Blitz: Secure Multi-Hop Payments Without Two-Phase Commits[*]

Lukas Aumayr
TU Wien
lukas.aumayr@tuwien.ac.at

Pedro Moreno-Sanchez
IMDEA Software Institute
pedro.moreno@imdea.org

Aniket Kate
Purdue University
aniket@purdue.edu

Matteo Maffei
TU Wien
matteo.maffei@tuwien.ac.at

Abstract

Payment-channel networks (PCN) are the most prominent approach to tackle the scalability issues of current permissionless blockchains. A PCN reduces the load on-chain by allowing arbitrarily many off-chain multi-hop payments (MHPs) between any two users connected through a path of payment channels. Unfortunately, current MHP protocols are far from satisfactory. One-round MHPs (e.g., Interledger) are insecure as a malicious intermediary can steal the payment funds. Two-round MHPs (e.g., Lightning Network (LN)) follow the 2-phase-commit paradigm as in databases to overcome this issue. However, when tied with economical incentives, 2-phase-commit brings other security threats (i.e., wormhole attacks), staggered collateral (i.e., funds are locked for a time proportional to the payment path length) and dependency on specific scripting language functionality (e.g., Hash Time-Lock Contracts) that hinders a wider deployment in practice.

We present Blitz, a novel MHP protocol that demonstrates for the first time that we can achieve the best of the two worlds: a single round MHP where no malicious intermediary can steal coins. Moreover, Blitz provides the same privacy for sender and receiver as current MHP protocols do, is not prone to the wormhole attack and requires only constant collateral. Additionally, we construct MHPs using only digital signatures and a timelock functionality, both available at the core of virtually every cryptocurrency today. We provide the cryptographic details of Blitz and we formally prove its security. Furthermore, our experimental evaluation on a LN snapshot shows that (i) staggered collateral in LN leads to in between 4x and 33x more unsuccessful payments than the constant collateral in Blitz; (ii) Blitz reduces the size of the payment contract by 26%; and (iii) Blitz prevents up to 0.3 BTC (3397 USD in October 2020) in fees being stolen over a three day period as it avoids wormhole attacks by design.

[*]This is the full version of the work accepted at 30th USENIX Security Symposium 2021.

1 Introduction

Permissonless cryptocurrencies such as Bitcoin enable secure payments in a decentralized, trustless environment. Transactions are verified through a consensus mechanism and all valid transactions are recorded in a public, distributed ledger, often called blockchain. This approach has inherent scalability issues and fails to meet the growing user demands: In Bitcoin, the transaction throughput is technically limited to tens of transactions per second and the transaction confirmation time is around an hour. In contrast, more centralized payment networks such as the Visa credit card network, can handle peaks of 47,000 transaction per second.

This scalability issue is an open problem in industry and academia alike [16, 33]. Among the approaches proposed so far, payment channels (PC) have emerged as one of the most promising solutions; implementations thereof are already widely used in practice, e.g., the Lightning Network (LN) [24] in Bitcoin. A PC enables two users to securely perform an arbitrary amount of instantaneous transactions between each other, while burdening the blockchain with merely two transactions, (i) for opening and (ii) for closing. In particular, following the unspent transaction output (UTXO) model, two users open a PC by locking some coins in a shared multi-signature output. By exchanging signed transactions that spend from the shared output in a peer-to-peer fashion, they can capture and redistribute their balances off-chain. Either one of the two users can terminate the PC by publishing the latest of these signed transactions on the blockchain.

As creating PCs requires locking up some coins, it is economically infeasible to set up a PC with every user one wants to interact with. Instead, PCs can be linked together forming a graph known as payment channel network (PCN) [21, 24]. In a PCN, a payment of α coins from a sender U_0 to a receiver U_n can be performed via a path $\{U_i\}_{i\in[0,n]}$ of intermediaries.

1.1 State-of-the-art PCNs

A possible way of achieving such a multi-hop payment (MHP) is an optimistic 1-round approach, e.g., Interledger [29]. Here, U_0 starts paying to its neighbor on the path U_1, who then pays to its neighbor U_2 and so on until U_n is reached. This protocol, however, relies on every intermediary behaving honestly, otherwise any intermediary can trivially steal coins by not forwarding the payment to its neighbor.

To achieve security in MHPs, most widely deployed PCNs (e.g., LN [24]) require an additional second round of communication (i.e., sequential, pair-wise communication between sender and receiver via intermediaries). Specifically, PCNs follow the principles of the 2-phase-commit protocol used to perform atomic updates in distributed databases. In the first communication round, the users on the payment path lock α coins of the PC with their right neighbor in a simple smart contract called Hash Time-Lock Contract (HTLC), which can be expressed even in restricted scripting languages such as the one used in Bitcoin. The money put into the HTLC by the left neighbor at each PC moves to the right neighbor, if this neighbor can present a secret chosen by U_n (i.e., the receiver of the payment); alternatively, it can be reclaimed by the left neighbor after some time has expired.

After HTLCs have been set up on the whole path, the users move to the second round, where they release the locks by passing the secret from U_n to U_0 via the intermediaries on the path before the time on the HTLCs has expired. Intermediaries are economically incentivized to assist in the 2-phase payment protocol. In the first round, when U_i receives α coins from the left neighbor U_{i-1}, it forwards only $\alpha - \text{fee}$ to the right neighbor U_{i+1}, charging fee coins for the forwarding service. In the second round, when U_{i+1} claims the $\alpha - \text{fee}$ coins from U_i, the latter is incentivized to recover the α coins from U_{i-1}.

1.2 Open problems in current PCNs

There are some fundamental problems with current PCNs that follow the 2-phase-commit paradigm. While 2-phase-commit has been successfully used for atomic updates in distributed databases, it is not well suited to applications where economic incentives are inherently involved. In particular, there exists a tradeoff between security, efficiency and number of rounds in the PCN setting that constitutes not only a challenging conceptual problem, but also one with strong practical impact, as we motivate below.

Staggered collateral After a user U_i has paid to U_{i+1}, it must have enough time to claim the coins put by U_{i-1}. If U_{i-1} is not cooperative, then this time is used to forcefully claim the funds with an on-chain transaction. The timing on the HTLCs (called collateral time in the blockchain folklore) grows therefore in a staggered manner from right to left, $t_i \geq t_{i+1} + \xi$. In practice, ξ has to be quite long: e.g., in the LN, it is set to one day (144 blocks). In the worst case, the funds are locked up for a time of $n \cdot \xi$. This means that a single payment of value α over n users can lock up a collateral of $\Theta(n^2 \cdot \alpha \cdot \xi)$. Reducing this locktime enables a faster release time of locked funds and directly improves the throughput of the network. Moreover, long locktimes are also problematic when looking at the high volatility of cryptocurrency prices, where prices can drop significantly within the same day.

Griefing attack A malicious user can start a MHP to itself, causing user U_i to lock up α coins for a time $(n-i) \cdot \xi$. The malicious user subsequently goes idle and lets the payment fail with the intention of reducing the overall throughput of the network by causing users to lock up their funds. In a different scenario, an intermediary could do the same by accepting payments in the first round, but going idle in the second. It is interesting also to observe the amplification factor: with the relatively small amount of α coins, an attacker can lock $(n-1) \cdot \alpha$ coins of the network. This attack is hard to detect and can even be used to target specific users in the PCN in order to lock up their funds.

Wormhole attack The wormhole attack [22] is an attack on PCNs where two colluding malicious users skip honest users in the open phase of the 2-phase-commit protocol and thereby cheat them out of their fees. The payment does not happen atomically anymore: For some users the payment is successful and for others it is not, i.e., for the ones encased by the malicious users. The users for whom it is unsuccessful have to lock up some of their funds, but do not get any fees for offering their services, nor can they use their locked funds for other payments. These fees go instead to the attacker.

HTLC contracts PCNs built on top of 2-phase-commit payments depend largely on HTLCs and the underlying cryptocurrencies supporting them in their scripts. However, there are a number of cryptocurrencies that do not have this functionality or that do not provide scripting capabilities at all, such as Stellar or Ripple. Instead, these currencies provide only digital signature schemes and timelocks.

On a conceptual level, one could actually wonder whether or not it is required to add an agreement protocol (in the database literature, a protocol where if an honest party delivers a message m, then m is eventually delivered by every honest party), like the HTLC-based 2-phase-commit paradigm, on top of the blockchain-inherited consensus protocol.

The current state of affairs thus leads to the following question: *Is it possible to design a PCN protocol with a single round of communication (and thus without HTLCs) while preserving security and atomicity?*

1.3 Our contributions

We positively answer this question by presenting *Blitz*, a novel payment protocol built on top of the existing payment channel constructions, which combines the advantages of both the optimistic 1-round and the 2-phase-commit paradigms. Our contributions are as follows.

• With Blitz, we introduce for the first time a payment protocol that achieves a MHP in one round of communication while preserving security in the presence of malicious intermediaries (i.e., as in the LN). The Blitz protocol has constant collateral of only $\Theta(n \cdot \alpha \cdot \xi)$, allowing for PCNs that are far more robust against griefing attacks and provide a higher transaction throughput. Additionally, the Blitz protocol is immune to the wormhole attack and having only one communication round reduces the chance of unsuccessful payments due to network faults.

• We show that Blitz payments can be realized with only timelocks and signatures, without requiring, in particular, HTLCs. This allows for a more widespread deployment, i.e., in cryptocurrencies that do not feature hashlocks or scripting, but only signatures and timelocks, e.g., Stellar or Ripple. Since Blitz builds on standard payment channel constructions, it can be smoothly integrated as an (alternative or additional) multi-hop protocol into all popular PCNs, such as the LN.

• We formally analyze the security and privacy of Blitz in the Universal Composability (UC) framework. We provide an ideal functionality modeling the security and privacy notions of interest and show that Blitz is a UC realization thereof.

• We evaluate Blitz and show that while the computation and communication overhead is inline with that of the LN, the size of the contract used in Blitz is around 26% smaller than an HTLC in the LN, which in practice opens the door for a higher number of simultaneous payments within each channel. We have additionally evaluated the effect of the reduction of collateral from staggered in the LN to constant in Blitz and observed that it reduces the number of unsuccessful payments due to locked funds by a factor between 4x and 33x, depending on payment amount and percentage of disrupted payments. Finally, the avoidance of the wormhole attack by design in Blitz can save up to 0.3 BTC (3397 USD in October 2020) of fees in our setting (over a three day period).

2 Background and notation

The notation used in this work is adopted from [5]. We provide here an overview on the necessary background and for more details we refer the reader to [5, 21, 22].

2.1 Transactions in the UTXO model

Throughout this work, we consider cryptocurrencies that are built with the *unspent transaction output* (UTXO) model, as Bitcoin is for instance. In such a model, the units of cash, which we will call *coins*, exist in *outputs* of *transactions*. Let us define such an output θ as a tuple consisting of two values, $\theta := (\mathsf{cash}, \phi)$, where $\theta.\mathsf{cash}$ denotes the amount of coins held in this output and $\theta.\phi$ is the condition which must be fulfilled in order to spend this output. The condition is encoded in the scripting language used by the underlying cryptocurrency. We say that a user U owns the coins in an output θ, if $\theta.\phi$ contains a digital signature verification script w.r.t. U's public key and the digital signature scheme of the underlying cryptocurrency. For this, we use the notation $\mathsf{OneSig}(U)$. If multiple signatures are required, we write $\mathsf{MultiSig}(U_1, \dots, U_n)$.

Ownership of outputs can change via transactions. A transaction maps a non-empty list of existing outputs to a non-empty list of new outputs. For better distinction, we refer to these existing outputs as *transaction inputs*. We

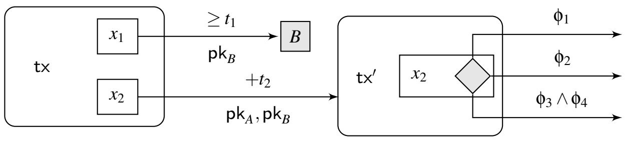


Figure 1: (Left) Transaction tx has two outputs, one of value x_1 that can be spent by B (indicated by the gray box) with a transaction signed w.r.t. pk_B at (or after) round t_1, and one of value x_2 that can be spent by a transaction signed w.r.t. pk_A and pk_B but only if at least t_2 rounds passed since tx was accepted on the blockchain. (Right) Transaction tx' has one input, which is the second output of tx containing x_2 coins and has only one output, which is of value x_2 and can be spent by a transaction whose witness satisfies the output condition $\phi_1 \vee \phi_2 \vee (\phi_3 \wedge \phi_4)$. The input of tx is not shown.

formally define a transaction body tx as an attribute tuple $\mathsf{tx} := (\mathsf{id}, \mathsf{input}, \mathsf{output})$. The identifier $\mathsf{tx.id} \in \{0,1\}^*$ is automatically assigned as the hash of the inputs and outputs, $\mathsf{tx.id} := \mathcal{H}(\mathsf{tx.input}, \mathsf{tx.output})$, where $\mathcal{H}$ is modelled as a random oracle. The attribute $\mathsf{tx.input}$ is a list of identifiers of the inputs of the transaction, while $\mathsf{tx.output} := (\theta_1, \dots, \theta_n)$ is a list of new outputs. A full transaction $\overline{\mathsf{tx}}$ contains additionally a list of witnesses, which fulfill the spending conditions of the inputs. We define $\overline{\mathsf{tx}} := (\mathsf{id}, \mathsf{input}, \mathsf{output}, \mathsf{witness})$ or for convenience $\overline{\mathsf{tx}} := (\mathsf{tx}, \mathsf{witness})$. Only a valid transaction can be published on the blockchain, i.e., one that has a valid witness for every input and has only inputs not used in other published transactions.

In fact, a transaction is not published on the blockchain immediately after it is submitted, but only after it is accepted through the consensus mechanism. We model that by defining a blockchain delay Δ, an upper bound on the time it takes for a transaction that is broadcast until it is added to the ledger.

For better readability we use charts to visualize transactions, their ordering and how they are used in protocols. The charts are expected to be read from left to right, i.e., the direction of the arrows. Every transaction is represented as a rectangle with rounded corners. Incoming arrows represent inputs. Every transaction has one or more output boxes inside it. Inside these boxes we write the amount of coins stored in the corresponding output. Every output box has one or more outgoing arrow. This arrow has the condition needed to spend the corresponding output written above and below it.

To present complex conditions in a compact way, we use the following notation. On a high level, we write the owner(s) of an output below the arrow and how they can spend it above. In a bit more detail, most output scripts require signature verification w.r.t. one or more public keys, a condition that we represent by writing the necessary public keys below a given arrow. Other conditions are written above the arrow. The conditions above can be any script supported by the underlying cryptocurrency, however in this paper we require only the following. We write "$+t$" or $\mathsf{RelTime}(t)$ to denote a relative timelock, i.e., the output with this condition can be spent, if and only if at least t rounds have passed since the transaction containing the output was published on the blockchain. Additionally, we consider absolute timelocks, denoted as "$\geq t$" or $\mathsf{AbsTime}(t)$: this condition is satisfied if and only if the blockchain is at least t blocks long. If an output condition is a disjunction of several conditions, i.e., $\phi = \phi_1 \vee \cdots \vee \phi_n$, we write a diamond shape in the output box and put each subcondition ϕ_i above/below its own arrow. For the conjunction of several conditions we write $\phi = \phi_1 \wedge \cdots \wedge \phi_n$. We illustrate an example of our transaction charts in Figure 1.

2.2 Payment channels

A payment channel is used by two parties P and Q to perform several payments between them while requiring only two on-chain transactions (for opening and closing). The balances are kept and updated in what is called a state. For brevity and readability, we hereby abstract away from the implementation details of a payment channel and provide a more detailed description in Appendix C.

We assume that there is an off-chain transaction $\mathsf{tx}^{\mathsf{state}}$ which holds the outputs representing the current state of the payment channel. We further assume that the current $\mathsf{tx}^{\mathsf{state}}$ can always be published on the blockchain and if an old state is published by a dishonest user, the honest user gets the total channel balance through some punishment mechanism.

Formally, we define a channel $\overline{\gamma}$ as the following attribute tuple $\overline{\gamma} := (\mathsf{id}, \mathsf{users}, \mathsf{cash}, \mathsf{st})$. Here, $\overline{\gamma}.\mathsf{id} \in \{0,1\}^*$ is a unique identifier of the channel, $\overline{\gamma}.\mathsf{users} \in \mathcal{P}^2$ denotes the two parties that participate in the channel out of the set of all parties $\mathcal{P}$. Further, $\overline{\gamma}.\mathsf{cash} \in \mathbb{R}_{\geq 0}$ stores the total number of coins held in the channel and $\overline{\gamma}.\mathsf{st} := (\theta_1, \ldots, \theta_n)$ is the current state of the channel consisting of a list of outputs. For convenience, we also define a channel skeleton γ with respect to a channel $\overline{\gamma}$ as the tuple $\gamma := (\overline{\gamma}.\mathsf{id}, \overline{\gamma}.\mathsf{users})$. When the channel is used along a payment path as shown in the next section, we say the $\gamma.\mathsf{left} \in \gamma.\mathsf{users}$ accesses the user that is closer to the sender and $\gamma.\mathsf{right} \in \gamma.\mathsf{users}$ the one closer to the receiver. The balance of each user can be inferred from the state $\overline{\gamma_i}.\mathsf{st}$, however for convenience we define a function $\overline{\gamma_i}.\mathsf{balance}(U)$, that returns the coins of user $U \in \gamma_i.\mathsf{users}$ in this channel.

2.3 Payment channel networks

Since maintaining a payment channel locks a certain amount of coins for a party, it is economically prohibitive to set up a payment channel with every party that one potentially wants to interact with. Instead, each party may open channels with a few other parties, creating thereby a network of channels. A payment channel network (PCN) [21] is thus a graph where vertices represent the users and edges represent channels between pairs of users. In a PCN, a user can pay any other user connected through a path of payment channels between them. Suppose user U_0 wants to pay some amount α to U_n, but does not have a payment channel directly with it. Now assume that instead, U_0 has a payment channel $\overline{\gamma_0}$ with U_1, who in turn has a channel $\overline{\gamma_1}$ with U_2 and so on, until the receiver U_n. We say that U_0 and U_n are connected by a path and denote a payment using it as *multi-hop payment* (MHP).

Optimistic payment schemes In an MHP, the main challenge is to ensure that the payment happens atomically and for everyone, so that no (honest) user loses any money. In fact, there exists payment-channel network constructions where this security property does not hold. We call them *optimisic payment schemes* and give Interledger [29] as an example. In this scheme, the users on the path simply forward the payment without any guarantee of the payment reaching the receiver. The sender U_0 starts by performing an update for channel $\overline{\gamma_0}$, where $\overline{\gamma_0}.\mathsf{balance}(U_1)$ is increased by α (and $\overline{\gamma_0}.\mathsf{balance}(U_0)$ is decreased by α) compared to the previous state. U_1 does the same with U_2 and this step is repeated until the receiver U_n is reached. This scheme works if every user is honest. However, a malicious intermediary can easily steal the money by simply stopping the payment and keeping the money for itself.

Secure MHPs Since the assumption that every user is honest is infeasible in practice, most widely deployed systems instead ensure that no honest user loses coins. The Lightning Network (LN) [24] uses so called Hash Time-Lock Contracts (HTLCs). An HTLC works as follows. In a payment channel between Alice and Bob, party Alice locks some coins that belong to her in an output that is spendable in the following fashion: (i) After a predefined time t, Alice gets her money back. (ii) Bob can also claim the money at any time, if he knows a pre-image r_A for a certain hash value $\mathcal{H}(r_A)$, which is set by Alice.

For an MHP in the LN, suppose again that we have a sender U_0 who wants to pay α to a receiver U_n via some intermediaries U_i with $i \in [1, n-1]$, and that two users U_j and U_{j+1} for $j \in [0, n-1]$ have an opened payment channel. Now for the first step, U_n samples a random number r, computes the hash of it $y := \mathcal{H}(r)$ and sends y to U_0. In the second step, the sender U_0 sets up an HTLC with U_1 by creating a new state with three outputs $\theta_1, \theta_2, \theta_3$ that correspondingly hold the amount of coins: α, U_0's balance minus α and U_1's balance. While θ_2 and θ_3 are spendable by their respective owners, θ_1 is the output used by the HTLC. The HTLC that is constructed spends the output containing α back to U_0 after n time, let us say n days, or to U_1 if it knows a value x such that $\mathcal{H}(x) = y$. Now U_1 repeats this step with its right neighbor, again using y but a different time, $(n-1)$ days, in the HTLC. This step is repeated until the receiver is reached, with a timeout of 1 day.

Now if constructed correctly, the receiver U_n can present r to its left neighbor U_{n-1}, which is the secret required in the HTLC for giving the money to U_n. We call this *opening the HTLC*. After doing that, the two parties can either agree to update their channel to a new state, where U_n has α coins more, or otherwise the receiver can publish the state and a transaction with witness r spending the money from the HTLC to itself on-chain. When a user U_i reveals the secret r to its left neighbor U_{i-1}, U_{i-1} can use r to continue this process. For this continuation, U_{i-1} needs to have enough time. Otherwise, U_i could claim the money of the HTLC it has with U_{i-1} by spending the HTLC on-chain at the last possible moment. Because of the blockchain delay, user U_{i-1} will notice this too late and will not be able to claim the money of the HTLC with U_{i-2} anymore. This is the reason why the timelocks on the HTLCs are staggered, i.e., increasing from right to left.

The aforementioned process where each user presents r to the left neighbor is repeated until the sender U_0 is reached, at which point the payment is completed. We call this approach of performing MHPs *2-phase-commit*.

3 Solution overview

The goal of this work is to achieve the best of the two multi-hop payment (MHP) paradigms existing nowadays (optimistic and 2-phase-commit), that is, an MHP protocol with a single round of communication that overcomes the drawbacks of the current LN MHP protocol and yet maintains the security and privacy notions of interest.

For that, we propose a paradigm shift, which we call *pay-or-revoke*. The idea is to update the payment channels from sender to receiver in a single round of communication. The key technical challenge is thus to design a single channel update that can be used *simultaneously* for sending coins from the left neighbor to the right one if the payment is successful and for a refund of the coins to the left neighbor if the payment is unsuccessful (e.g., one intermediary is offline).

We present the pay-or-revoke paradigm in an incremental way, starting with a naive design, discussing the problems with it, and presenting a tentative solution. We iterate these steps until we finally reach our solution.

Naive approach Assume a setting with a sender U_0 who wants to pay α coins to a receiver U_n via a known path of some intermediaries U_i ($i \in [1, n-1]$), where each pair of consecutive users U_j and U_{j+1} for $j \in [0, n-1]$ has a payment channel $\overline{\gamma_j}$, where $\overline{\gamma_j}.\mathsf{balance}(U_j) \geq \alpha$. We start out with an optimistic payment scheme, as presented in Section 2.3. We already explained that the success of such a payment relies on every intermediary behaving honestly and really forwarding the α coins. Should an intermediary not forward the payment, U_n will never receive anything. Additionally, a receiver could claim that it never received the money even though it actually did and it would be difficult for the sender to prove otherwise.

To solve these problems the sender faces when using this form of payment we introduce a possibility for the sender to step back from a payment, that is, refund itself and all subsequent users the α coins that they initially put, should the payment not reach U_n. With such a refund functionality, the sender can now check if a receiver is giving a confirmation that it got the payment. This confirmation is external to the system (e.g., a digital payment receipt) and serves additionally as a proof that the money was received. If such a confirmation is not received, the sender simply steps back from the payment and the payments in every channel are reverted.

Adding refund functionality Adding a refund functionality while avoiding additional security problems is challenging. Two neighbors can no longer simply update their channel $\overline{\gamma_i}$ to a state where α coins are moved from the left to the right neighbor, as this only encodes the payment. Instead, we need to introduce an intermediate channel state $\mathsf{tx}^{\mathsf{state}}$, which encodes the possibility for both a refund and a payment.

We realize that as follows. This new state has an output holding α coins coming from $\overline{\gamma_i}.\mathsf{left}$ $(= U_i)$ while leaving the rest of the balance in the channel untouched. The output containing α coins becomes then the input for two mutually exclusive transactions: refund and payment. We denote the refund transaction as $\mathsf{tx}_i^{\mathsf{r}}$, which spends the money back to $\overline{\gamma_i}.\mathsf{left}$ $(= U_i)$. We denote the payment transaction as $\mathsf{tx}_i^{\mathsf{p}}$, which spends the money to $\overline{\gamma_i}.\mathsf{right}$ $(= U_{i+1})$. The refund should only be possible until a certain time T. This gives the sender time to wait for the payment to reach the receiver and for the receiver to give a (signed) confirmation. Should something go wrong, the sender starts the refund procedure. After time T, if no refund happened, the payment is considered successful and the payment transaction becomes valid.

The latter condition can easily be expressed in the scripting language of virtually any cryptocurrency including Bitcoin, by making use of absolute timelocks, which in this work we defined as $\mathsf{AbsTime}(T)$, meaning an output can be spent only after some time T. Unfortunately, the same cannot be done for expressing the condition that an output is spendable *only before* time T (e.g., see [14] for details).

