Extras and Premiums: Local PCN Routing with Redundancy and Fees

Yu Shen[1], Oğuzhan Ersoy[2,1], and Stefanie Roos[1]

[1] Delft University of Technology, The Netherlands
{y.shen-5,s.roos}@tudelft.nl
[2] Radboud University, The Netherlands
oguzhan.ersoy@ru.nl

Abstract. Payment channel networks (PCNs) are a promising solution to the blockchain scalability problem. In PCNs, a sender can route a multi-hop payment to a receiver via intermediaries. Yet, Lightning, the only prominent payment channel network, has two major issues when it comes to multi-hop payments. First, the sender decides on the path without being able to take local capacity restrictions into account. Second, due to the atomicity of payments, any failure in the path causes a failure of the complete payment.

In this work, we propose **F**orward-**U**pdate-**Fi**nalize (FUFi): The sender adds redundancy to a locally routed payment by initially committing to sending a higher amount than the actual payment value. Intermediaries decide on how to forward a received payment, potentially splitting it between multiple paths. If they cannot forward the total payment value, they may reduce the amount they forward. If paths for sufficient funds are found, the receiver and sender jointly select the paths and amounts that will actually be paid. Payment commitments are updated accordingly and fulfilled. In order to guarantee atomicity and correctness of the payment value, we use a modified Hashed Time Lock Contract (HTLC) for paying that requires both the sender and the receiver to provide a secret preimage. FUFi furthermore is the first local routing protocol to include fees and specify a fee policy to intermediaries on how to determine their fair share of fees.

We prove that the proposed protocol achieves all key security properties of multi-hop payments. Furthermore, our evaluation on both synthetic and real-world Lightning topologies shows FUFi outperforms existing algorithms in terms of fraction of successful payments by about 10%.

1 Introduction

Payment channel networks (PCNs) enable blockchain scalability by increasing the throughput of transactions and reducing latency, and fees [14]. They move payments off-chain, i.e., not all payments have to be included in the blockchain. Thus, they do not require that every payment is broadcast to all participants and verified by them.

Two parties fund a payment channel by depositing coins into a joined account using a blockchain transaction, and then they can make direct transactions in the channel [4]. Opening such a channel only pays off for frequent transaction partners due to the need for the initial blockchain transaction. If a sender S wants to send funds to a receiver R without having a direct channel, they can route the funds via multiple existing channels. For instance if both S and R have a channel with P, P can act as intermediary. It is important that both the channel between S and P and the one between P and R have sufficient funds [17].

Designing routing algorithms to find sufficiently funded paths between senders and receivers successfully and swiftly is thus the key for a usable PCN. In the literature, there are two types of routing algorithms: source routing, in which the sender decides on the path, and local routing, where intermediaries decide on which neighbor they forward the payment to. Lightning, Bitcoin's PCN, uses source routing: The sender determines one path to the receiver based on a publicly available snapshot of the network topology [14]. However, the snapshot does not include the exact amount of funds available in each channel. So payments may fail due to channels in the path having insufficient funds.

There are approaches for the sender to split payments into multiple sub-payments, each routed via a different path [13, 19]. Smaller payments are less likely to exceed the amount of available funds in a channel and congestion information can be utilized to determine the most suitable paths [19]. Yet, if only one channel in one path has insufficient funds, the payment fails. You can retry the payment using different paths and splits of the total payment amount but it might take a large number of tries until you find a distribution of funds on paths that works. Boomerang [3] adds redundancy to the payment, i.e., the sender initially sends more funds than the actual payment value. In this manner, some failures along individual paths do not result in the overall payment failing. If more than the payment value arrive at the receiver, they are returned to the sender.

Boomerang mitigates the lack of knowledge about local distributions of funds but does not fundamentally address it. In contrast, local routing protocols leave the decision of how to find a path to the intermediaries of a payment, who are clearly aware of the amount of funds in their channels [6, 17]. However, failures are still possible if the routing ends up at a node that does not have any outgoing channels with sufficient funds. So, redundancy should be included here as well. Applying Boomerang is possible, however, it requires that the sender splits the payment whereas local routing protocols leave the splitting to intermediaries [6]. In addition, intermediaries typically receive a fee to incentivize participation. In source routing, the sender can compute the required fee as they know the path or paths but in local routing, the sender cannot know the fee in advance and it is challenging to find a suitable algorithm for fee computation. So, local routing due to its awareness of local constraints is more suitable for PCNs than source routing but the existing local routing algorithms are not useful in practice due to their lack of redundancy and incentives.

We here address both issues by presenting **F**orward-**U**pdate-**Fi**nalize(FUFi). We integrate fees into local routing by having the sender add a separate amount dedicated to fees to each payment, which they compute based on the expected number of intermediaries. Intermediaries know how much of the payment they receive is dedicated to fees and can then subtract an amount as their own fees. We suggest but do not enforce a policy on how to choose the amount, which provides more fees to the parties early on in the path who take a higher risk as they need to lock up funds for longer than parties close to the receiver. While we do not enforce the policy as such, taking more funds than suggested results in insufficient fees for successors on the path who then do not forward the payment. If a payment fails, no fees are paid, so nodes are disincentivized from deviating from the fee policy.

For redundancy in FUFi, the sender increases the payment amount by a factor r. Intermediaries may then reduce the amount they forward. Concretely, when an intermediary receives amount a to **forward**[3], they determine a set of neighbors that can provide them with a path to the receiver. If the amount a can be split between these neighbors, the intermediary splits the amount a; otherwise, a smaller amount is split. In the first phase of routing, only commitments are made to pay the respective amounts but payments and amounts are not yet finalized. Once sufficient funds reach the receiver, commitments are **updated** such that incoming funds (minus fees) at an intermediaries match the outgoing funds and any redundant funds are returned to the sender. In order to ensure updated commitments, parties are incentivized to revoke the old commitment as the alternative is a failed payment, which implies no fees. After the amounts are updated, the payments can be **finalized**. In Lightning, finalization means that the receiver reveals a hash preimage that allows them and all intermediaries to claim their funds. However, with redundancy present, a rational receiver should reveal the secret before updating to gain additional funds. To prevent this loss of funds, the sender needs to also provide a second hash preimage, which they only provide once all commitments are updated.

We prove that intermediaries do not lose any funds by participating in FUFi, the sender only loses the original payment value plus fees, and the payment terminates. Furthermore, the sender loses funds only if they obtain a signed receipt of the receiver. In turn, if the receiver provides a receipt, they are indeed paid. To show that FUFi indeed achieves better performance than state-of-the-art routing algorithms, we extend an existing payment channel network simulator. Our results on both real-world Lightning snapshots and synthetic topologies indicates that a redundancy factor $r = 1.8$ achieves the best results. Furthermore, for such redundancy values, FUFi improves upon Boomerang and the local routing protocol Interdimensional SpeedyMurmurs [6] by about 10% in terms of fraction of successful payments.

[3] so a is the amount after they subtracted their fees

2 Background and Related Work

In this section, we first provide the necessary background on PCNs and the routing of the payments. Then, we discuss closely related works on routing protocols.

2.1 Payment channel networks

A payment channel allows two parties to exchange transactions without publishing them on the blockchain [9]. First, two parties open a channel by publishing a funding transaction on the blockchain. Then, they can send and receive coins by exchanging authenticated messages to update the channel state. Assume in a channel between v_1 and v_2, v_1 contributes c_1 and v_2 contributes c_2 coins. Then v_1 can send up to c_1 coins to P_2 locally. After sending a coins, v_1 can now still send $c_1 - a$ coins but there are more coins available for v_2, mainly $c_2 + a$. We call $c = c_1 + c_2$ the capacity of the channel. The balance in the direction of v_1 to v_2 is the maximum amount that can be sent by v_1 and changes with every local payment. Later on, parties can close the channel by publishing the latest state of the channel on the blockchain. Disputes about the balance are also handled on the blockchain.

A payment channel network (PCN) allows parties (nodes) who do not have a direct channel (edge) to make payments by using the channels in the network. In such a *multi-hop payment* (MHP), the payment between the sender and the receiver is forwarded via a path of connected channels [2, 8, 12, 14, 22, 23]. The Lightning Network [14], a PCN on top of Bitcoin, realizes MHPs via Hash-Time-Locked-Contracts (HTLCs). A HTLC between a payer and a payee is defined wrt. a payment amount a, a hash value h and a timelock t, and implements the following two logical conditions:

1. If a value x is given such that $H(x) = h$ (where H is a hash function), then the payee can claim the a coins.
2. If t has expired, then the payer can claim the a coins.

