance, and it manages the closure of merges ($\mathtt{chan\text{-}closeM}$) by removing φ and adjusting balances. Channel closure ($\mathtt{closeC}$) involves a two-phase commit process where both participants exchange and validate closing messages before their final balances are added back to the ledger, and the channel is marked as $\mathtt{closedC}$.

B. Ideal Functionality and Real Protocol

The ideal functionality $\mathcal{F}$ communicates with the party set $\mathcal{P}$, the simulator $\mathcal{S}$, and the ledger $\mathcal{L}$. To simplify the protocol description without compromising its security, we impose certain restrictions on the inputs provided by the environment $\mathcal{E}$. Specifically, the environment $\mathcal{E}$ should not provide illogical inputs, such as requesting the activation of a channel that is already open or requesting a channel balance that exceeds the user's actual coins in the ledger.

Our ideal functionality $\mathcal{F}$ comprises seven procedures, as shown in Fig. 5. (A) *Open Channel* ensures channel activation only upon mutual agreements of both involved parties. (B) *Update Channel* enables any party to request a state update of the channel, typically for executing payments. This procedure is completed only upon receiving the consent of the channel's other party. (C) *Open Merge*, initialed by the hub of the channels that require merging, is successfully concluded only after obtaining unanimous agreement from all end users. (D) *Update Edge*, initiated by a party within an edge, facilitates the transfer of the merged balances and is finalized only with the approval of the other involved parties. (E) *Update Merge* is designed to shift merged balances between two edges, commencing with the intermediate party and concluding upon unanimous consent from all parties involved in the merge. (F) *Close Merge*, which any party within an edge can initiate, results in returning the capacity back to the channel balance upon its completion. (G) *Close Channel*, open to initiation by either party of a channel, leads to the transfer of the channel balance to the ledger $\mathcal{L}$ at the end of the process.

Next, we provide an analysis regarding how the ideal functionality $\mathcal{F}$ achieves the security properties.

Consensus on Open Channel and Update Channel. A channel can only be opened when A initiates the request and B agrees, requiring a mutual consensus between the two parties.

Ideal Functionality $\mathcal{F}$

(A) Open Channel

Upon receiving $(\texttt{open}, \beta)$ from A in t, let $B := \beta.\text{other-party}(A)$, proceed as follows:

1) Within $t_1 \leq t + \Delta$, send $(\texttt{remove}, A, \beta.\text{balanceC}(A))$ to ledger $\mathcal{L}$.
2) After completing step 1, send $(\texttt{opening}, \beta)$ to B in t_1.
3) Upon receiving $(\texttt{open}, \beta)$ from B in t_1, send $(\texttt{remove}, B, \beta.\text{balance}(B))$ to $\mathcal{L}$ within $t_2 \leq t_1 + \Delta$, output $(\texttt{opened})$ to $\beta.\text{users}$ and simulator $\mathcal{S}$, then stop; otherwise, send $(\texttt{add}, A, \beta.\text{balance}(A))$ to $\mathcal{L}$ after $t_2 > t_1 + \Delta$, output $(\texttt{not-opened})$ to A, then stop.

(B) Update Channel

Upon receiving $(\texttt{updateC}, id, \theta)$ from A in t, proceed as follows:

1) Send $(\texttt{updateC-req}, id, \theta)$ to B in $t_1 := t + 1$. To maintain synchronization with the rounds of our protocol, $\mathcal{F}$ waits for one round before sending the message.
2) Upon receiving $(\texttt{updateC-ok})$ from B in t_1, set $\beta.\text{balance} := \beta.\text{balance} + \theta$, output $(\texttt{updated})$ to A in $t_2 := t_1 + 1$, then stop; otherwise, output $(\texttt{not-updated})$ to A, then stop.

(C) Open Merge

Upon receiving $(\texttt{merge}, \varphi)$ from hub in t, proceed as follows:

1) If hub is honest, send $(\texttt{merge-req}, \varphi)$ to $P \in \varphi.\text{users}$ in $t_1 := t + 1$; otherwise, upon receiving $(\texttt{send-req}, P)$ from $\mathcal{S}$ in t, send $(\texttt{merge-req}, \varphi)$ to P in t_1.
2) Upon receiving $(\texttt{merge}, \varphi)$ from all users in t_1, send $(\texttt{merge-confirm}, \varphi)$ to hub in $t_2 := t_1 + 1$.
3) Upon receiving $(\texttt{merge-confirmed}, \varphi)$ from hub in t_2, let β be the underlying channel for each edge $\epsilon \in \varphi.\text{edges}$. Add φ to $\beta.\text{mergeSet}$. Set $\beta.\text{balanceC}(\text{hub}) := \beta.\text{balanceC}(\text{hub}) - \varphi.\text{capacity}(\epsilon)$. Output $(\texttt{merged})$ to all users and $\mathcal{S}$ within $t_3 \leq t_2 + \Delta$, then stop; otherwise, output $(\texttt{not-merged})$, then stop.

(D) Update Edge

Upon receiving $(\texttt{updateE}, \tilde{id}, \tilde{\theta}, \epsilon)$ from hub in t, proceed as follows:

1) Send $(\texttt{updateE-req}, \tilde{id}, \tilde{\theta})$ to user in $t_1 := t + 1$.
2) Upon receiving $(\texttt{updateE-ok})$ from user in t_1, output $(\texttt{updatedE})$ to hub in $t_2 := t_1 + 1$, then stop; otherwise, output $(\texttt{not-updatedE})$, then stop.

(E) Update Merge

Let ϵ_1 and ϵ_2 be the edges involved, connecting to end-users P and Q, respectively. Let $R \in \varphi.\text{users}$. Upon receiving $(\texttt{updateM}, \tilde{id}, \hat{\theta}, \epsilon_1, \epsilon_2)$ from hub in t, proceed as follows:

1) If hub is honest, send $(\texttt{updateM-req}, \tilde{id}, \hat{\theta}, \epsilon_1, \epsilon_2)$ to P and Q in $t_1 := t + 1$; otherwise, upon receiving $(\texttt{send-req}, P)$ from $\mathcal{S}$ in t, send $(\texttt{updateM-req}, \tilde{id}, \hat{\theta}, \epsilon_1, \epsilon_2)$ to P in t_1, and similarly for Q.
2) Upon receiving $(\texttt{updateM-ok})$ from P and Q in t_1, send $(\texttt{updateM-confirm}, \tilde{id}, \hat{\theta}, \epsilon_1, \epsilon_2)$ to hub in $t_2 := t_1 + 1$. Upon receiving $(\texttt{updateM-confirmed})$ from hub in t_2, send $(\texttt{updateM-pending}, \tilde{id}, \hat{\theta}, \epsilon_1, \epsilon_2)$ to users in $t_3 := t_2 + 2$.
3) Upon receiving $(\texttt{updateM-wrong})$ from R and R is honest in t_3, output $(\texttt{not-updatedM})$ to $\varphi.\text{users}$, and stop. Otherwise, update the $\texttt{capacity}$ and $\texttt{balanceE}$ of ϵ_1 and ϵ_2, respectively. Output $(\texttt{updatedM})$ to $\varphi.\text{users}$, and stop.

(F) Close Merge

Upon receiving $(\texttt{closeM}, \tilde{id}, \epsilon)$ from hub in t, proceed as follows:

1) Assume user is the end party, remove user from $\varphi.\text{users}$, remove ϵ from $\varphi.\text{edges}$.
2) Let β denote the underlying channel corresponding to ϵ, remove φ from $\beta.\text{mergeSet}$, set $\beta.\text{balanceC} := \beta.\text{balanceC} + \epsilon.\text{balanceE}$. Output $(\texttt{closedM}, \tilde{id}, \epsilon)$ to $\varphi.\text{users}$ and user within $t_1 \leq t + 3\Delta$ and stop.