We overcome this problem in a different way. Instead of making the refund transaction $\mathsf{tx}_i^{\mathsf{r}}$ only valid before T, we allow both $\mathsf{tx}_i^{\mathsf{r}}$ and the payment transaction $\mathsf{tx}_i^{\mathsf{p}}$ to be valid after time T and encode a condition that, should both be posted after T, $\mathsf{tx}_i^{\mathsf{p}}$ will always be accepted over $\mathsf{tx}_i^{\mathsf{r}}$. We can achieve this by adding a relative timelock on the input of $\mathsf{tx}_i^{\mathsf{r}}$ of the blockchain delay Δ. In other words, should a user try to close the channel with $\mathsf{tx}^{\mathsf{state}}$ appearing on the chain after time T, the other user will have enough time to react and post $\mathsf{tx}_i^{\mathsf{p}}$, which will get accepted before the relative timelock of $\mathsf{tx}_i^{\mathsf{r}}$ expires. For the honest refund case nothing changes: If $\mathsf{tx}^{\mathsf{state}}$ is on-chain and $\mathsf{tx}_i^{\mathsf{r}}$ gets posted before $T - \Delta$, it will always be accepted over $\mathsf{tx}_i^{\mathsf{p}}$, since the latter transaction is only valid after time T.

Making the refund atomic So far, we added a refund functionality that is (i) not atomic and (ii) triggerable by every user on the path. An obvious attack on this scheme would be for any user on the path to commence the refund in a way that $\mathsf{tx}_i^{\mathsf{r}}$ is accepted on the ledger just before T. Other users would not have enough time to react accordingly and lose their funds. Also, allowing intermediary users to start the refund opens up the door to griefing, where malicious users start a refund even though the payment reached the receiver. We therefore need a mechanism that (i) ensures the atomicity of the refund (or payment) and (ii) is triggerable only by the sender.

Following the LN protocol, one could add a condition

$\mathcal{H}(r_A)$ on the refund transaction, such that the refund can only happen when a pre-image r_A chosen by the sender is known. To prevent the sender from publishing at the last moment however, the timing for the refund in the next channel would have to be $T+\Delta$ to give U_1 enough time to react. In subsequent channels, this time would grow by Δ for every hop and we would then have an undesirable staggered time delay. Additionally, this approach would rely on the scripting language supporting hash-lock functionality.

To keep the time delay constant, we instead make the refund transactions dependent on a transaction being published by the sender. First, the sender creates a transaction that we name *enable-refund* and denote by $\mathsf{tx}^{\mathsf{er}}$. The unsigned transaction $\mathsf{tx}^{\mathsf{er}}$ is then passed through the path and is used at each channel $\overline{\gamma_j}$ as an additional input for $\mathsf{tx}^{\mathsf{r}}_i$.

This makes the refund transaction at every channel dependent on $\mathsf{tx}^{\mathsf{er}}$ and gives the sender and only the sender the possibility to abort the payment until time T in case something goes wrong along the path (e.g., a user is offline or the enable-refund transaction is tampered), and the receiver the guarantee to get the payment after time T otherwise.

In order to use the same $\mathsf{tx}^{\mathsf{er}}$ for the refund transaction $\mathsf{tx}^{\mathsf{r}}_i$ of every channel $\overline{\gamma_i}$, we proceed as follows. For every user on the path (except for the receiver) there needs to exist an output in $\mathsf{tx}^{\mathsf{er}}$ which belongs to it. Additionally, we observe that an intermediary U_i whose left neighbor U_{i-1} has used $\mathsf{tx}^{\mathsf{er}}$ as input for its refund transaction $\mathsf{tx}^{\mathsf{r}}_{i-1}$ can safely construct a refund transaction $\mathsf{tx}^{\mathsf{r}}_i$ dependent on the same $\mathsf{tx}^{\mathsf{er}}$, because it will know that if its left neighbor refunded, $\mathsf{tx}^{\mathsf{er}}$ has to be on-chain, which means that it can refund itself. Also, since the appearance of $\mathsf{tx}^{\mathsf{er}}$ on the ledger is a global event that is observable by everyone at the same time, the time T used for the refund can be the same for every channel, i.e., constant.

Putting everything together Our approach is depicted in Figure 2, $\mathsf{tx}^{\mathsf{er}}$ is shown in Figure 3, and the transaction structure between two users is shown in Figure 4. Note that we change the payment value from α to α_i to embed a per-hop fee (see Appendix A for details). After the payment is set up from sender to receiver, the receiver sends a confirmation of $\mathsf{tx}^{\mathsf{er}}$ back to U_0, which acts both as verification that $\mathsf{tx}^{\mathsf{er}}$ was not tampered and as a payment confirmation. Should the sender receive this in time, it will wait until time T, after which the payment will be successful. If no confirmation was received in time, or $\mathsf{tx}^{\mathsf{er}}$ was tampered, the sender will publish $\mathsf{tx}^{\mathsf{er}}$ in time to trigger the refund.

We remark that it is crucial that every intermediate user can safely construct $\mathsf{tx}^{\mathsf{r}}_i$ only observing $\mathsf{tx}^{\mathsf{er}}$, but not the input funding it (or not even knowing whether it will be funded at all in the first place). Indeed, an intermediary U_i does not care if the transaction $\mathsf{tx}^{\mathsf{er}}$ is spendable at all, it only cares that its left neighbor U_{i-1} uses an output of the same transaction $\mathsf{tx}^{\mathsf{er}}$ as input for its refund transaction $\mathsf{tx}^{\mathsf{r}}_{i-1}$, as U_i does in $\mathsf{tx}^{\mathsf{r}}_i$.

In UTXO based cryptocurrencies, using the j^{th} output of a transaction tx as input of another transaction tx' means ref-

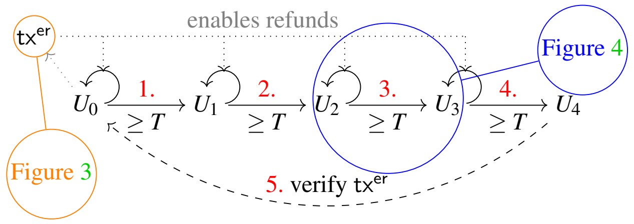


Figure 2: Illustration of the pay-or-revoke paradigm.

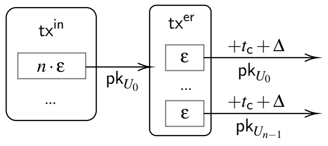


Figure 3: Transaction $\mathsf{tx}^{\mathsf{er}}$, which enables the refunds and, here, spends the output of some other transaction $\mathsf{tx}^{\mathsf{in}}$.

erencing the hash of the transaction body $\mathcal{H}(\mathsf{tx})$, which we defined as $\mathsf{tx.id}$, plus an index j. A transaction $\mathsf{tx}^{\mathsf{r}}_i$ that was created with an input referencing $\mathsf{tx}^{\mathsf{er}}.\mathsf{id}$ and some index j, can only be valid if $\mathsf{tx}^{\mathsf{er}}$ is published. This means, in particular, that it is computationally infeasible to create a different transaction $\mathsf{tx}^{\mathsf{er}\prime} \neq \mathsf{tx}^{\mathsf{er}}$ and use one of $\mathsf{tx}^{\mathsf{er}\prime}$'s outputs as input of $\mathsf{tx}^{\mathsf{r}}_i$ without finding a collision in $\mathcal{H}$. Further, as $\mathsf{tx}^{\mathsf{r}}_i$ requires the signatures of both U_i and U_{i+1}, a malicious U_i on its own cannot create a different refund transaction $\mathsf{tx}^{\mathsf{r}\prime}_i$ that does not depend on $\mathsf{tx}^{\mathsf{er}}$.

A final timelock There is however still one subtle problem with the construction up to this point regarding the timing coming from the fact that the sender has the advantage of being the only one able to trigger the refund by publishing $\mathsf{tx}^{\mathsf{er}}$. In a bit more detail, as closing a channel takes some time, a malicious sender U_0 can forcefully close its channel with U_1 beforehand. Then, when $\mathsf{tx}^{\mathsf{state}}_0$ is on the ledger, the sender publishes $\mathsf{tx}^{\mathsf{er}}$ so that it appears just before $T-\Delta$. The sender is able to publish $\mathsf{tx}^{\mathsf{r}}_0$ just in time before T. All other intermediaries however, who did not yet close their channel, with the result that $\mathsf{tx}^{\mathsf{state}}_i$ is not on the ledger, will not be able to do this and publish $\mathsf{tx}^{\mathsf{r}}_i$ in time.

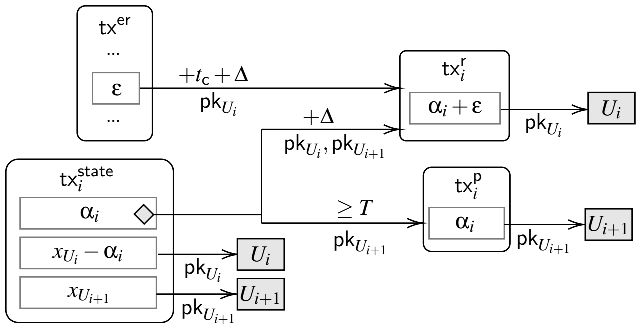


Figure 4: Payment setup in the channel $\overline{\gamma_i}$ of two neighboring users U_i and U_{i+1} with the new state $\mathsf{tx}^{\mathsf{state}}$. x_{U_i} and $x_{U_{i+1}}$ are the amounts that U_i and U_{i+1} own in the state prior to $\mathsf{tx}^{\mathsf{state}}$.

To solve this problem, we introduce a relative timelock on the outputs of $\mathsf{tx}^{\mathsf{er}}$ of exactly $t_{\mathsf{c}} + \Delta$, as shown in Figure 3 and Figure 4. This relative time delay is an upper bound on the time it takes to (i) forcefully close the channel and (ii) wait for the time delay needed to publish $\mathsf{tx}_i^{\mathsf{r}}$. With this, we ensure that no user gains an advantage by closing its channel in advance, since this can be entirely done in this relative timelock on $\mathsf{tx}^{\mathsf{er}}$'s outputs. Honest intermediaries can easily check that this relative timelock is present in $\mathsf{tx}^{\mathsf{er}}$'s outputs and every user on the payment path has the same time.

A timeline of when the transactions have to appear on the ledger is given in Appendix F. Note that for the payment to be refunded, $\mathsf{tx}^{\mathsf{er}}$ has to be posted to the ledger at the latest at time $T - t_{\mathsf{c}} - 3\Delta$. Still, for better readability we sometimes refer to this case simply as $\mathsf{tx}^{\mathsf{er}}$ being published before time T.

Improving anonymity of the path Until this point, we have shown a design of the pay-or-revoke paradigm, that, while ensuring that honest users do not lose coins, has an obvious drawback in terms of anonymity. In particular, the transaction outputs of $\mathsf{tx}^{\mathsf{er}}$ contain the addresses of every user on the path in the clear (except for the receiver who does not need to refund and therefore needs no such output). This means that every intermediary (or any other user that sees $\mathsf{tx}^{\mathsf{er}}$) learns about the identity of every user on the payment path as soon as it sees $\mathsf{tx}^{\mathsf{er}}$. To prevent this leak, we use stealth addresses [31]. We overview our use of stealth addresses here and refer to Section 4.2 for technical details. On a high level, instead of spending to existing addresses, the sender uses fresh addresses for the outputs of $\mathsf{tx}^{\mathsf{er}}$. These addresses were never used before, but are under the control of the respective users. With this approach, if $\mathsf{tx}^{\mathsf{er}}$ is leaked, the identities of all users on the path, especially the identity of the sender and the receiver, remain hidden. Note that we assume the input of $\mathsf{tx}^{\mathsf{er}}$ to be an unused and unlinkable input of the sender.

Fast track payments The design considered so far has still a practical drawback compared to MHPs in the LN. In the LN, if every user is honest, the payment is carried out almost instantaneously, i.e. the channels are updated as soon as the HTLCs are opened. Obviously, users of a payment do not want to wait until some time T until the payment is carried out, even if all users are honest. To enable the same fast payments in Blitz, we extend the protocol design with an *optional* second communication round, called the fast track (we compare this second round to the one adopted in the LN below). Specifically, the users on the path can honestly update their channels from the sender to the receiver to a state where the α coins move from left to right.

For this, the sender does not go idle upon receiving the confirmation in time from the receiver. Instead, U_0 starts updating the channel $\overline{\gamma_0}$ with its neighbor U_1 to a state where the α coins are paid to U_1. Since U_0 is the only one able to publish $\mathsf{tx}^{\mathsf{er}}$, U_0 is safe when performing this update. After this update, U_1 does the same with U_2. All users on the path repeat this step until the receiver is reached. If everyone is honest, the payment will be carried out as quick as in the LN honest case. If someone stops the update or some honest users are skipped by colluding malicious users, honest users simply wait until time T, and claim their money (and fees) either by cooperatively updating the channel with their neighbor or forcefully on-chain. Intuitively, since intermediary users only update their right channel after updating their left channel, they cannot lose any money, even if $\mathsf{tx}^{\mathsf{er}}$ is published.

Using the fast track seems to be a better choice for normal payments. However, there are applications, where the non fast track is more suitable, e.g., a service with a trial period or a subscription model, where a user might want to set up a payment, that gets confirmed after some time. Should the user decide against it, he/she can cancel the payment. The choice of fast track is up to the user. Having this second round is completely optional and for efficiency reasons only. A payment that is carried out in one round has the same security properties as one carried out in two rounds.

Fast revoke In the case that an intermediary is offline and the payment is unsuccessful, the refund can happen without necessarily publishing $\mathsf{tx}^{\mathsf{er}}$, saving the cost to put a transaction on-chain. Say U_{i+1} is offline and U_i has already set up the construction with U_{i-1}. As soon as an honest U_i notices that U_{i+1} is unresponsive, it can start asking U_{i-1} to update their channel to the state before the payment was set up. After doing this, U_{i-1} asks its left neighbor to do the same and so on until the sender is reached and the payment is reverted without $\mathsf{tx}^{\mathsf{er}}$ being published. Should some intermediary refuse to honestly revoke, then $\mathsf{tx}^{\mathsf{er}}$ can still be published. Apart from funds being locked for a shorter time, one could add additional incentives to the fast revocation (or fast track) by giving a small fee to the users that are willing to participate in it. Of course, users need a mechanism to find out whether others are offline. For that, we note that the LN protocol mandates users to periodically broadcast a heartbeat message. We consider such default messages orthogonal to payment protocols and do not count them in round complexity.

Honest update The transactions in Figure 4 between users are exchanged off-chain and used to guarantee that honest users do not lose any coins. However, should one of the users in a channel be able to convince the other that it is able to enforce either $\mathsf{tx}_i^{\mathsf{r}}$ or $\mathsf{tx}_i^{\mathsf{p}}$ on-chain (that is if $\mathsf{tx}^{\mathsf{er}}$ is on-chain before time T or time T has already passed, respectively), two collaborating users can simply perform an honest update. For this, they update their channel to a state where both have their corresponding balance, with the benefit that no transaction has to be put on-chain and their channel remains open.

Blitz vs. ILP/LN/AMHL We claim that Blitz is a solution for the issues presented in Section 1 and allows for PCNs that have higher throughput, less communication complexity, additional security against certain attacks, and are implementable in cryptocurrencies without scripting capabilities. We highlight the differences between Blitz and other state-of-the-art payment methods such as Interledger Payments (ILP), the LN

Table 1: Features of different payment methods: Interledger (ILP), Lightning Network (LN), Anonymous Multi-Hop Locks (AMHL), Blitz and Blitz using the fast track payment (FT). We abbreviate timelocks as TL and signature functionality as σ. * The requirement of HTLC can be dropped from the LN using scriptless scripts when feasible.

	ILP	LN	AMHL	Blitz	Blitz FT
Bal. Security	No	Yes	Yes	Yes	Yes
Rounds	1	2	2	1	2
Atomicity	No	No (Wormhole)	Yes	Yes	Yes
Scripting	σ	σ, TL, HTLCs*	σ, TL	σ, TL	σ, TL
Collateral	n/a	linear	linear	constant	constant

Table 2: Collateral time for the LN, AMHL and Blitz for unsuccessful (refund) and successful payments (pay) as well as different threat models. We say *instant* when noone on the path stops the payment in either round. ξ denotes the time users need to claim their funds (e.g., in the LN 144 blocks).

	LN / AMHL		Blitz	
	refund	pay	refund	pay
anyone malicious	$n \cdot \xi$	$n \cdot \xi$	ξ	ξ
sender honest	$n \cdot \xi$	$n \cdot \xi$	Δ	ξ
everyone honest	instant	instant	instant	instant

and the wormhole secure construction Anonymous Multi-Hop Locks (AMHL) [22] in Table 1.

First, Blitz offers balance security with only one round of communication, while ILP does not provide that and the LN requires two rounds. While the fast track optimization does involve a second round (from left to right, as opposed to right to left as in the LN), it is optional and affects only the efficiency (in the case everyone is honest) and not security: a payment that had a successful first round will be successful regardless of any network faults in the second round.

Indeed, the same holds true for the wormhole attack: Once a user has successfully set up a Blitz payment, it cannot be skipped anymore in the second round, even with the fast track. The payment is successful for everyone or no one, achieving thus the atomicity property missing in ILP and the LN, and honest intermediaries are not cheated out of their fees.

Secondly, Blitz reduces the collateral from linear (in the size of the path) to constant in the case some of the parties are malicious, while offering comparable performance in the optimistic case, as shown in Section 6. For a corner case where the sender is honest, the collateral can even be unlocked almost instantaneously. We show in which cases Blitz outperforms the LN in Table 2. Finally, in terms of interoperability, we require only signatures and timelocks from the underlying blockchain, with the LN additionally requiring HTLCs and ILP only signatures.

Concurrent payments In Blitz, multiple payments can be carried out in parallel, analogous to concurrent HTLC-based payments in the LN (see Appendix A for further discussion and an illustrative example).

4 Our construction

4.1 Security and privacy goals

We informally review the security and privacy goals of a PCN, deferring the formal definitions to Appendix K.

Balance security Honest intermediaries do not lose money [21].

Sender/Receiver privacy In the case of a successful payment, malicious intermediaries cannot determine if the left neighbor along the path is the actual sender or just an honest user connected to the sender through a path of non-compromised users. Similarly, malicious intermediaries cannot determine if the right neighbor is the actual receiver or an honest user connected to the receiver through a path of non-compromised users.

Path privacy In the case of a successful payment, malicious intermediaries cannot determine which users participated in the payment aside from their direct neighbors.

4.2 Assumptions and building blocks

System assumptions We assume that every party has a publicly known pair of public keys (A, B) as required for stealth address creation (see below). We further assume that honest parties are required to stay online for the duration of the protocol. Finally, we consider the route finding algorithm an orthogonal problem and assume that every user (U_0) has access to a function $\mathsf{pathList} \leftarrow \mathsf{GenPath}(U_0, U_n)$, which generates a valid path from U_0 to U_n over some intermediaries. We refer the reader to [26, 27] for more details on recent routing algorithms for PCNs. We now introduce the cryptographic building blocks that we require in our protocol.

Ledger and payment channels We rely, as a blackbox, on a public ledger to keep track of all balances and transactions and a PCN that supports the creation, update, and closure of channels (see Section 2). We further assume that payment channels between users that want to conduct payments are already opened. We denote the standard operations to interact with the blockchain and the channels as follows:

$\mathsf{publishTx}(\overline{\mathsf{tx}})$: If $\overline{\mathsf{tx}}$ is a valid transaction (Section 2), it will be accepted on the ledger after at most time Δ.

$\mathsf{updateChannel}(\overline{\gamma_i}, \mathsf{tx}_i^{\mathsf{state}})$: When called by a user $\in \overline{\gamma_i}.\mathsf{users}$, initiates an update in $\overline{\gamma_i}$ to the state $\mathsf{tx}_i^{\mathsf{state}}$. If the update is successful, $(\mathsf{update-ok})$ is returned to both users of the channel, else $(\mathsf{update-fail})$ is returned to them. We define t_{u} as an upper bound on the time it takes for a channel update after this procedure is called.

$\mathsf{closeChannel}(\overline{\gamma_i})$: When called by a user $\in \overline{\gamma_i}.\mathsf{users}$, closes the channel, such that the latest state transaction $\mathsf{tx}_i^{\mathsf{state}}$ will appear on the ledger. We define t_{c} as an upper bound on the time it takes for $\mathsf{tx}_i^{\mathsf{state}}$ to appear on the ledger after this procedure is called.

Digital signatures A digital signature scheme is a tuple of algorithms $\Sigma := (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Vrfy})$ defined as follows:

$(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(\lambda)$ is a PPT algorithm that on input the security parameter λ, outputs a pair of public and private keys $(\mathsf{pk}, \mathsf{sk})$.

$\sigma \leftarrow \mathsf{Sign}(\mathsf{sk}, m)$ is a PPT algorithm that on input the private key sk and a message m outputs a signature σ.

$\{0,1\} \leftarrow \mathsf{Vrfy}(\mathsf{pk}, \sigma, m)$ is a DPT algorithm that on input the public key pk, an authentication tag σ and a message m, outputs 1 if σ is a valid authentication for m.

We require that the digital signature scheme is correct, that is, $\forall(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(\lambda)$ it must hold that $1 \leftarrow \mathsf{Vrfy}(\mathsf{pk}, \mathsf{Sign}(\mathsf{sk}, m), m)$. We additionally require a digital signature scheme that is strongly unforgeable against message-chosen attacks (EUF-CMA) [15].

Stealth addresses [31] On a high level, this scheme allows a user (say Alice) to derive a fresh public key in a digital signature scheme Σ controlled by another user (say Bob) on input two of Bob's public keys. In a bit more detail, a *stealth addresses* scheme is a tuple of algorithms $\Phi := (\mathsf{GenPk}, \mathsf{GenSk})$ defined as follows:

$(P, R) \leftarrow \mathsf{GenPk}(A, B)$ is a PPT algorithm that on input two public keys A, B controlled by some user U, creates a new public key P under U's control. This is done by first sampling some randomness $r \xleftarrow{\$} [0, l-1]$, where l is the prime order of the group used in the underlying signature scheme Σ, and computing $P := g^{\mathcal{H}(A^r)} \cdot B$, where $\mathcal{H}$ is a hash function modelled as a random oracle. Then, the value $R := g^r$ is calculated. P is the public key under U's control and R is the information required to construct the private key.

$p \leftarrow \mathsf{GenSk}(a, b, P, R)$ is a DPT algorithm that on input two secret keys a, b corresponding to the two public keys A, B and a pair (P, R) that was generated as $P \leftarrow \mathsf{GenPk}(A, B)$, creates the secret key p corresponding to P. This is done by computing $p := \mathcal{H}(R^a) + b$.

We see that correctness follows directly: $g^p = g^{\mathcal{H}(R^a)+b} = g^{\mathcal{H}(g^{r \cdot a})} \cdot g^b = g^{\mathcal{H}(A^r)} \cdot B = P$. In [31] it is argued that this new one-time public key P is *unlinkable* for a spectator even when observing R, meaning on a high level that P for some user U cannot be linked to any existing public key of U. For simplicity, we denote $\widetilde{U_i}, \mathsf{pk}_{\widetilde{U_i}}$ when referring to the stealth identity or the stealth public key under the control of user U_i.

Anonymous communication network (ACN) An ACN allows users to communicate anonymously with each other. One such ACN is based on onion routing, whose ideal functionality is defined in [8]. Sphinx [11] is a realization of this and (extended with a per-hop payload) is used in the Lightning Network (LN). We use this functionality here as well in a blackbox way. On a high level, routing information and a per-hop payload is encrypted and layered for every user along a path, in what is called an *onion*. Every user on the path can then, when it is its turn, "peel off" such a layer, revealing: (i) the next neighbor; (ii) the payload meant for it; and (iii) the rest of the data, which is again an onion that can only be opened by the next neighbor. This rest of data is then forwarded to the next user and so on until the receiver is reached.

For readability, we use two algorithms, where $\mathsf{onion} \leftarrow \mathsf{CreateRoutingInfo}(\{U_i\}_{i\in[1,n]}, \{\mathsf{msg}_i\}_{i\in[1,n]})$ creates such a routing object (an onion) using (publicly known) public encryption keys of the corresponding users on the path. Moreover, when called by correct user U_i, the algorithm $\mathsf{GetRoutingInfo}(\mathsf{onion}_i, U_i)$ returns $(U_{i+1}, \mathsf{msg}_i, \mathsf{onion}_{i+1})$, that is, the next user on the path, a message and a new onion or returns msg_n if called by the recipient. A wrong user $U \neq U_i$ calling $\mathsf{GetRoutingInfo}(\mathsf{onion}_i, U_i)$ will result in an error $\perp$.

4.3 2-party protocol for channel update

In this section, we show the necessary steps to update a single channel $\overline{\gamma_i}$ between two consecutive users U_i and U_{i+1} on a payment path to a state encoding our payment functionality as shown in Figure 4. We will describe later in Section 4.4 the complete multi-hop payment (MHP) protocol.