An HTLC-based MHP works as follows: First, the receiver chooses a random value x shares $h = H(x)$ with the sender. Then, each channel on the path (from the sender to the receiver) locks coins wrt. hash condition h, corresponding payment value (plus fee), and a timelock. The first of two subsequent nodes on the path acts a the payer and the second as the payee for the HTLC. Once the last channel has locked coins, the receiver reveals the value x to the last intermediary in the path and claims the payment amount. The last intermediary shares the same x with the previous party and obtains the corresponding amount of coins in their channel. This continues until the sender has paid the first intermediary.

Routing protocols find a path between a sender and a receiver. In Lightning, the sender decides on the path based on snapshot of the network topology [20]. Apart from the nodes and edges/channels, the snapshot includes the following information per channel: the fee policy, i.e., how to determine the fee claimed by the nodes for a given payment value, the timelock t that the nodes in the channel want to use for a HTLC, and the capacity c. Based on the provided information, the sender determines a least costly path. The cost function used to evaluate the cost of a path differs between Lightning clients [21].

We abstract payment channel functionalities through APIs. Possible realizations of these APIs are discussed in [6], including realizations that can be instantiated over Bitcoin. Different parties can call those APIs to communicate and make payments. We use four APIs for different events:

- *cPay* is called when a party wants to establish an HTLC with a neighbor. Unlike the HTLC of Lightning Network, two hashes are passed to cPay as our payments require both the agreement of the sender and the receiver to be finalized, as detailed below.
- *updateHTLC* is called when a party wants to modify the amount locked in a HTLC.[4] Modifying the payment amount can be necessary to remove redundancy after paths have been found.
- *cPay-unlock* is called when a party provides the two preimages of an HTLC and wants to unlock the funds in this HTLC.
- *refund* is used if the time-lock expires and a party wants to have their locked coins returned.

For simplicity, we denote the calling to an API as $API \longrightarrow F$ where F is a Turning machine that implements those APIs. Appendix A gives the formal descriptions of the APIs.

2.2 State-of-the-art PCN Routing Protocols

To improve the success ratio of multi-hop payments, there have been several works that can be divided into two categories: splitting the payment amount [13, 19] with redundancy [3, 15] and local routing [5, 6, 17]. Here, we briefly explain the most important state-of-the-art protocols.

AMP [13]: The Atomic Multi-Path payment (AMP) protocol allows a sender to forward a payment through multiple paths to improve the success ratio of Lightning's single-path HTLCs. Since the payment is divided into smaller amounts, the probability of having sufficient funds is higher for one channel. However, if any channel involved in the payment does not have a positive balance, the payment still fails.

Boomerang [3]: Boomerang extends the AMP protocol by adding redundancy to payments. Concretely, Boomerang makes k sub-payments of an equal amount b such that $k \cdot b > a$ for payment amount a. Thus, even if some of the sub-payments fail, the amount reaching the receiver may still be sufficient. Boomerang ensures that receivers cannot claim more than amount a, i.e., any funds reaching the receiver that exceed a are returned to the sender. However, for the protocol

[4] We introduce *updateHTLC* API since FUFi allows parties to modify the locked amount in the update phase. The realization of *updateHTLC* can be done by simply revoking the existing HTLC while creating the new one in the same channel update.

to work, all sub-payments have to be of the same size and splitting can only be done by the sender, not by intermediaries.

Spear [15]: Spear, like Boomerang, integrates redundancy into source routing. It is more flexible than Boomerang as it can have sub-payments of varying amounts. Spear uses a modified HTLC to realize the redundancy of payments. Each HTLC has two hash conditions: one chosen by the sender and one chosen by the receiver. Spear still requires that the amount of each sub-payment and the path taken by the payment are fixed by the sender before starting the routing. We use the idea of the two hash conditions in FUFi but only require one hash from the sender for all sub-payments. By using local routing, we enable flexibility and allow parties to adapt the sub-payment amount.

Spider [19]: Spider splits a payment into small sub-payments at the source and forwards them separately. Rather than forwarding these sub-payments at once, a sender can forward them over a longer period of time. During this time, they react to feedback about congestion along the paths used to forward and adjust the rate using a waterfilling algorithm to balance between paths. Communication load and latency are drastically increased and the authors do not provide a concrete method on how to achieve atomicity, i.e., ensure that either all sub-payments are claimed by the receiver or all funds are returned to the sender.

Ethna [5]. Ethna is a local routing protocol that supports payment splitting without atomicity. Intermediaries can split a payment into sub-payments and forward them to different neighbors, and they can decrease the payment amount. In this case, the payment can still be partly completed with a smaller payment size. It is unclear which applications can profit from such partial payments as usually the full payment value is expected for a purchase. Furthermore, Ethna requires smart contract functionality that PCNs over Bitcoin, like the Lightning Network, do not provide.

SpeedyMurmurs [17]. SpeedyMurmurs is a local routing algorithm: It establishes spanning trees in a distributed manner. Intermediaries then locally determine which of their neighbors provide short paths to the receiver based on the spanning tree positions of the neighbors and the receiver. They forward to one neighbor that provides a path to the receiver and has a channel with sufficient balance. If no such channels exist, the routing fails. Splitting at the source is possible but not at intermediaries. The paper only focuses on the routing and does not specify the cryptography used to achieve atomicity.

Splitting Payments Locally [6] Eckey et al. designed a protocol to enable intermediate nodes to split a payment and still achieve atomicity. They show how the protocol can be integrated into a number of routing protocols, including SpeedyMurmurs. For deciding how much funds to give to each neighbor they present two variants: SplitIfNecessary only locally splits payments if there is no single channel that can handle the payment. SplitClosest minimizes the path length and forwards as much as possible to the neighbor that is closest to the receiver, in terms of the path length in the spanning trees. In contrast to original SpeedyMurmurs, the paper provides a cryptographic protocol to guarantee that payments are atomic and intermediaries do not lose funds. However, if only one of the split subpayments fails, the complete payment still fails.

3 Our Protocol

After specifying our system and threat model, we first present the protocol without fees. Afterwards, we show how to integrate fees into the protocol.

3.1 System and Threat model

Let V be the set of nodes in a PCN and $E \subset V \times V$ be the set of channels. We model a PCN as a directed graph $G = (V, E)$ with a capacity function $C \colon V \times E \to \mathbb{R}$. The function C returns the balance in a channel, i.e., $C(v_i, (v_i, v_j))$ gives the available coins of v_i in the channel (v_i, v_j). We assume that there is synchronous communication and the protocol advances in rounds, which correspond to the maximal delay of communication. It takes at Δ rounds to publish information (e.g., disputes) on the chain.

We assume a local internal active adversary, i.e., the adversary can compromise nodes in the network and adapt their behaviour arbitrarily. The attacker cannot observe and control the behaviour of uncompromised parties. They further do not control the network, e.g., they cannot delay messages of uncompromised parties to cause time-locks to expire. The set of corrupted parties is static during the execution of the protocol. The adversary is computationally bounded and hence cannot break cryptographic primitives.

We focus on a rational adversary that aims to gain funds through an attack. Thus, denial-of-service attacks where the adversary refuses to forward payments to undermine the routing without causing other parties to lose funds are not treated here. Such denial-of-service attacks have been evaluated in the context of local routing [24]. We furthermore assume that all parties communicate via secure authenticated channels.

3.2 Security Goals

We now define our security goals. Concretely, we modify the security goals — balance security, bounded loss for the sender, atomicity, and finality — from [6] to include fees. Informally, balance security implies that no honest node, excluding the sender, loses funds during a payment. Bounded loss for the sender means that the sender loses at most the payment value plus any fees paid. Atomicity means that i) the sender only loses funds if they obtain a valid receipt in return and ii) the receiver only provides the sender with a valid receipt if they are paid. Last, finality states that the payment terminates.

The formal definitions of the above properties require us to first define the concept of a receipt formally. Note that in contrast to [6], in line with our "c-Pay" operation, two preimages are used for a receipt. Payments are routed from sender S to receiver R. We assume a EUF-CMA-secure signature algorithm, which is given by a triple of algorithms $(KGen, Sign, Verify)$ for key generation, signing, and verification, and a preimage-resistant hash function H.