(G) Close Channel

Upon receiving $(\texttt{closeC}, id)$ from A in t, proceed as follows:

1) If $\beta.\text{mergeSet} = \emptyset$, within round $t_1 \leq t + 2\Delta$, send $(\texttt{add}, A, \beta.\text{balanceC}(A))$ to ledger $\mathcal{L}$, and send $(\texttt{add}, B, \beta.\text{balanceC}(B))$ to ledger $\mathcal{L}$. Output $(\texttt{closedC})$ to $\beta.\text{users}$, and stop.
2) Otherwise, for each merge φ in $\beta.\text{mergeSet}$, execute procedure (F). Let t_2 be the current round. Within $t_3 \leq t_2 + 2\Delta$, send $(\texttt{add}, A, \beta.\text{balanceC}(A))$ to ledger $\mathcal{L}$, and send $(\texttt{add}, B, \beta.\text{balanceC}(B))$ to ledger $\mathcal{L}$. Output $(\texttt{closedC})$ to $\beta.\text{users}$, and stop.

Fig. 5: Ideal Functionailty of Starfish

The time interval from the initiation of the open channel request to the actual opening is 2Δ. Similarly, an update of the channel also requires a consensus between both parties. Under the premise of both parties being honest, updating the channel takes 2 rounds.

Consensus on Open Merge and Update Edge. The procedure of open merge requires initiation by a hub and consent from all end users. The entire merging process is expected to take $\Delta + 2$ rounds. The update edge process is similar to the update channel, necessitating consensus among the edge parties. If both parties remain honest, the operation is expected to take 2 rounds.

Consensus on Update Merge. The update merge process mandates initiation by the hub, followed by confirmations from all the end users. In this procedure, the parties associated with the edges involved in the update merge are required to reach a consensus regarding the update merge request. Subsequently, the hub executes an atomic broadcast to synchronize the

current update merge state. If all parties are honest, the procedure is expected to take 4 rounds.

Guaranteed Close Merge and Close Channel. Within any merge edge, any party can issue a close merge request to remove its channel from the merge contract. The procedure is expected to be completed within 3Δ time. Any party within a channel can issue a close channel request to close the channel. Should the channel not be integrated into a merge contract, it is expected to close within 2Δ time; otherwise, it first requires 3Δ time to close merge the channel before the closure can be executed.

Guaranteed balance payout for users. Once a channel has been close merged, the current merge balance is transferred to the channel balance. When the channel is closed, the latest state of the channel balance is stored in the parties' account in the ledger $\mathcal{L}$.

Then we provide a comprehensive description on the real-world Starfish protocol as depicted in Fig. 6 and Fig. 7.

Initially, we outline the procedure for opening a channel, as shown in Procedure (A), with the corresponding contract functionalities illustrated in Fig. 4. Upon receiving $(\texttt{open}, \beta)$ from the environment $\mathcal{E}$, A sends a request to the contract instance $\mathcal{C}(\beta.id)$ for channel activation. Subsequently, $\mathcal{C}(\beta.id)$ informs B about the initiation of the channel. If B agrees to open the channel within Δ time, $\mathcal{C}(\beta.id)$ records the opening of channel β, reallocates the channel balances from ledger $\mathcal{L}$ into the contract, and outputs $(\texttt{opened})$ to both A and B. Otherwise, $\mathcal{C}(\beta.id)$ outputs $(\texttt{not-opened})$ to A.

The process of channel updating is depicted in Procedure (B). Updating the channel does not require interaction with the contract. Upon receiving $(\texttt{updateC}, id, \theta)$ from $\mathcal{E}$, A first increments its local version number (i.e., $\beta.\mathsf{versionC}$), and updates the latest balance allocation based on θ. A then sends the updated state along with its signature to B, requesting a channel state update. After verifying the correctness of the signature and the state version number, B seeks approval from $\mathcal{E}$ for the channel update. If $\mathcal{E}$ approves the update, B responds to A with an updated state signed by its signature. If B does not respond, A outputs $(\underline{\texttt{not-updatedC}})$ and terminates the update channel process.

The procedure for open merge is shown in Procedure (C), with the corresponding contract functionalities detailed in Fig. 4. The open merge process requires initiation by the hub. To prevent replay attacks, the hub first signs the open merge message (φ, t) and sends this signed open merge request to the end users. Upon receiving the request, each end user verifies the validity of the signature and then requests permission to merge from $\mathcal{E}$. Once permission is granted, each end user forwards the merge message along with its own signature back to the hub. The hub aggregates all the signatures and sends the complete merge message, now containing the collective signatures, to the contract $\mathcal{C}(\varphi.\tilde{id})$. Subsequently, $\mathcal{C}(\varphi.\tilde{id})$ invokes the relevant channel instances to record the merge information and outputs $(\texttt{merged})$ to all users.

Following the merge process, parties can transact using the edge balances, as shown in Procedure (D). Upon receiving $(\texttt{updateE}, \tilde{id}, \tilde{\theta})$ from $\mathcal{E}$, the hub first increments its local edge version number $\epsilon.\mathsf{versionE}$ by 1 and adjusts the edge balance allocation according to $\tilde{\theta}$. The hub then sends the updated edge state, along with its signature, to the corresponding end user. After verifying the correctness of the updated merge state number and signature, the end user requests confirmation from $\mathcal{E}$ for the state update. Once the confirmation is received, the end user replies to the hub with a signed update edge message. If the end user does not respond, the hub outputs $(\underline{\texttt{not-updatedE}})$ and terminates the process.

The update merge process, depicted in Procedure (E), enables the transfer of capacity between different edges. Initiated by the hub, this procedure typically reallocates capacity from edge ϵ_1 to ϵ_2, where P and Q are the respective end users of these edges. Upon receiving $(\texttt{updateM}, \tilde{id}, \hat{\theta}, \epsilon_1, \epsilon_2)$ from the environment $\mathcal{E}$, the hub increments its local merge version number (i.e., $\varphi.\mathsf{versionM}$), updates the capacity allocation according to $\hat{\theta}$, and disseminates the updated merge state and signatures to P and Q. Once P and Q validate these updates and obtain approval from $\mathcal{E}$, they return their signed confirmations to the hub. Subsequently, the hub initiates an atomic broadcast to synchronize all users within the merge group to the latest merge version number. During this broadcast, each user verifies the proposed capacity update and the merge version number. If any user detects inconsistencies, it ignores the message; otherwise, it votes to accept the broadcast. If the atomic broadcast succeeds, all users update their local $\varphi.\mathsf{versionM}$ and the capacities of the involved edges. Specifically, for the users directly connected to ϵ_1 and ϵ_2 (i.e., P, Q, and the hub), they additionally update their local $\epsilon.\mathsf{versionE}$ and $\epsilon.\mathsf{balanceE}$ values to reflect the new edge states. Upon completing these updates, each user outputs $(\underline{\texttt{updatedM}})$ and terminates the process. If the atomic broadcast fails, all users output $(\underline{\texttt{not-updatedM}})$.

The procedure for close merge is shown in Procedure (F), with the associated contract functions detailed in Fig. 4. Any user within a merge edge can submit a request to close merge, thereby removing its edge from the merge contract. Specifically, suppose the hub submits a close merge request; then, $\mathcal{C}(\varphi.\tilde{id})$ notifies the corresponding user and waits for a response within Δ time. If the user responds with a valid close merge state, namely by providing its local edge and merge version numbers that are higher than those of the hub, $\mathcal{C}(\varphi.\tilde{id})$ records this state. Otherwise, if the user does not respond within Δ time, $\mathcal{C}(\varphi.\tilde{id})$ defaults to the state provided by the hub. Thereafter, $\mathcal{C}(\varphi.\tilde{id})$ sends a close merge check message, containing the current highest valid merge version number, to all other users in the merge. If any user challenges by submitting its local merge version number, $\mathcal{C}(\varphi.\tilde{id})$ adopts the highest valid merge version number. Finally, letting β denote the underlying channel corresponding to the edge ϵ, $\mathcal{C}(\varphi.\tilde{id})$ invokes $\mathcal{C}(\beta.id)$ to reallocate the channel balances based on the finalized highest valid edge and merge version numbers.

The procedure for close channel is illustrated in Procedure (G), with the related contract functions detailed in Fig. 4. Either party of the channel has the capability to initiate its closure. Upon receiving input from $\mathcal{E}$, A sends a close channel request to $\mathcal{C}(\beta.id)$. If at this time channel β is still part of a merge contract, procedure (F) is invoked to close merge

---

Starfish Protocol

Assume A is the caller and B is the responder. Let $\beta^{(A)}$ denote the channel state form A's perspective, and similarly for B.