As overviewed in Section 3, a channel update requires to create a series of transactions to realize the "pay-or-revoke" semantics at a given channel. In particular, for readability, we define the following transaction creation methods and in Figure 7 some macros to be used hereby in the paper:

$\mathsf{tx}^{\mathsf{p}}_i := \mathsf{GenPay}(\mathsf{tx}^{\mathsf{state}}_i)$ This transaction takes $\mathsf{tx}^{\mathsf{state}}_i.\mathsf{output}[0]$ as input and creates a single $\mathsf{output} := (\alpha_i, \mathsf{OneSig}(U_{i+1}))$.

$\mathsf{tx}^{\mathsf{r}}_i := \mathsf{GenRef}(\mathsf{tx}^{\mathsf{state}}_i, \mathsf{tx}^{\mathsf{er}}, \theta_{\varepsilon_i})$ This transaction takes as input $\mathsf{tx}^{\mathsf{state}}_i.\mathsf{output}[0]$ and $\theta_{\varepsilon_i} \in \mathsf{tx}^{\mathsf{er}}.\mathsf{output}$. The calling user U_i makes sure that this output belongs to a stealth address under U_i's control. It creates a single output $\mathsf{tx}^{\mathsf{r}}_i.\mathsf{output} := (\alpha_i + \varepsilon, \mathsf{OneSig}(U_i))$, where α_i, U_i, U_{i+1} are taken from $\mathsf{tx}^{\mathsf{state}}_i$.

We now explain in detailed order, how these transactions have to be created, signed and exchanged. A full description in pseudocode is given in Figure 5. This two party update procedure, which we call `pcSetup`, is called by a user U_i giving as parameters the channel $\overline{\gamma_i}$ with its right neighbor U_{i+1}, the transaction $\mathsf{tx}^{\mathsf{er}}$, a list containing the values R_i for the stealth addresses of each user on the path, onion_{i+1} containing some routing information for the next user, the output $\theta_{\varepsilon_i} \in \mathsf{tx}^{\mathsf{er}}.\mathsf{output}$ that belongs to a stealth address of U_i, the amount to be paid α_i and the time T. The user U_i knows these values either from performing `pcSetup` with its left neighbor U_{i-1} or because U_i is the sender.

The first step for U_i is to create the new channel state from the channel $\overline{\gamma_i}$ and the amount α_i by calling $\mathsf{tx}^{\mathsf{state}}_i := \mathtt{genState}(\overline{\gamma_i}, \alpha)$. In the second step, U_i creates the transaction $\mathsf{tx}^{\mathsf{r}}_i$ from $\mathsf{tx}^{\mathsf{state}}_i.\mathsf{output}[0]$ and θ_{ε_i}. Then, U_i sends $\mathsf{tx}^{\mathsf{er}}$, $\mathsf{tx}^{\mathsf{state}}_i$, $\mathsf{tx}^{\mathsf{r}}_i$, rList and onion_{i+1} to its right neighbor U_{i+1}.

Now U_{i+1} checks if $\mathsf{tx}^{\mathsf{er}}$ is well-formed and, if it is not the receiver, has an output $\theta_{\varepsilon_{i+1}}$, which belongs to its stealth address (using its stealth address private keys a, b) under some $R_i \in \mathsf{rList}$. Moreover, it checks that onion_{i+1} contains the correct routing information and a message indicating that the $\mathsf{tx}^{\mathsf{er}}$ was not tampered, for instance a hash of it. All this is done using the macro (see Figure 7) $(\mathsf{sk}_{\widetilde{U_{i+1}}}, \theta_{\varepsilon_{i+1}}, R_{i+1}, U_{i+2}, \mathsf{onion}_{i+2}) :=$

```
$\underline{\texttt{pcSetup}(\overline{\gamma}_i, \mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_{i+1}, \theta_{\varepsilon_i}, \alpha_i, T)}$:

$\underline{U_i}$
 1. $\mathsf{tx}_i^{\mathsf{state}} := \texttt{genState}(\alpha_i, T, \overline{\gamma}_i)$
 2. $\mathsf{tx}_i^{\mathsf{r}} := \mathsf{GenRef}(\mathsf{tx}_i^{\mathsf{state}}, \theta_{\varepsilon_i})$
 3. Send $(\mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_{i+1}, \mathsf{tx}^{\mathsf{state}}, \mathsf{tx}_i^{\mathsf{r}})$ to $U_{i+1}$ $(= \overline{\gamma}_i.\mathsf{right})$

$\underline{U_{i+1}}$ upon $(\mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_{i+1}, \mathsf{tx}^{\mathsf{state}}, \mathsf{tx}_i^{\mathsf{r}})$ from $U_i$
 4. Check that $\texttt{checkTxEr}(U_{i+1}, U_{i+1}.a, U_{i+1}.b, \mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_{i+1}) \neq \bot$, but returns some values $(\mathsf{sk}_{\widetilde{U_{i+1}}}, \theta_{\varepsilon_{i+1}}, R_{i+1}, U_{i+2}, \mathsf{onion}_{i+2})$
 5. Extract $\alpha_i$ and $T$ from $\mathsf{tx}^{\mathsf{state}}$ and check $\mathsf{tx}_i^{\mathsf{state}} = \texttt{genState}(\alpha_i, T, \overline{\gamma}_i)$
 6. Check that for one output $\theta_{\varepsilon_x} \in \mathsf{tx}^{\mathsf{er}}.\mathsf{output}$ it holds that $\mathsf{tx}_i^{\mathsf{r}} := \mathsf{GenRef}(\mathsf{tx}_i^{\mathsf{state}}, \theta_{\varepsilon_x})$. If one of these previous checks failed, return $\bot$.
 7. $\mathsf{tx}_i^{\mathsf{p}} := \mathsf{GenPay}(\mathsf{tx}_i^{\mathsf{state}})$
 8. Send $(\sigma_{U_{i+1}}(\mathsf{tx}_i^{\mathsf{r}}))$ to $U_{i+1}$

$\underline{U_i}$ upon $(\sigma_{U_{i+1}}(\mathsf{tx}_i^{\mathsf{r}}))$
 9. If $\sigma_{U_{i+1}}(\mathsf{tx}_i^{\mathsf{r}})$ is not a correct signature of $U_{i+1}$ for the $\mathsf{tx}_i^{\mathsf{r}}$ created in step 2, return $\bot$.
10. $\texttt{pcSetup}(\overline{\gamma}_i, \mathsf{tx}_i^{\mathsf{state}})$
11. If, after $t_{\mathsf{u}} + t_{\mathsf{c}}$ time has expired, the message $(\mathsf{update{-}ok})$ is returned, return $\top$. Else return $\bot$.

$\underline{U_{i+1}}$
12. Upon $(\mathsf{update{-}ok})$, return $(\mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_{i+2}, U_{i+2}, \theta_{\varepsilon_{i+1}}, \alpha_i, T)$
13. Upon $(\mathsf{update{-}fail})$, return $\bot$
```

Figure 5: Protocol for 2-party channel update

$\texttt{checkTxEr}(U_{i+1}, U_{i+1}.a, U_{i+1}.b, U_{i+1}.\mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_{i+1})$, which returns $\bot$ if any of the checks fail.

Then, U_{i+1} checks if $\mathsf{tx}_i^{\mathsf{state}}$ and $\mathsf{tx}_i^{\mathsf{r}}$ were well-constructed and in particular, that $\mathsf{tx}_i^{\mathsf{r}}$ uses an output of $\mathsf{tx}^{\mathsf{er}}$ as input. If everything is ok, then U_{i+1} can independently create $\mathsf{tx}_i^{\mathsf{p}}$, since it requires only its own signature. Next, U_{i+1} pre-signs $\mathsf{tx}_i^{\mathsf{r}}$ and sends this signature to U_i. U_i checks if this signature is correct and then invokes a channel update with U_{i+1} to $\mathsf{tx}_i^{\mathsf{state}}$.

After this step, the $\texttt{pcSetup}$ function is finished and returns either $(\mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_{i+2}, U_{i+2}, \theta_{\varepsilon_{i+1}}, \alpha_i, T)$ to U_{i+1} and $\top$ to U_i if successful or $\bot$ otherwise to the users $\overline{\gamma}_i.\mathsf{users}$. If U_{i+1} is not the receiver, it will continue this process with its own neighbor as shown in the next section.

4.4 Multi-hop payment description

In this section we describe the MHP protocol. The pseudocode for carrying out MHPs in Blitz is shown in Figure 6, the macros used in are listed in Figure 7. For the full description of the macros, see Appendix I.

Setup Say the sender wants to pay α coins to U_n via a path $\mathsf{channelList}$ and for some timeout T. In the setup phase, the sender derives a new stealth address $\mathsf{pk}_{\widetilde{U_i}}$ and some R_i for every user except the receiver. Then, the sender creates a list rList of entries R_i and onions encoding the right neighbor U_{i+1} for every user U_i. Moreover, the sender constructs $\mathsf{tx}^{\mathsf{er}}$. Then, it adds the sum of all per-hop fees to the initial amount α: $\alpha_i := \alpha + (n-1) \cdot \mathsf{fee}$ where fee is the fee charged by every user (see Appendix A). The setup ends when the sender starts the open phase with its right neighbor U_1.

Open After successfully setting up the payment with its left user U_{i-1}, U_i knows $\mathsf{tx}^{\mathsf{er}}$, rList, onion_{i+1} α_{i-1}, T and its stealth output for $\theta_{\varepsilon_i} \in \mathsf{tx}^{\mathsf{er}}.\mathsf{output}$. Using these values and reducing α_{i-1} by fee, U_i carries out the 2-party channel update with U_{i+1}. The right neighbor continues this step with its right neighbor until the receiver is reached.

Finalize Once the receiver has finished the open phase with its left neighbor, it sends back a signature of $\mathsf{tx}^{\mathsf{er}}$ as a confirmation to the sender, who will then check if that transaction was tampered with. If yes, or if the sender did not receive such a confirmation in time, the sender publishes $\mathsf{tx}^{\mathsf{er}}$ on the blockchain. Otherwise the sender goes idle.

Respond At any given time after opening a payment construction, users need to check if $\mathsf{tx}^{\mathsf{er}}$ was published. If it was, they need to refund themselves via $\mathsf{tx}_i^{\mathsf{r}}$. Also, if some user's left neighbor tries to publish $\mathsf{tx}_i^{\mathsf{r}}$ after time T, the user publishes $\mathsf{tx}_i^{\mathsf{p}}$. This ensures, that if the refund did not happen before time T, the users have a way to enforce the payment. Note that due to the relative timelock on both $\mathsf{tx}^{\mathsf{er}}$ and $\mathsf{tx}^{\mathsf{state}}$, $\mathsf{tx}_i^{\mathsf{p}}$ will always be possible if $\mathsf{tx}^{\mathsf{er}}$ is published after T (or if the left neighbor tries to refund after T by closing the channel).

The protocol is shown in Figure 6. Note that we simplified the protocol for readability purposes, (e.g., by omitting the payment ids that are required for multiple concurrent payments). The full protocol modelled in the Universal Composability framework can be seen in Appendix J.5.

5 Security analysis

5.1 Security model

The security model we use closely follows [5, 12, 13]. We model the security of Blitz in the synchronous, global universal composability (GUC) framework [10]. We use a global ledger $\mathcal{L}$ to capture any transfer of coins. The ledger is parameterized by a signature scheme Σ and a blockchain delay Δ, which is an upper bound on the number of rounds it takes between when a transaction is posted to $\mathcal{L}$ and when said transaction is added to $\mathcal{L}$. Our security analysis is fully presented in Appendix J and briefly outlined here.

Firstly, we provide an ideal functionality $\mathcal{F}_{Pay}$, which is an idealized description of the behavior we expect of our pay-or-revoke payment paradigm. This description stipulates any input/output behavior and the impact on the ledger of a payment protocol, as well as how adversaries can influence the execution. In this idealized setting, all parties communicate only with $\mathcal{F}_{Pay}$, which acts as a trusted third party.

We then provide our protocol Π formally defined in the UC framework and show that Π *emulates* $\mathcal{F}_{Pay}$. On a high level, we show that any attack that can be performed on Π can also be simulated on $\mathcal{F}_{Pay}$ or in other words that Π is at least as

```
Setup
$U_0$ upon receiving $(\texttt{setup}, \mathsf{channelList}, \mathsf{tx}^{\mathsf{in}}, \alpha, T)$
1. If $\texttt{checkChannels}(\mathsf{channelList}, U_0) = \bot$, abort.
2. Let $n := |\mathsf{channelList}|$. If $\texttt{checkT}(n, T) = \bot$, abort.
3. If $\texttt{checkTxIn}(\mathsf{tx}^{\mathsf{in}}, n, U_0) = \bot$, abort.
4. $(\mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}) := \texttt{genTxEr}(U_0, \mathsf{channelList}, \mathsf{tx}^{\mathsf{in}})$
5. $\alpha_0 := \alpha + \mathsf{fee} \cdot (n-1)$
6. $(\mathsf{sk}_{\widetilde{U_0}}, \theta_{\varepsilon_0}, R_0, U_1, \mathsf{onion}_1) :=$
   $\texttt{checkTxEr}(U_0, U_0.a, U_0.b, \mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion})$
7. $\texttt{pcSetup}(\overline{\gamma_0}, \mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_1, U_1, \theta_{\varepsilon_0}, \alpha_0, T)$

Open
$U_{i+1}$ upon receiving $(\mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_{i+2}, U_{i+2}, \theta_{\varepsilon_{i+1}}, \alpha_i, T)$
1. If $U_{i+1}$ is the receiver $U_n$, send $(\texttt{confirm}, \sigma_{U_n}(\mathsf{tx}^{\mathsf{er}})) \hookrightarrow U_0$
   and go idle.
2. $\texttt{pcSetup}(\overline{\gamma_{i+1}}, \mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_{i+2}, U_{i+2}, \theta_{\varepsilon_{i+1}}, \alpha_i - \mathsf{fee}, T)$

Finalize
$U_0$: Upon $(\texttt{confirm}, \sigma_{U_n}(\mathsf{tx}^{\mathsf{er}})) \hookleftarrow U_n$, check that $\sigma_{U_n}(\mathsf{tx}^{\mathsf{er}})$ is
$U_n$'s valid signature for the transaction $\mathsf{tx}^{\mathsf{er}}$ created in the Setup
phase. If not, or if $\mathsf{tx}^{\mathsf{er}}$ was changed, or no such confirmation was
received until $T - t_c - 3\Delta$, $\mathsf{publishTx}(\mathsf{tx}^{\mathsf{er}}, \sigma_{U_0'}(\mathsf{tx}^{\mathsf{er}}))$.

Respond (Executed in every round $\tau_x$)
1. If $\tau_x < T - t_c - 2\Delta$ and $\mathsf{tx}^{\mathsf{er}}$ on the blockchain,
   $\mathsf{closeChannel}(\overline{\gamma_i})$ and, after $\mathsf{tx}_i^{\mathsf{state}}$ is accepted on the
   blockchain within at most $t_c$ rounds, wait $\Delta$ rounds.
   Let $\sigma_{\widetilde{U_i}}(\mathsf{tx}_i^{\mathsf{r}})$ be a signature using the secret key $\mathsf{sk}_{\widetilde{U_i}}$.
   $\mathsf{publishTx}(\mathsf{tx}_i^{\mathsf{r}}, (\sigma_{\widetilde{U_i}}(\mathsf{tx}_i^{\mathsf{r}}), \sigma_{U_i}(\mathsf{tx}_i^{\mathsf{r}}), \sigma_{U_{i+1}}(\mathsf{tx}_i^{\mathsf{r}})))$.
2. If $\tau_x > T$, $\overline{\gamma_i}$ is closed and $\mathsf{tx}^{\mathsf{er}}$ and $\mathsf{tx}_i^{\mathsf{state}}$ is on the blockchain,
   but not $\mathsf{tx}_i^{\mathsf{r}}$, $\mathsf{publishTx}(\mathsf{tx}_{i-1}^{\mathsf{p}}, (\sigma_{U_i}(\mathsf{tx}_{i-1}^{\mathsf{p}})))$.
```

Figure 6: The Blitz payment protocol

Macros (see Appendix I)

checkTxIn$(\mathsf{tx}^{\mathsf{in}}, n, U_0)$**:** If $\mathsf{tx}^{\mathsf{in}}$ is well-formed and has enough coins, returns $\top$. **checkChannels**$(\mathsf{channelList}, U_0)$**:** If channelList forms a valid path, returns the receiver U_n, else $\bot$. **checkT**(n, T)**:** If T is sufficiently large, return $\top$. Otherwise, return $\bot$ **genTxEr**$(U_0, \mathsf{channelList}, \mathsf{tx}^{\mathsf{in}})$**:** Generates $\mathsf{tx}^{\mathsf{er}}$ from $\mathsf{tx}^{\mathsf{in}}$ along with a list of values rList to redeem their stealth adresses and an onion containing the routing information. **genState**$(\alpha_i, T, \overline{\gamma_i})$**:** Generates and returns a new channel state carrying transaction $\mathsf{tx}_i^{\mathsf{state}}$ from the given parameters, shown in Figure 4. **checkTxEr**$(U_i, a, b, \mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_i)$**:** Checks if $\mathsf{tx}^{\mathsf{er}}$ is correct, U_i has a stealth address in it and onion_i holds routing information. If unsuccessful, returns $\bot$. If U_i is the receiver, returns $(\top, \top, \top, \top, \top)$. Else, returns $(\mathsf{sk}_{\widetilde{U_i}}, \theta_{\varepsilon_i}, R_i, U_{i+1}, \mathsf{onion}_{i+1})$ containing the output belonging to U_i θ_{ε_i}, the secret key to spend it $\mathsf{sk}_{\widetilde{U_i}}$, the next user and the next onion.

Figure 7: Subprocedures used in the protocol

secure as $\mathcal{F}_{Pay}$. To prove this, we design a simulator $\mathcal{S}$, which translates any attack on the protocol into an attack on the ideal functionality. Then, we show that no PPT *environment* can distinguish between interacting with the real world and interacting with the ideal world. In the real world, the environment sends instructions to a real attacker $\mathcal{A}$ and interacts with Π. In the ideal world, the environment sends attack instructions to $\mathcal{S}$ and interacts with $\mathcal{F}_{Pay}$.

We need to show that the same messages are output in the same rounds and the same transactions are posted on the ledger in the same rounds in both the real and the ideal world, regardless of adversarial presence. To achieve this, the simulator needs to instruct the ideal functionality to output a message whenever one is output in the real protocol and the simulator needs to post the same transactions on the ledger. By achieving this, the environment cannot trivially distinguish between the real and the ideal world anymore just be looking at the messages and transactions as well as their respective timing. Formally, in Appendix J we prove Theorem 1.

Theorem 1. *(informal) Let Σ be a EUF-CMA secure signature scheme. Then, for any ledger delay $\Delta \in \mathbb{N}$, the protocol Π UC-realizes the ideal functionality $\mathcal{F}_{Pay}$.*

5.2 Informal security discussion

Due to space constraints, we only argue informally here why Blitz achieves security and privacy (see Section 4.1). We give a more formal discussion in Appendix K and consider the security against some concrete attacks in Appendix E.

Balance security An honest intermediary will forward a payment to its right neighbor only if first invoked by its left neighbor. If constructed correctly, the refund transactions in both channels depend on $\mathsf{tx}^{\mathsf{er}}$ being published and the timing is identical. Also, the payment transactions have identical conditions in both channels. The only possible way for an intermediary to lose money is, if it were to pay its money to the right neighbor, while the left neighbor refunded. However, if the left neighbor is able to refund, this means that also the intermediary itself can refund. Similarly, if the right neighbor is able to claim the money, the intermediary can also claim it.

Honest sender A sender that does not receive a confirmation of the receiver that it received the money in time, can trigger a refund by publishing $\mathsf{tx}^{\mathsf{er}}$. In the setup phase of the protocol, the sender ensures that there is enough time for this.

Honest receiver The receiver gets the money in exchange for some service. It will wait until being certain that the money will be received before shipping the product. The transaction $\mathsf{tx}^{\mathsf{er}}$ on the blockchain is a proof that a refund has occurred.

Privacy Blitz requires to share with intermediaries $\mathsf{tx}^{\mathsf{er}}$, routing information and the value that is being paid. The transaction $\mathsf{tx}^{\mathsf{er}}$ uses stealth addresses for its outputs and an unlinkable input, thereby granting sender, receiver and path privacy in the honest case, as defined in Section 4.1. As in the LN however, the stronger notion of relationship anonymity [21] does not hold; the payment can be linked by comparing (i) in Blitz, $\mathsf{tx}^{\mathsf{er}}$ and (ii) in the LN, the hash value. In the pessimistic case, the balance is claimed on-chain. In both Blitz and the LN, this breaks sender, path and receiver privacy. We defer the reader to Appendix A for a more detailed discussion on all privacy properties mentioned in this paragraph.

6 Evaluation

In this section, we evaluate the benefits that Blitz offers over the LN. The source code for our simulation is at [1].

Testbed We took a snapshot of the LN graph (October 2020) from https://ln.bigsun.xyz/ containing 11.6k nodes, 6.5k of which have 30.9k active channels with a total capacity of 1166.7 BTC, which account for around 13.2M USD in October 2020. We ignore the nodes without active channels. The initial distribution of the channel balance is unknown. We assume that initially the balance at each channel is available to both users. It is assigned to a user as required by payments in a first come, first serve basis. Naturally, the balance that has already been used and thus assigned to one user in the channel, is not reassigned to the other user. Since we use this strategy consistently throughout all our experiments, this assignment does not introduce any bias in the results.

Simulation setup We discretize the time in rounds and each round represents the collateral time per hop (i.e., 1 day or 144 blocks as in the LN). In such a setting, we simulate payments in batches as follows. Assume that we want to simulate *NPay* payments for an amount of *Amt* and with a failure rate of *FRate*. For that, in a first batch we simulate the *FRate* % of *NPay* payments, where each payment is between two nodes s and r (such that $s \neq r$) selected at random in the graph and routed through the cheapest path according to fees. Moreover, each payment in this batch is disrupted at an intermediary node chosen at random in the path between s and r. Finally, for each payment, some balance is marked to be locked at the channels for a certain number of rounds during the second batch, depending on whether we are evaluating the LN (i.e., staggered rounds) or Blitz (i.e., single round). We model thereby a setting where the network contains locked collateral due to disrupted payments.

After the first batch, we simulate a second one of *NPay* payments over 3 rounds as before, assuming that they are not disrupted (e.g., go over paths of honest nodes). We remark that here each payment may still be unsuccessful because there are not enough unlocked funds in the path between s and r. We focus thus on the effect that staggered vs. constant collateral has in the number of successful payments.

Setting parameters Due to the off-chain nature of the LN, there is no ground truth for payment data, a common limitation in PCN related work. We try to make reasonable assumptions for these unknown parameters in our simulation. We sample the payment amount *Amt* for each payment from the range $[1000, \mathsf{ub}]$. We use a lower bound of 1000, as technically the minimum is 546 satoshis (=1 dust) and we additionally account for fees. We select an upper bound (ub) out of $\{3000, 6000, 9000\}$, which is around 0.1%, 0.2% and 0.3% of the average channel capacity. We consider two different number of payments *NPay*, 78k and 978k. The former corresponds to four payments per active node and per round (ppnpr) modeling a setting with sporadic payments (e.g., a banking system), whereas the latter corresponds to 50 ppnpr, modeling a higher payment frequency (e.g., micropayments).

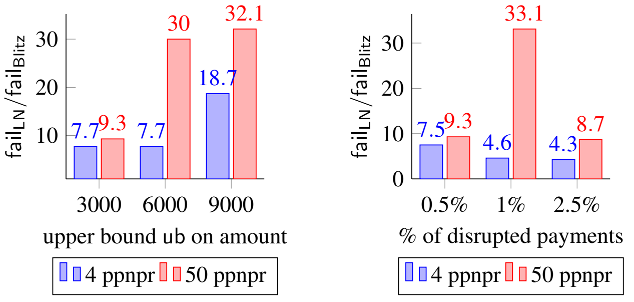


Figure 8: Ratio $\mathsf{fail}_{\mathsf{LN}}/\mathsf{fail}_{\mathsf{Blitz}}$. (Left) we fix the number of disrupted payments at 0.5% and vary ub. (Right) we fix ub at 3000 and vary the number of disrupted payments.

Finally, we vary the amount of disrupted payments *FRate* as $\{0.5, 1, 2.5\}$ % of the total payments *NPay*. We divide these disrupted payments into two groups of equal size. In the first half, the payment is stopped during the setup phase (from s to r). In the LN, the channels before the faulty/malicious node are locked with a staggered collateral lock time. In Blitz, due to the sender publishing $\mathsf{tx}^{\mathsf{er}}$, the funds are immediately unlocked. In the second half, the payment fault occurs in the second phase, which in the LN is the unlocking and in Blitz the fast track. This models the case where a node is offline or an attacker delays the completion of the payment until the last possible moment. In the LN, the collateral left of the malicious node is again staggered, whereas in Blitz the channels right of that node are locked for one simulation round. Finally, we note that distributing the disrupted payments differently into these groups will alter the results accordingly (see Appendix H).