Definition 1. *A receipt is defined as*

$$receipt(S, R, a, h_s, h_r) = Sign_{sk_R}(S, R, a, h_s, h_r) \tag{1}$$

where sk_R is the secret key of the receiver R with pk_R being the corresponding public key, a indicates the payment amount and h_s and h_r are two hash values. We define a validation function validate such that

$$validate(receipt(S, R, a, h_s, h_r)) = true \text{ iff}$$

1. *$Verify(pk_R, receipt(S, R, a, h_s, h_r)) = true$*
2. *S provides $receipt(S, R, a, h_s, h_r)$, x_s, x_r, where $H(x_s) = h_s$, $H(x_r) = h_r$.*

In our security definitions, we look at the capacity function C before a payment is executed and the function C' after the execution of the payment. For clarity, we here assume that there are no concurrent payments that affect the function C. Our evaluation considers concurrency. Let furthermore B be the set of honest or benign nodes.

Definition 2 (Balance security for intermediaries).
$\forall\ v_i \in B \setminus \{S\}, \quad \sum_{(v_i,v_j)\in E} C^{'}(v_i, (v_i, v_j)) - \sum_{(v_i,v_j)\in E} C(v_i, (v_i, v_j)) \geq 0$

Definition 3 (Bounded lose for sender). *For a payment of amount a with fee f: if $S \in B$, then $\sum_{(S,v_j)\in E} C(S, (S, v_j)) - \sum_{(S,v_j)\in E} C^{'}(S, (S, v_j)) \leq a + f$*

Definition 4 (Atomicity). *For a payment of amount a:*

1. *if $\sum_{(S,v_j)\in E} C^{'}(S, (S, v_j)) - \sum_{(S,v_j)\in E} C(S, (S, v_j)) < 0 \wedge S \in B$ then $validate(receipt(S, R, size, h_s, h_r)) = true$*
2. *if $validate(receipt(S, R, size, h_s, h_r)) = true \wedge R \in B$ then $\sum_{(R,v_j)\in E} C^{'}(R, (R, v_j)) - \sum_{(R,v_j)\in E} C(R, (R, v_j)) \geq a$*

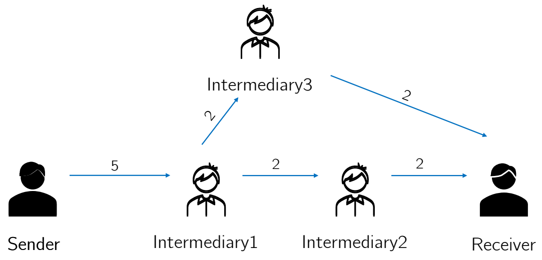
Definition 5 (Finality). *The protocol terminates for all honest parties, i.e., on all locked channels, either "refund" or "cPay-unlock" is eventually executed.*

3.3 Protocol Description

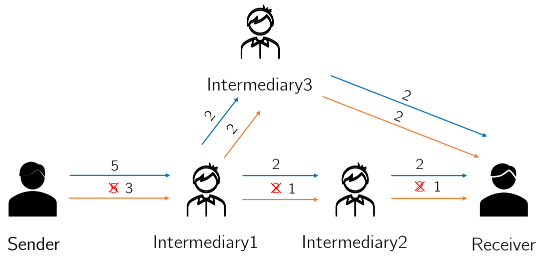
The key idea of FUFi is to forward payments with redundancy and revoke those redundant payments later. For this purpose, we divide the protocol into three phases: Forward, Update, and Finalize. In the forward phase, sender and intermediaries split a payment into several sub-payments and forward them to neighbors until the receiver is reached. In the update phase, intermediaries and the receiver may modify the payment size. Only if the correct payment amount arrives at the receiver, the payment can go through. The update phase is the key difference of FUFi to previous routing algorithms, as it enables the use of redundancy. The last phase, the finalize phase, completes the payment or revokes it. Figure 1 displays an example of the forward and update phase of FUFi, the finalize phase merely executes the red payments that are agreed upon during the update phase. We now go over each of the phases. Detailed pseudocode is given in Appendix B. In the following, we refer to HTLCs for which a node is a payee as incoming HTLCs while outgoing HTLCs are those for which they are a payer.

Initialization Before starting the actual routing, the sender S first uses a random value x_S and sends the hash $h_S = H(x_S)$ to the receiver R. In response, the receiver chooses their own random value x_R and sends the corresponding hash h_R to S. Both preimages are necessary to obtain the funds promised in the HTLCs applied during the finalize phase. Afterwards, S decides on the amount they want to send, which is the payment amount a times a redundancy factor r. Once the amount is fixed, the actual routing of the payment starts.

Forward During the routing, both sender and intermediaries have to decide which of their neighbours they forward sub-payments to. In [6], multiple methods are proposed for splitting the payment such that the combined amount of all sub-payments equals the total payment value. Since we include redundancy in our payment, FUFi can also proceed if the amounts of the sub-payments sum up to less than the total amount, as the total amount includes redundancy.



(a) Forward Phase



(b) Update Phase

Fig. 1: Forward and update phase of FUFi. Sender wants to make a payment of value 3 but initially sends 5 (blue). The first intermediary only forwards 4, split into two payments of 2. After 4 coins arrive at the receiver, one payment path has to be updated to only have 1 coin. Red indicates final payment after update.

We abstract the splitting procedure as follows: A party P — sender or intermediary — calls a function $Route_G(a_P, P, R, aux)$ where a_P is the total amount P wants to forward and aux is any auxiliary information the routing algorithm may require. For instance, Interdimensional SpeedyMurmurs requires the set of nodes that have previously been on the path to prevent routing loops. $Route_G(a_P, P, R, aux)$ returns i) a set of tuples (e_j, a_j), such that e_j indicates a payment channel adjacent to P and a_j the amount to be forwarded via this channel, and ii) an amount a_{Rest} that is not forwarded. We have $a_{Rest} + \sum_j a_j = a_P$, so the payment value is split over adjacent channels with a possible leftover. There are many possible instantiations for $Route_G(a_P, P, R, aux)$, some of which are introduced in Section 4 for our evaluation. The function is mostly identical to [6] but in [6], $a_{Rest} = 0$ or the payment fails.

For each (e_j, a_j), P establishes a HTLC for the channel e_j by calling "cPay" stating that P will pay a_j if they receive preimages for h_S and h_R within a certain time. We will discuss how to choose the time-lock at the end of this section. R keeps track of all sub-payments that arrive. If less than the payment amount a arrives, the payment fails. After a HTLC timeouts, all involved parties call "refund" to have their invested funds returned. Otherwise, if enough funds reach R, the payment proceeds to the update phase.

Update The update phase, which is a new phase that none of the previous protocols have, has to deal with the fact that S in the end only agrees to paying a while initially sending $r \cdot a$. S thus only provides the preimage for h_S, which is necessary to complete the payment, if the HTLCs are updated such that S loses at most a coins.

At the end of the forward phase, let $a + \delta$ coins being locked in R's channels. R has to update the HTLCs such that only a coins are locked. They hence select HTLCs whose values are to be reduced. The HTLCs can also be completely cancelled. Any method of choosing which HTLCs to reduce or cancel can be applied as long as the final results restricts the incoming funds from the payment at R to a. For the HTLCs that should be changed, R calls "updateHTLC" to change the locked amount. One straight-forward method that we use in the evaluation is to simply keep the HTLCs that are established first.

Now, for the sender's outgoing HTLCs to have a combined value of a as well, intermediaries have to update their HTLCs. During the update phase, intermediaries check whether the funds they promise in their outgoing HTLCs, i.e., the funds they pay if the payment succeeds, is lower than the amount they are promised to receive from incoming HTLCs. The two may differ for two reasons: i) a successor updated one of the outgoing HTLCs and ii) they were unable to split the total incoming amount among their neighbors in the forward phase. Independently of the reason, the intermediary updates their incoming HTLCs such that incoming and outgoing funds match. Like for the receiver, the exact protocol used to decide on which HTLCs to update does not matter for the protocol to work. Once the incoming and outgoing funds of all intermediaries match, the HTLCs of S should amount to exactly a because no funds are 'lost' to intermediaries.