(A) Open Channel

(1) **[A]** Upon receiving (open, β) from the environment $\mathcal{E}$ in t, A initializes a new channel contract instance $\mathcal{C}(\beta.id)$ and sends a Δ-bounded contract construction message (open, β), then proceeds to step (4).

(2) **[B]** Upon receiving (opening, β) from $\mathcal{C}(\beta.id)$ within $\tau \leq t + \Delta$, B sends (opening, β) to $\mathcal{E}$. B sends (open, β) to $\mathcal{C}(\beta.id)$ after receiving (open) from $\mathcal{E}$ in τ, then proceeds to step (3).

(3) **[B]** Upon receiving (opened) from $\mathcal{C}(\beta.id)$ within $\tau_1 \leq \tau + \Delta$, B outputs (opened) and stops.

(4) **[A]** Upon receiving (opened) from $\mathcal{C}(\beta.id)$ within $\tau_2 \leq t + 2\Delta$, A outputs (opened) and stops. If receiving (not-opened) from $\mathcal{C}(\beta.id)$ after $\tau_3 > t + 2\Delta$, A outputs (not-opened) and stops.

(B) Update Channel

(1) **[A]** Upon receiving (updateC, id, θ) from $\mathcal{E}$ in t, A sets $\mathsf{msgC} = (\beta^{(A)}.\mathsf{versionC} + 1, \beta^{(A)}.\mathsf{balanceC} + \theta)$ and signs it with σ_A. A sends (updateC, id, msgC, σ_A) to B and proceeds to step (3).

(2) **[B]** Upon receiving (updateC, id, $\mathsf{msgC} = (v, b)$, σ_A) from A in $t_1 := t + 1$, B checks $v \overset{?}{=} \beta^{(B)}.\mathsf{versionC} + 1$. If the condition holds, B sends (updateC-req, id, $b - \beta^{(B)}.\mathsf{balanceC}$) to $\mathcal{E}$. Upon receiving (updateC-ok) from $\mathcal{E}$, B generates the signature σ_B on msgC and sends (updateC, σ_B) to A in t_1. If the condition does not hold, B ignores the message.

(3) **[A]** Upon receiving (updateC, σ_B) from B in $t_2 := t_1 + 1$, A outputs (updatedC) and stops; otherwise, A outputs (not-updatedC) and stops.

(G) Close Channel

(1) **[A]** Upon receiving (closeC, id) from $\mathcal{E}$ in t, if $\beta.\mathsf{mergeSet} \neq \emptyset$, A executes procedure (F) for each $\varphi \in \beta.\mathsf{mergeSet}$. After closing all merges, A sets $\mathsf{msgC} = (\beta^{(A)}.\mathsf{versionC}, \beta^{(A)}.\mathsf{balanceC}, \Sigma_C)$, where Σ_C is the signature set of the current channel state. A sends (closeC, id, msgC) to $\mathcal{C}(\beta.id)$ and goes to step (3).

(2) **[B]** Upon receiving (closingC, id) from $\mathcal{C}(\beta.id)$ in τ, if $\beta.\mathsf{mergeSet} \neq \emptyset$, B executes procedure (F) for each $\varphi \in \beta.\mathsf{mergeSet}$. After closing all merges, B generates msgC like A and sends (closeC, id, msgC) to $\mathcal{C}(\beta.id)$ and goes to step (3).

(3) **[A, B]** Upon receiving (closedC) from $\mathcal{C}(\beta.id)$ within $\tau_1 \leq \tau + 4\Delta$, A and B output (closedC) and stops.

Fig. 6: Channel Operations

the channel; otherwise, $\mathcal{C}(\beta.id)$ notifies B and waits for the response within Δ time. If B responds, $\mathcal{C}(\beta.id)$ allocates the channel balance based on the highest valid channel version number agreed upon by both parties; if B does not respond within Δ time, the allocation is based on the valid channel version number provided by A. Ultimately, the contract refunds the users' coins and closes the channel.

Theorem 1. *The protocol Starfish executing in the $\mathcal{C}$-hybrid world UC-realizes the ideal functionality $\mathcal{F}$ with respect to the global ledger $\mathcal{L}$ and blockchain delay Δ.*

Proof. The formal proof is provided in the supplementary material. $\square$

VII. EVALUATION

To demonstrate the practicality of Starfish, we implemented a functional prototype leveraging the Ethereum platform. We then conducted a simulation study on the Lightning Network dataset to evaluate its performance within a payment channel network, contrasting our findings with those reported for Revive and Shaduf.

A. Implementation

Our Starfish implementation is built on the Ethereum platform and comprises two contracts, namely the merge contract and the channel contract. The Starfish, Shaduf, and Revive projects have about 320, 440, and 260 LOCs, respectively. To use the Starfish protocol, users need to deploy only one merge contract, which serves them permanently, and consumes minimal on-chain resources. However, when creating new channels or making functional calls while running the protocol, there will be on-chain expenses. These expenses are calculated in gas, which is a unit of measurement used for quantifying the computational cost of transactions and smart contract executions on the Ethereum network. The initiator of a transaction needs to provide enough gas fee[2]. Deploying the merge and channel contracts requires 1.8 million and 1.4 million gas, respectively.

B. Gas Cost

1) Settings: We vary the number of channels involved in rebalancing from 2 to 10, measuring the total gas cost of on-chain operations for Starfish, Shaduf and Revive. The implementation of Starfish requires Open/Close channels and Merge/Close merge contracts while that of Shaduf involves on-chain operations such as Open/Close channels and Bind/Unbind; but Revive only deals with Open/Close channel operations. Therefore, in our context, we compare Starfish with Shaduf and Revive, but won't include Revive's figures to save space.

In Starfish, we calculate the total gas cost involved in performing various operations such as opening, merging, closing the merge contract, and closing N channels. It is important to note that N channels require only one Merge operation and one Close Merge operation. In Shaduf, the binding strategy

---

[2]The gas fee is calculated by multiplying the gas price, measured in Gwei, by the amount of gas required for the transaction or smart contract execution.

Starfish Protocol

(C) Open Merge

(1) **[Hub]** Upon receiving $(\underline{\texttt{merge}}, \varphi)$ from $\mathcal{E}$ in t, hub generates $\sigma_{\texttt{hub}}$ for (φ, t), sends $(\underline{\texttt{merge}}, \varphi, \sigma_{\texttt{hub}})$ to $P \in \varphi.\texttt{users}$ in t, and goes to step (3).

(2) **[Users]** Upon receiving $(\overline{\underline{\texttt{merge}}, \varphi, \sigma_{\texttt{hub}}})$ from hub in $t_1 := t + 1$, P sends $(\underline{\texttt{merge-req}}, \varphi)$ to $\mathcal{E}$. Upon receiving $(\underline{\texttt{merge-ok}})$ from $\mathcal{E}$, P generates σ_P for (φ, t), sends $(\underline{\texttt{merge}}, \varphi, \sigma_P)$ to hub, and goes to step (5).

(3) **[Hub]** Upon receiving all $(\underline{\texttt{merge}}, \varphi, \sigma_P)$ in $t_2 := t_1 + 1$, hub sends $(\underline{\texttt{merge-confirm}}, \varphi, \Sigma)$ to $\mathcal{E}$, where Σ collects signatures from all users. Upon receiving $(\underline{\texttt{merge-confirmed}})$ from $\mathcal{E}$, hub creates a merge contract $\mathcal{C}(\varphi.\tilde{id})$ and sends a Δ-bounded message $(\underline{\texttt{merge}}, \varphi, \Sigma)$ to $\mathcal{C}(\varphi.\tilde{id})$. Then hub goes to step (4).

(4) **[Hub]** Upon receiving $(\underline{\texttt{merged}}, \varphi)$ from $\mathcal{C}(\varphi.\tilde{id})$ within $\tau \le t_2 + \Delta$, hub updates balanceC and balanceE according to the contracts. hub outputs $(\underline{\texttt{merged}}, \varphi)$ and stops. Otherwise, hub outputs $(\underline{\texttt{not-merged}})$ and stops.