Collateral effect We calculate the number of unsuccessful payments in a baseline case (i.e., omitting the first batch of disrupted payments), in Blitz as well as in the LN and we say that $\mathsf{fail}_{\mathsf{Blitz}}$ (correspondingly $\mathsf{fail}_{\mathsf{LN}}$) is the number of payments that fail in Blitz (correspondingly the LN) when subtracting those failing also in the baseline case. We carry out every experiment for a given setting eight times and calculate the average. In Figure 8 we show the ratio $\mathsf{fail}_{\mathsf{LN}}/\mathsf{fail}_{\mathsf{Blitz}}$. For all choices of parameters, there are more unsuccessful payments in the LN than in Blitz, showing thus the practical advantage of Blitz by requiring only constant collateral. We also observe that difference grows in favor of Blitz with the number of payments, showing that the advantage in terms of collateral is higher in use cases for which initially the LN was designed such as micropayments. Finally, we observe that Blitz offers higher transaction throughput even with an arguably small ratio of disrupted payments (i.e., a reduced adversarial effect).

Wormhole attack We measure an upper bound on the amount of fees potentially at risk in the LN, due to it being prone to the wormhole attack. We observe that the amount of coins at risk grows with the number of payments and their amount. In particular, with 50 ppnpr and an upper bound of

3000 (modeling e.g., a micropayment setting), we observe that the LN put at risk 0.25 BTC (2831 USD in October 2020). Increasing the upper bound to 9000 while keeping 65 ppnpr, we observe that the LN put at risk 0.30 BTC. Blitz prevents the wormhole attack and the stealing of these fees by design.

Computation overhead The Blitz protocol does not require any costly cryptography. In particular, it requires that each user verifies locally the signatures for the involved transactions. Moreover, each user must compute three signatures (see Figure 4) independently on the number of channels involved in the payment. In the LN, each user requires to compute only two signatures, one per each commitment transaction representing the new state. We remark, however, that these are all simple computations that can be executed in negligible time even with commodity hardware.

Communication overhead We find that the contract size in Blitz is 26% smaller than the size of the HTLCs in the LN. This advantage is crucial in practice as current LN payment channels cannot hold more than 483 HTLC (and thus 483 in-flight payments) simultaneously, because otherwise, the size of the off-chain state would be higher than a valid Bitcoin transaction [25, 30]. The reduced communication overhead in Blitz implies then that it allows for more simultaneous in-flight payments per channel than in the LN.

In the pessimistic case, the LN requires to include on-chain one transaction per channel (158 Bytes for refund, 192 Bytes for payment), while Blitz requires not only one on-chain transaction per channel (307 Bytes for refund, 158 Bytes for payment), but also that the sender includes the transaction $\mathsf{tx}^{\mathsf{er}}$ to ensure that the refund is atomic. In this sense, the LN requires a smaller overhead than Blitz for the pessimistic case. We remark that there exist incentives in PCNs for the nodes to follow the optimistic case and reduce entering the pessimistic case because it requires to close the channels and cannot be used for further off-chain payments without re-opening them, with the consequent cost in time and fees. We give detailed results about communication overhead in Appendix G.

7 Related work

PCNs have attracted plenty of attention from academia [14, 20, 22, 23, 29] and have been deployed in practice [24]. These PCNs, with the exception of Interledger [29], follow the 2-phase-commit paradigm and suffer from (some of) the drawbacks we have discussed in this work, namely, prone to the wormhole attack, griefing attacks, staggered collateral or rely on scripting functionality not widely available. Interledger is a 1-phase protocol that however does not provide security.

Sprites [23] is the first multi-hop payment (MHP) that achieves constant collateral. It, however, relies on Turing complete smart contracts (available in, e.g., Ethereum) thereby reducing its applicability in practice. Other constructions that require Turing complete smart contracts, e.g., State channels [12], achieve constant collateral, but have similar privacy issues as the LN when used for MHPs. AMCU [14] achieves constant collateral and is compatible with Bitcoin. AMCU, however, reveals every participant to each other, a privacy leakage undesirable in the MHP setting.

To improve privacy, [21] introduced MHTLCs. In [32], CHTLCs based on Chameleon hash functions were introduced, a functionality that is again not supported in most cryptocurrencies (e.g., in Bitcoin). AMHL [22] replaces the HTLC contract with novel cryptographic locks to avoid the wormhole attack. MHTLC, CHTLC or AMHL based MHPs all follow the 2-phase-commit paradigm and require staggered collateral. We defer to Appendix B for works on 1-phase commits in the context of distributed databases.

8 Conclusion

Payment-channel networks (PCNs) are the most prominent solution to the scalability problem of cryptocurrencies with practical adoption (e.g., the LN). While optimistic 1-round payments (e.g., Interledger) are prone to theft by malicious intermediaries, virtually all PCNs today follow the 2-phase-commit paradigm and are thus prone to a combination of: (i) security issues such as wormhole attacks; (ii) staggered collateral; and (iii) limited deployability as they rely on either HTLC or Turing complete smart contracts.

We find a redundancy implementing a 2-phase-commit protocol on top of the consensus provided by the blockchain and instead design Blitz, a multi-hop payment protocol that demonstrates for the first time that it is possible to have a 1-round payment protocol that is secure, resistant to wormhole attacks by design, has constant collateral, and builds upon digital signatures and timelock functionality from the underlying blockchain's scripting language. Our experimental evaluation shows that Blitz reduces the number of unsuccessful payments by a factor of between 4x and 33x, reduces the size of the payment contract by a 26% and saves up to 0.3 BTC (3397 USD in October 2020) in fees over a three day period as it avoids wormhole attacks by design.

Blitz can be seamlessly deployed as a (additional or alternative) payment protocol in the current LN. We believe that Blitz opens possibilities of performing more efficient and secure payments across multiple different cryptocurrencies and other applications built on top, research directions which we intend to pursue in the near future.

Acknowledgements We thank Lloyd Fournier for his valuable feedback on an earlier version of this work. This work has been supported by the European Research Council (ERC) under the Horizon 2020 research (grant 771527-BROWSEC); by the Austrian Science Fund (FWF) through the projects PROFET (grant P31621), the Meitner program (grant M-2608) and the project W1255-N23; by the Austrian Research Promotion Agency (FFG) through the Bridge-1 project PR4DLT (grant 13808694) and the COMET K1 SBA; by the Vienna Business Agency through the project Vienna Cybersecurity and Privacy Research Center (VISP); by CoBloX Labs; by the National Science Foundation (NSF) under grant CNS-1846316.

References

[1] Blitz simulation: Github repository, 2020. https://github.com/blitz-payments/simulation.

[2] M. Abdallah, R. Guerraoui, and P. Pucheral. One-phase commit: does it make sense? *Conference on Parallel and Distributed Systems*, 1998.

[3] Yousef J. Al-houmaily and Panos K. Chrysanthis. Two-Phase Commit in Gigabit-Networked Distributed Databases. In *Parallel and Distributed Computing Systems*, 1995.

[4] Yousef J Al-Houmaily and Panos K Chrysanthis. 1-2PC: the one-two phase atomic commit protocol. In *Symposium on Applied Computing*, 2004.

[5] Lukas Aumayr, Oguzhan Ersoy, Andreas Erwig, Sebastian Faust, Kristina Hostakova, Matteo Maffei, Pedro Moreno-Sanchez, and Siavash Riahi. Generalized Bitcoin-Compatible Channels. Cryptology ePrint Archive. https://eprint.iacr.org/2020/476.

[6] Christian Badertscher, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. Bitcoin as a Transaction Ledger: A Composable Treatment. In *CRYPTO*, 2017.

[7] Vivek Bagaria, Joachim Neu, and David Tse. Boomerang: Redundancy Improves Latency and Throughput in Payment-Channel Networks. In *FC*, 2020.

[8] Jan Camenisch and Anna Lysyanskaya. A Formal Treatment of Onion Routing. In *CRYPTO*, 2005.

[9] Ran Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *FOCS*, 2001.

[10] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. Universally Composable Security with Global Setup. In *TCC*, 2007.

[11] G. Danezis and I. Goldberg. Sphinx: A Compact and Provably Secure Mix Format. In *IEEE S&P*, 2009.

[12] Stefan Dziembowski, Lisa Eckey, Sebastian Faust, Julia Hesse, and Kristina Hostáková. Multi-party Virtual State Channels. In *EUROCRYPT*, 2019.

[13] Stefan Dziembowski, Lisa Eckey, Sebastian Faust, and Daniel Malinowski. PERUN: Virtual Payment Channels over Cryptographic Currencies. In *IEEE S&P*, 2019.

[14] Christoph Egger, Pedro Moreno-Sanchez, and Matteo Maffei. Atomic Multi-Channel Updates with Constant Collateral in Bitcoin-Compatible Payment-Channel Networks. In *CCS*, 2019.

[15] Shafi Goldwasser, Silvio Micali, and Ronald L Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on computing*, 1988.

[16] Lewis Gudgeon, Pedro Moreno-Sanchez, Stefanie Roos, Patrick McCorry, and Arthur Gervais. SoK: Off The Chain Transactions. In *FC*, 2020.

[17] Rachid Guerraoui and Jingjing Wang. How Fast can a Distributed Transaction Commit? In *PODS*, 2017.

[18] Maurice Herlihy, Liuba Shrira, and Barbara Liskov. Cross-chain Deals and Adversarial Commerce. *VLDB*, 2019.

[19] Jonathan Katz and Ueli Maurer and Björn Tackmann and Vassilis Zikas. Universally Composable Synchronous Computation. In *TCC*, 2013.

[20] Aggelos Kiayias and Orfeas Stefanos Thyfronitis Litos. A Composable Security Treatment of the Lightning Network. In *CSF*, 2019.

[21] Giulio Malavolta, Pedro Moreno-Sanchez, Aniket Kate, Matteo Maffei, and Srivatsan Ravi. Concurrency and Privacy with Payment-Channel Networks. In *CCS*, 2017.

[22] Giulio Malavolta, Pedro Moreno-Sanchez, Clara Schneidewind, Aniket Kate, and Matteo Maffei. Anonymous Multi-Hop Locks for Blockchain Scalability and Interoperability. In *NDSS*, 2019.

[23] Andrew Miller, Iddo Bentov, Ranjit Kumaresan, and Patrick McCorry. Sprites: Payment Channels that Go Faster than Lightning. In *FC*, 2019.

[24] Joseph Poon and Thaddeus Dryja. The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments.

[25] EmelyanenkoK (pseudonym). Payment channel congestion via spam-attack. https://github.com/lightningnetwork/lightning-rfc/issues/182.

[26] Stefanie Roos, Pedro Moreno-Sanchez, Aniket Kate, and Ian Goldberg. Settling Payments Fast and Private: Efficient Decentralized Routing for Path-Based Transactions. In *NDSS*, 2018.

[27] Vibhaalakshmi Sivaraman, Shaileshh Bojja Venkatakrishnan, Kathleen Ruan, Parimarjan Negi, Lei Yang, Radhika Mittal, Giulia C. Fanti, and Mohammad Alizadeh. High Throughput Cryptocurrency Routing in Payment Channel Networks. In *NSDI*, 2020.

[28] James W Stamos and Flaviu Cristian. Coordinator log transaction execution protocol. *Distributed and Parallel Databases*, 1993.

[29] Stefan Thomas and Evan Schwartz. A protocol for interledger payments, 2015. https://interledger.org/interledger.pdf.

[30] Sergei Tikhomirov, Pedro Moreno-Sanchez, and Matteo Maffei. A Quantitative Analysis of Security, Anonymity and Scalability for the Lightning Network. In *IEEE S&B Workshop*, 2020.

[31] Nicolas Van Saberhagen. Cryptonote v 2.0, 2018. https://cryptonote.org/whitepaper.

[32] Bin Yu, Shabnam Kasra Kermanshahi, Amin Sakzad, and Surya Nepal. Chameleon Hash Time-lock Contract for Privacy Preserving Payment Channel Networks. In *Conference on Provable Security*, 2019.

[33] Alexei Zamyatin, Mustafa Al-Bassam, Dionysis Zindros, Eleftherios Kokoris-Kogias, Pedro Moreno-Sanchez, Aggelos Kiayias, and William J. Knottenbelt. SoK: Communication Across Distributed Ledgers. In *FC*, 2021.

A Discussion on practical deployment

Payment fees We encode a fee mechanism in our construction. For simplicity, we assume that every intermediary charges the same fee amount: fee. However, it is trivial to extend this mechanism to allow for different fees. The sender initially puts an amount $\alpha_0 := \alpha + \mathsf{fee} \cdot (n-1)$ in the output $\theta_{i,0}$. Every intermediary now deducts fee from this amount when opening the construction with its own right neighbor. Specifically, an intermediary U_i receives α_{i-1} and forwards only $\alpha_i := \alpha_{i-1} - \mathsf{fee}$. Thereby, every intermediary effectively gains fee coins in the case of a successful payment.

Refund tradeoff In the case of a refund, where a fast refund (see Section 3) is not possible, the sender has to publish $\mathsf{tx}^{\mathsf{er}}$. Doing this will have the cost of publishing this transaction (and possibly the transaction containing its input) plus the $(n-1) \cdot \varepsilon$ that go to the intermediaries. The amount ε can be the smallest possible amount of cash, since it is just used to enable the payment. In other words, for Bitcoin we can say $\varepsilon := 1$ satoshi,[1] which is currently around 0.00011 USD. However, the refund of Blitz payments has a fundamental advantage over the one in the Lightning Network (LN). The refund time is only consant in the worst case and if the sender is honest, is only the time it takes to publish $\mathsf{tx}^{\mathsf{er}}$ (i.e., Δ) instead of $n \cdot \xi$. We presented this advantage in Section 3.

So the tradeoff is a more expensive, but much faster refund. This immensely reduces the effect of griefing attacks and increases the overall transaction throughput.

Race We already mentioned that only the sender can publish $\mathsf{tx}^{\mathsf{er}}$ and because of the time delays, the timing is the same for every user on the path. We claimed that the latest possible time to safely publsih $\mathsf{tx}^{\mathsf{er}}$ and still be able to claim the refund is $T - t_{\mathsf{c}} - 3\Delta$. However, there is a time frame after $T - t_{\mathsf{c}} - 3\Delta$ up until $T - t_{\mathsf{c}} - 2\Delta$, where the sender could publish $\mathsf{tx}^{\mathsf{er}}$ and still, $\mathsf{tx}_i^{\mathsf{r}}$ would be sent to the ledger before time T. However now, everyone is at risk, because we said that accepting a transaction takes at most Δ time and at time T, already $\mathsf{tx}_i^{\mathsf{p}}$ might be sent to the ledger and there might be a race over which of these two transactions is accepted first. We argue, that a sender will not do this, as this puts himself at the same risk as ever other intermediary. For a way of preventing this race entirely, we defer the reader to Appendix D.

Obfuscate the length of the path By adding additional dummy outputs (that belong to fresh addresses of the sender) to $\mathsf{tx}^{\mathsf{er}}$, a sender can obfuscate the path length. Note that the rList has to include some random values as well, so that it has the same number of elements as $\mathsf{tx}^{\mathsf{er}}$ has outputs. Note that by looking at the time lock in the LN, the path length or at least ones position within the path is leaked to some degree.

Extended privacy discussion As mentioned in Section 4.1, Blitz achieves sender, receiver and path privacy, which provide a measure of privacy in the case of a successful payment. To hide the path from users observing $\mathsf{tx}^{\mathsf{er}}$, we use stealth addresses for the outputs of $\mathsf{tx}^{\mathsf{er}}$. This allows to have path privacy as defined in Section 4.1, where malicious intermediaries cannot determine the participants of the payment other than their direct neighbors. We stress that as in the LN, the stronger notion of relationship anonymity [21] does not hold. Two users can link a payment by comparing the transaction $\mathsf{tx}^{\mathsf{er}}$ in Blitz, or the hash value in the LN.

To make an on-chain linking of the sender impossible, we require the input of $\mathsf{tx}^{\mathsf{er}}$ to be fresh and unlinkable to the sender. In practice, this can be achieved as follows. The sender creates off-chain an intermediary transaction $\mathsf{tx}^{\mathsf{in}}$ that spends from an output under the sender's control $\mathsf{tx}^{\mathsf{sdr}}$ to a newly generated address of the sender, never used before. Then, $\mathsf{tx}^{\mathsf{er}}$ uses this output with the new address of $\mathsf{tx}^{\mathsf{in}}$ as input. Since $\mathsf{tx}^{\mathsf{in}}$ is off-chain, users observing $\mathsf{tx}^{\mathsf{er}}$ are unable to link the payment to an on-chain identity. Again, this is due to inputs referring to a transaction hash plus an id of the output.

In the pessimistic case, these properties do not hold anymore. If the transactions go on-chain, they can be linked together by observing a shared transaction $\mathsf{tx}^{\mathsf{er}}$ or time T. The same holds true in the LN, where transactions that spend from an HTLC with the same hash value, can be linked.

Redundancy for improving throughput and latency Routing a payment through a path can fail or be delayed due to unknown channel balances, offline or malicious users or other reasons. Following Boomerang [7], a sender can construct several redundant payments across several paths, that differ in one or more users. For this, the sender creates a transaction $\mathsf{tx}^{\mathsf{er}}$ for each of these redundant payments and forwards them. Intermediary users have to open a payment construction (build $\mathsf{tx}_i^{\mathsf{r}}$ and $\mathsf{tx}_i^{\mathsf{p}}$) for every $\mathsf{tx}^{\mathsf{er}}$ that they receive.

Should an intermediary user have a choice of forwarding a payment to several different neighbors, it can choose one and start a fast refund (Section 3) for the other payments. Should several different payments reach the receiver, it can start the fast refund for all but one of them. In the worst case, if the sender sees that after some time more than one payment is active, it can start the refund by publishing the according transaction $\mathsf{tx}^{\mathsf{er}}$. With this, the sender can ensure that at most one of the redundant payments is carried out. This technique is useful to improve transaction throughput and latency and we achieve it without any additional cryptography.

Concurrent payments Two parties of a payment channel can achieve concurrent payments as follows. They agree to update their current channel state $\mathsf{tx}_i^{\mathsf{state}}$ to a new state $\mathsf{tx}_i^{\mathsf{state}'}$, where any unresolved in-flight Blitz payments are carried over. More concretely, for every unresolved payment the transactions $\mathsf{tx}_i^{\mathsf{r}}$ and $\mathsf{tx}_i^{\mathsf{p}}$ are recreated, but the input for these transactions is changed from using an output of $\mathsf{tx}_i^{\mathsf{state}}$ to using an output of $\mathsf{tx}_i^{\mathsf{state}'}$. Afterwards, the right user's signature

[1] In practice, Bitcoin transactions need to carry a total amount of one dust, which is 546 satoshis. Having individual outputs of one satoshi is not a problem, as the sender can include an additional output to a stealth address under its control, such that the sum is greater than one dust. In $\mathsf{tx}_i^{\mathsf{r}}$, the output of $\mathsf{tx}^{\mathsf{er}}$ holding one satoshi is combined with the first output of the state $\mathsf{tx}^{\mathsf{state}}$, resulting in a sum larger than one dust.

for $\mathsf{tx}^{\mathsf{r}}_i$ is given to the left user and only then, the old state $\mathsf{tx}^{\mathsf{state}}_i$ is revoked using the revocation technique in the LN (outlined in Appendix C). In other words, the same channel state-management of the LN is reused in Blitz, but changing the HTLC contract for the Blitz contract. We show an illustrative example of concurrent payments in Figure 9.

B 1-phase commits in distributed databases

The concepts of 1-phase commits [2, 3, 28] and one-two commit [4] have been studied for distributed databases in general. These protocols introduce recovery mechanisms such as coordinator Log [28], implicit Yes-Vote [3] or logical logging [2] towards avoiding the voting/commit/prepare phase of 2-phase commits. However, extending observation by Herlihy, Liskov, and Shrira [18], traditional 1-phase commit ideas are not directly applicable to PCNs: while PCNs (with blockchain-based conflict resolution) are structurally similar to transactions over distributed database, they are fundamentally different in terms of the ACID properties and the adversarial assumptions. Nevertheless, analyses such as [17] can still be interesting to understand lower-bounds for PCNs.

C Payment channels in more detail

In this section, we give a more detailed account on payment channels. A payment channel is used by two parties P and Q to perform several payments between them while requiring only two on-chain transactions. It is set up by two parties spending some coins to a shared multisig output (i.e., an output θ with $\theta.\phi := \mathsf{MultiSig}(P,Q)$). Before signing and publishing this transaction however, they create transactions (so called *commitment transactions* tx^{c}) that spend this shared output in some way, e.g., giving each party some balance. We also refer to this as the (current) *state* of the channel. Now after publishing this tx^{f} on-chain, they can update their balances by creating new commitment transactions tx^{c}, rebalancing the funds of the channel and thereby carrying out payments. We note that there are implementations that use two commitment transactions per state (in other words, one per party) such as the Lightning Network (LN) [24] whereas a more recent construction called generalized channels [5] requires one commitment transaction per state. In this work, we leverage the latter construction, although other ledger channel protocols such as the one of the LN would work as well.

After a channel has been updated several times, there exist several tx^{c} that can be published. In order to prevent misbehavior, where one party publishes an older state of the channel, which perhaps is financially more advantageous to it, we employ a punishment mechanism. If an old state is published, the other, honest user can carry out this punishment to gain all funds of the channel. For this to work, both parties exchange revocation secrets every time a state is succeeded by a new one. This secret, together with the outdated tx^{c} that is published by the misbehaving user is enough to claim all funds of the channel. The latest state can always be safely published as the corresponding revocation secret was not yet revealed. This mechanism provides an economical incentive not to publish an old tx^{c}.

To close a payment channel, the parties can merely publish the latest tx^{c} to the ledger, which terminates the channel. In summary, two parties can use a payment channel to carry out arbitrary many off-chain payments that rebalance some funds, but only need to publish two transactions on the blockchain, one to open the channel and one to close it, saving both fees and increasing the cryptocurrency's transaction throughput.

D Preventing the race condition when the sender is irrational

We mentioned in Appendix A, that if the sender posts $\mathsf{tx}^{\mathsf{er}}$ after $T - t_{\mathsf{c}} - 3\Delta$ and before $T - t_{\mathsf{c}} - 2\Delta$, there is an unwanted race condition. We argue, that this race condition is also unwanted by the sender, but a small tweak to the protocol allows us to prevent it completely. We need to tweak the spending condition of each output θ_{ϵ_i} in $\mathsf{tx}^{\mathsf{er}}$ from $\mathsf{RelTime}(t_{\mathsf{c}} + \Delta) \wedge \mathsf{OneSig}(\widetilde{U_i})$ to $(\mathsf{RelTime}(t_{\mathsf{c}} + \Delta) \wedge \mathsf{OneSig}(\widetilde{U_i})) \vee \mathsf{AbsTime}(T - 2\Delta)$. So in other words, we introduce an additional way of spending the outputs: Anyone can spend any or all outputs of $\mathsf{tx}^{\mathsf{er}}$ if they are still unspent by the time $T - 2\Delta$.[2]

Having this condition ensures that there is no race. If the sender publishes $\mathsf{tx}^{\mathsf{er}}$ such, that the relative timelock of the outputs expires only after $T - 2\Delta$, anyone, especially an observant intermediary, can claim all or at least its left neighbor's output to prevent a refund. And if the sender publishes $\mathsf{tx}^{\mathsf{er}}$ such, that the relative timelock expires before $T - 2\Delta$, then there is no race between the refund and the payment.

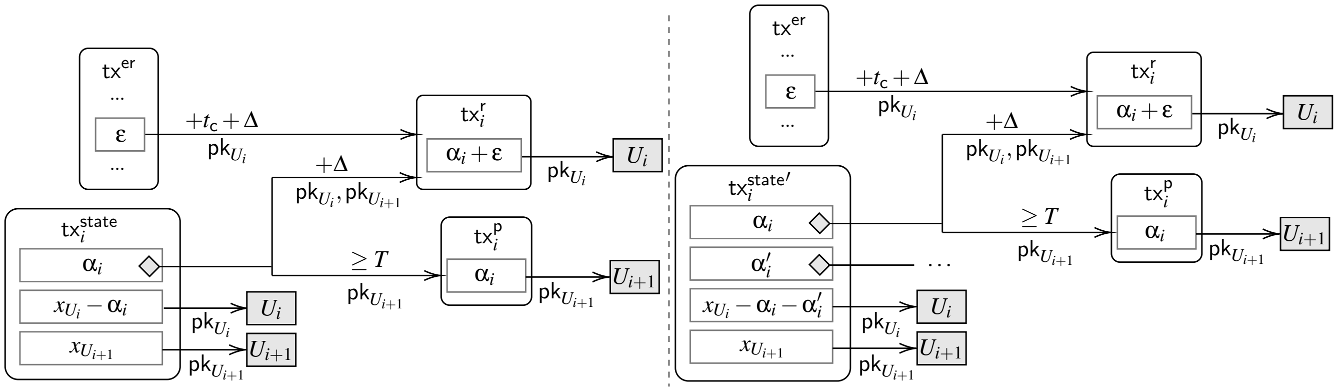
E Concrete attack scenarios (informal)

In this section, we consider some attacks against Blitz and argue informally, why balance security still holds.