The above protocol relies on the fact that intermediaries may not be willing to update the HTLC, an operation that requires the agreement of both payer and payee. In such a case, the funds are not reduced and the payment fails as the sender does not provide their preimage x_S. Intermediaries hence do not gain fees if they refuse to update. Note that they even receive fees if the HTLC is cancelled (e.g., modified to have an amount of 0), as we detail in Section 3.4.

Finalize Once all the HTLCs are updated, the finalize phase completes the payment: S provides the preimage x_S to R. R then provides both x_S and x_R to resolve the HTLCs with their neighbors, which then forward the preimages to their predecessors on the path until all payments have been executed.

Time-locks Now, we can discuss the choice of time-locks. For HTLCs, we need to ensure there is enough time for honest parties to update payments and publish their HTLCs on chain in a dispute. It takes $(\Delta + 1)$ rounds for an intermediary to know that its payment is published on chain by neighbours in the worst case. To get money back, this node needs another $(\Delta+1)$ rounds to publish the HTLC with its predecessors on chain. So, if we want to make sure honest nodes have enough time to publish their HTLCs, the difference of time-locks for subsequent nodes on a path should be at least $2\cdot(\Delta+1)$. Besides the time to publish HTLCs, there is also one round of communication for both establishing the original HTLC and possibly for updating it. Thus, the time-lock set by the sender should be $t_0 + n \cdot (2 + 2 \cdot (\Delta + 1))$ where n is an upper bound on the expected number of nodes on a path, with n depending on the routing algorithm, and t_0 is the current time.

3.4 Fees

In PCNs, nodes are incentivized by fees to participate. Yet, previous local routing protocols disregard fees and how to assign them [6,17]. In the Lightning network, fees are computed in advance and added to the payment amount. However, the computation is only possible as the source decides on the path and knows the fee policies of all nodes. As the paths in local routing are determined by intermediaries, the exact fee cannot be computed in advance. In addition to this known challenge in local routing, FUFi suffers from a second challenge: nodes need to be incentivized to revoke their HTLCs. Such revocation fees need to be paid even if the receiver decides not to use a channel for routing as the nodes would otherwise refuse to revoke and let the payment fail.

We use a relatively simple idea for fees: The sender S decides on an amount f they are willing to pay as fees and then route the amount $a + r + f$ consisting of the actual payment amount, the redundancy, and the maximal fee. Intermediaries learn the amount f and can then decide how much they take as a fee. They forward the remaining fees to the subsequent nodes. If they take a large amount, subsequent nodes may refuse to route the payment due to insufficient compensation, meaning that greedy intermediaries may not receive funds due to the payment failing. Moreover, the more fee they take, the less likely the receiver will choose theirs in the case of receiving more than a amount.

Ersoy et al. [7] analyzed how to determine propagation fees for forwarding transactions in the Bitcoin network. Two requirements defined in their fee policy: i) nodes should not gain more fees by acting maliciously like introducing Sybil nodes, ii) rational nodes should benefit from forwarding. They showed that honest intermediary nodes are incentivized to claim a fraction C of the remaining fee that they receive, and the receiver obtains the remaining part. Here, C is a globally agreed-upon constant. Any remaining fees are taken by the receiver. We apply this fee policy in our evaluation.

Note that the previously discussed fees are only paid upon success. We now discuss the revocation fees. Revocation fees should not exceed the fees for a successful payment to prevent intermediaries from intentionally failing payments. We use a separate transaction with a new HTLC for the same two hash conditions to forward revoke fees. So, two transactions with different temporary secret keys are required for one sub-payment: one for the normal payment and the other one for revoking fees. With this construction, the transaction for revocation fees still exists after the revocation of a normal payment and the node can claim their fee once the preimages are revealed. If a party refuses to revoke, the payment amount exceeds v and the sender S does not provide their preimage, meaning that the party does not gain any fees, not even revocation fees.

We have now specified the phases of our protocol and how it handles fees. We show that FUFi indeed achieves the claimed security goals in Appendix C. In the security proofs, we prove termination separately for sender, intermediaries, and receiver. For balance security, we note that parties never promise to pay more than they are promised to receive and if they pay, they are also paid. Similarly, for bounded loss for the sender, we argue that the sender does not reveal their preimage unless the bounded loss is guaranteed. Atomicity is argued similarly to [6].

4 Evaluation

We simulate FUFi's performance in a simulator by extending a known PCN simulator [5]. Our simulator executes payments concurrently. We adapt routing algorithms from previous work to include an update phase and compute their success ratio.

4.1 Routing Algorithms

The performance of three different routing protocols is compared in our simulations. *SplitClosest* is a local routing protocol with splitting, introduced as a variant of SpeedyMurmurs [6]. It consumes channels' capacities in the order of closeness to the receiver and has been shown to have the best success ratio of all the algorithms evaluated in [6].

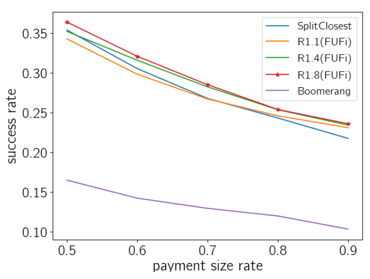
[5] https://github.com/stef-roos/PaymentRouting

A new routing algorithm has been designed for FUFi. Like SplitClosest, it is a variant of SpeedyMurmurs with splitting. It differs from SplitClosest in two aspects: i) it utilizes redundancy and fees as introduced in Section 3 and ii) it uses a waterfilling algorithm for splitting the forwarded amount between neighbors that offer a shorter path to the receiver. Concretely, a node splits the payment value to forward such that the available funds in the channels are as close to equal as possible. As stated in Section 3, parties may have to update incoming HTLCs. They choose the HTLC in order of arrival, i.e., they prioritise older HTLCs and revoke the ones most recently established. To determine the impact of each of the two changes i) and ii), we also consider SplitClosest with redundancy, i.e., only change i), and FUFi without redundancy, i.e., only change ii). As a third algorithm, we use Boomerang. It is a source routing algorithm with redundancy. In our simulations, we use the parameters that achieved the best performance in the original paper (100 sub-payments redundancy 1.33 [3]).

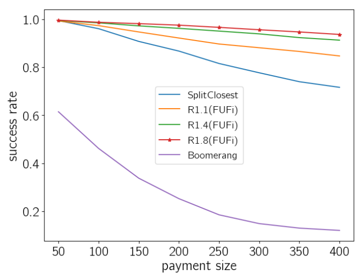
4.2 Setup

We evaluate the different routing algorithms on a snapshot of the Lightning network and a randomly generated scale-free graph. The Lightning Network snapshot is from December 30, 2021. We delete disconnected nodes and channels without capacity. Then, we obtain a graph with 18081 nodes and 76427 channels. To simulate the size of the Lightning Network in the future, we use the Barabasi-Albert (BA) model [1] to generate the topology of network. BA graph is a scale-free model that means only a few nodes have a high degree, similar to Lightning [16]. We use the BA graph to approximate Lightning in the future, with a larger network size, and generate a graph with 25000 nodes where each new node is connected to 6 existing nodes. Most of channels in the Lightning Network snapshot have a low capacity. To simulate the capacity distribution of the Lightning Network, we use an exponential distribution with an average value of 200 to generate channels' balances in the random graph.

In our simulations, the delay of payment forwarding is set to 10 seconds and C is set to 0.4. In [7], C is chosen in relation to the average degree of the nodes, which is 9 for the snapshot. $C = 0.4$ has been identified as a good choice for incentivizing intermediaries to forward if the average degree is 9 or higher. For redundancy in FUFi, we consider 1.1, 1.4, and 1.8. In our first experiment, we change the payment amount and simulate 100000 payments in 1000 seconds. We start from a relatively small payment amount that is smaller than a single channel's capacity on average. To study the impact of payment splitting and redundancy, the payment amount is increased to a larger number that makes payment splitting necessary. For the random graph, the payment size varies from 50 to 400 because the expected capacity of a channel is 200. In the Lightning network, the capacity of channels varies a lot. Thus, using a constant payment amount frequently results in the payment amount exceeding the total capacities of outgoing channels of the sender, meaning that the payment fails in the first step regardless of the protocol. To exclude such unavoidable failures, we instead



(a) Lightning Network



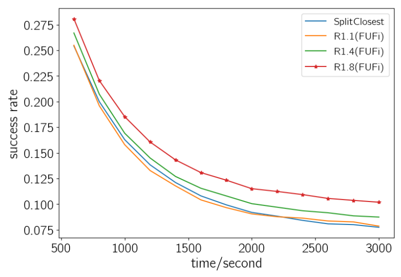
(b) Random Graph (BA)

Fig. 2: Success rate with different payment sizes

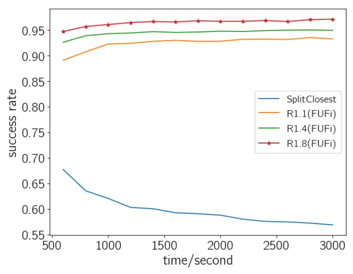
set a payment amount rate $p \in (0,1)$. When a sender starts a payment, the payment amount is p times the combined balance.