(5) **[Users]** Upon receiving $(\underline{\texttt{merged}}, \varphi)$ from $\mathcal{C}(\varphi.\tilde{id})$ within round $\tau \le t_1 + \Delta + 1$, P updates balanceC and balanceE according to the contracts. P outputs $(\underline{\texttt{merged}})$, then stops; otherwise, P outputs $(\underline{\texttt{not-merged}})$ after receiving $(\underline{\texttt{not-merged}})$ from $\mathcal{C}(\varphi.\tilde{id})$, and stops.

(D) Update Edge

(1) **[Hub]** Upon receiving $(\underline{\texttt{updateE}}, \tilde{id}, \tilde{\theta}, \epsilon)$ from $\mathcal{E}$ in t, hub sets msgE $= (\epsilon^{(\texttt{hub})}.\texttt{versionE} + 1, \epsilon^{(\texttt{hub})}.\texttt{balanceE} + \tilde{\theta})$ and signs it with $\sigma_{\texttt{hub}}$. hub sends $(\underline{\texttt{updateE}}, \tilde{id}, \text{msgE}, \sigma_{\texttt{hub}})$ to user and proceeds to step (3).

(2) **[User]** Upon receiving $(\underline{\texttt{updateE}}, \tilde{id}, \text{msgE} = (v, b), \sigma_{\texttt{hub}})$ from hub in $t_1 := t + 1$, user checks $v \overset{?}{=} \epsilon^{(\texttt{user})}.\texttt{versionE} + 1$. If the condition holds, user sends $(\underline{\texttt{updateE-req}}, \tilde{id}, b - \epsilon^{(\texttt{user})}.\texttt{balanceE})$ to $\mathcal{E}$. user updates $\epsilon^{(\texttt{user})}.\texttt{versionE} := v$ and $\epsilon^{(\texttt{user})}.\texttt{balanceE} := b$ after receiving $(\underline{\texttt{updateE-ok}})$ from $\mathcal{E}$. user generates $\sigma_{\texttt{user}}$ on msgE and sends $(\underline{\texttt{updateE}}, \sigma_{\texttt{user}})$ to hub, and stops. Otherwise, user ignores the message.

(3) **[Hub]** Upon receiving $(\underline{\texttt{updateE}}, \sigma_{\texttt{user}})$ from user in $t_2 := t_1 + 1$, hub outputs $(\underline{\texttt{updatedE}})$, and stops; otherwise, hub outputs $(\underline{\texttt{not-updateE}})$.

(E) Update Merge

Let ϵ_1 and ϵ_2 be the edges involved, connecting to end-users P and Q, respectively. Let cap denote the set of all edge capacities, and let $R \in \varphi.\texttt{users}$.

(1) **[Hub]** Upon receiving $(\underline{\texttt{updateM}}, \tilde{id}, \hat{\theta}, \epsilon_1, \epsilon_2)$ from $\mathcal{E}$ in t, hub sets msgM $= (\varphi^{(\texttt{hub})}.\texttt{versionM} + 1, \texttt{cap}^{(\texttt{hub})})$ and msgE $= (\epsilon_1^{(\texttt{hub})}.\texttt{versionE} + 1, \epsilon_2^{(\texttt{hub})}.\texttt{versionE} + 1, \epsilon^{(\texttt{hub})}.\texttt{capacity} + \hat{\theta}, \epsilon^{(\texttt{hub})}.\texttt{balanceE}(\texttt{hub}) + \hat{\theta})$, and generates $\sigma_{\texttt{hub}}$ on them. hub sends $(\underline{\texttt{updateM}}, \tilde{id}, \text{msgM}, \text{msgE}, \sigma_{\texttt{hub}})$ to P, Q, and goes to step (3).

(2) **[Users]** Upon receiving $(\underline{\texttt{updateM}}, \tilde{id}, \text{msgM} = (vM, c), \text{msgE} = (vE_1, vE_2, \hat{c}, \hat{b}), \sigma_{\texttt{hub}})$ in $t_1 := t + 1$, P (or Q) checks $vM \overset{?}{=} \varphi^{(P)}.\texttt{versionM} + 1$, $vE_1 \overset{?}{=} \epsilon_1^{(P)}.\texttt{versionE} + 1$. If the condition holds, P sends $(\underline{\texttt{updateM-req}}, \tilde{id}, \hat{c} - \epsilon^{(P)}.\texttt{capacity}, \epsilon_1, \epsilon_2)$ to $\mathcal{E}$. Upon receiving $(\underline{\texttt{updateM-ok}})$ from $\mathcal{E}$, P generates σ_P for msgM and msgE, sends $(\underline{\texttt{updateM}}, \sigma_P)$ to hub, and goes to step (4). Otherwise, P ignores the message.

(3) **[Hub]** Upon receiving $(\underline{\texttt{updateM}}, \sigma_{P(Q)})$ from both P and Q in $t_2 := t_1 + 1$, hub sends $(\underline{\texttt{updateM-confirm}})$ to $\mathcal{E}$. Upon receiving $(\underline{\texttt{updateM-confirmed}})$ from $\mathcal{E}$, hub runs AtomicBroadcast$(\underline{\texttt{updateM}}, \tilde{id}, \text{msgM}, \text{msgE}, \Sigma)$ to $\varphi.\texttt{users}$, where Σ collects the above signatures. If the AtomicBroadcast succeeds, hub outputs $(\underline{\texttt{updatedM}})$ in $t_3 := t_2 + 2$ and stops; otherwise, hub outputs $(\underline{\texttt{not-updatedM}})$, and stops.

(4) **[Users]** Upon receiving $(\underline{\texttt{updateM}}, \tilde{id}, \text{msgM} = (vM, c), \text{msgE} = (vE_1, vE_2, \hat{c}, \hat{b}), \Sigma)$, R checks $vM \overset{?}{=} \varphi^{(R)}.\texttt{versionM} + 1$. If the condition holds, R broadcasts the message; if not, R sends $(\underline{\texttt{updateM-pending}})$ to $\mathcal{E}$. Upon receiving $(\underline{\texttt{updateM-wrong}})$ from $\mathcal{E}$, R ignores the message. If the AtomicBroadcast succeeds, R updates both the capacity of each edge and versionM to remain consistent with hub. If R is either P or Q, R then updates versionE and balanceE of the corresponding edge to remain consistent with hub. R outputs $(\underline{\texttt{updatedM}})$ in t_3 and stops; otherwise, R outputs $(\underline{\texttt{not-updatedM}})$, and stops.

(F) Close Merge

Assume hub is the caller and user is the responder. Let cap denote the set of all edge capacities. Let users denote $\varphi.\texttt{users} \setminus \{\texttt{user}\}$, and $R \in \texttt{users}$.

(1) **[Hub]** Upon receiving $(\underline{\texttt{closeM}}, \tilde{id}, \epsilon)$ from $\mathcal{E}$ in t, hub sets msgM $= (\varphi^{(\texttt{hub})}.\texttt{versionM}, \texttt{cap}^{(\texttt{hub})}, \Sigma_R)$, where Σ_R is the signature set of the current merge state. hub sets msgE $= (\epsilon^{(\texttt{hub})}.\texttt{versionE}, \epsilon^{(\texttt{hub})}.\texttt{balanceE}, \Sigma_E)$, where Σ_E is the signature set of the current edge state. hub sends $(\underline{\texttt{closeM}}, \tilde{id}, \epsilon, \text{msgM}, \text{msgE})$ to $\mathcal{C}(\varphi.\tilde{id})$ in t, then goes to step (3).

(2) **[User]** Upon receiving $(\underline{\texttt{closingM}}, \tilde{id}, \epsilon)$ from $\mathcal{C}(\varphi.\tilde{id})$ in τ, user generates msgM and msgE like hub. user sends $(\underline{\texttt{closeM}}, \tilde{id}, \epsilon, \text{msgM}, \text{msgE})$ to $\mathcal{C}(\varphi.\tilde{id})$ in τ, and goes to step (5).

(3) **[Hub]** If not receiving $(\underline{\texttt{closedM}}, \tilde{id}, \epsilon)$ within 2Δ time, hub sends $(\underline{\texttt{timeout}}, \tilde{id})$ to $\mathcal{C}(\varphi.\tilde{id})$, and goes to step (5).