$\mathsf{tx}^{\mathsf{er}}$ is tampered If $\mathsf{tx}^{\mathsf{er}}$ is tampered by some intermediary, the next intermediary will see that the message embedded in the routing information is not $\mathcal{H}(\mathsf{tx}^{\mathsf{er}})$ anymore. Assuming that a malicious intermediary does not know the routing information especially not the receiver, changing the routing information will result in the receiver not being reached.

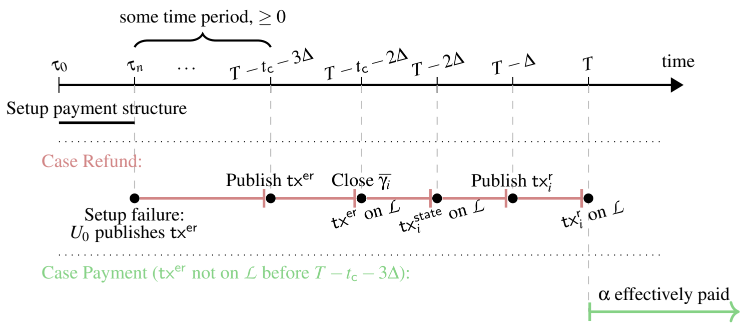
Also, note that balance security holds even in the case where $\mathsf{tx}^{\mathsf{er}}$ is tampered, as long as every intermediary U_i makes sure, that its refund $\mathsf{tx}^{\mathsf{r}}_i$ depends on the same $\mathsf{tx}^{\mathsf{er}}$ as the refund of its neighbor $\mathsf{tx}^{\mathsf{r}}_{i-1}$. Also note, that intermediaries have to ensure the same for the time T, in order to have the same time as their neighbor. Should an intermediary change the time T to a smaller value, it potentially only hurts itself by not being able to refund in time, while its left neighbor actually is. If the time T is changed to a larger value, this may

[2] In Bitcoin, an output can also be made spendable by anyone by putting OP_TRUE or requiring a signature that verifies under the private key 0x1, known by everybody.



$\mathsf{tx}^{\mathsf{er}}$
ε
$+t_c + \Delta$
pk_{U_i}
$\mathsf{tx}_i^{\mathsf{r}}$
$\alpha_i + \varepsilon$
U_i
$+\Delta$
$\mathsf{pk}_{U_i}, \mathsf{pk}_{U_{i+1}}$
$\mathsf{tx}_i^{\mathsf{state}}$
α_i
$x_{U_i} - \alpha_i$
$x_{U_{i+1}}$
$\geq T$
$\mathsf{pk}_{U_{i+1}}$
$\mathsf{tx}_i^{\mathsf{p}}$
U_{i+1}
$\mathsf{tx}_i^{\mathsf{state}'}$
α_i'
$x_{U_i} - \alpha_i - \alpha_i'$

Figure 9: Concurrent payments between users U_i and U_{i+1}: (left) a Blitz channel with a single payment; (right) an updated channel that has this payment and a second concurrent one. To add a second payment of value α_i' to the channel, the transactions for the in-flight payment of value α_i are recreated with the new state $\mathsf{tx}_i^{\mathsf{state}'}$ as input, the channel is updated to $\mathsf{tx}_i^{\mathsf{state}'}$ and finally, the old state $\mathsf{tx}_i^{\mathsf{state}}$ is revoked. In the LN, this process is the same, except that the HTLC contract and transactions are recreated, instead of the Blitz ones.



some time period, ≥ 0
τ_0
τ_n
$T - t_c - 3\Delta$
$T - t_c - 2\Delta$
$T - 2\Delta$
$T - \Delta$
T
time
Setup payment structure
Case Refund:
Setup failure:
U_0 publishes $\mathsf{tx}^{\mathsf{er}}$
Publish $\mathsf{tx}^{\mathsf{er}}$
Close $\overline{\gamma}_i$
Publish $\mathsf{tx}_i^{\mathsf{r}}$
$\mathsf{tx}^{\mathsf{er}}$ on $\mathcal{L}$
$\mathsf{tx}_i^{\mathsf{state}}$ on $\mathcal{L}$
$\mathsf{tx}_i^{\mathsf{r}}$ on $\mathcal{L}$
Case Payment ($\mathsf{tx}^{\mathsf{er}}$ not on $\mathcal{L}$ before $T - t_c - 3\Delta$):
α effectively paid

Figure 10: Timeline of when transactions appear on the ledger $\mathcal{L}$ in the case payment and refund. $\tau_n - \tau_0$ denotes the time needed for the setup of the whole payment.

delay the execution of the payment, however it is detectable, if the receiver sends this time T back to the sender, who can check if it was tampered.

Some users are skipped (wormhole) Users cannot be skipped, as the routing information can only be opened by the next user. A malicious user would not know the receiver and would not be able to forge the sender's signature of $\mathcal{H}(\mathsf{tx}^{\mathsf{er}})$ that is embedded as a message to the receiver in this onion. The only thing for the malicious user is to stop forwarding the payment (griefing attack). Users that are skipped in the fast-track payment will not be cheated out of their fees or funds, rather this money will be locked until at most until T instead of being accessible immediately (see Section 3).

Sender publishes $\mathsf{tx}^{\mathsf{er}}$ after starting fast track Assume a malicious sender started the fast track with its neighbor, but the fast track updates have not yet reached the receiver. Should the sender now publish $\mathsf{tx}^{\mathsf{er}}$, the intermediaries that did not yet perform the fast track will refund. The receiver will say that it did not receive the money and will not ship the promised product. The sender cannot prove that the receiver got the money, even though it has the payment confirmation in form of the receiver's signature of $\mathsf{tx}^{\mathsf{er}}$. The transaction $\mathsf{tx}^{\mathsf{er}}$ on the blockchain is a proof of revocation, and the sender will have lost its money without getting anything in return. The sender should thus not publish $\mathsf{tx}^{\mathsf{er}}$ after starting the fast track.

F Timeline

We show a timeline of posting the transaction of the Blitz payment construction between two users in Figure 10. Red shows the refund case, green the payment case.

G Communication overhead

To evaluate our payment scheme, we created an implementation that creates the transactions necessary for setting up the payment. The source code is publicly available at `https://github.com/blitz-payments/overhead`. We tested the compatibility by deploying the transactions on the Bitcoin testnet and checking if the transactions achieve our intended functionalities. Furthermore, we measured the transaction sizes in Bytes and compare them to multi-hop payments

Table 3: Communication overhead of the LN and Blitz. The pessimistic transactions are on-chain, the rest off-chain.

Cases	LN		Blitz	
	# txs	size	# txs	size
Pay (pessimistic)	1	192	1	158
Refund (pessimistic) per channel	1	158	1	307
Additional pess. refund cost for sender	0	0	1	$157+34\cdot n$
Cost of p in-flight payments	1	$225+119\cdot p$	1	$225+88\cdot p$

in the Lightning Network (LN) in a case-by-case analysis.

We present the number of transactions and their sizes for the different sizes in Table 3. Note that the size of the contract in our construction is only 88 Bytes compared to the 119 of the HTLC, a difference mostly due to the part of the script that verifies the hash pre-image. This means, that state transactions holding several different in-flight payments, which directly implement the contract in their outputs, can hold around 26% more Blitz payments than LN payments. For one payment, this difference results in a state of size 311 Bytes for Blitz and a state of 345 Bytes for the LN. In Blitz, additionally to the state we require the refund transaction to be exchanged, which is 307 Bytes, resulting in 618 Bytes for a 2-party setup.

For the rest of the cases, the Blitz payments and the LN payments are similar. In the pessimistic case, both Blitz and the LN require to publish one transaction (after closing the channel) per disputed channel. In the pessimistic refund case, the it is 158 Bytes in the LN and 307 Bytes in Blitz, due to the additional signature of the input spending from $\mathsf{tx}^{\mathsf{er}}$. In the pay case it is 192 Bytes in the LN and 158 Bytes in Blitz, due to the additional hash in the LN. The most notable difference in comparing the transaction overhead comes from the fact that in the Blitz payment, the sender has to publish $\mathsf{tx}^{\mathsf{er}}$ in the pessimistic refund case, which is a total of $157+34\cdot(n)$ Bytes, for a payment path of length $n+1$. However, in the LN there is an additional communication overhead of sending the hash pre-image of 32 Bytes per channel back in the open phase.

H Extended simulation results

In this section, we include results for the simulation when we do not distribute the disrupted payments equally between the two types. As expected, letting 75% of the disrupted payments be of the second type is more favorable for Blitz, while having 25% is less favoring than dividing equally. We show the results in Table 4.

I Extended macros

In this section, we give concrete pseudo-code for the used subprocedures.

Subprocedures

```
checkTxIn($\mathsf{tx}^{\mathsf{in}}, n, U_0$):

1. Check that $\mathsf{tx}^{\mathsf{in}}$ is a transaction on the ledger $\mathcal{L}$.
```

Table 4: Extended results of our simulation.

ub	*FRate*	ppnpr	$\mathsf{fail}_{\mathsf{Blitz}}$	$\mathsf{fail}_{\mathsf{LN}}$	ratio
25% disrupted type 1, 75% type 2					
3000	0.5%	4	4	33	8.25
3000	0.5%	50	13	4343	334.08
3000	1%	4	15	56	3.73
3000	1%	50	751	32807	43.68
3000	2.5%	4	28	182	6.50
3000	2.5%	50	1076	77213	71.76
75% disrupted type 1, 25% type 2					
3000	0.5%	4	18	31	1.72
3000	0.5%	50	505	4422	8.76
3000	1%	4	19	61	3.21
3000	1%	50	1458	33386	22.90
3000	2.5%	4	78	195	2.50
3000	2.5%	50	15427	77574	5.03

```
2. If $\mathsf{tx}^{\mathsf{in}}.\mathsf{output}[0].\mathsf{cash} \geq n \cdot \varepsilon$ and $\mathsf{tx}^{\mathsf{in}}.\mathsf{output}[0].\phi = \mathsf{OneSig}(U_0')$, that is spendable by an unused address of $U_0$,
   return $\top$. Otherwise, return $\bot$. When using this transaction (to fund $\mathsf{tx}^{\mathsf{er}}$), the sender will pay any superfluous coins back to a
   fresh address of itself.

checkChannels($\mathsf{channelList}, U_0$):

Check that $\mathsf{channelList}$ forms a valid path from $U_0$ via some intermediaries to a receiver $U_n$ and that no users are in the path
twice. If not, return $\bot$. Else, return $U_n$.

checkT($n, T$):

Let $\tau$ be the current round. If $T \geq \tau + n(2+t_{\mathsf{u}}) + 3\Delta + t_{\mathsf{c}} + 1$, return $\top$. Otherwise, return $\bot$

genTxEr($U_0, \mathsf{channelList}, \mathsf{tx}^{\mathsf{in}}$):

1. Let $\mathsf{outputList} := \emptyset$ and $\mathsf{rList} := \emptyset$
2. For every channel $\gamma_i$ in $\mathsf{channelList}$:
   • $(\mathsf{pk}_{\widetilde{U_i}}, R_i) \leftarrow \mathsf{GenPk}(\gamma_i.\mathsf{left}.A, \gamma_i.\mathsf{left}.B)$
   • $\mathsf{outputList} := \mathsf{outputList} \cup (\varepsilon, \mathsf{OneSig}(\mathsf{pk}_{\widetilde{U_i}}) \wedge \mathsf{RelTime}(t_{\mathsf{c}} + \Delta))$
   • $\mathsf{rList} := \mathsf{rList} \cup R_i$
3. Let $\mathcal{P} := \{\gamma_i.\mathsf{left}, \gamma_i.\mathsf{right}\}_{\gamma_i \in \mathsf{channelList}}$ and let $\mathsf{nodeList}$ be a list, where $\mathcal{P}$ is sorted from sender to receiver. Let $n := |\mathcal{P}|$.
4. Shuffle $\mathsf{outputList}$ and $\mathsf{rList}$.
5. Let $\mathsf{tx}^{\mathsf{er}} := (\mathsf{tx}^{\mathsf{in}}.\mathsf{output}[0], \mathsf{outputList})$
6. Create a list $[\mathsf{msg}_i]_{i \in [0,n]}$, where $\mathsf{msg}_i := \mathcal{H}(\mathsf{tx}^{\mathsf{er}})$
7. $\mathsf{onion} \leftarrow \mathsf{CreateRoutingInfo}(\mathsf{nodeList}, [\mathsf{msg}_i]_{i \in [0,n]})$
8. Return $(\mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion})$

genState($\alpha_i, T, \overline{\gamma_i}$):

1. For the users $U_i := \overline{\gamma_i}.\mathsf{left} =$ and $U_{i+1} := \overline{\gamma_i}.\mathsf{right}$, create the output vector $\vec{\theta}_i := (\theta_0, \theta_1, \theta_2)$, where
   • $\theta_0 := (\alpha_i, (\mathsf{MultiSig}(U_i, U_{i+1}) \wedge \mathsf{RelTime}(T)) \vee (\mathsf{OneSig}(U_{i+1}) \wedge \mathsf{AbsTime}(T)))$
   • $\theta_1 := (x_{U_i} - \alpha_i, \mathsf{OneSig}(U_i))$
   • $\theta_2 := (x_{U_{i+1}}, \mathsf{OneSig}(U_{i+1}))$
```

```
    where $x_{U_i}$ and $x_{U_{i+1}}$ is the amount held by $U_i$ and $U_{i+1}$ in the
    channel, respectively.
2. Let $\mathsf{tx}_i^{\mathsf{state}}$ be a channel transaction carrying the state with
   $\mathsf{tx}^{\mathsf{state}}.\mathsf{output} = \vec{\theta}_i$. Return $\mathsf{tx}_i^{\mathsf{state}}$.
```

```
checkTxEr($U_i, a, b, \mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_i$):
1. $x := \mathsf{GetRoutingInfo}(\mathsf{onion}_i, U_i)$. If $x = \perp$, return $\perp$. If $U_i$ is
   the receiver and $x = \mathcal{H}(\mathsf{tx}^{\mathsf{er}})$ , return $(\top,\top,\top,\top,\top)$. Else, if
   $x \neq (U_{i+1}, \mathcal{H}(\mathsf{tx}^{\mathsf{er}}), \mathsf{onion}_{i+1})$, return $\perp$.
2. For all outputs $(\mathsf{cash}, \phi) \in \mathsf{tx}^{\mathsf{er}}.\mathsf{output}$ it must hold that:
   • $\mathsf{cash} = \varepsilon$
   • $\phi = \mathsf{OneSig}(\mathsf{pk}_x) \wedge \mathsf{RelTime}(t_{\mathsf{c}} + \Delta)$ for some identity $\mathsf{pk}_x$
3. For exactly one output $\theta_{\varepsilon_i} := (\varepsilon, \mathsf{OneSig}(\widetilde{U_i}) \wedge$
   $\mathsf{RelTime}(t_{\mathsf{c}} + \Delta)) \in \mathsf{tx}^{\mathsf{er}}.\mathsf{output}$ and one element $R_i \in \mathsf{rList}$ it
   must hold that
      • Let $\mathsf{pk}_{\widetilde{U_i}}$ be the corresponding public key of $\mathsf{OneSig}(\widetilde{U_i})$
      • $\mathsf{sk}_{\widetilde{U_i}} := \mathsf{GenSk}(a, b, \mathsf{pk}_{\widetilde{U_i}}, R_i)$ must be the corresponding
        secret key of $\mathsf{pk}_{\widetilde{U_i}}$
4. If the checks in 2 or 3 do not hold, return $\perp$
5. Return $(\mathsf{sk}_{\widetilde{U_i}}, \theta_{\varepsilon_i}, R_i, U_{i+1}, \mathsf{onion}_{i+1})$
```

Subprocedures used exclusively in UC model

```
createMaps($U_0, \mathsf{nodeList}, \mathsf{tx}^{\mathsf{in}}$):
1. For every $U_i \in \mathsf{nodeList} \setminus U_n$ do:
   • $(\mathsf{pk}_{\widetilde{U_i}}, R_i) \leftarrow \mathsf{GenPk}(U_i.A, U_i.B)$
   • $\mathsf{outputMap}(U_i) := (\varepsilon, \mathsf{OneSig}(\mathsf{pk}_{\widetilde{U_i}}) \wedge \mathsf{RelTime}(t_{\mathsf{c}} + \Delta))$
   • $\mathsf{rMap}(U_i) := R_i$
2. $\mathsf{rList} = \mathsf{rMap}.\mathtt{values().shuffle()}$
3. $\mathsf{tx}^{\mathsf{er}} := (\mathsf{tx}^{\mathsf{in}}.\mathsf{output}[0], \mathsf{outputMap}.\mathtt{values().shuffle()})$
4. Create a map stealthMap that stores for every user $U_i$ that is
   a key in outputMap the corresponding output of $\mathsf{tx}^{\mathsf{er}}$ corre-
   sponding to $\mathsf{outputMap}(U_i)$
5. Create two empty lists $\emptyset$ named msgList, userList
6. For every $U_i \in \mathsf{nodeList}$ from $U_n$ to $U_0$ (in descending order):
   • Append $[\mathcal{H}(\mathsf{tx}^{\mathsf{er}})]$ to msgList
   • Prepend $[U_i]$ to userList.
   • $\mathsf{onion}_i := \mathsf{CreateRoutingInfo}(\mathsf{userList}, \mathsf{msg})$
   • $\mathsf{onions}(U_i) := \mathsf{onion}_i$
7. Return $(\mathsf{tx}^{\mathsf{er}}, \mathsf{onions}, \mathsf{rMap}, \mathsf{rList}, \mathsf{stealthMap})$
```

```
genStateOutputs($\overline{\gamma_i}, \alpha_i, T$):
1. Let $\vec{\theta}'_i := \overline{\gamma_i}.\mathsf{st}$ be the current state of the channel $\overline{\gamma_i}$.
2. Let $U_i := \overline{\gamma_i}.\mathsf{left} =$ and $U_{i+1} := \overline{\gamma_i}.\mathsf{right}$.
3. $\vec{\theta}'_i$ consists of the outputs $\theta'_{U_i} := (x_{U_i}, \mathsf{OneSig}(U_i))$ and
   $\theta'_{U_{i+1}} := (x_{U_{i+1}}, \mathsf{OneSig}(U_{i+1}))$ holding the balances of the two
   users.[a] If $x_{U_i} < \alpha_i$, return $\perp$
4. Create the output vector $\vec{\theta}_i := (\theta_0, \theta_1, \theta_2)$, where
   • $\theta_0 := (\alpha_i, (\mathsf{MultiSig}(U_i, U_{i+1}) \wedge \mathsf{RelTime}(T)) \vee$
     $(\mathsf{OneSig}(U_{i+1}) \wedge \mathsf{AbsTime}(T)))$
   • $\theta_1 := (x_{U_i} - \alpha_i, \mathsf{OneSig}(U_i))$
   • $\theta_2 := (x_{U_{i+1}}, \mathsf{OneSig}(U_{i+1}))$
5. Return $\vec{\theta}_i$.
```

```
genRefTx($\theta, \theta_{\varepsilon_i}, U_i$):
1. Create a transaction $\mathsf{tx}_i^{\mathsf{r}}$ with $\mathsf{tx}_i^{\mathsf{r}}.\mathsf{input} := [\theta, \theta_{\varepsilon_i}]$ and
   $\mathsf{tx}_i^{\mathsf{r}}.\mathsf{output} := (\theta.\mathsf{cash} + \theta_{\varepsilon_i}.\mathsf{cash}, \mathsf{OneSig}(U_i))$.
2. Return $\mathsf{tx}_i^{\mathsf{r}}$
```

```
genPayTx($\theta, U_{i+1}$):
1. Create a transaction $\mathsf{tx}_i^{\mathsf{p}}$ with $\mathsf{tx}_i^{\mathsf{p}}.\mathsf{input} := [\theta]$ and
   $\mathsf{tx}_i^{\mathsf{p}}.\mathsf{output} := (\theta.\mathsf{cash}, \mathsf{OneSig}(U_{i+1}))$.
2. Return $\mathsf{tx}_i^{\mathsf{p}}$
```

[a]Possibly other outputs $\{\theta'_j\}_{j \geq 0}$ could also be present in this state. They, along with the off-chain objects there (e.g., other payments) would have to be recreated in the new state while adapting the index of the output these objects are referring to. For simplicity, we say this here in prose and omit it in the protocol, only handling the two outputs mentioned.

J Modeling in the UC framework

We formally model our construction in the global UC framework (GUC) [10], an extension of the standard UC framework [9] that allows for a global setup, which we use for instance for modelling the ledger. In this section, we provide some preliminaries and then present the code for the ideal functionality of the multi-hop payment construction presented in this work. Our model follows closely the model in [5].

J.1 Preliminaries, communication model and threat model

A protocol Π runs between parties of the set $\mathcal{P}$. A protocol is executed in the presence of an adversary $\mathcal{A}$ that receives as input a security parameter $\lambda \in \mathbb{N}$ and an auxiliary input $z \in \{0,1\}^*$. We assume a static corruption model, where $\mathcal{A}$ can corrupt any party $P_i \in \mathcal{P}$ at the beginning of the execution, which means learning P_i's internal state and taking full control over P_i. The environment $\mathcal{E}$ is a special entity, that sends inputs to every party and the adversary $\mathcal{A}$ and observes every message output by the parties. Note that $\mathcal{E}$ is used to model anything that can happen outside the protocol execution.

We model communication in a synchronized network setting, where the protocol execution takes place in rounds. This abstraction allows for arguing about time more naturally. The global ideal functionality $\mathcal{G}_{clock}$ [19] represents a global clock, that proceeds to the next round when all honest parties agree to do so. Every entity is aware of what the current round is.

On top of this notion of rounds, we use a functionality $\mathcal{F}_{GDC}$ that models authenticated channels with guaranteed delivery after one round between the parties. Messages sent from a party P to Q in round t are guaranteed to reach Q in round $t+1$ with Q knowing that the sender was P. The adversary $\mathcal{A}$ can observe the content of messages and reorder the ones that were sent within the same round. It cannot however drop, modify or delay messages. See [12] for a formal description of $\mathcal{F}_{GDC}$.

Every other message, that is not sent between two party, but rather involves for instance $\mathcal{E}$ or $\mathcal{A}$, takes zero rounds. Also, we assume that any computation by any party takes zero rounds as well.

J.2 Ledger and channels

To model a UTXO cryptocurrency, we use a global functionality $\mathcal{G}_{Ledger}(\Delta)$, parameterized by an upper bound of the blockchain delay Δ, i.e., the number of rounds it at most takes for a valid transaction to be accepted on the blockchain, after being posted, and a signature scheme Σ. This functionality interacts with a fixed set of parties $\mathcal{P}$. To initialize $\mathcal{G}_{Ledger}$, $\mathcal{E}$ sets up a key pair $(\mathsf{sk}_P, \mathsf{pk}_P)$ for every $P \in \mathcal{P}$, sends $(\mathtt{sid}, \mathtt{REGISTER}, \mathsf{pk}_p)$ to $\mathcal{G}_{Ledger}$ and sets the intial state of $\mathcal{L}$, the set of all published transactions. After the initialization, the state of $\mathcal{L}$ is publicly accessible by every entity. When a valid transaction (i.e., a transacation that has correct witnesses for each input, a unique id, and the inputs have not been spent) is posted via $(\mathtt{sid}, \mathtt{POST}, \overline{\mathsf{tx}})$, it will be accepted on $\mathcal{L}$ after at most Δ rounds. The adversary chooses the exact number of rounds.

In this simplified model, the set of users is fixed and we do not model the fact, that in reality, transactions are usually bundled in blocks. We chose this simplification to increase readability and refer to works such as [6] for a more accurate formalization.

To model channels, we use the functionality $\mathcal{F}_{Channel}$ [5] that builds on top of $\mathcal{G}_{Ledger}$. It provides the functionality to create, update and close a payment channel between two users. We say that updating a channel takes at most t_{u} rounds and closing a channel, regardless if the parties are cooperating or not, takes at most t_{c} rounds.

For our Blitz payments, we assume that all participating parties have been registered with the ledger functionality and have had channels created beforehand already. For the complete API of $\mathcal{F}_{Channel}$ and $\mathcal{G}_{Ledger}$ see below. For better readability, we use the following notation instead of calling $\mathcal{G}_{clock}$ or $\mathcal{F}_{GDC}$. We let $(\mathsf{msg}) \overset{t}{\hookrightarrow} X$ denote sending message (msg) to X in round t. Moreover, $(\mathsf{msg}) \overset{t}{\hookleftarrow} X$ means receiving message (msg) from X at time t. Note that X as well as the sending/receiving identity are either a party $P \in \mathcal{P}$, the environment $\mathcal{E}$, the simulator $\mathcal{S}$ or another ideal functionality.