In the second experiment, we simulate 300000 payments in 3000 seconds and monitor how the success rate changes over time. The payment amount rate of the Lightning Network is 0.8 while the payment amount is set to 400 in the random graph.

Finally, we have an experiment to measure the influence of redundancy and the waterfilling algorithm separately. In this experiment, the redundancy is set to 1.4, payment amount rate of the Lightning Network is 0.8, payment amount of the random graph is 400, and 100000 payments are simulated in 1000 seconds.



(a) Lightning Network



(b) Random Graph (BA)

Fig. 3: The change of success rate over time

4.3 Simulation Results

Figure 2a and Figure 2b show the success ratio of different payment amounts. In the Lightning Network, FUFi with 1.8 redundancy improves the success ratio of SplitClosest by about 10%. Boomerang is outperformed by other protocols because it uses a source routing algorithm that can not adapt to the changes of channels' capacities. This result shows the effectiveness of combining local splitting and redundancy. The result of the random graph is similar to the Lightning

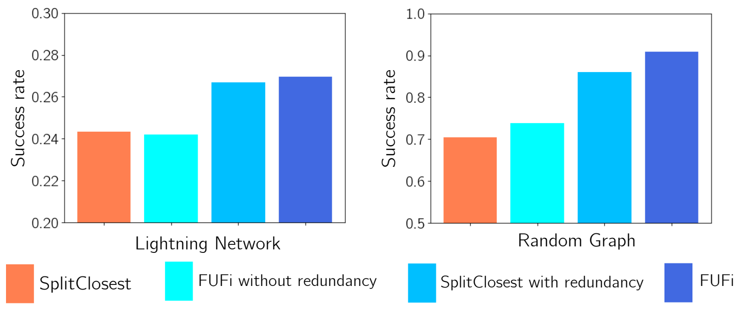


Fig. 4: Impact of redundancy in comparison to varying splitting protocol.

Network. However, the differences in performance between the protocols are more pronounced, which can be explained by the better connectivity of the random graph. Nodes have more choices to forward and hence the payment is less likely to fail.

Figure 3a shows the success rate over time, which decreases over time in the Lightning Network. However, FUFi's success rate is retained at a higher level than SplitClosest because of the use of waterfilling. SplitClosest tends to use up all the funds in channels to have short paths whereas waterfilling tries to balance the funds. For the random graph, there is no negative impact over time for waterfilling. The higher number of paths enables nodes to better balance their channels and hence avoid depletion, i.e., channels with no or hardly any funds on one direction. For SplitClosest, there still is a negative impact as it does actively deplete channels.

Figure 4 compares the impact of our two changes, with the result that redundancy has more impact than waterfilling, which has no significant impact on Lightning and a smaller impact on the random graph.

5 Conclusion

We have introduced FUFi, which increases the performance of local routing by about 10% and is the first local routing protocol for PCNs that integrates fees.

Yet, we mainly disregarded the aspect of privacy: Hiding the identity of sender and receiver as well as channel capacities are important privacy properties for PCNs [11]. While Lightning was initially thought to be private, it has been shown that it is vulnerable to multiple attacks [10, 18]. The exact attacks are not possible for local routing; yet, it is likely that FUFi is vulnerable to similar attacks. In future work, we thus aim to investigate which privacy attacks are possible, how they affect FUFi, and how to defend against the attacks.

Acknowledgments

This research was partially funded by Ripple's University Blockchain Research Initiative.

References

1. Albert, R., Barabási, A.L.: Statistical mechanics of complex networks. Rev. Mod. Phys. 74, 47–97 (Jan 2002), https://link.aps.org/doi/10.1103/RevModPhys.74.47
2. Aumayr, L., Moreno-Sanchez, P., Kate, A., Maffei, M.: Blitz: Secure multi-hop payments without two-phase commits. In: Bailey, M., Greenstadt, R. (eds.) 30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021. pp. 4043–4060. USENIX Association (2021), https://www.usenix.org/conference/usenixsecurity21/presentation/aumayr
3. Bagaria, V.K., Neu, J., Tse, D.: Boomerang: Redundancy improves latency and throughput in payment-channel networks. In: Bonneau, J., Heninger, N. (eds.) Financial Cryptography and Data Security - 24th International Conference, FC 2020, Kota Kinabalu, Malaysia, February 10-14, 2020 Revised Selected Papers. Lecture Notes in Computer Science, vol. 12059, pp. 304–324. Springer (2020), https://doi.org/10.1007/978-3-030-51280-4_17
4. Decker, C., Wattenhofer, R.: A fast and scalable payment network with bitcoin duplex micropayment channels. In: Symposium on Self-Stabilizing Systems. pp. 3–18. Springer (2015)
5. Dziembowski, S., Kedzior, P.: Ethna: Channel network with dynamic internal payment splitting. IACR Cryptol. ePrint Arch. p. 166 (2020), https://eprint.iacr.org/2020/166
6. Eckey, L., Faust, S., Hostáková, K., Roos, S.: Splitting payments locally while routing interdimensionally. IACR Cryptol. ePrint Arch. p. 555 (2020), https://eprint.iacr.org/2020/555
7. Ersoy, O., Ren, Z., Erkin, Z., Lagendijk, R.L.: Transaction propagation on permissionless blockchains: Incentive and routing mechanisms. In: Crypto Valley Conference on Blockchain Technology, CVCBT 2018, Zug, Switzerland, June 20-22, 2018. pp. 20–30. IEEE (2018), https://doi.org/10.1109/CVCBT.2018.00008
8. Green, M., Miers, I.: Bolt: Anonymous payment channels for decentralized currencies. In: Thuraisingham, B., Evans, D., Malkin, T., Xu, D. (eds.) Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017. pp. 473–489. ACM (2017), https://doi.org/10.1145/3133956.3134093
9. Gudgeon, L., Moreno-Sanchez, P., Roos, S., McCorry, P., Gervais, A.: Sok: Layer-two blockchain protocols. In: Bonneau, J., Heninger, N. (eds.) Financial Cryptography and Data Security - 24th International Conference, FC 2020, Kota Kinabalu, Malaysia, February 10-14, 2020 Revised Selected Papers. Lecture Notes in Computer Science, vol. 12059, pp. 201–226. Springer (2020), https://doi.org/10.1007/978-3-030-51280-4_12
10. Herrera-Joancomartí, J., Navarro-Arribas, G., Ranchal-Pedrosa, A., Pérez-Solà, C., Garcia-Alfaro, J.: On the difficulty of hiding the balance of lightning network channels. In: Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security. pp. 602–612 (2019)
11. Malavolta, G., Moreno-Sanchez, P., Kate, A., Maffei, M., Ravi, S.: Concurrency and privacy with payment-channel networks. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 455–471 (2017)
12. Malavolta, G., Moreno-Sanchez, P., Schneidewind, C., Kate, A., Maffei, M.: Anonymous multi-hop locks for blockchain scalability and interoperability. In: 26th Annual Network and Distributed System Security Symposium, NDSS

2019, San Diego, California, USA, February 24-27, 2019. The Internet Society (2019), https://www.ndss-symposium.org/ndss-paper/anonymous-multi-hop-locks-for-blockchain-scalability-and-interoperability/