(4) **[Users]** Upon receiving $(\underline{\texttt{closeM-check}}, \tilde{id}, \text{msgM} = (v, c))$ from $\mathcal{C}(\varphi.\tilde{id})$ in $\tau_1 \le \tau + \Delta$, R checks if $\varphi^{(R)}.\texttt{versionM} > v$. If the condition holds, R generates msgM like hub. R sends $(\underline{\texttt{closeM-challenge}}, \tilde{id}, \text{msgM})$ to $\mathcal{C}(\varphi.\tilde{id})$ and goes to step (6); otherwise, R goes to step (6).

(5) **[Hub, User]** Upon receiving $(\underline{\texttt{closedM}}, \tilde{id}, \epsilon)$ from $\mathcal{C}(\varphi.\tilde{id})$ within $\tau_2 \le \tau + 2\Delta$, hub and user remove φ, update balanceC based on the contracts, output $(\underline{\texttt{closedM}}, \tilde{id}, \epsilon)$ and then stop.

(6) **[Users]** Upon receiving $(\underline{\texttt{closedM}}, \tilde{id}, \epsilon)$ from $\mathcal{C}(\varphi.\tilde{id})$ in τ_2, R updates φ based on the contracts. R outputs $(\underline{\texttt{closedM}}, \tilde{id}, \epsilon)$ and then stops.

Fig. 7: Merge Operations

---

determines the number of on-chain operations required by Bind and Unbind. Therefore, we explore three different binding strategies: "High to Low", "All to One", and "All Bind". We respectively denote the Shaduf corresponding to these three strategies as HL-Shaduf, AO-Shaduf, and AB-Shaduf. In Revive, a globally impartial node oversees the rebalancing process. Only when there is a dispute over the rebalancing outcome while closing channels, on-chain operations related to rebalancing are needed. Therefore, we estimate the possible

maximum consumption of gas in Revive when opening and closing N channels.

2) Results and Discussion: We analyze the assessment outcomes from two different angles.

Gas Cost of Starfish. There are four types of on-chain operations for Starfish: Open Channel, Merge, Close Merge, and Close Channel. Fig. 8(a) illustrates that the gas cost for opening and closing all channels is directly proportional to the number of channels. This is because the gas cost for a single

(a) Starfish   (b) HL-Shaduf   (c) AO-Shaduf   (d) AB-Shaduf

Fig. 8: The gas cost comparison of Starfish and Shaduf (using three different strategies).

(a) Uniform small payments   (b) Skew small payments   (c) Fixed Skewness for small payments   (d) Uniform large payments

Fig. 9: The success ratio of Starfish when varying the channel capacity, payment skewness and payment values.

Open Channel or Close Channel operation remains unchanged, but the number of channels impacts the total gas cost. The gas cost for merging and unmerging all channels also increases linearly with the number of channels. Although N channels only require one Merge and Close Merge operation, with each additional channel added, every Merge and Close Merge operation necessitates storing the state of the new channel and performing an extra signature verification. Consequently, the gas cost increases at a constant rate with the addition of one channel.

Cost Comparison. We first compare Shaduf's Bind/Unbind operations with Starfish's Merge/Close Merge operations. The gas cost for binding and unbinding all channels scales linearly with the number of channels in HL-Shaduf (Fig. 8(b)) and AO-Shaduf (Fig. 8(c)), while in AB-Shaduf (Fig. 8(d)), it follows a quadratic relationship with the number of channels. This is because the gas cost for Bind and Unbind operations remains constant, while noting that for HL-Shaduf, AO-Shaduf, and AB-Shaduf, N channels respectively require N, $2N$, and $N(N-1)$ On-chain operations. It's important to highlight that, in HL-Shaduf and AO-Shaduf, the slope of the two lines is respectively $1.6\times$ and $3.3\times$ that of Starfish. This is because the linear growth in Shaduf is due to multiple Bind/Unbind operations, which result in higher gas cost compared to Starfish, because of the additional storage and computation required. When the number of channels reaches 10, the gas cost for Bind/Unbind operations in HL-Shaduf, AO-Shaduf, and AB-Shaduf is approximately $1.4\times$, $2.5\times$, and $12.6\times$, respectively, compared to Starfish's Merge/Close Merge.

C. Success Ratio

1) Settings: To enhance simulation realism, we use the real-world Lightning Network topology for performance evaluation. It is worth noting that Starfish is adaptable to any blockchain supporting Turing-complete smart contracts. As the initial balance allocation in channels is invisible, we evenly distribute the total balances among all channels. We use two methods for sampling payment initiators and receivers: uniform sampling, where each user has an equal chance of initiating or receiving a payment, and skewed sampling, where a subset of users who are more likely to initiate payments is selected. Transaction values are randomly sampled from Bitcoin transaction data from 2021-03-01 to 2021-03-31, resulting in a dataset of 2.65 million small payments and 6.65 million large payments. For Revive, if a channel (referred to as the target channel) of a node along the payment path is depleted, it is rebalanced using other channels (referred to as the source channels) of the same node. Specifically, we select the shortest circle that concurrently includes both the target channel and the source channel. We evaluate three methods for Shaduf: HL-Shaduf, AO-Shaduf, and AB-Shaduf, and merge all balances across all channels for each node in Starfish. To mitigate the impact of randomness, we evaluate the success ratio of 50,000 payments ten times and average the results for the final success ratio.

2) Results: We investigate how the success rates of six off-chain payment methods, with LN (Lightning Network) as a baseline, are affected by their respective capacity, skewness, and payment size.

The impact of channel capacity. From Fig.s 9(a), 9(d), and 9(c), it is apparent that as the channel capacity increases, the success rate of each protocol tends to rise. This is expected

since a larger channel capacity enables the handling of a greater volume and size of transactions. However, notably, throughout the range of channel capacity from $1\times$ to $25\times$, Starfish consistently maintains a superior success ratio compared to other protocols.

The impact of payment skewness. We explore the impact of payment skewness on the success ratio as shown in Fig. 9(b). It can be observed that under different skewness levels, Starfish consistently exhibits the highest success ratio. Specifically, compared to LN with an average success ratio, Revive increases by around 9%, HL-Shaduf, AO-Shaduf, and AB-Shaduf show enhancements of around 15%, 14%, and 18%, respectively, while Starfish sees an increase of approximately 22%. We fix the skewness at 8 and evaluate the success ratio when channel capacities varying from $1\times$ to $25\times$. The results are presented in Fig. 9(c). One can see that compared to LN's average success rate, Revive shows an increase of around 8%, while HL-Shaduf, AO-Shaduf, and AB-Shaduf demonstrate increases of around 12%, 12%, and 15%, respectively. Additionally, Starfish exhibits an increase of around 19%.

The impact of payment size. We explore the impact of payment size on the success ratio. Fig.s 9(a) and 9(d) illustrate that Starfish consistently outperforms in both small and large payment scenarios. Specifically, we compare the average success rates with LN. Under the uniform small payment scenario, Revive shows an increase of around 8%, HL-Shaduf, AO-Shaduf, and AB-Shaduf demonstrate increases of around 15%, 14%, and 19%, respectively, while Starfish exhibits an increase of around 23%. Under the uniform large payment scenario, Revive shows an increase of around 3%, HL-Shaduf, AO-Shaduf, and AB-Shaduf demonstrate increases of around 11%, 7%, and 15%, respectively, while Starfish exhibits an increase of around 23%.

VIII. CONCLUSION

In conclusion, our research introduces Starfish, a novel payment network addressing scalability challenges in blockchain Payment Channel Networks (PCNs). Starfish enhances rebalancing efficiency by allowing channels to borrow funds, demonstrating optimal performance in simulations. The Ethereum-based implementation proves practical feasibility. Compared to existing protocols, Starfish consistently outperforms in success ratios for off-chain payments, making it a promising solution for scalable and efficient blockchain transactions.

REFERENCES

[1] Y. Chen, H. Chen, Y. Zhang, M. Han, M. Siddula, and Z. Cai, "A survey on blockchain systems: Attacks, defenses, and privacy preservation," *High-Confidence Computing*, vol. 2, no. 2, p. 100048, 2022.

[2] M. Hearn, "Micro-payment channels implementation now in bitcoinj," *Bitcointalk. org*, 2013.

[3] S. Dziembowski, L. Eckey, S. Faust, and D. Malinowski, "Perun: Virtual payment hubs over cryptocurrencies," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 106–123.