Interface of $\mathcal{F}_{Channel}(T,k)$ [5]

Parameters:

T:	upper bound on the maximum number of consecutive off-chain communication rounds between channel users
k:	number of ways the channel state can be published on the ledger

API:
Messages from $\mathcal{E}$ via a dummy user P:

- $(\mathtt{sid}, \mathtt{CREATE}, \overline{\gamma}, \mathsf{tid}_P) \overset{\tau}{\hookleftarrow} P$:
  Let $\overline{\gamma}$ be the attribute tuple $(\overline{\gamma}.\mathsf{id}, \overline{\gamma}.\mathsf{users}, \overline{\gamma}.\mathsf{cash}, \overline{\gamma}.\mathsf{st})$, where $\gamma.\mathsf{id} \in \{0,1\}^*$ is the identifier of the channel, $\overline{\gamma}.\mathsf{users} \subset \mathcal{P}$ are the users of the channel (and $P \in \overline{\gamma}.\mathsf{users}$), $\overline{\gamma}.\mathsf{cash} \in \mathbb{R}^{\geq 0}$ is the total money in the channel and $\overline{\gamma}.\mathsf{st}$ is the initial state of the channel. tid_P defines P's input for the funding transaction of the channel. When invoked, this function asks $\overline{\gamma}.\mathsf{otherParty}$ to create a new channel.
- $(\mathtt{sid}, \mathtt{UPDATE}, \mathsf{id}, \vec{\theta}) \overset{\tau}{\hookleftarrow} P$:
  Let $\overline{\gamma}$ be the channel where $\overline{\gamma}.\mathsf{id} = \mathsf{id}$. When invoked by $P \in \overline{\gamma}.\mathsf{users}$ and both parties agree, the channel $\overline{\gamma}$ (if it exists) is updated to the new state $\vec{\theta}$. If the parties disagree or at least one party is dishonest, the update can fail or the channel can be forcefully closed to either the old or the new state. Regardless of the outcome, we say that t_{u} is the upper bound that an update takes. In the successful case, $(\mathtt{sid}, \mathtt{UPDATED}, \mathsf{id}, \vec{\theta}) \overset{\leq \tau + t_{\mathsf{u}}}{\hookrightarrow} \overline{\gamma}.\mathsf{users}$ is output.
- $(\mathtt{sid}, \mathtt{CLOSE}, \mathsf{id}) \overset{\tau}{\hookleftarrow} P$:
  Will close the channel $\overline{\gamma}$, where $\overline{\gamma}.\mathsf{id} = \mathsf{id}$, either peacefully or forcefully. After at most t_{c} in round $\leq \tau + t_{\mathsf{c}}$, a transaction tx with the current state $\overline{\gamma}.\mathsf{st}$ as output $(\mathsf{tx.output} := \overline{\gamma}.\mathsf{st})$ appears on $\mathcal{L}$ (the public ledger of $\mathcal{G}_{Ledger}$).

Interface of $\mathcal{G}_{Ledger}(\Delta, \Sigma)$ [5]

This functionality keeps a record of the public keys of parties. Also, all transactions that are posted (and accpeted, see below) are stored in the publicly accessible set $\mathcal{L}$ containing tuples of all accepted transactions .

Parameters:

Δ:	upper bound on the number of rounds it takes a valid transaction to be published on $\mathcal{L}$
Σ:	a digital signature scheme

API:
Messages from $\mathcal{E}$ via a dummy user $P \in \mathcal{P}$:

- $(\mathtt{sid}, \mathtt{REGISTER}, \mathsf{pk}_P) \overset{\tau}{\hookleftarrow} P$:
  This function adds an entry (pk_P, P) to PKI consisting of the public key pk_P and the user P, if it does not already exist.
- $(\mathtt{sid}, \mathtt{POST}, \overline{\mathsf{tx}}) \overset{\tau}{\hookleftarrow} P$:
  This function checks if $\overline{\mathsf{tx}}$ is a valid transaction and if yes, accepts it on $\mathcal{L}$ after at most Δ rounds.

J.3 The UC-security definition

We denote Π as a *hybrid* protocol that accesses the ideal functionalities $\mathcal{F}_{\mathsf{prelim}}$ consisting of $\mathcal{F}_{Channel}$, $\mathcal{G}_{Ledger}$, $\mathcal{F}_{GDC}$ and $\mathcal{G}_{clock}$. An environment $\mathcal{E}$ that interacts with Π and an adversary $\mathcal{A}$ will on input a security parameter λ and an auxiliary input z output $\mathtt{EXEC}^{\mathcal{F}_{\mathsf{prelim}}}_{\Pi,\mathcal{A},\mathcal{E}}(\lambda, z)$. Moreover, $\phi_{\mathcal{F}_{Pay}}$ denotes the ideal protocol of ideal functionality $\mathcal{F}_{Pay}$, where the dummy users simply forward their input to $\mathcal{F}_{Pay}$. It has access to the same functionalities $\mathcal{F}_{\mathsf{prelim}}$. The output of $\phi_{\mathcal{F}_{Pay}}$ on input λ and z when interacting with $\mathcal{E}$ and a simulator $\mathcal{S}$ is denoted as $\mathtt{EXEC}^{\mathcal{F}_{\mathsf{prelim}}}_{\phi_{\mathcal{F}_{Pay}},\mathcal{S},\mathcal{E}}(\lambda, z)$.

If a protocol Π GUC-realizes an ideal functionality $\mathcal{F}_{Pay}$, then any attack that is possible on the real world protocol Π can be carried out against the ideal protocol $\phi_{\mathcal{F}_{Pay}}$ and vice versa. Our security definition is as follows.

Definition 1. A protocol Π GUC-realizes an ideal functionality $\mathcal{F}_{Pay}$, w.r.t. $\mathcal{F}_{\mathsf{prelim}}$, if for every adversary $\mathcal{A}$ there exists a simulator $\mathcal{S}$ such that we have

$$\left\{\mathtt{EXEC}^{\mathcal{F}_{\mathsf{prelim}}}_{\Pi,\mathcal{A},\mathcal{E}}(\lambda,z)\right\}_{\substack{\lambda\in\mathbb{N},\\ z\in\{0,1\}^*}} \overset{c}{\approx} \left\{\mathtt{EXEC}^{\mathcal{F}_{\mathsf{prelim}}}_{\phi_{\mathcal{F}_{Pay}},\mathcal{S},\mathcal{E}}(\lambda,z)\right\}_{\substack{\lambda\in\mathbb{N},\\ z\in\{0,1\}^*}}$$

where $\approx^c$ denotes computational indistinguishability.

J.4 Ideal functionality

In this section we will describe the ideal functionality (IF) $\mathcal{F}_{Pay}$ in prose. We are only interested in protocols that realize this IF and never output an ERROR. For cases where ERROR is output, any guarantees are lost. These cases are are not meaningful to us, they occur for instance when a transaction does not appear on the ledger as it should. We use the subprocedures defined in Appendix I. We divide the ideal functionality in three main parts: (i) Pay, (ii) Finalize and (iii) Respond.

Pay This sequence starts with setup, which is executed when queried by the sender U_0. In it, $\mathcal{F}_{Pay}$ sets up all initial objects and does the following. For every neighbor distinguish two cases: (i) the neighbor is honest, then $\mathcal{F}_{Pay}$ takes care of computing the objects and updating the channel or (ii) the neighbor is dishonest, then $\mathcal{F}_{Pay}$ instructs the simulator to simulate the view of the attacker. Should an attack ask an honest node, simulated by the simulator, to continue opening a payment with a legitimate request, the simulator will first let $\mathcal{F}_{Pay}$ perform Check and then Register. This process is repeated until the receiver is reached. At this point the Finalize part starts.

Finalize If U_0 is honest, $\mathcal{F}_{Pay}$ will expect a confirmation in the correct round, which is given either by itself if U_n is honest or sent by U_n via the simulator. If the confirmation is not well formed or no confirmation is received in the correct round, $\mathcal{F}_{Pay}$ instructs the simulator to publish $\mathsf{tx}^{\mathsf{er}}$. In case that U_n is honest, but not U_0, either $\mathcal{F}_{Pay}$ via the simulator or the simulator directly will simulate the view to the attacker by constructing and sending the confirmation to U_0.

Respond In this phase $\mathcal{F}_{Pay}$ reacts to transactions $\mathsf{tx}^{\mathsf{er}}$, that it has registered for payments in step Pay, appearing on $\mathcal{L}$. In the case of $\mathsf{tx}^{\mathsf{er}}$ for an honest user in a channel being published before the time where a refund is possible, $\mathcal{F}_{Pay}$ will close the channel and ask the simulator to publish a refund transaction. In the case that the time T has already passed and the neighbor closes the channel, $\mathcal{F}_{Pay}$ will instruct the simulator to claim the money by publishing the payment transaction.

Ideal Functionality $\mathcal{F}_{Pay}(\Delta)$

Parameters:

Δ :	Upper bound on the time it takes a transaction to appear on $\mathcal{L}$.

Local variables:

idSet :	A set of containing pairs of ids and users $(\mathtt{pid}, U_i)$ to prevent duplicate ids to avoid loops in payments.
Φ :	A map, storing for a given key $(\mathtt{pid}, U_0)$ of an id $\mathtt{pid}$ and a user U_0, a tuple $(\tau_{\mathsf{f}}, \mathsf{tx}^{\mathsf{er}}, U_n)$, where τ_{f} is the round in which the payment confirmation is expected from the receiver, the transaction $\mathsf{tx}^{\mathsf{er}}$ and the receiver U_n. The map is initially empty and read write access is written as $\Phi(\mathtt{pid}, U_0)$. $\Phi.\mathtt{keyList}()$ returns a set of all keys.
Γ :	A set of tuples $(\mathtt{pid}, \overline{\gamma_i}, \vec{\theta}_i, \mathsf{tx}^{\mathsf{er}}, T, \theta_{\varepsilon_i}, R_i)$ for channels with opened payment construction, containing a payment id $\mathtt{pid}$, the channel $\overline{\gamma_i}$, the state the payment builds upon $\vec{\theta}_i$, the time T, the output used in the refund by $\overline{\gamma_i}$.left and value R_i to reconstruct the secrect key of the stealth address used. It is initially empty.
Ψ :	A set of tuples $(\mathtt{pid}, \mathsf{tx}^{\mathsf{er}})$ containing payments, that have been opened and where the receiver is honest.
t_{u} :	Time required to perform a ledger channel update honestly.
t_{c} :	Time it at most takes to close a channel.

Init (executed at initialization in round t_{init}.)

Send $(\mathtt{sid}, \mathtt{init}) \overset{t_{\mathsf{init}}}{\hookrightarrow} \mathcal{S}$ and upon $(\mathtt{sid}, \mathtt{init\text{-}ok}, t_{\mathsf{u}}, t_{\mathsf{c}}) \overset{t_{\mathsf{init}}}{\hookleftarrow} \mathcal{S}$ set t_{u} and t_{c} accordingly.

Pay

Let τ be the current round.
Setup:

1. Upon $(\mathtt{sid}, \mathtt{pid}, \mathtt{SETUP}, \mathsf{channelList}, \mathsf{tx}^{\mathsf{in}}, \alpha, T, \overline{\gamma_0}) \overset{\tau}{\hookleftarrow} U_0$, if $(\mathtt{pid}, U_0) \in \mathsf{idSet}$ go idle. $\mathsf{idSet} := \mathsf{idSet} \cup \{(\mathtt{pid}, U_0)\}$
2. Let $x := \mathtt{checkChannels}(\mathsf{channelList}, U_0)$. If $x = \bot$, go idle. Else, let $U_n := x$. If $\overline{\gamma_0}$ is not the full channel between U_0 and his right neighbor $U_1 := \overline{\gamma_0}$.right (corresponding to the channel skeleton γ_0 in channelList), go idle. Let nodeList be a list of all the users on the path sorted from U_0 to U_n.
3. Let $n := |\mathsf{channelList}|$. If $\mathtt{checkT}(n, T) = \bot$, go idle.
4. If $\mathtt{checkTxIn}(\mathsf{tx}^{\mathsf{in}}, n, U_0) = \bot$, go idle.
5. $(\mathsf{tx}^{\mathsf{er}}, \mathsf{onions}, \mathsf{rMap}, \mathsf{rList}, \mathsf{stealthMap}) := \mathtt{createMaps}(U_0, \mathsf{nodeList}, \mathsf{tx}^{\mathsf{in}})$.
6. Set $\alpha_0 := \alpha + \mathsf{fee} \cdot (n-1)$.
7. Set $\Phi(\mathtt{pid}, U_0) := (\tau_{\mathsf{f}} := \tau + n \cdot (2 + t_{\mathsf{u}}) + 1, \mathsf{tx}^{\mathsf{er}}, U_n)$.
8. If U_1 honest, execute **Open**$(\mathtt{pid}, \mathsf{nodeList}, \mathsf{tx}^{\mathsf{er}}, \mathsf{onions}, \mathsf{rMap}, \mathsf{rList}, \mathsf{stealthMap}, \alpha_0, T, \overline{\gamma_0})$.
9. Else, let $\mathsf{onion}_1 := \mathsf{onions}(U_1)$ and $\theta_{\varepsilon_0} := \mathsf{stealthMap}(U_0)$. Send $(\mathtt{sid}, \mathtt{pid}, \mathtt{open}, \mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_1, \alpha_0, T, \overline{\gamma_0}, \theta_{\varepsilon_0}) \overset{\tau}{\hookrightarrow} \mathcal{S}$.

Continue:

1. Upon $(\mathtt{sid}, \mathtt{pid}, \mathtt{continue}, \mathsf{nodeList}, \mathsf{tx}^{\mathsf{er}}, \mathsf{onions}, \mathsf{rMap}, \mathsf{rList}, \mathsf{stealthMap}, \alpha_{i-1}, T, \overline{\gamma_{i-1}}) \overset{\tau}{\hookleftarrow} \mathcal{S}$
2. **Open**$(\mathtt{pid}, \mathsf{nodeList}, \mathsf{tx}^{\mathsf{er}}, \mathsf{onions}, \mathsf{rMap}, \mathsf{rList}, \mathsf{stealthMap}, \alpha_{i-1}, T, \overline{\gamma_{i-1}})$.

Check:

1. Upon $(\texttt{sid},\texttt{pid},\texttt{check-id},\mathsf{tx}^{\mathsf{er}},\theta_{\varepsilon_i},R_i,U_{i-1},U_i,U_{i+1},\alpha_i,T)$ $\overset{\tau}{\hookleftarrow} \mathcal{S}$
2. If $(\texttt{pid},U_i) \notin \mathsf{idSet}$, let $\mathsf{idSet} := \mathsf{idSet} \cup \{(\texttt{pid},U)\}$ and send the message $(\texttt{sid},\texttt{pid},\texttt{OPEN},\mathsf{tx}^{\mathsf{er}},\theta_{\varepsilon_i},R_i,U_{i-1},U_{i+1},\alpha_{i-1},T)$ $\overset{\tau}{\hookrightarrow} U_i$
3. If $(\texttt{sid},\texttt{pid},\texttt{ACCEPT},\overline{\gamma_i}) \overset{\tau}{\hookleftarrow} U_i$, $(\texttt{sid},\texttt{pid},\texttt{ok},\overline{\gamma_i}) \overset{\tau}{\hookrightarrow} \mathcal{S}$.

Payment-Open:

1. Upon $(\texttt{sid},\texttt{pid},\texttt{payment-open},\mathsf{tx}^{\mathsf{er}}) \overset{\tau}{\hookleftarrow} \mathcal{S}$, let $\Psi := \Psi \cup \{(\texttt{pid},\mathsf{tx}^{\mathsf{er}})\}$.

Register:

1. Upon $(\texttt{sid},\texttt{pid},\texttt{register},\overline{\gamma_i},\vec{\theta}_i,\mathsf{tx}^{\mathsf{er}},T,\theta_{\varepsilon_i},R) \overset{\tau}{\hookleftarrow} \mathcal{S}$
2. $\Gamma := \Gamma \cup \{(\texttt{pid},\overline{\gamma_i},\vec{\theta}_i,\mathsf{tx}^{\mathsf{er}},T,\theta_{\varepsilon_i},R)\}$

Open$(\texttt{pid},\mathsf{nodeList},\mathsf{tx}^{\mathsf{er}},\mathsf{onions},\mathsf{rMap},\mathsf{rList},\mathsf{stealthMap},\alpha_{i-1},T,\overline{\gamma_{i-1}})$:
Let τ be the current round and $U_i := \overline{\gamma_{i-1}}.\mathsf{right}$

1. If $(\texttt{pid},U_i) \in \mathsf{idSet}$, go idle.
2. $\mathsf{idSet} := \mathsf{idSet} \cup \{(\texttt{pid},U_i)\}$
3. If an entry after U_i in $\mathsf{nodeList}$ exists and is $\perp$, go idle.
4. If $U_i = U_n$ (i.e., last entry in $\mathsf{nodeList}$), set $U_{i+1} := \top$. Else, get U_{i+1} from $\mathsf{nodeList}$ (the entry after U_i).
5. $R_i := \mathsf{rMap}(U_i)$ and $\theta_{\varepsilon_i} := \mathsf{stealthMap}(U_i)$
6. $\vec{\theta}_{i-1} := \texttt{genStateOutputs}(\overline{\gamma_{i-1}},\alpha_{i-1},T)$. If $\vec{\theta}_{i-1} = \perp$, go idle. Else, wait 1 round.
7. $(\texttt{sid},\texttt{pid},\texttt{OPEN},\mathsf{tx}^{\mathsf{er}},\theta_{\varepsilon_i},R_i,U_{i-1},U_{i+1},\alpha_{i-1},T) \xhookrightarrow{\tau+1} U_i$
8. If not $(\texttt{sid},\texttt{pid},\texttt{ACCEPT},\overline{\gamma_i}) \xhookleftarrow{\tau+1} U_i$, go idle. Else,wait 1 round.
9. $(\texttt{ssid}_C,\texttt{UPDATE},\overline{\gamma_{i-1}}.\mathsf{id},\vec{\theta}_{i-1}) \xhookrightarrow{\tau+2} \mathcal{F}_{Channel}$
10. $(\texttt{ssid}_C,\texttt{UPDATED},\overline{\gamma_{i-1}}.\mathsf{id}) \xhookleftarrow{\tau+2+t_{\mathsf{u}}} \mathcal{F}_{Channel}$, else go idle.
11. $\Gamma := \Gamma \cup (\texttt{pid},\overline{\gamma_i},\vec{\theta}_i,\mathsf{tx}^{\mathsf{er}},T,\theta_{\varepsilon_i},R_i)$
12. If $U_i = U_n$:
    - $\Psi := \Psi \cup \{(\texttt{pid},\mathsf{tx}^{\mathsf{er}})\}$
    - $(\texttt{sid},\texttt{pid},\texttt{PAYMENT-OPEN},\mathsf{tx}^{\mathsf{er}},T,\alpha_i) \xhookrightarrow{\tau+2+t_{\mathsf{u}}} U_i$
    - If U_0 is dishonest, send $(\texttt{sid},\texttt{pid},\texttt{finalize},\mathsf{tx}^{\mathsf{er}}) \xhookrightarrow{\tau+2+t_{\mathsf{u}}} \mathcal{S}$
13. Else:
    - $(\texttt{sid},\texttt{pid},\texttt{OPENED}) \xhookrightarrow{\tau+2+t_{\mathsf{u}}} U_i$
    - If U_{i+1} honest, execute **Open**$(\texttt{pid},\mathsf{nodeList},\mathsf{tx}^{\mathsf{er}},\mathsf{onions},\mathsf{rMap},\mathsf{rList},\mathsf{stealthMap},\alpha_{i-1}-\mathsf{fee},\overline{\gamma_i})$
    - Else, send $(\texttt{sid},\texttt{pid},\texttt{open},\mathsf{tx}^{\mathsf{er}},\mathsf{rList},\mathsf{onion}_{i+1},\alpha_i-\mathsf{fee},T,\overline{\gamma_i},\theta_{\varepsilon_i}) \overset{\tau}{\hookrightarrow} \mathcal{S}$, where $\mathsf{onion}_{i+1} := \mathsf{onions}(U_{i+1})$

<u>Finalize (executed at every round)</u>

For every $(\texttt{pid},U_0) \in \Phi.\texttt{keyList}()$ do the following:

1. Let $(\tau_{\mathsf{f}},\mathsf{tx}^{\mathsf{er}},U_n) = \Phi(\texttt{pid},U_0)$. If for the current round τ it holds that $\tau = \tau_f$, do the following.
2. If U_n honest, check if $(\texttt{pid},\mathsf{tx}^{\mathsf{er}}) \in \Psi$. If yes, let $\Psi := \Psi \setminus \{(\texttt{pid},\mathsf{tx}^{\mathsf{er}})\}$ and go idle.
3. If U_n dishonest and $(\texttt{sid},\texttt{pid},\texttt{confirmed},\mathsf{tx}_x^{\mathsf{er}},\sigma_{U_n}(\mathsf{tx}_x^{\mathsf{er}})) \overset{\tau_n}{\hookleftarrow} \mathcal{S}$, such that $\mathsf{tx}_x^{\mathsf{er}} = \mathsf{tx}^{\mathsf{er}}$ and $\sigma_{U_n}(\mathsf{tx}_x^{\mathsf{er}})$ is U_n's valid signature of $\mathsf{tx}^{\mathsf{er}}$, go idle.
4. Send $(\texttt{sid},\texttt{pid},\texttt{denied},\mathsf{tx}^{\mathsf{er}},U_0) \xhookrightarrow{\tau_n+1} \mathcal{S}$. $\mathsf{tx}^{\mathsf{er}}$ must appear on $\mathcal{L}$ in round $\tau' \leq \tau_{\mathsf{f}} + \Delta$. Otherwise, output $(\texttt{sid},\texttt{ERROR}) \overset{t_1}{\hookrightarrow} U_0$.

<u>Respond (executed at the end of every round)</u>

Let t be the current round. For every element $(\texttt{pid},\overline{\gamma_i},\vec{\theta}_i,\mathsf{tx}^{\mathsf{er}},T,\theta_{\varepsilon_i},R_i) \in \Gamma$, check if $\overline{\gamma_i}.\mathsf{st} = \vec{\theta}_i$ and $\mathsf{tx}^{\mathsf{er}}$ is on $\mathcal{L}$. If yes, do the following:
Revoke: If $\gamma_i.\mathsf{left}$ honest and $t < T - t_{\mathsf{c}} - 2\Delta$ do the following.

- Set $\Gamma := \Gamma \setminus \{(\texttt{pid},\overline{\gamma_i},\vec{\theta}_i,\mathsf{tx}^{\mathsf{er}},T,\theta_{\varepsilon_i},R_i)\}$.
- $(\texttt{ssid}_C,\texttt{CLOSE},\overline{\gamma_i}.\mathsf{id}) \overset{t}{\hookrightarrow} \mathcal{F}_{Channel}$
- At time $t + t_{\mathsf{c}}$, a transaction tx with $\mathsf{tx.output} = \overline{\gamma_i}.\mathsf{st}$ has to be on $\mathcal{L}$. If not, do the following. If $(\texttt{ssid}_C,\texttt{PUNISHED},\overline{\gamma_i}.\mathsf{id}) \xhookleftarrow{\tau<T} \mathcal{F}_{Channel}$, go idle. Else, send $(\texttt{sid},\texttt{ERROR}) \overset{T}{\hookrightarrow} \gamma_i.\mathsf{users}$.
- Wait for Δ rounds and send $(\texttt{sid},\texttt{pid},\texttt{post-refund},\overline{\gamma_i},\theta_{\varepsilon_i},R_i) \xhookrightarrow{t'<T-\Delta} \mathcal{S}$
- At time $t'' < T$, check whether a transaction tx' appears on $\mathcal{L}$ with $\mathsf{tx}'.\mathsf{input} = [\theta_{\varepsilon_i}, tx.\mathsf{output}[0]]$ and $\mathsf{tx}'.\mathsf{output} = [(\mathsf{tx.output}[0].\mathsf{cash} + \theta_{\varepsilon_i}.\mathsf{cash}, \mathsf{OneSig}(U_i))]$. If it does, send $(\texttt{sid},\texttt{pid},\texttt{REVOKED}) \overset{t''}{\hookrightarrow} \overline{\gamma_i}.\mathsf{left}$. If not, send $(\texttt{sid},\texttt{ERROR}) \overset{T}{\hookrightarrow} \gamma_i.\mathsf{users}$.

Force-Pay: Else, if a transaction tx with $\mathsf{tx.output} = \overline{\gamma_i}.\mathsf{st}$ is on-chain and $\mathsf{tx.output}[0]$ is unspent (i.e., there is no transaction on $\mathcal{L}$, that uses is as input), $t \geq T$ and U_{i+1} is honest, do the following.

- Set $\Gamma := \Gamma \setminus \{(\texttt{pid},\overline{\gamma_i},\vec{\theta}_i,\mathsf{tx}^{\mathsf{er}},T,\theta_{\varepsilon_i},R_i)\}$.
- Send $(\texttt{sid},\texttt{pid},\texttt{post-pay},\overline{\gamma_i}) \overset{t}{\hookrightarrow} \mathcal{S}$
- In round $t + \Delta$ transaction tx' with $\mathsf{tx}'.\mathsf{input} = [\mathsf{tx.output}[0]]$ and $\mathsf{tx}'.\mathsf{output} = (\mathsf{tx.output}[0].\mathsf{cash}, \mathsf{OneSig}(U_{i+1}))$ must have appeared on $\mathcal{L}$. If yes, $(\texttt{sid},\texttt{pid},\texttt{FORCE-PAY}) \xhookrightarrow{t+\Delta} \overline{\gamma_i}.\mathsf{right}$. Otherwise, $(\texttt{sid},\texttt{ERROR}) \xhookrightarrow{t+\Delta} \gamma_i.\mathsf{users}$.