13. Osuntokun, O.: Amp: Atomic multi-path payments over lightning (2018-02-06)
14. Poon, J., Dryja, T.: The bitcoin lightning network: Scalable off-chain instant payments (2016), https://www.bitcoinlightning.com/wp-content/uploads/2018/03/lightning-network-paper.pdf
15. Rahimpour, S., Khabbazian, M.: Spear: fast multi-path payment with redundancy. In: Baldimtsi, F., Roughgarden, T. (eds.) AFT '21: 3rd ACM Conference on Advances in Financial Technologies, Arlington, Virginia, USA, September 26 - 28, 2021. pp. 183–191. ACM (2021), https://doi.org/10.1145/3479722.3480997
16. Rohrer, E., Malliaris, J., Tschorsch, F.: Discharged payment channels: Quantifying the lightning network's resilience to topology-based attacks. In: 2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW). pp. 347–356. IEEE (2019)
17. Roos, S., Moreno-Sanchez, P., Kate, A., Goldberg, I.: Settling payments fast and private: Efficient decentralized routing for path-based transactions. In: 25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018. The Internet Society (2018), http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_09-3_Roos_paper.pdf
18. Sharma, P.K., Gosain, D., Diaz, C.: On the anonymity of peer-to-peer network anonymity schemes used by cryptocurrencies. arXiv preprint arXiv:2201.11860 (2022)
19. Sivaraman, V., Venkatakrishnan, S.B., Alizadeh, M., Fanti, G., Viswanath, P.: Routing cryptocurrency with the spider network. In: Proceedings of the 17th ACM Workshop on Hot Topics in Networks, HotNets 2018, Redmond, WA, USA, November 15-16, 2018. pp. 29–35. ACM (2018), https://doi.org/10.1145/3286062.3286067
20. Sunshine, C.A.: Source routing in computer networks. Comput. Commun. Rev. 7(1), 29–33 (1977), https://doi.org/10.1145/1024853.1024855
21. Tochner, S., Zohar, A., Schmid, S.: Route hijacking and dos in off-chain networks. In: Proceedings of the 2nd ACM Conference on Advances in Financial Technologies. pp. 228–240 (2020)
22. Tripathy, S., Mohanty, S.K.: MAPPCN: multi-hop anonymous and privacy-preserving payment channel network. In: Bernhard, M., Bracciali, A., Camp, L.J., Matsuo, S., Maurushat, A., Rønne, P.B., Sala, M. (eds.) Financial Cryptography and Data Security - FC 2020 International Workshops, AsiaUSEC, CoDeFi, VOTING, and WTSC, Kota Kinabalu, Malaysia, February 14, 2020, Revised Selected Papers. Lecture Notes in Computer Science, vol. 12063, pp. 481–495. Springer (2020), https://doi.org/10.1007/978-3-030-54455-3_34
23. Tsabary, I., Yechieli, M., Manuskin, A., Eyal, I.: MAD-HTLC: because HTLC is crazy-cheap to attack. In: 42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021. pp. 1230–1248. IEEE (2021), https://doi.org/10.1109/SP40001.2021.00080
24. Weintraub, B., Nita-Rotaru, C., Roos, S.: Structural attacks on local routing in payment channel networks. In: 2021 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW). pp. 367–379. IEEE (2021)

A APIs

Meaning of parameters

pid: payment id
e: corresponding payment channel's id
$size$: payment size
fee: routing fees
h_S, h_R: hashes for HTLC
x_S, x_R: preimages for HTLC
T: time lock
R: the receiver's id

APIs

$(cPay, pid, e, size, fee, h_S, h_R, T, R) \longrightarrow F$

If the caller has enough funds in channel e, construct an HTLC among channel e. The amount of locked funds equals $size + fee$, the timelock is T, and hashes of preimages are h_S and h_R. Afterwards, sends $(cPaid, pid, e, size, fee, h_S, h_R, T, R)$ to the other party in channel e.

$(updateHTLC, pid, size, fee) \longrightarrow F$

Check whether the sender has an unfinished payment whose ID is pid. If so, compare this payment's $size$ and fee with the new $size$ and fee. If the new $size$ and fee are smaller, invalidate the old HTLC and make a new one with the new $size$ and fee.

$(cPay\text{–}unlock, pid, x_S, x_R) \longleftarrow F$

If x_S and x_R are valid preimages for payment pid's hashes, unlock the locked funds and assign it to the payee.

$(refund, pid) \longleftarrow F$

If current timestamp is larger than payment pid's timelock, unlock the locked funds and return it to the payer.

B Pseudocode

Meaning of variables and functions

$a \longrightarrow b$: send message a to b
$x \longleftarrow \mathbb{Z}_P$: sample a number in the number field $\mathbb{Z}_P$ and assign it to variable x
$id \longleftarrow \{0,1\}^*$: chose a random id
F: calling to APIs
S: sender's ID
R: receiver's ID
G: the topology of PCN
C_i: node i's capacity function, can output this node's capacity in each channel
$x_S \ and \ x_R$: sampled preimages for HTLCs
$H(x)$: a hash function
$Sign_k(m)$: sign the message m using key k
$Vrfy_k(m, Sig)$: verify the signature Sig of message m using key k
T: the latest time to complete a payment
$C_{redundant}(\geq 1)$: redundancy of payments
C_{revoke}: the ratio of routing fees to revoke fees
$fee^{routing}$: fees if the whole payment succeeds
fee^{revoke}: fees if the whole payment is revoked
Δ: time needed to publish a payment on chain
$[k]$: Natural numbers from 1 to k
aux: auxiliary information for routing algorithms, different routing algorithms may require different auxiliary information
$Route(amount, I, R, aux)$: For the payment from I to R with size $size$ and auxiliary information aux, decide how to split it and what is the next node
$Fee(totalfees)$: decide fees for current node based on a fee policy
$Reschedule_G(candidates, amount)$: choose some sub-payments from candidates to let the sum of them equal amount, the size of those sub-payments also can be adjusted

In the initialization phase, different parties obtain necessary information and initialize their variables.

Initialization

```
Sender:
  1: $Receive\ (G, S, R, size, fee, C_S)$
  2: $out := \emptyset$                          ▷ The set of all sub-payments
  3: $sum_{pay} := 0$                            ▷ Sum of updated payment size
  4: $sum_{fee} := 0$                            ▷ Sum of updated fees
Receiver:
```

```
1: $Receive(G, S, R, size, fee, C_R)$
2: $in := \emptyset$                      ▷ The set of received sub-payments
3: $candidate := \emptyset$
4: $sum_{pay} := 0$                       ▷ Sum of received sub-payments
5: $finish := 0$                          ▷ Are all candidates decided
```

Intermediaries:

```
1: $Receive(G, C_I)$
2: $fw := \emptyset$                      ▷ The set of forwarded sub-payments
3: $cnt := \emptyset$                     ▷ Count sub-payments for each received payment
4: $sum_{pay} := \emptyset$               ▷ Sum of updated sub-payments' sizes
5: $sum_{fee} := \emptyset$               ▷ Sum of updated sub-payments' fees
6: $old_{pay} := \emptyset$               ▷ The initial payment size of each payment
7: $oldFee^{routing} := \emptyset$        ▷ The initial routing fee of each payment
8: $oldFee^{revoke} := \emptyset$         ▷ The initial revoke fee of each payment
```

In step 1, the sender sends its hash to the receiver. Upon receiving the sender's hash, the receiver checks the signature of this hash and sends back its hash if the signature is valid. Upon receiving the receiver's hash, the sender checks the receiver's signature. If the signature of the receiver is valid, the sender decides how to forward its payment using the "Route" function in line 3 of step 3. In lines 4 to 9, the sender forwards and stores each sub-payment. In step 4, F checks whether the party who calls "cPay" API has enough capacity. If so, F maintains this party's capacity accordingly and sends a "cPaid" message to the next node. After a "cPaid" message is received, the intermediary calculates the fees and timelock in lines 2 to 4 of step 5. From line 5 to line 12, it routes and forwards the received payment like the sender. In step 6, the receiver receives a "cPaid" message at first. In line 2, the receiver calculates the sum of received sub-payments. In line 3, the receiver stores the received sub-payment into the candidate set. If the amount of received sub-payments is larger than the payment size, the receiver chooses sub-payments using the "Reschedule" function in line 7. From line 6 to line 10, the receiver updates rescheduled sub-payments by "updateHTLC" API. In lines 13 and 14, the receiver sets all later payments' size to 0.