[4] R. Network-Fast, "cheap, scalable token transfers for ethereum," *Accessed: Jul*, vol. 7, p. 2020, 2018.

[5] M. Xu, Y. Guo, Q. Hu, Z. Xiong, D. Yu, and X. Cheng, "A trustless architecture of blockchain-enabled metaverse," *High-confidence computing*, vol. 3, no. 1, p. 100088, 2023.

[6] J. Poon and T. Dryja, "The bitcoin lightning network: Scalable off-chain instant payments," 2016.

[7] L. Aumayr, M. Maffei, O. Ersoy, A. Erwig, S. Faust, S. Riahi, K. Hostáková, and P. Moreno-Sanchez, "Bitcoin-compatible virtual channels," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 901–918.

[8] A. Miller, I. Bentov, S. Bakshi, R. Kumaresan, and P. McCorry, "Sprites and state channels: Payment networks that go faster than lightning," in *International conference on financial cryptography and data security*. Springer, 2019, pp. 508–526.

[9] C. Egger, P. Moreno-Sanchez, and M. Maffei, "Atomic multi-channel updates with constant collateral in bitcoin-compatible payment-channel networks," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 801–815.

[10] P. McCorry, S. Bakshi, I. Bentov, S. Meiklejohn, and A. Miller, "Pisa: Arbitration outsourcing for state channels," in *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, 2019, pp. 16–30.

[11] W. Yu, M. Xu, D. Yu, X. Cheng, Q. Hu, and Z. Xiong, "Zk-pcn: A privacy-preserving payment channel network using zk-snarks," in *2022 IEEE International Performance, Computing, and Communications Conference (IPCCC)*. IEEE, 2022, pp. 57–64.

[12] Y. Guo, M. Xu, D. Yu, Y. Yu, R. Ranjan, and X. Cheng, "Cross-channel: Scalable off-chain channels supporting fair and atomic cross-chain operations," *IEEE Transactions on Computers*, vol. 72, no. 11, pp. 3231–3244, 2023.

[13] P. Li, T. Miyazaki, and W. Zhou, "Secure balance planning of off-blockchain payment channel networks," in *IEEE INFOCOM 2020-IEEE conference on computer communications*. IEEE, 2020, pp. 1728–1737.

[14] X. Luo and P. Li, "Learning-based off-chain transaction scheduling in prioritized payment channel networks," *IEEE Journal on Selected Areas in Communications*, vol. 40, no. 12, pp. 3589–3599, 2022.

[15] G. Avarikioti, G. Janssen, Y. Wang, and R. Wattenhofer, "Payment network design with fees," in *Data Privacy Management, Cryptocurrencies and Blockchain Technology: ESORICS 2018 International Workshops, DPM 2018 and CBT 2018, Barcelona, Spain, September 6-7, 2018, Proceedings 13*. Springer, 2018, pp. 76–84.

[16] V. Sivaraman, S. B. Venkatakrishnan, K. Ruan, P. Negi, L. Yang, R. Mittal, G. Fanti, and M. Alizadeh, "High throughput cryptocurrency routing in payment channel networks," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2020, pp. 777–796.

[17] L. Yang, X. Dong, S. Gao, Q. Qu, X. Zhang, W. Tian, and Y. Shen, "Optimal hub placement and deadlock-free routing for payment channel network scalability," *arXiv preprint arXiv:2305.19182*, 2023.

[18] X. Wang, R. Yu, D. Yang, G. Xue, H. Gu, Z. Li, and F. Zhou, "Fence: Fee-based online balance-aware routing in payment channel networks," *IEEE/ACM Transactions on Networking*, 2023.

[19] Y. Liu, Y. Wu, F. Zhao, and Y. Ren, "Balanced off-chain payment channel network routing strategy based on weight calculation," *The Computer Journal*, p. bxad029, 2023.

[20] S. Jiang, J. Wu, F. Zuo, and A. Mei, "Balance-aware cost-efficient routing in the payment channel network," in *2023 IEEE/ACIS 21st International Conference on Software Engineering Research, Management and Applications (SERA)*. IEEE, 2023, pp. 8–15.

[21] S. Lin, J. Zhang, and W. Wu, "Fstr: Funds skewness aware transaction routing for payment channel networks," in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2020, pp. 464–475.

[22] R. Khalil and A. Gervais, "Revive: Rebalancing off-blockchain payment networks," in *Proceedings of the 2017 acm sigsac conference on computer and communications security*, 2017, pp. 439–453.

[23] Z. Hong, S. Guo, R. Zhang, P. Li, Y. Zhan, and W. Chen, "Cycle: Sustainable off-chain payment channel network with asynchronous rebalancing," in *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2022, pp. 41–53.

[24] W. Ni, P. Chen, L. Chen, P. Cheng, C. J. Zhang, and X. Lin, "Utility-aware payment channel network rebalance," *Proceedings of the VLDB Endowment*, vol. 17, no. 2, pp. 184–196, 2023.

[25] Z. Ge, Y. Zhang, Y. Long, and D. Gu, "Shaduf: Non-cycle payment channel rebalancing," in *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24–28, 2022*.

[26] L. Labs, "Loop," https://lightning.engineering/loop/, 2019.

[27] I. Tsabary, M. Yechieli, A. Manuskin, and I. Eyal, "Mad-htlc: because htlc is crazy-cheap to attack," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 1230–1248.

[28] R. Canetti, "Universally composable security: A new paradigm for cryptographic protocols," in *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*. IEEE, 2001, pp. 136–145.

Minghui Xu (Member, IEEE) received the BS degree in physics from Beijing Normal University, Beijing, China, in 2018, and the PhD degree in computer science from George Washington University, Washington DC, USA, in 2021. He is currently an associate professor with the School of Computer Science and Technology, Shandong University, China. His research focuses on blockchain, distributed computing, and applied cryptography.

Wenxuan Yu received the B.E. degree in School of Computer Science and Technology from Harbin Engineering University, Harbin, China, in 2021. He is currently a Ph.D. candidate in the School of Computer Science and Technology, Shandong University, Qingdao, China. His current research interests include applied cryptography, secure multiparty computation, and blockchain.

Guangyong Shang is currently with Inspur Yunzhou Industrial Internet Co., Ltd, Jinan, China. His research interests include blockchain and artificial intelligence.

Guangpeng Qi is currently with Inspur Yunzhou Industrial Internet Co., Ltd, Jinan, China. His research interests include cloud computing, blockchain, and industrial internet systems.

Dongliang Duan received the M.S. degree in school computer science and technology from Shandong University, Qingdao, China, in 2024. He is currently an engineer at ByteDance, Beijing, China. His research interests include blockchain and privacy.

Shan Wang received the B.S. and Ph.D. degrees in computer science from Southeast University, Nanjing, China, in 2016 and 2022, respectively. She is currently a Postdoctoral Fellow with the Department of Computing, The Hong Kong Polytechnic University, Hong Kong, China. Her current research interests include permissioned blockchain systems, blockchain user anonymity, and applied cryptography.

Kun Li received the B.S. degree in information science and technology from Beijing Normal University, Beijing, China, in 2017, and the Ph.D. degree in artificial intelligence from the School of Artificial Intelligence, Beijing Normal University, in 2023. She is currently an Assistant Professor at Shandong University. Her research interests include mobile computing, and blockchain.

Yue Zhang received the Ph.D. degree in computer science from Jinan University, Guangzhou, China, in 2020. He is currently a Professor with the School of Computer Science and Technology, Shandong University, Qingdao, China. His current research interests include cybersecurity, program analysis, mobile security, IoT security, and blockchain.

Xiuzhen Cheng (Fellow, IEEE) received the MS and PhD degrees in computer science from the University of Minnesota - Twin Cities, in 2000 and 2002, respectively. She is a professor with the School of Computer Science and Technology, Shandong University. Her current research interests include wireless and mobile security, cyber physical systems, wireless and mobile computing, sensor networking, and algorithm design and analysis. She has served on the editorial boards of several technical journals and the technical program committees of various professional conferences/workshops. She also has chaired several international conferences. She worked as a program director for the US National Science Foundation (NSF) from April to October in 2006 (full time), and from April 2008 to May 2010 (part time). She received the NSF CAREER Award in 2004. She is a member of ACM.