J.5 Protocol

Here we present the formal protocol Π and a brief description thereof. For simplicity, we assume that users involved in the payment do not use (e.g., update, close) the channels involved in the payment.[3] Moreover, for any payment the sender knows the receiver and the receiver knows the sender. Also, every user knows if it is the sender in a payment or if it is the receiver in a payment. Therefore, when the simulator simulates the behavior of an honest user, the simulator also knows if the user is the sender/receiver or not and, if it is the sender (receiver), the simulator also knows the receiver (sender).

The protocol itself is similar to the simpler version presented in Section 4.4, but extended with payment ids and UC formalism, most notably we introduce rounds and the environment $\mathcal{E}$. To reiterate briefly, the protocol is divided into three parts. In Pay, the initial objects are setup by U_0 after

[3]We refer the reader to Appendix A for an outline on how to perform concurrent payments or use the channel otherwise while a payment is active.

being invoked by $\mathcal{E}$. Afterwards, the neighbor is contacted and they open a payment construction by creating a new state, the appropriate transactions, signing them and then updating the channel. When first asked, a user will forward an open message to $\mathcal{E}$, which responds with accept (or nothing). In Finalize, the receiver sends a confirmation to the sender. The sender expects the correct confirmation in the correct round, otherwise it will publish $\mathsf{tx}^{\mathsf{er}}$. In the Respond phase, users will react to $\mathsf{tx}^{\mathsf{er}}$ being published and, if possible, either refund or force the payment.

Protocol Π

Let $\mathsf{fee} \in \mathbb{N}$ be a system parameter known to every user.
Local variables of U_i (all initially empty):

pidSet :	A set storing every payment id `pid` that a user has participated in to prevent duplicates.
paySet :	A map storing tuples $(\texttt{pid}, \tau_{\mathsf{f}}, U_n)$ where `pid` is an id, τ_{f} is the round in which a confirmation is expected from the receiver U_n for the payments that have been opened by this user.
local :	A map, storing for a given `pid` U_i's local copy of $\mathsf{tx}^{\mathsf{er}}$ and T in a tuple $(\mathsf{tx}^{\mathsf{er}}, T)$.
left :	A map, storing for a given `pid` a tuple $(\overline{\gamma_{i-1}}, \vec{\theta}_{i-1}, \mathsf{tx}^{\mathsf{r}}_{i-1})$ containing channel with its left neighbor U_{i-1}, the state and the transaction $\mathsf{tx}^{\mathsf{r}}_{i-1}$ for U_i's left channel in the payment `pid`.
right :	A map, sotring for a given `pid` a tuple $(\overline{\gamma_i}, \vec{\theta}_i, \mathsf{tx}^{\mathsf{r}}_i, \mathsf{sk}_{\widetilde{U_i}})$ containing the channel with its right neighbor, the state, the transaction $\mathsf{tx}^{\mathsf{r}}_i$ and the key necessary for signing the refund transaction in the payment `pid`.
$\mathsf{rightSig}$:	A map, storing for a given `pid` the signature for $\mathsf{tx}^{\mathsf{r}}_i$ of the right neighbor $\sigma_{U_{i+1}}(\mathsf{tx}^{\mathsf{r}}_i)$ in the payment `pid`.

<u>Pay</u>

Setup: In every round, every node $U_0 \in \mathcal{P}$ does the following. We denote τ_0 as the current round.

<u>U_0 upon $(\texttt{sid}, \texttt{pid}, \texttt{SETUP}, \mathsf{channelList}, \mathsf{tx}^{\mathsf{in}}, \alpha, T, \overline{\gamma_0}) \overset{\tau_0}{\hookleftarrow} \mathcal{E}$</u>

1. If $\texttt{pid} \in \mathsf{pidSet}$, abort. Add `pid` to pidSet.
2. Let $x := \texttt{checkChannels}(\mathsf{channelList}, U_0)$. If $x = \bot$, abort. Else, let $U_n := x$. If $\overline{\gamma_0}$ is not the full channel between U_0 and his right neighbor $U_1 := \overline{\gamma_0}.\mathsf{right}$ (corresponding to the channel skeleton γ_0 in $\mathsf{channelList}$), go idle.
3. Let $n := |\mathsf{channelList}|$. If $\texttt{checkT}(n, T) = \bot$, abort.
4. If $\texttt{checkTxIn}(\mathsf{tx}^{\mathsf{in}}, n, U_0) = \bot$, abort
5. $(\mathsf{tx}^{\mathsf{er}}, \mathsf{onions}, \mathsf{rMap}, \mathsf{rList}, \mathsf{stealthMap}) := \texttt{createMaps}(U_0, \mathsf{nodeList}, \mathsf{tx}^{\mathsf{in}})$.
6. $(\mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_0) := \texttt{genTxEr}(U_0, \mathsf{channelList}, \mathsf{tx}^{\mathsf{in}})$
7. $\mathsf{paySet} := \mathsf{paySet} \cup \{(\texttt{pid}, \tau_{\mathsf{f}} := \tau + n \cdot (2 + t_{\mathsf{u}}) + 1, U_n)\}$
8. $(\mathsf{sk}_{\widetilde{U_0}}, \theta_{\varepsilon_0}, R_0, U_1, \mathsf{onion}_1) := \texttt{checkTxEr}(U_0, U_0.a, U_0.b, \mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_0)$
9. Set $\mathsf{local}(\texttt{pid}) := (\mathsf{tx}^{\mathsf{er}}, T)$.
10. Set $\alpha_0 := \alpha + \mathsf{fee} \cdot (n-1)$ and compute:
    - $\vec{\theta}_0 := \texttt{genState}(\alpha_0, T, \overline{\gamma_0})$
    - $\mathsf{tx}^{\mathsf{r}}_0 := \texttt{genRefTx}(\overline{\gamma_0}.\mathsf{st}[0], \theta_{\varepsilon_0}, U_0)$
11. Set $\mathsf{right}(\texttt{pid}) := (\overline{\gamma_0}, \vec{\theta}_0, \mathsf{tx}^{\mathsf{r}}_0, \mathsf{sk}_{\widetilde{U_0}})$.
12. Send $(\texttt{sid}, \texttt{pid}, \texttt{open-req}, \mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_1, \vec{\theta}_0, \mathsf{tx}^{\mathsf{r}}_0) \overset{\tau_0}{\hookrightarrow} U_1$.

Open: In every round, every node $U_{i+1} \in \mathcal{P}$ does the following. We denote τ_x as the current round.

<u>U_{i+1} u. $(\texttt{sid}, \texttt{pid}, \texttt{open-req}, \mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_{i+1}, \vec{\theta}_i, \mathsf{tx}^{\mathsf{r}}_i) \overset{\tau_x}{\hookleftarrow} U_i$</u>

1. Perform the following checks:
   - Verify that $\texttt{pid} \notin \mathsf{pidSet}$. Add `pid` to pidSet
   - Let $x := \texttt{checkTxEr}(U_{i+1}, U_{i+1}.a, U_{i+1}.b, \mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_{i+1})$. Check that $x \neq \bot$, but instead $x = (\mathsf{sk}_{\widetilde{U_{i+1}}}, \theta_{\varepsilon_{i+1}}, R_{i+1}, U_{i+2}, \mathsf{onion}_{i+2})$.
   - Set $\alpha_i = \vec{\theta}_i[0].\mathsf{cash}$ and extract T from $\vec{\theta}_{i-1}[0].\phi$ (the parameter of $\mathsf{AbsTime}()$).
   - Check that there exists a channel between U_i and U_{i+1} and call this channel $\overline{\gamma_i}$. Verify that $\vec{\theta}_i = \texttt{genState}(\alpha_i, T, \overline{\gamma_i})$.
   - Check that $\mathsf{tx}^{\mathsf{r}}_i := \texttt{genRefTx}(\overline{\gamma_i}.\mathsf{st}[0], \theta_{\varepsilon_x}, U_i)$, where θ_{ε_x} is an output of $\mathsf{tx}^{\mathsf{er}}$.
2. If one or more of the previous checks fail, abort. Otherwise, send $(\texttt{sid}, \texttt{pid}, \texttt{OPEN}, \mathsf{tx}^{\mathsf{er}}, \theta_{\varepsilon_{i+1}}, R_i, U_i, U_{i+2}, \alpha_i, T) \overset{\tau_x}{\hookrightarrow} \mathcal{E}$.
3. If $(\texttt{sid}, \texttt{pid}, \texttt{ACCEPT}, \overline{\gamma_{i+1}}) \overset{\tau_x}{\hookleftarrow} \mathcal{E}$, generate $\sigma_{U_{i+1}}(\mathsf{tx}^{\mathsf{r}}_i)$. Otherwise stop.
4. Set $\mathsf{local}(\texttt{pid}) := (\mathsf{tx}^{\mathsf{er}}_i, T)$, $\mathsf{left}(\texttt{pid}) := (\overline{\gamma_i}, \vec{\theta}_i, \mathsf{tx}^{\mathsf{r}}_i)$ and $(\texttt{sid}, \texttt{pid}, \texttt{open-ok}, \sigma_{U_{i+1}}(\mathsf{tx}^{\mathsf{r}}_i)) \overset{\tau_x}{\hookrightarrow} U_i$.

<u>U_i upon $(\texttt{sid}, \texttt{pid}, \texttt{open-ok}, \sigma_{U_{i+1}}(\mathsf{tx}^{\mathsf{r}}_i)) \overset{\tau_i+2}{\hookleftarrow} U_{i+1}$</u>

(The round τ_i given U_i and `pid` is defined in Setup or in Open step (6), the round when the update is successful.)

5. Check that $\sigma_{U_{i+1}}(\mathsf{tx}^{\mathsf{r}}_i)$ is a valid signature for $\mathsf{tx}^{\mathsf{r}}_i$. If yes, set $\mathsf{rightSig}(\texttt{pid}) := (\mathsf{tx}^{\mathsf{r}}_i, \sigma_{U_{i+1}}(\mathsf{tx}^{\mathsf{r}}_i))$ and $(\texttt{ssid}_C, \texttt{UPDATE}, \overline{\gamma_i}.\mathsf{id}, \vec{\theta}_i) \overset{\tau_i+2}{\hookrightarrow} \mathcal{F}_{Channel}$.

<u>U_{i+1} upon $(\texttt{ssid}_C, \texttt{UPDATED}, \overline{\gamma_i}.\mathsf{id}, \vec{\theta}_i) \overset{\tau_x+1+t_{\mathsf{u}}}{\hookleftarrow} \mathcal{F}_{Channel}$</u>

6. Define $\tau_{(i+1)} := \tau_x + 1 + t_{\mathsf{u}}$.
7. If U_{i+1} is not the receiver, using the values of step 1:
   - Send $(\texttt{sid}, \texttt{pid}, \texttt{OPENED}) \overset{\tau_{i+1}}{\hookrightarrow} \mathcal{E}$.
   - $(\mathsf{sk}_{\widetilde{U_{i+1}}}, \theta_{\varepsilon_{i+1}}, R_{i+1}, U_{i+2}, \mathsf{onion}_{i+2}) := \texttt{checkTxEr}(U_{i+1}, U_{i+1}.a, U_{i+1}.b, \mathsf{tx}^{\mathsf{er}}_i, \mathsf{rList}, \mathsf{onion}_{i+1})$
   - $\vec{\theta}_{i+1} := \texttt{genState}(\alpha_i - \mathsf{fee}, T, \overline{\gamma_{i+1}})$
   - $\mathsf{tx}^{\mathsf{r}}_{i+1} := \texttt{genRefTx}(\overline{\gamma_{i+1}}.\mathsf{st}[0], \theta_{\varepsilon_{i+1}}, U_{i+1})$
   - Set $\mathsf{right}(\texttt{pid}) := (\overline{\gamma_{i+1}}, \vec{\theta}_{i+1}, \mathsf{tx}^{\mathsf{s}}_{i+1}, \mathsf{sk}_{\widetilde{U_{i+1}}})$
   - Send $(\texttt{sid}, \texttt{pid}, \texttt{open-req}, \mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_{i+2}, \vec{\theta}_{i+1}, \mathsf{tx}^{\mathsf{s}}_{i+1}) \overset{\tau_{i+1}}{\hookrightarrow} U_{i+2}$.
8. If U_{i+1} is the receiver:
   - $\mathsf{msg} := \mathsf{GetRoutingInfo}(\mathsf{onion}_{i+1}, U_{i+1})$

- Create the signature $\sigma_{U_n}(\mathsf{tx}^{\mathsf{er}}_i)$ as confirmation and send $(\mathtt{sid},\mathtt{pid},\mathtt{finalize},\mathsf{tx}^{\mathsf{er}},\sigma_{U_n}(\mathsf{tx}^{\mathsf{er}})) \xrightarrow{\tau_{i+1}} U_0$. Send the message $(\mathtt{sid},\mathtt{pid},\mathtt{PAYMENT\text{-}OPEN},\mathsf{tx}^{\mathsf{er}},T,\alpha_i) \xhookrightarrow{\tau_{i+1}} \mathcal{E}$.

<u>Finalize</u>

<u>U_0 in every round τ</u>

For every entry $(\mathtt{pid},\tau_{\mathsf{f}},U_n) \in \mathsf{paySet}$ do the following if $\tau = \tau_{\mathsf{f}}$:

1. Remove $(\mathtt{pid},\tau_{\mathsf{f}},U_n)$ from paySet.
2. Upon receiving $(\mathtt{sid},\mathtt{pid},\mathtt{finalize},\mathsf{tx}^{\mathsf{er}},\sigma_{U_n}(\mathsf{tx}^{\mathsf{er}})) \overset{\tau}{\hookleftarrow} U_n$, continue if $\sigma_{U_n}(\mathsf{tx}^{\mathsf{er}})$ is a valid signature for $\mathsf{tx}^{\mathsf{er}}$. Otherwise, go to step (4).
3. If $\mathsf{local}(\mathtt{pid}) = \mathsf{tx}^{\mathsf{er}}$, go idle. Otherwise, continue with the next step.
4. Sign $\mathsf{tx}^{\mathsf{er}}$ yielding $\sigma_{U_0}(\mathsf{tx}^{\mathsf{er}})$ and set $\overline{\mathsf{tx}^{\mathsf{er}}} := (\mathsf{tx}^{\mathsf{er}},(\sigma_{U_0}(\mathsf{tx}^{\mathsf{er}})))$. Send $(\mathtt{ssid}_L,\mathtt{POST},\overline{\mathsf{tx}^{\mathsf{er}}}) \overset{\tau}{\hookrightarrow} \mathcal{G}_{Ledger}$.

<u>Respond</u>

<u>U_i at the end of every round</u>

Let t be the current round. Do the following:

1. For every $\mathtt{pid}$ in $\mathsf{right}.\mathtt{keyList}()$, let $(\overline{\gamma_i},\vec{\theta}_i,\mathsf{tx}^{\mathsf{r}}_i,\mathsf{sk}_{\widetilde{U}_i}) := \mathsf{right}(\mathtt{pid})$, let $(\mathsf{tx}^{\mathsf{er}},T) := \mathsf{local}(\mathtt{pid})$ and do the following. If $t < T - t_{\mathsf{c}} - 2\Delta$, $\mathsf{tx}^{\mathsf{er}}$ is on the ledger $\mathcal{L}$ and $\overline{\gamma_i}.\mathsf{st} = \vec{\theta}_i$, do the following:
   - Remove the entry for $\mathtt{pid}$ from right, send $(\mathtt{ssid}_C,\mathtt{CLOSE},\overline{\gamma_i}.\mathsf{id}) \overset{t}{\hookrightarrow} \mathcal{F}_{Channel}$ and wait Δ rounds.
   - If a transaction tx with $\mathsf{tx}.\mathsf{output} = \vec{\theta}_i$ is on $\mathcal{L}$ in round $t + t_{\mathsf{c}}$, sign $\mathsf{tx}^{\mathsf{r}}_i$ yielding $\sigma_{U_i}(\mathsf{tx}^{\mathsf{r}}_i)$
   - Use $\mathsf{sk}_{\widetilde{U}_i}$ to sign $\mathsf{tx}^{\mathsf{r}}_i$ yielding $\sigma_{\widetilde{U}_i}(\mathsf{tx}^{\mathsf{r}}_i)$
   - Set $\overline{\mathsf{tx}^{\mathsf{r}}_i} := (\mathsf{tx}^{\mathsf{r}}_i,(\sigma_{U_i}(\mathsf{tx}^{\mathsf{r}}_i),\mathsf{rightSig}(\mathtt{pid}),\sigma_{\widetilde{U}_i}(\mathsf{tx}^{\mathsf{r}}_i)))$ and send $(\mathtt{ssid}_L,\mathtt{POST},\overline{\mathsf{tx}^{\mathsf{r}}_i}) \xhookrightarrow{t+t_{\mathsf{c}}+\Delta} \mathcal{G}_{Ledger}$. When it appears on $\mathcal{L}$ in round $t_1 < T$, send $(\mathtt{sid},\mathtt{pid},\mathtt{REVOKED}) \overset{t_2}{\hookrightarrow} \mathcal{E}$
2. For every $\mathtt{pid}$ in $\mathsf{left}.\mathtt{keyList}()$, let $(\overline{\gamma_{i-1}},\vec{\theta}_{i-1},\mathsf{tx}^{\mathsf{r}}_{i-1}) := \mathsf{left}(\mathtt{pid})$, let $(\mathsf{tx}^{\mathsf{er}},T) := \mathsf{local}(\mathtt{pid})$ and do the following. If $t \geq T$ and a transaction tx with $\mathsf{tx}.\mathsf{output} = \vec{\theta}_{i-1}$ is on the ledger $\mathcal{L}$, but not $\mathsf{tx}^{\mathsf{r}}_{i-1}$, do the following:
   - Remove the entry for $\mathtt{pid}$ from left and create $\mathsf{tx}^{\mathsf{p}}_{i-1} := \mathtt{genPayTx}(\overline{\gamma_{i-1}}.\mathsf{st},U_i)$.
   - Sign $\mathsf{tx}^{\mathsf{p}}_{i-1}$ yielding $\sigma_{U_i}(\mathsf{tx}^{\mathsf{p}}_{i-1})$.
   - Set $\overline{\mathsf{tx}^{\mathsf{p}}_{i-1}} := (\mathsf{tx}^{\mathsf{p}}_{i-1},\sigma_{U_i}(\mathsf{tx}^{\mathsf{p}}_{i-1}))$ and send $(\mathtt{ssid}_L,\mathtt{POST},\overline{\mathsf{tx}^{\mathsf{p}}_{i-1}}) \overset{t}{\hookrightarrow} \mathcal{G}_{Ledger}$.
   - If it appears on $\mathcal{L}$ in round $t_1 \leq t + \Delta$, send $(\mathtt{sid},\mathtt{pid},\mathtt{FORCE\text{-}PAID}) \overset{t_1}{\hookrightarrow} \mathcal{E}$

J.6 Proof

In this section, we describe the simulator as well as the formal proof that the Blitz protocol (see Appendix J.5) UC-realizes the ideal functionality $\mathcal{F}_{Pay}$ shown in Appendix J.4.

Simulator

Local variables:

right	A map, storing the transaction $\mathsf{tx}^{\mathsf{r}}_i$ for a given keypair consisting of a payment id $\mathtt{pid}$ and a user U_i.
$\mathsf{rightSig}$	A map, storing the signature of the right neighbor for the transaction stored in right for a given keypair consisting of a payment id $\mathtt{pid}$ and a user U_i.

Simulator for init phase

Upon $(\mathtt{sid},\mathtt{init}) \xhookleftarrow{t_{\mathsf{init}}} \mathcal{F}_{Pay}$ and send $(\mathtt{sid},\mathtt{init\text{-}ok},t_{\mathsf{u}},t_{\mathsf{c}}) \xhookrightarrow{t_{\mathsf{init}}} \mathcal{F}_{Pay}$.

Simulator for pay phase

<u>a) Case U_i is honest, U_{i+1} dishonest</u>

1. Upon $(\mathtt{sid},\mathtt{pid},\mathtt{open},\mathsf{tx}^{\mathsf{er}},\mathsf{rList},\mathsf{onion}_{i+1},\alpha_i,T,\overline{\gamma_i},\theta_{\varepsilon_i}) \overset{\tau}{\hookleftarrow} \mathcal{F}_{Pay}$ or upon being called by the simulator $\mathcal{S}$ itself in round τ with parameters $(\mathtt{pid},\mathsf{tx}^{\mathsf{er}},\mathsf{rList},\mathsf{onion}_{i+1},\alpha_i,T,\overline{\gamma_i},\theta_{\varepsilon_i})$.
2. Let $U_i := \overline{\gamma_i}.\mathsf{left}$ and $U_{i+1} := \overline{\gamma_i}.\mathsf{right}$.
3. $\vec{\theta}_i := \mathtt{genState}(\alpha_i,T,\overline{\gamma_i})$
4. $\mathsf{tx}^{\mathsf{r}}_i := \mathtt{genRefTx}(\overline{\gamma_i}.\mathsf{st}[0],\theta_{\varepsilon_i},U_i)$
5. $(\mathtt{sid},\mathtt{pid},\mathtt{open\text{-}req},\mathsf{tx}^{\mathsf{er}},\mathsf{rList},\mathsf{onion}_{i+1},\vec{\theta}_i,\mathsf{tx}^{\mathsf{r}}_i) \overset{\tau}{\hookrightarrow} U_{i+1}$
6. Upon $(\mathtt{sid},\mathtt{pid},\mathtt{open\text{-}ok},\sigma_{U_{i+1}}(\mathsf{tx}^{\mathsf{r}}_i)) \xleftarrow{\tau+2} U_{i+1}$, check that $\sigma_{U_{i+1}}(\mathsf{tx}^{\mathsf{r}}_i)$ is a valid signature for $\mathsf{tx}^{\mathsf{r}}_i$. If yes, set $\mathsf{rightSig}(\mathtt{pid},U_i) := \sigma_{U_{i+1}}(\mathsf{tx}^{\mathsf{r}}_i)$, $\mathsf{right}(\mathtt{pid},U_i) := \mathsf{tx}^{\mathsf{r}}_i$ and $(\mathtt{ssid}_C,\mathtt{UPDATE},\overline{\gamma_i}.\mathsf{id},\vec{\theta}_i) \xrightarrow{\tau+2} \mathcal{F}_{Channel}$. Send$(\mathtt{sid},\mathtt{pid},\mathtt{register},\overline{\gamma_i},\vec{\theta}_i,\mathsf{tx}^{\mathsf{er}},T,\theta_{\varepsilon_i},R) \overset{\tau}{\hookrightarrow} \mathcal{F}_{Pay}$. Otherwise, go idle.

<u>b) Case U_i is honest, U_{i-1} dishonest</u>

1. Upon $(\mathtt{sid},\mathtt{pid},\mathtt{open\text{-}req},\mathsf{tx}^{\mathsf{er}},\mathsf{rList},\mathsf{onion}_i,\vec{\theta}_{i-1},\mathsf{tx}^{\mathsf{r}}_{i-1}) \overset{\tau}{\hookleftarrow} U_{i-1}$. Let $\alpha_{i-1} := \vec{\theta}_{i-1}[0].\mathsf{cash}$ and extract T from $\vec{\theta}_{i-1}[0].\phi$ (the parameter of $\mathsf{AbsTime}()$).
2. Let $x := \mathtt{checkTxEr}(U_i,U_i.a,U_i.b,\mathsf{tx}^{\mathsf{er}},\mathsf{rList},\mathsf{onion}_i)$. Check that $x \neq \bot$, but instead $x = (\mathsf{sk}_{\widetilde{U}_i},\theta_{\varepsilon_i},R_i,U_{i+1},\mathsf{onion}_{i+1})$. Otherwise, go idle.
3. Check that there exists a channel between U_i and U_{i+1} and call this channel $\overline{\gamma_i}$. Verify that $\vec{\theta}_i = \mathtt{genState}(\alpha_i,T,\overline{\gamma_i})$.
4. $(\mathtt{sid},\mathtt{pid},\mathtt{check\text{-}id},\mathsf{tx}^{\mathsf{er}},\theta_{\varepsilon_i},R_i,U_{i-1},U_i,U_{i+1},\alpha_i,T) \overset{\tau}{\hookrightarrow} \mathcal{F}_{Pay}$
5. If not $(\mathtt{sid},\mathtt{pid},\mathtt{ok},\overline{\gamma_i}) \overset{\tau}{\hookleftarrow} \mathcal{F}_{Pay}$, go idle. Let $U_{i+1} := \overline{\gamma_i}.\mathsf{right}$.
6. Sign $\mathsf{tx}^{\mathsf{r}}_{i-1}$ on behalf of U_i yielding $\sigma_{U_i}(\mathsf{tx}^{\mathsf{r}}_{i-1})$ and $(\mathtt{sid},\mathtt{pid},\mathtt{open\text{-}ok},\sigma_{U_i}(\mathsf{tx}^{\mathsf{r}}_{i-1})) \overset{\tau}{\hookrightarrow} U_{i-1}$.
7. Upon $(\mathtt{ssid}_C,\mathtt{UPDATED},\overline{\gamma_{i-1}}.\mathsf{id},\vec{\theta}_{i-1}) \xleftarrow{\tau+1+t_{\mathsf{u}}} \mathcal{F}_{Channel}$, send $(\mathtt{sid},\mathtt{pid},\mathtt{register},\overline{\gamma_{i-1}},\vec{\theta}_{i-1},\mathsf{tx}^{\mathsf{er}},T,\bot,\bot) \overset{\tau}{\hookrightarrow} \mathcal{F}_{Pay}$. Otherwise, go idle.