Forward phase

(1) Sender initiates a payment:
In round t_0, the sender samples a preimage and calculate the hash value of it. Then, this hash is signed and sent to the receiver.

```
1: $x_S \longleftarrow \mathbb{Z}_P, h_S := H(x_S)$
2: Send $(h_S, Sign_{sk_S}(h_S))$ to the receiver
```

(2) Receiver upon receiving (h_S, Sig_{h_S}):
If the received signature is valid, the receiver sends a signed message back.

```
1: if $Sig_{h_S}$ is a valid signature of the sender then
2:     $x_R \longleftarrow \mathbb{Z}_P, h_R := H(x_R)$
3:     $Sig = Sign_{sk_R}(S, R, size, h_S, h_R)$
4:     Send $(init, h_S, h_R, Sig)$ to the sender
5: else
6:     $Terminate$
7: end if
```

(3) Sender upon receiving $(init, h_S, h_R, Sig)$**:**
In round $t_0 + 2$, the sender checks the validity of the receiver's message. If it is valid, the sender tries to route and forward its payment.

```
1: if $Sig$ is a valid signature for $(S, R, size, h_S, h_R)$ then
2:     $T := t_0 + 2 + |V|(2 + 2 \cdot (\Delta + 1))$
3:     $(e_j, size_j)_{j \in [k]} = Route_G(size \cdot C_{redundant}, S, R, aux)$
4:     for $j \in [k]$ do
5:         $fee_j = size_j / size \cdot fee$
6:         $pid_j \longleftarrow \{0,1\}^*$
7:         $out := out \cup \{(e_j, size_j, fee_j, pid_j)\}$
8:         $(cPay, pid_j, e_j, size_j, fee_j, fee_j \cdot C_{revoke}, h_S, h_R, T, R) \longrightarrow F$
9:     end for
10: else
11:     $Terminate$
12: end if
```

(4) F receives $(cPay, pid, e, size, fee_{routing}, fee_{revoke}, h_S, h_R, T, R)$ **from** I **or** S**:**

```
1: if $I$'s capacity in channel $e \geq size + fee$ then
2:     decrease $I$'s capacity by $size + fee$
3:     send $(cPaid, pid, e, size, fee^{routing}, fee^{revoke}, h_S, h_R, T, R)$ to the
   other party of channel $e$
4: else
5:     $Terminate$
6: end if
```

(5) Intermediary upon receiving $(cPaid, pid, e, size, fee^{routing}, fee^{revoke}, h_S, h_R, T, R)$**:**

```
1: if $current\ round \leq t_0 + 1 + |V|$ then
2:     $T := T - 2(\Delta + 1)$
3:     $cnt[T] := 0, sum_{pay}[T] := 0, sum_{fee}[T] := 0, old_{pay}[T] := size$
4:     $oldFee^{routing}[T] := fee^{routing}, oldFee^{revoke}[T] := fee^{revoke}$
5:     $(e_j, size_j)_{j \in [k]} = Route_G(size, I, R, aux)$
6:     for $j \in [k]$ do
7:         $fee_j^{routing} = Fee(size_j / size \cdot fee^{routing})$
8:         $fee_j^{revoke} = Fee(size_j / size \cdot fee^{revoke})$
9:         $pid_j \longleftarrow \{0,1\}^*$
```

```
10:         $fw[T] := fw[T] \cup \{(e_j, size_j, fee_j^{routing}, fee_j^{revoke}, pid_j, pid)\}$
11:         $cnt[T] := cnt[T] + 1$
12:         $(cPay, pid_j, e_j, size_j, fee_j^{routing}, fee_j^{revoke}, h_S, h_R, T, R) \longrightarrow F$
13:     end for
14: else
15:     $Terminate$
16: end if
```

(6) Receiver upon receiving $(cPaid, pid, e, size, fee^{routing}, fee^{revoke}, h_S, h_R, T, R)$**:**
The receiver calculates the sum of received sub-payments. If the sum reaches the payment size, the receiver tries to reschedule received sub-payments.

```
 1: if $finish = 0$ then
 2:     $sum_{pay} := sum_{pay} + size$
 3:     $candidate = candidate \cup (pid, size, fee^{routing}, fee^{revoke})$
 4:     if $sum_{pay} >= size$ then
 5:         $(pid_j, size_j, fee_j)_{j \in [k]} = Reschedule_G(candidate)$
 6:         for $j \in [k]$ do
 7:             $(updateHTLC, pid_j, size_j, fee_j) \longrightarrow F$
 8:             $in = in \cup \{pid_j\}$
 9:             $finish = 1$
10:         end for
11:     end if
12: else
13:     $(updateHTLC, pid, 0, fee^{revoke}) \longrightarrow F$
14:     $in = in \cup \{pid\}$
15: end if
```

In the update phase, F receives participants' update requests first. It checks whether the update request is sent from a payment's payee. If so, F changes the corresponding party's capacity and sends an "updatedHTLC" message to the found payment's payer. Upon receiving an "updatedHTLC" message, the intermediary I finds the corresponding split sub-payment from the fw set. In line 2 of step 8, I determines whether the updated payment size and fees are correct. If so, it updates the stored sub-payment with the new payment size and fees. In lines 8 and 9, I sums split sub-payments' fees and payment size for each corresponding unsplit sub-payment. If all split sub-payments are updated, I updates the unsplit sub-payment with recalculated payment size and fees in lines 11 and 12. In line 13, I requests the former node to update its payment using "updateHTLC" API. In step 9, the sender receives a "updatedHTLC" message from F. From line 2 to line 4, the sender update stored sub-payments and sums the updated payment size and fees. If the updated payment size and fees are correct, it sends its preimage to the receiver. Otherwise, the sender terminates immediately and the forwarded payment fails.

Update phase

(7) F upon receiving $(updateHTLC, pid, size_{new}, fee_{new})$ **from** I **or** R**:**

```
1: if $I$ has a payment with $pid$ then
2:     increase the capacity of the former node based on $size_{new}$ and $fee_{new}$
3:     send $(updatedHTLC, pid_0, size_{new}, fee_{new})$ to the former node
4: end if
```

(8) Intermediary upon receiving $(updatedHTLC, pid_0, size_{new}, fee_{new})$**:**

Intermediate node needs to make sure the updated payment size and fees are correct. If so, it waits until all sub-payments are updated. Afterwards, it tries to update the former payment

```
1: $Let\ \ x_j := (e_j, size_j, fee_j^{routing}, fee_j^{revoke}, pid_j, pid) \in fw[T] \quad s.t.\ pid_j = pid_0$
2: if $size_{new} > size_j \vee fee_{new} > size_{new}/size_j \cdot fee_j^{routing} + (1 - size_{new}/size_j) \cdot fee_j^{revoke}$ then
3:     $Terminate$
4: else
5:     $fw[T] := fw[T] \backslash \{x_j\}$
6:     $fw[T] := fw[T] \cup \{(e_j, size_{new}, fee_{new}, 0, pid_j, pid)\}$
7:     $cnt[T] := cnt[T] - 1$
8:     $sum_{pay}[T] := sum_{pay}[T] + size_{new}$
9:     $sum_{fee}[T] := sum_{fee}[T] + fee_{new}$
10:    if $cnt[T] = 0$ then
11:        $sum_{fee}[T] += sum_{pay}[T]/old_{pay}[T] \cdot oldFee^{routing}[T]$
12:        $sum_{fee}[T] += (1 - sum_{pay}[T]/old_{pay}[T]) \cdot oldFee^{revoke}[T]$
13:        $(updateHTLC, pid, sum_{pay}[T], sum_{fee}[T]) \longrightarrow F$
14:    end if
15: end if
```

(9) Sender upon receiving $(updatedHTLC, pid_0, size_{new}, fee_{new})$**:**
When the sender receives update requests, it counts the sum of those updated sub-payments. If the sum equals the payment size plus fees, it sends its preimage to the receiver.

```
1: $Let\ x_j := (e_j, size_j, fee_j, pid_j) \in out[T] \quad s.t.\ pid_j = pid_0$
2: $out := out \backslash \{x_j\}$
3: $out := out \cup \{(e_j, size_{new}, fee_{new}, pid_j)\}$
4: $sum_{pay} += size_{new}, sum_{fee} += fee_{new}, cnt -= 1$
5: if $sum_{pay} = size \wedge cnt = 0$ then
6:     $fee = fee/C_{redundant} + fee \cdot C_{revoke} \cdot (redundant - 1)/redundant$
7:     if $sum_{pay} > size \vee sum_{fee} > fee$ then
8:         $Terminate$
9:     else
```

```
10:         Send $x_S$ to the receiver
11:     end if
12: end if
```

In step 10, the receiver checks whether the received preimage is correct. If so, it unlocks its funds by calling "cPay-unlock" with the preimages it holds. In step 11, F receives preimages from the receiver or an intermediary. If the preimages are correct, F changes the corresponding party's capacity and sends a "cPay-unlocked" message to the sender or an intermediary. Afterwards, intermediaries use the preimages in "cPay-unlocked" messages to unlock their funds in line 2 of step 12. Next, the sender receives "cPay-unlocked" messages from F. It checks whether all sub-payments are completed. If so, it returns the received preimages.