APPENDIX

Theorem 2. *The protocol Starfish executing in the $\mathcal{C}$-hybrid world UC-realizes the ideal functionality $\mathcal{F}$ with respect to the global ledger $\mathcal{L}$ and blockchain delay Δ.*

Proof. Simulator $\mathcal{S}$ facilitates interaction with the environment $\mathcal{E}$ and the ideal functionality $\mathcal{F}$. To achieve indistinguishability between the real and ideal worlds, $\mathcal{S}$ is required to simulate the behavior of the predefined adversary $\mathcal{A}$. This involves corrupting the same parties in the ideal world as adversary $\mathcal{A}$ does in the real world. $\mathcal{S}$ also needs to generate public-private key pairs for all parties, distributing the public keys to the corrupted parties. For each corrupted party, $\mathcal{S}$ sends the corresponding private key individually. $\mathcal{S}$ observes the actions of adversary $\mathcal{A}$ in the real world and the instructions given to the corrupted parties, then selects identical inputs to submit to the ideal functionality $\mathcal{F}$. Furthermore, $\mathcal{S}$ represents the contract function $\mathcal{C}$ and honest parties (i.e., dummy parties) to send messages to the corrupted parties. It is important to note that $\mathcal{S}$ does not simulate scenarios where all parties are either honest or corrupt. In cases involving two-party channels or edges, our simulation considers scenarios where either one party is honest or the other is corrupt. For multi-party interactions within the merge operations, we differentiate scenarios based on the hub being either honest or corrupt. Additionally, we simulate situations where the end users, according to this categorization, is portrayed as either honest or corrupt.

□

Simulation

(A) Open Channel
Case: A is honest and B is corrupt

Upon A sending (open, β) to ideal functionality $\mathcal{F}$ in round t, send (open, β) to contract instance $\mathcal{C}(\beta.id)$ on behalf of A in the same round. Assume the message reaches the ledger $\mathcal{L}$ within $\tau \leq t + \Delta$. If B sends (open, β) to $\mathcal{C}(\beta.id)$ in τ, send (open, β) to $\mathcal{F}$ on behalf of B in the same round.

Case: A is corrupt and B is honest

Upon A sending (open, β) to contract instance $\mathcal{C}(\beta.id)$ in round t, send (open, β) to ideal functionality $\mathcal{F}$ on behalf of A in the same round. Assume the message reaches the ledger $\mathcal{L}$ within $\tau \leq t + \Delta$. If B sends (open, β) to $\mathcal{F}$ in τ, send (open, β) to $\mathcal{C}(\beta.id)$ on behalf of B in the same round.

(B) Update Channel
Case: A is honest and B is corrupt

Upon A sending $(\mathsf{updateC}, id, \theta)$ to ideal functionality $\mathcal{F}$ in round t, sign σ_A on $\mathsf{msgC} = (\beta^{(A)}.\mathsf{versionC} + 1, \beta^{(A)}.\mathsf{balanceC} + \theta)$ and send $(\mathsf{updateC}, id, \mathsf{msgC}, \sigma_A)$ to B on behalf of A in the same round. If B sends $(\mathsf{updateC}, \sigma_B)$ to A in $t_1 := t + 1$, send $(\mathsf{updateC\text{-}ok})$ to $\mathcal{F}$ on behalf of B in the same round.

Case: A is corrupt and B is honest

Let $\mathsf{msgC} = (v, b)$. Upon A sending $(\mathsf{updateC}, id, \mathsf{msgC}, \sigma_A)$ to B in round t, if $v = \beta^{(B)}.\mathsf{versionC} + 1$, send $(\mathsf{updateC}, id, b - \beta^{(B)}.\mathsf{balanceC})$ to $\mathcal{F}$ on behalf of A in the same round; otherwise, ignore the message and stop. If B sends $(\mathsf{updateC\text{-}ok})$ to $\mathcal{F}$ in $t_1 := t + 1$, send $(\mathsf{updateC}, \sigma_B)$ to A on behalf of B in the same round.

(C) Open Merge
Case: hub is honest

Upon hub sending $(\mathsf{merge}, \varphi)$ to ideal functionality $\mathcal{F}$ in round t, send $(\mathsf{merge}, \varphi, \sigma_{\mathsf{hub}})$ to $P \in \varphi.\mathsf{users}$ on behalf of hub in the same round. Proceed as follows:
1) If P is corrupt and sends $(\mathsf{merge}, \varphi, \sigma_P)$ to hub in $t_1 := t + 1$, send $(\mathsf{merge}, \varphi)$ to $\mathcal{F}$ on behalf of P in the same round. If P is honest and sends $(\mathsf{merge}, \varphi)$ to $\mathcal{F}$ in t_1, send $(\mathsf{merge}, \varphi, \sigma_P)$ to hub on behalf of P in the same round.
2) Upon hub sending $(\mathsf{merge\text{-}confirmed}, \varphi)$ to $\mathcal{F}$ in $t_2 := t_1 + 1$, send $(\mathsf{merge}, \varphi, \Sigma)$ to $\mathcal{C}(\varphi.\tilde{id})$ on behalf of hub in the same round, where Σ represents the set of signatures from all users.

Case: hub is corrupt

Upon hub sending $(\mathsf{merge}, \varphi, \sigma_{\mathsf{hub}})$ to $P \in \varphi.\mathsf{users}$ and P is honest in round t, send $(\mathsf{merge}, \varphi)$ to $\mathcal{F}$ on behalf of hub and send $(\mathsf{send\text{-}req}, P)$ to $\mathcal{F}$ in the same round. Proceed as follows:
1) If P is corrupt, send $(\mathsf{merge}, \varphi)$ to $\mathcal{F}$ on behalf of P in $t_1 := t + 1$. If P is honest and sends $(\mathsf{merge}, \varphi)$ to $\mathcal{F}$ in t_1, send $(\mathsf{merge}, \varphi, \sigma_P)$ to hub on behalf of P in the same round.
2) Upon hub sending $(\mathsf{merge}, \varphi, \Sigma)$ to $\mathcal{C}(\varphi.\tilde{id})$ in $t_2 := t_1 + 1$, where Σ represents the set of signatures from all users, send $(\mathsf{merge\text{-}confirmed}, \varphi)$ to $\mathcal{F}$ on behalf of hub in the same round.

(D) Update Edge
Case: hub is honest and user is corrupt

Upon hub sending $(\mathsf{updateE}, \tilde{id}, \tilde{\theta}, \epsilon)$ to ideal functionality $\mathcal{F}$ in round t, sign σ_{hub} on $\mathsf{msgE} = (\epsilon^{(\mathsf{hub})}.\mathsf{versionE} + 1, \epsilon^{(\mathsf{hub})}.\mathsf{balanceE} + \tilde{\theta})$ and send $(\mathsf{updateE}, \tilde{id}, \mathsf{msgE}, \sigma_{\mathsf{hub}})$ to user on behalf of hub in the same round. If user sends $(\mathsf{updateE}, \sigma_{\mathsf{user}})$ to hub in $t_1 := t + 1$, send $(\mathsf{updateE\text{-}ok})$ to $\mathcal{F}$ on behalf of user in the same round.

Case: hub is corrupt and user is honest

Let $\mathsf{msgE} = (v, b)$, upon hub sending $(\mathsf{updateE}, \tilde{id}, \mathsf{msgE}, \sigma_{\mathsf{hub}})$ to user in round t, if $v = \epsilon^{(\mathsf{user})}.\mathsf{versionE} + 1$, send $(\mathsf{updateE}, \tilde{id}, b - \epsilon^{(\mathsf{user})}.\mathsf{balanceE})$ to $\mathcal{F}$ on behalf of hub in the same round; otherwise, ignore the message and stop. If user sends $(\mathsf{updateE\text{-}ok})$ to $\mathcal{F}$ in $t_1 := t + 1$, send $(\mathsf{updateE}, \sigma_{\mathsf{user}})$ to hub on behalf of user in the same round.