Finalize phase

(10) Receiver upon receiving x_S:
The receiver uses preimages to unlock funds.

```
1: if $H(x_S) = h_S$ then
2:     for $pid \in in$ do
3:         $(cPay\text{–}unlock, pid, x_S, x_R) \longrightarrow F$
4:     end for
5:     $wait\ for\ \Delta + 1\ rounds\ to\ return\ (x_S, x_R)$
6: end if
```

(11) F upon receiving $(cPay\text{–}unlock, pid, x_S, x_R)$ from I or R:

```
1: if $x_S$ and $x_R$ are valid preimages for the payment with pid then
2:     increase $I$'s capacity by $size + fee$
3:     send $(cPay\text{–}unlocked, pid, x_S, x_R)$ to the former node of $I$
4: end if
```

(12) Intermediary upon receiving $(cPay\text{–}unlocked, pid_0, x_S, x_R)$:

```
1: $Let\ x_j := (e_j, size_j, fee_j, fee_j^{revoke}, pid_j, pid) \in fw[T] \quad s.t.\ pid_j = pid_0$
2: $(cPay\text{–}unlock, pid, x_S, x_R) \longrightarrow F$
3: $fw[T] := fw[T] \backslash \{x_j\}$
4: if $\forall\ T, fw[T] = \emptyset$ then
5:     $wait\ for\ \Delta + 1\ rounds\ to\ return\ (x_S, x_R)$
6: end if
```

(13) Sender upon receiving $(cPay\text{–}unlocked, pid_0, x_S, x_R)$:

```
1: $Let\ x_j := (e_j, size_j, fee_j, fee_j^{revoke}, pid_j, pid) \in fw[T] \quad s.t.\ pid_j = pid_0$
2: $out := out \backslash \{x_j\}$
3: if $out = \emptyset$ then
4:     $return\ (x_S, x_R)$
5: end if
```

If some parties do not behave as expected, the error handling functions below make sure that honest parties always terminate and never lose money. For the sender, if there are still some unfinished payments after the timelock, it unlocks its funds by "refund" and terminates after $\Delta + 1$ rounds. For each "refund" request, F checks the timelock of the corresponding payment. If the corresponding payment expires, F increases the capacity of the refund initiator. If the receiver does not terminate in $t_0 + 2 + 2|V|$ rounds, it simply terminates in this round. In every round, the intermediary checks whether there are expired payments and sends a "refund" request for each expired payment. When the forward set fw is empty, the intermediary terminates.

Error handling

```
Sender in round $T$:
1: for $pid \in out$ do
2:     $(refund, pid) \longrightarrow F$
3: end for
4: $wait\ for\ \Delta + 1\ rounds\ to\ Terminate$
F upon receiving $(refund, pid)$ from $I$ or $S$:
1: if the payment with $pid$ expires then
2:     increase $I$'s capacity by $size + fee$
3: end if
Receiver in round $t_0 + 2 + 2|V|$:
1: $Terminate$
Intermediary in every round:
1: if $fw[now] \neq \emptyset$ then
2:     for $(e_j, v_j, fee_j, pid_0, pid) \in fw[now]$ do
3:         $(refund, pid_0) \longrightarrow F$
4:     end for
5:     if $fw = \emptyset$ then
6:         $wait\ for\ 2(\Delta + 1)\ rounds\ to\ Terminate$
7:     end if
8: end if
```

C Security Analysis

Our security analysis follows the proofs in [6]. As we extended local routing by allowing for fees and redundancy, these changes also need to be reflected in the proofs. In particular, we add a modify phase that is not present in the original protocol and thus requires new proofs.

C.1 Termination

Lemma 1. *Finality is satisfied in FUFi.*

Proof. In the modify phase, the sender sums up the updated payment amount. It terminates immediately if the amount of updated payments is incorrect. In the complete phase, the sender terminates when all sub-payments are finished. If some parties behave incorrectly, the sender's error handling function also ensures that it terminates in round $T+\Delta+1$ where T is the time-lock set by the sender.

For an honest receiver, if the preimage of the sender is received, it terminates after $\Delta+1$ rounds from the current round. Otherwise, its error handling function ensures that it terminates in round $t_0+2+2|V|$ where t_0 is the start round of payment and $|V|$ is the number of nodes in a PCN.

An intermediary stores all sub-payments in the set fw. Upon the receiving of a "cPay-unlocked" message, it deletes the corresponding element in fw. When $fw=\emptyset$ which means every sub-payment is completed, it terminates after $\Delta+1$ rounds from the current round. If some sub-payments expire which means it is impossible to let $fw=\emptyset$, it terminates after $2(\Delta+1)$ rounds from the current round.

Since all honest parties terminate in finite rounds, we can conclude that FUFi satisfies the definition of finality. □

C.2 Bounded loss

Lemma 2. *Bounded loss for the sender is satisfied in FUFi.*

Proof. Based on the definition of APIs, a party loses money only when a valid cPay-unlock message is sent. A cPay-unlock message includes the ID of a payment. APIs can look up the payment size and fees of payment by this ID. If this message is valid, the capacity of the corresponding party decreases by $payment\ size + fees$. A valid cPay-unlock message requires correct preimages for h_S and h_R. In FUFi, only the sender knows the preimage of h_S and it only sends this preimage when the sum of payment size and fees of all sub-payments equals or is smaller than the desired value. Thus, an honest sender never loses money more than $payment\ size + fees$ in FUFi. □

C.3 Balance security of intermediate nodes

Lemma 3. *FUFi satisfies the balance security for intermediate nodes.*

Proof. For an honest intermediate node, the definition of the routing function ensures that the sum of $size$ and fee in cPay messages it sent is always smaller than the sum of $size$ and fee in cPaid messages it received. In the protocol without unlinkability, the same hashes are used for all sub-payments. An intermediate node can use later nodes' preimages directly. In the protocol with unlinkability, we use different hashes for different sub-payments. An honest intermediary samples a random number x in $\mathbb{Z}_P$ for each sub-payment. The new hash value h equals the sum of the original hash value and the hash of x. Because of the additive homomorphism, $Hash(x_0+x)=Hash(x_0)+Hash(x)$ holds where x_0 is the preimage for the original hash value. So, an intermediary can obtain x_0 by simple arithmetic when later nodes reveal the preimage of

$Hash(x_0) + Hash(x)$. To ensure a node always has enough time to claim coins if it pays, a node always subtracts $2 \cdot (\Delta + 1))$ from the next HTLC's time-lock. $\Delta + 1$ is the maximum time to know it has lost some money. The other $\Delta + 1$ is the maximum time to publish an HTLC on-chain. Thus, if an intermediate node pays, it learns preimages for payment and can unlock more funds using received preimages.

In the modify phase, intermediate nodes calculate the sum of $size$ and $fees$ of all updated sub-payments. Afterward, it adds fees for itself. Then, it calls updateHTLC API to update the $size$ and $fees$ of the former payment. Such a procedure ensures that the sum of updated HTLCs with later nodes is still smaller than the sum of HTLCs with former nodes. Because the time-lock T and conditions are not changed in the modify phase, we can conclude that an honest intermediate node never loses coins in the modify and complete phase too.

Because intermediate nodes never lose money during the whole process, we can conclude that the balance security of intermediate nodes is ensured in FUFi. □

C.4 Atomicity

Lemma 4. *FUFi satisfies atomicity of definition 4.*

Proof. Atomicity means if an honest sender loses any coins, it can get a valid receipt that shows it has paid the promised money. In FUFi, the capacity of the sender decreases only when other parties call cPay-unlock API to obtain money from the sender. If this call is valid, API sends $(cPay\text{–}unlocked, pid_0, x_S, x_R)$ to the sender. So, a sender learns preimages of payment if it loses money because of this payment. The initialization process of payments ensures that an honest sender locks its funds only after it has received a signed receipt. If a sender loses money, it knows the preimages of the signed receipt. According to equation 4.1, the receipt kept by the sender is valid if it loses any money. So, atomicity is satisfied in FUFi. □