(E) Update Merge
Case: hub is honest

Upon hub sending $(\mathsf{updateM}, \tilde{id}, \hat{\theta}, \epsilon_1, \epsilon_2)$ to ideal functionality $\mathcal{F}$ in round t, sign σ_{hub} on $\mathsf{msgM} = (\varphi^{(\mathsf{hub})}.\mathsf{versionM} + 1, \mathsf{cap}^{(\mathsf{hub})})$ and $\mathsf{msgE} = (\epsilon_1^{(\mathsf{hub})}.\mathsf{versionE} + 1, \epsilon_2^{(\mathsf{hub})}.\mathsf{versionE} + 1, \epsilon^{(\mathsf{hub})}.\mathsf{capacity} + \hat{\theta}, \epsilon^{(\mathsf{hub})}.\mathsf{balanceE}(\mathsf{hub}) + \hat{\theta})$, send $(\mathsf{updateM}, \tilde{id}, \mathsf{msgM}, \mathsf{msgE}, \sigma_{\mathsf{hub}})$ to P and Q (the end-users of ϵ_1 and ϵ_2, respectively) on behalf of hub in the same round. Proceed as follows:
1) If P (or Q) is corrupt and sends $(\mathsf{updateM}, \sigma_P)$ to hub in $t_1 := t + 1$, send $(\mathsf{updateM\text{-}ok})$ to $\mathcal{F}$ on behalf of P (or Q) in the same round. If P (or Q) is honest and sends $(\mathsf{updateM\text{-}ok})$ to $\mathcal{F}$ in t_1, send $(\mathsf{updateM}, \sigma_P)$ to hub on behalf of P (or Q) in the same round.
2) If hub sends $(\mathsf{updateM\text{-}confirmed})$ to $\mathcal{F}$ in $t_2 := t_1 + 1$, run $\mathsf{AtomicBroadcast}(\mathsf{updateM}, \tilde{id}, \mathsf{msgM}, \mathsf{msgE}, \Sigma)$ to $\varphi.\mathsf{users}$ on behalf of hub in the same round, where Σ collects the above signatures.

Case: hub is corrupt

Let $\mathsf{msgM} = (vM, c)$ and $\mathsf{msgE} = (vE_1, vE_2, \hat{c}, \hat{b})$, upon hub sending $(\mathsf{updateM}, \tilde{id}, \mathsf{msgM}, \mathsf{msgE}, \sigma_{\mathsf{hub}})$ to P (or Q) and P is honest in round t, send $(\mathsf{updateM}, \tilde{id}, \hat{\theta}, \epsilon_1, \epsilon_2)$ to $\mathcal{F}$ on behalf of hub and send $(\mathsf{send\text{-}req}, P)$ to $\mathcal{F}$ in the same round. Proceed as follows:
1) If P (or Q) is corrupt, send $(\mathsf{updateM\text{-}ok})$ to $\mathcal{F}$ on behalf of P in $t_1 := t + 1$. If P is honest and sends $(\mathsf{updateM\text{-}ok})$ to $\mathcal{F}$ in t_1, send $(\mathsf{updateM}, \sigma_P)$ to hub on behalf of P in the same round.
2) If hub runs $\mathsf{AtomicBroadcast}(\mathsf{updateM}, \tilde{id}, \mathsf{msgM}, \mathsf{msgE}, \Sigma)$ to $\varphi.\mathsf{users}$ in $t_2 := t_1 + 1$, where Σ collects the above signatures. Send $(\mathsf{updateM\text{-}confirmed})$ to $\mathcal{F}$ on behalf of hub in the same round.
3) If $R \in \varphi.\mathsf{users}$ is honest, and sends $(\mathsf{updateM\text{-}wrong})$ to $\mathcal{F}$ in $t_3 := t_2 + 2$, ignore the $\mathsf{AtomicBrodcast}$ messages on behalf of R.

(F) Close Merge
Case: hub is honest and user is corrupt

Upon hub sending ($\texttt{closeM}, \tilde{id}, \epsilon$) to ideal functionality $\mathcal{F}$ in round t, set $\mathsf{msgM} = (\varphi^{(\mathsf{hub})}.\mathsf{versionM}, \mathsf{cap}^{(\mathsf{hub})}, \Sigma_R)$, where Σ_R is the signature set of the current merge state. Set $\mathsf{msgE} = (\epsilon^{(\mathsf{hub})}.\mathsf{versionE}, \epsilon^{(\mathsf{hub})}.\mathsf{balanceE}, \Sigma_E)$, where Σ_E is the signature set of the current edge state. Send ($\texttt{closeM}, \tilde{id}, \epsilon, \mathsf{msgM}, \mathsf{msgE}$) to $\mathcal{C}(\varphi.\tilde{id})$ on behalf of hub in the same round. Proceed as follows:

1) Assume the message reaches ledger $\mathcal{L}$ within round $\tau \leq t + \Delta$. If user does not send any message, send ($\texttt{timeout}, \tilde{id}$) to $\mathcal{C}(\varphi.\tilde{id})$ on behalf of hub in round $\tau_1 > \tau + \Delta$.
2) Let $R \in \varphi.\mathsf{users}\backslash\{\mathsf{user}\}$. If R is corrupt and does not send message to $\mathcal{C}(\varphi.\tilde{id})$ within round $\tau_2 \leq \tau + 2\Delta$. Send confirmation to $\mathcal{F}$ on behalf of R in the same round.

Case: hub is corrupt and user is honest

Upon hub sending ($\texttt{closeM}, \tilde{id}, \epsilon, \mathsf{msgM}, \mathsf{msgE}$) to $\mathcal{C}(\varphi.\tilde{id})$ in round t, send ($\texttt{closeM}, \tilde{id}, \epsilon$) to ideal functionality $\mathcal{F}$ on behalf of hub in the same round. Proceed as follows:

1) Assume the message reaches ledger $\mathcal{L}$ within round $\tau \leq t + \Delta$, send ($\texttt{closeM}, \tilde{id}, \epsilon, \mathsf{msgM}, \mathsf{msgE}$) to $\mathcal{C}(\varphi.\tilde{id})$ on behalf of user in τ.
2) Let $R \in \varphi.\mathsf{users}\backslash\{\mathsf{user}\}$. If R is honest and sends confirmation to $\mathcal{F}$ within round $\tau_2 \leq \tau + 2\Delta$, send ($\texttt{closeM-challenge}, \tilde{id}, \mathsf{msgM}$) to $\mathcal{C}(\varphi.\tilde{id})$ on behalf of R in the same round.

(G) Close Channel
Case: A is honest and B is corrupt

Upon A sending ($\texttt{closeC}, id$) to ideal functionality $\mathcal{F}$ in round t, generates $\mathsf{msgC} = (\beta^{(A)}.\mathsf{versionC}, \beta^{(A)}.\mathsf{balanceC}, \Sigma_C)$, where Σ_C denotes the signatures of A and B for the current channel state. Sends ($\texttt{closeC}, id, \mathsf{msgC}$) to $\mathcal{C}(\beta.id)$ on behalf of A in the same round. Assume the message reaches ledger $\mathcal{L}$ within round $\tau \leq t + \Delta$. If $\beta.\mathsf{mergeSet} = \emptyset$ and B sends ($\texttt{closeC}, id, \mathsf{msgC}$) to $\mathcal{C}(\beta.id)$ in round τ, where $\mathsf{msgC} = (\beta^{(B)}.\mathsf{versionC}, \beta^{(B)}.\mathsf{balanceC}, \Sigma_C)$, sends confirmation to $\mathcal{F}$ on behalf of B in the same round. Otherwise, for each merge in $\beta.\mathsf{mergeSet}$, A executes Procedure (F) in round τ.

Case: A is corrupt and B is honest

Upon A sending ($\texttt{closeC}, id, \mathsf{msgC}$) to $\mathcal{C}(\beta.id)$ in round t, where $\mathsf{msgC} = (v, b, \Sigma_C)$, sends ($\texttt{closeC}, id$) to ideal functionality $\mathcal{F}$ on behalf of A in the same round. Assume the message reaches ledger $\mathcal{L}$ within round $\tau \leq t + \Delta$. If $\beta.\mathsf{mergeSet} = \emptyset$, sends ($\texttt{closeC}, id, \mathsf{msgC}$) to $\mathcal{C}(\beta.id)$ on behalf of B in round τ, where $\mathsf{msgC} = (\beta^{(B)}.\mathsf{versionC}, \beta^{(B)}.\mathsf{balanceC}, \Sigma_C)$. Otherwise, for each merge in $\beta.\mathsf{mergeSet}$, B executes Procedure (F) in round τ.