8. If $U_i = U_n$ (if $(\mathsf{sk}_{\widetilde{U_i}}, \theta_{\varepsilon_i}, R_i, U_{i+1}, \mathsf{onion}_{i+1}) = (\top,\top,\top,\top,\top)$ holds), and U_0 is honest,[a] send $(\texttt{sid}, \texttt{pid}, \texttt{payment-open}, \mathsf{tx}^{\mathsf{er}}) \xhookrightarrow{\tau+1+t_{\mathsf{u}}} \mathcal{F}_{Pay}$. If U_0 is dishonest, send $(\texttt{sid}, \texttt{pid}, \texttt{finalize}, \mathsf{tx}^{\mathsf{er}}, \sigma_{U_0}(\mathsf{tx}^{\mathsf{er}})) \xhookrightarrow{\tau+1+t_{\mathsf{u}}} U_n$.
9. If U_{i+1} honest, call process$(\texttt{sid}, \texttt{pid}, \mathsf{tx}^{\mathsf{er}}, \overline{\gamma_{i-1}}, \overline{\gamma_i}, R_i, \mathsf{onion}_i, \alpha_i, T)$
10. If dishonest, go to step Simulator U_i honest, U_{i+1} dishonest step 1 with parameters $(\texttt{pid}, \mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_{i+1}, \alpha_{i-1} - \mathsf{fee}, T, \overline{\gamma_i}, \theta_{\varepsilon_i})$.

<u>process$(\texttt{sid}, \texttt{pid}, \mathsf{tx}^{\mathsf{er}}, \overline{\gamma_{i-1}}, \overline{\gamma_i}, R_i, \mathsf{onion}_i, \alpha_{i-1}, T)$</u>

Let τ be the current round.

1. Initialize $\mathsf{nodeList} := \{U_i\}$ and $\mathsf{onions}, \mathsf{rMap}, \mathsf{stealthMap}$ as empty maps.
2. $(U_{i+1}, \mathsf{msg}_i, \mathsf{onion}_{i+1}) := \mathsf{GetRoutingInfo}(\mathsf{onion}_i)$
3. $\mathsf{stealthMap}(U_i) := \theta_{\varepsilon_i}$
4. $\mathsf{rMap}(U_i) := R_i$
5. While U_i and U_{i+1} honest:
   - $x := \texttt{checkTxEr}(U_{i+1}, U_{i+1}.a, U_{i+1}.b, \mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_{i+1})$:
     - If $x = \bot$, append U_{i+1} and then $\bot$ to $\mathsf{nodeList}$ and break the loop.
     - If $x = (\top,\top,\top,\top,\top)$, append U_{i+1} to $\mathsf{nodeList}$ and break the loop.
     - Else, if $x = (\mathsf{sk}_{\widetilde{U_{i+1}}}, \theta_{\varepsilon_{i+1}}, U_{i+2}, \mathsf{onion}_{i+2})$, do the following.
   - Append U_{i+1} to $\mathsf{nodeList}$
   - $\mathsf{onions}(U_{i+2}) := \mathsf{onion}_{i+2}$
   - $\mathsf{rMap}(U_{i+1}) := R_{i+1}$
   - $\mathsf{stealthMap}(U_{i+1}) := \theta_{\varepsilon_{i+1}}$
   - If U_{i+2} is dishonest, append U_{i+2} to $\mathsf{nodeList}$ and break the loop.
   - Set $i := i+1$ (i.e., continue loop for U_{i+1} and U_{i+2})
6. Send $(\texttt{sid}, \texttt{pid}, \texttt{continue}, \mathsf{nodeList}, \mathsf{tx}^{\mathsf{er}}, \mathsf{onions}, \mathsf{rMap}, \mathsf{rList}, \mathsf{stealthMap}, \alpha_{i-1}, T, \overline{\gamma_{i-1}}) \overset{\tau}{\hookrightarrow} \mathcal{F}_{Pay}$

[a] For simplicity, assume that the U_n (and in the case it is honest, the simulator) knows the sender. As the payment is usually tied to the exchange of some goods, this is a reasonable assumption. Note that in practice, this is not necessary, as the sender can be embedded in the routing information onion_n.

Simulator for finalize phase

<u>a) Publishing $\mathsf{tx}^{\mathsf{er}}$</u>

Upon receiving a message $(\texttt{sid}, \texttt{pid}, \texttt{denied}, \mathsf{tx}^{\mathsf{er}}, U_0) \overset{\tau}{\hookleftarrow} \mathcal{F}_{Pay}$ and U_0 honest, sign $\mathsf{tx}^{\mathsf{er}}$ on behalf of U_0 yielding $\sigma_{U_0}(\mathsf{tx}^{\mathsf{er}})$. Set $\overline{\mathsf{tx}^{\mathsf{er}}} := (\mathsf{tx}^{\mathsf{er}}, \sigma_{U_0}(\mathsf{tx}^{\mathsf{er}}))$ and send $(\texttt{ssid}_L, \texttt{POST}, \overline{\mathsf{tx}^{\mathsf{er}}}) \overset{\tau}{\hookrightarrow} \mathcal{G}_{Ledger}$.

<u>b) Case U_n honest, U_0 dishonest</u>

Upon message $(\texttt{sid}, \texttt{pid}, \texttt{finalize}, \mathsf{tx}^{\mathsf{er}}) \overset{\tau}{\hookleftarrow} \mathcal{F}_{Pay}$, sign $\mathsf{tx}^{\mathsf{er}}$ on behalf of U_n yielding $\sigma_{U_n}(\mathsf{tx}^{\mathsf{er}})$. Send $(\texttt{sid}, \texttt{pid}, \texttt{finalize}, \mathsf{tx}^{\mathsf{er}}, \sigma_{U_n}(\mathsf{tx}^{\mathsf{er}})) \overset{\tau}{\hookrightarrow} U_0$.

<u>c) Case U_n dishonest, U_0 honest</u>

Upon message $(\texttt{sid}, \texttt{pid}, \texttt{finalize}, \mathsf{tx}^{\mathsf{er}}, \sigma_{U_n}(\mathsf{tx}^{\mathsf{er}})) \overset{\tau}{\hookleftarrow} U_n$, send $(\texttt{sid}, \texttt{pid}, \texttt{confirmed}, \mathsf{tx}^{\mathsf{er}}, \sigma_{U_n}(\mathsf{tx}^{\mathsf{er}})) \overset{\tau}{\hookrightarrow} \mathcal{F}_{Pay}$.

Simulator for respond phase

In every round τ, upon receiving the following two messages, react accordingly.

1. Upon $(\texttt{sid}, \texttt{pid}, \texttt{post-refund}, \overline{\gamma_i}, \mathsf{tx}^{\mathsf{er}}, \theta_{\varepsilon_i}, R_i) \overset{\tau}{\hookleftarrow} \mathcal{F}_{Pay}$.
   - Extract α_i and T from $\overline{\gamma_i}.\mathsf{st}.\mathsf{output}[0]$.
   - If U_{i+1} is honest, create the transaction $\mathsf{tx}^{\mathsf{r}}_i := \texttt{genRefTx}(\overline{\gamma_i}.\mathsf{st}[0], \theta_{\varepsilon_i}, U_i)$. Else, let $\mathsf{tx}^{\mathsf{r}}_i := \mathsf{right}(\texttt{pid}, U_i)$
   - Extract $\mathsf{pk}_{\widetilde{U_i}}$ from the output θ_{ε_i} of $\mathsf{tx}^{\mathsf{er}}$ and let $\mathsf{sk}_{\widetilde{U_i}} := \mathsf{GenSk}(U_i.a, U_i.b, \mathsf{pk}_{\widetilde{U_i}}, R_i)$.
   - Generate signatures $\sigma_{U_i}(\mathsf{tx}^{\mathsf{r}}_i)$ and, using $\mathsf{sk}_{\widetilde{U_i}}$, $\sigma_{\widetilde{U_i}}(\mathsf{tx}^{\mathsf{r}}_i)$ on behalf of U_i.
   - If $U_{i+1} := \overline{\gamma_i}.\mathsf{right}$ is honest, generate signature $\sigma_{U_{i+1}}(\mathsf{tx}^{\mathsf{r}}_i)$ on behalf of U_{i+1}. Else, let $\sigma_{U_{i+1}}(\mathsf{tx}^{\mathsf{r}}_i) := \mathsf{rightSig}(\texttt{pid}, U_i)$
   - Set $\overline{\mathsf{tx}^{\mathsf{r}}_i} := (\mathsf{tx}^{\mathsf{r}}_i, (\sigma_{U_i}(\mathsf{tx}^{\mathsf{r}}_i), \sigma_{U_{i+1}}(\mathsf{tx}^{\mathsf{r}}_i), \sigma_{\widetilde{U_i}}(\mathsf{tx}^{\mathsf{r}}_i)))$.
   - Send $(\texttt{ssid}_L, \texttt{POST}, \overline{\mathsf{tx}^{\mathsf{r}}_i}) \overset{\tau}{\hookrightarrow} \mathcal{G}_{Ledger}$.
2. Upon $(\texttt{sid}, \texttt{pid}, \texttt{post-pay}, \overline{\gamma_i}) \overset{\tau}{\hookleftarrow} \mathcal{F}_{Pay}$
   - Extract α_i and T from $\overline{\gamma_i}.\mathsf{st}.\mathsf{output}[0]$. Create the transaction $\mathsf{tx}^{\mathsf{p}}_i := \texttt{genPayTx}(\overline{\gamma_i}.\mathsf{st}, U_{i+1})$.
   - Generate signatures $\sigma_{U_{i+1}}(\mathsf{tx}^{\mathsf{p}}_i)$ and set $\overline{\mathsf{tx}^{\mathsf{p}}_i} := (\mathsf{tx}^{\mathsf{p}}_i, (\sigma_{U_{i+1}}(\mathsf{tx}^{\mathsf{p}}_i)))$.
   - Send $(\texttt{ssid}_L, \texttt{POST}, \overline{\mathsf{tx}^{\mathsf{p}}_i}) \overset{\tau}{\hookrightarrow} \mathcal{G}_{Ledger}$.

Lemma 1. *Let Σ be a EUF-CMA secure signature scheme. Then, the Pay phase of protocol Π GUC-emulates the Pay phase of functionality $\mathcal{F}_{Pay}$.*

Proof. We show that the simulator $\mathcal{S}$ presented above interacting with the Pay phase of $\mathcal{F}_{Pay}$ is indistinguishable for any environment $\mathcal{E}$ from an interaction with Π and a dummy adversary $\mathcal{A}$. A bit more formally, we show that the ensembles $\texttt{EXEC}_{\mathcal{F}_{Pay},\mathcal{S},\mathcal{E}}$ and $\texttt{EXEC}_{\Pi,\mathcal{A},\mathcal{E}}$ are indistinguishable for the environment $\mathcal{E}$.

In our description, we write $m[\tau]$ to denote that message m is observed at round τ. Moreover, we interact with other ideal functionalities, which in turn interact with either the environment $\mathcal{E}$ or other parties, who are possibly under adversarial control, by sending messages. These interactions can have an additional impact on publicly observable variables, i.e., the ledger $\mathcal{L}$. When sending a message m to a ideal functionality $\mathcal{F}$ in round τ, we denote the set of all by $\mathcal{E}$ observable actions triggered by this as a function $\mathsf{obsSet}(m, \mathcal{F}, \tau)$.

In the following, we analyze the different corruption cases. For each case, we first describe the view of the environment in Π and then the view of the environment as simulated by $\mathcal{S}$. For the Pay phase, we consider three different cases of the interaction between two users U_i and U_{i+1}. We match the sequences of this phase, that we use in the proof below, and where they are used in the ideal and real world in Table 5. Note that for $U_i = U_0$ `SETUP` is performed initially, otherwise `CREATE_STATE`. We define the following messages.

- $m_0 := (\mathtt{sid}, \mathtt{pid}, \mathtt{open\text{-}req}, \mathsf{tx^{er}}, \mathsf{rList}, \mathsf{onion}_{i+1}, \vec{\theta}_i, \mathsf{tx}^{\mathsf{r}}_i)$
- $m_1 := (\mathtt{sid}, \mathtt{pid}, \mathtt{OPEN}, \mathsf{tx^{er}}, \theta_{\varepsilon_{i+1}}, \mathsf{R_i}, \mathsf{U_i}, \mathsf{U_{i+2}}, \alpha_\mathsf{i}, \mathsf{T})$
- $m_2 := (\mathtt{sid}, \mathtt{pid}, \mathtt{ACCEPT}, \overline{\gamma_{\mathsf{i+1}}})$
- $m_3 := (\mathtt{sid}, \mathtt{pid}, \mathtt{open\text{-}ok}, \sigma_{\mathsf{U_{i+1}}}(\mathsf{tx}^{\mathsf{r}}_\mathsf{i}))$
- $m_4 := (\mathtt{ssid}_\mathsf{C}, \mathtt{UPDATE}, \overline{\gamma_\mathsf{i}}.\mathsf{id}, \tilde{\theta}_\mathsf{i})$
- $m_5 := (\mathtt{ssid}_\mathsf{C}, \mathtt{UPDATED}, \overline{\gamma_\mathsf{i}}.\mathsf{id}, \tilde{\theta}_\mathsf{i})$
- $m_6 := (\mathtt{sid}, \mathtt{pid}, \mathtt{OPENED})$ or, if sent by the receiver, $m_6 := (\mathtt{sid}, \mathtt{pid}, \mathtt{PAYMENT\text{-}OPEN}, \mathsf{tx^{er}}, \mathsf{T}, \alpha_\mathsf{i})$

1. U_i honest, U_{i+1} corrupted

Real world: After U_i performs either `SETUP` or `CREATE_STATE`, it sends m_0 to U_{i+1} in the current round τ. The environment $\mathcal{E}$ controls $\mathcal{A}$ and therefore U_{i+1} and will see m_0 in round $\tau+1$. Iff U_{i+1} replies with a correct message m_3 in $\tau+2$, U_i will perform `CHECK_SIG` and call $\mathcal{F}_{Channel}$ with message m_4 in the same round. The ensemble is $\mathtt{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_0[\tau+1]\} \cup \mathsf{obsSet}(m_4, \mathcal{F}_{Channel}, \tau+2)$

Ideal world: After $\mathcal{F}_{Pay}$ performs either `SETUP` or simulator performs `CREATE_STATE`, the simulator sends m_0 to U_{i+1} in the current round τ. $\mathcal{E}$ will see m_0 in round $\tau+1$. Iff U_{i+1} replies with a correct message m_3 in $\tau+2$, the simulator will perform `CHECK_SIG` and call $\mathcal{F}_{Channel}$ with message m_4 in the same round. The ensemble is $\mathtt{EXEC}_{\mathcal{F}_{Pay},\mathcal{S},\mathcal{E}} := \{m_0[\tau+1]\} \cup \mathsf{obsSet}(m_4, \mathcal{F}_{Channel}, \tau+2)$

2. U_i honest, U_{i+1} honest

Real world: After U_i performs either `SETUP` or `CREATE_STATE`, it sends m_0 to U_{i+1} in the current round τ. U_{i+1} performs `CHECK_STATE` and sends m_1 to $\mathcal{E}$ in round $\tau+1$. Iff $\mathcal{E}$ replies with m_2, U_{i+1}, U_{i+1} replies with m_3. U_i receives this in round $\tau+2$, performs `CHECK_SIG` and sends m_4 to $\mathcal{F}_{Channel}$. U_{i+1} expects the message m_5 in round $\tau+2+t_\mathsf{u}$ and will then send m_6 to $\mathcal{E}$. Afterwards it continues with either `CREATE_STATE` or `FINALIZE`. The ensemble is $\mathtt{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_1[\tau+1], m_6[\tau+2+t_\mathsf{u}]\} \cup \mathsf{obsSet}(m_4, \mathcal{F}_{Channel}, \tau+2)$

Ideal world: After $\mathcal{F}_{Pay}$ performs either `SETUP` or is invoked by itself (in step Open.13) or by the simulator (in step process.6) in the current round τ, $\mathcal{F}_{Pay}$ perform the procedure Open. This behaves exactly like `CREATE_STATE`, `CHECK_STATE` and `CHECK_SIG`. However, since every object is created by $\mathcal{F}_{Pay}$, the checks are omitted. The procedure Open outputs the messages m_1 in round $\tau+1$ and iff $\mathcal{E}$ replies with m_2, calls $\mathcal{F}_{Channel}$ with m_4 in $\tau+2$. Finally, if m_5 is received in round $\tau+2+t_\mathsf{u}$, outputs m_6 to $\mathcal{E}$. The ensemble is $\mathtt{EXEC}_{\mathcal{F}_{Pay},\mathcal{S},\mathcal{E}} := \{m_1[\tau+1], m_6[\tau+2+t_\mathsf{u}]\} \cup \mathsf{obsSet}(m_4, \mathcal{F}_{Channel}, \tau+2)$

3. U_i corrupted, U_{i+1} honest

Real world: After U_{i+1} receives the message m_0 from U_i, it performs `CHECK_STATE` and sends m_1 to $\mathcal{E}$ in the current round τ. Iff $\mathcal{E}$ replies with m_2, U_{i+1} sends m_3 to U_i. If U_{i+1} receives the message m_5 from $\mathcal{F}_{Channel}$ in round $\tau+1+t_\mathsf{u}$, it sends m_6 to $\mathcal{E}$. The ensemble is $\mathtt{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_1[\tau], m_3[\tau+1], m_6[\tau+1+t_\mathsf{u}]\}$

Ideal world: After the simulator receives m_0 from U_i, it performs `CHECK_STATE` together with $\mathcal{F}_{Pay}$ and $\mathcal{F}_{Pay}$ sends m_1 to $\mathcal{E}$. Iff $\mathcal{E}$ replies with m_2, $\mathcal{F}_{Pay}$ asks the simulator to send m_3 to U_i. All of this happens in the current round τ. If the simulator receives m_5 in round $\tau+1+t_\mathsf{u}$, it sends m_6 to $\mathcal{E}$. The ensemble is $\mathtt{EXEC}_{\mathcal{F}_{Pay},\mathcal{S},\mathcal{E}} := \{m_1[\tau], m_3[\tau+1], m_6[\tau+1+t_\mathsf{u}]\}$

Note that we do not care about the case were both U_i and U_{i+1} are corrupted, because the environment is communincating with itself, which is trivially the same in the ideal and the real world. We see that in these three cases, the execution ensembles of the ideal and the real world are identical, thereby proving Lemma 1.

□

Lemma 2. *Let Σ be a EUF-CMA secure signature scheme. Then, the Finalize phase of protocol Π GUC-emulates the Finalize phase of functionality $\mathcal{F}_{Pay}$.*

Proof. Again, we consider the execution ensembles of the interaction between users U_n and U_0 for three different cases. We match the sequences and where they are used in the ideal and real world in Table 6. We define the following messages.

- $m_7 := (\mathtt{sid}, \mathtt{pid}, \mathtt{finalize}, \mathsf{tx^{er}})$
- $m_8 := (\mathtt{ssid}_\mathcal{L}, \mathtt{POST}, \overline{\mathsf{tx^{er}}})$

1. U_n honest, U_0 corrupted

Real world: After performing `FINALIZE` in the current round τ, U_n sends m_7 to U_0. The ensemble is $\mathtt{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_7[\tau]\}$

Ideal world: After either $\mathcal{F}_{Pay}$ or the simulator performs `FINALIZE` in the current round τ, the simulator sends m_7 to U_0. The ensemble is $\mathtt{EXEC}_{\mathcal{F}_{Pay},\mathcal{S},\mathcal{E}} := \{m_7[\tau]\}$

2. U_n honest, U_0 honest

Real world: After performing `FINALIZE` in the current round τ, U_n sends m_7 to U_0. In the meantime, U_0 performs `CHECK_FINALIZE` and should it not receive a correct message m_7 in the correct round, will send m_8 to $\mathcal{G}_{Ledger}$ in round τ'. This will result in the sets of message The ensemble is $\mathtt{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \mathsf{obsSet}(m_8, \mathcal{G}_{Ledger}, \tau')$

Ideal world: Either $\mathcal{F}_{Pay}$ or the simulator performs `FINALIZE` in the current round τ. In the meantime, $\mathcal{F}_{Pay}$ performs `CHECK_FINALIZE` and will, if the checks in `FINALIZE` failed or it was performed in a incorrect round τ', $\mathcal{F}_{Pay}$ will instruct the simulator

Table 5: Explanation of the sequence names used in Lemma 1 and where they can be found in the ideal functionality (IF), Protocol (Prot) or Simulator (Sim).

	Real World	Ideal World			Output	Description
		U_i honest, U_{i+1} corrupted	U_i honest, U_{i+1} honest	U_i corrupted, U_{i+1} honest		
`SETUP`	Prot.Pay.Setup 1-12	IF.Pay.Setup 1-7,9, Sim.Pay.a 1-5	IF.Pay.Setup 1-8	n/a	m_0	Does setup and contacts next user
`CREATE_STATE`	Prot.Pay.Open 6-8	n/a	IF.Pay.Open 12, 13 Sim.Pay.a 1-5	Sim.Pay.b 8-10	m_6, m_0	Upon m_5, sends message m_6 to $\mathcal{E}$. Then, ceates the objects to send in m_0 and sends it to U_{i+1} (or finalize).
`CHECK_STATE`	Prot.Pay.Open 1-4	n/a	IF.Pay.Open 1-8	Sim.Pay.b 1-4 IF.Check Sim.Pay.b 5-7 IF.Register	m_1, m_3	Checks if objects in m_0 are correct, sends m_1 to $\mathcal{E}$ and on m_2, sends m_3 to U_i
`CHECK_SIG`	Prot.Pay.Open 5	Sim.Pay.a 6	IF.Pay.Open 9-11	n/a	m_4	Checks if signature of $\mathsf{tx}^{\mathsf{r}}_i$ is correct

Table 6: Explanation of the sequence names used in Lemma 2 and where they can be found.

	Real World	Ideal World			Output	Description
		U_n honest, U_0 corrupted	U_n honest, U_0 honest	U_n corrupted, U_0 honest		
`FINALIZE`	Prot.Pay.Open 8	IF.Pay.12 and Sim.Finalize.b or Sim.Pay.b 8	IF.Pay.12 and Sim.Finalize.b or Sim.Pay.b 8	n/a	m_7	Sends finalize message to U_0
`CHECK_FINALIZE`	Prot.Finalize 1-6	n/a	IF.Finalize 1,2,4 Sim.Finalize.a	Sim.Finalize.c IF.Finalize 1,3,4 Sim.Finalize.a	m_8	Checks if $\mathsf{tx}^{\mathsf{er}}$ is the same, if not, publishes it to ledger with m_8.

to send m_8 to $\mathcal{G}_{Ledger}$ in rounds τ'. The ensemble is $\texttt{EXEC}_{\mathcal{F}_{Pay},\mathcal{S},\mathcal{E}} := \mathsf{obsSet}(m_8, \mathcal{G}_{Ledger}, \tau')$

3. U_n corrupted, U_0 honest

Real world: U_0 performs `CHECK_FINALIZE` and should it not receive a correct message m_7 in the correct round, will send m_8 to $\mathcal{G}_{Ledger}$ in round τ'. The ensemble is $\texttt{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \mathsf{obsSet}(m_8, \mathcal{G}_{Ledger}, \tau')$

Ideal world: The simulator and $\mathcal{F}_{Pay}$ perform `CHECK_FINALIZE` and should the simulator not receive a correct message m_7 in the correct round, $\mathcal{F}_{Pay}$ will instruct the simulator to send m_8 to $\mathcal{G}_{Ledger}$ in round τ'.The ensemble is $\texttt{EXEC}_{\mathcal{F}_{Pay},\mathcal{S},\mathcal{E}} := \mathsf{obsSet}(m_8, \mathcal{G}_{Ledger}, \tau')$

□

Lemma 3. *Let Σ be a EUF-CMA secure signature scheme. Then, the Respond phase of protocol Π GUC-emulates the Respond phase of functionality $\mathcal{F}_{Pay}$.*

Proof. Again, we consider the execution ensembles. This time only for the case were a user U_i is honest, however we distinguish between the case of revoke and force-pay. We match the sequences and where they are used in the ideal and real world in Table 7. We define the following messages.

- $m_9 := (\texttt{ssid}_C, \texttt{CLOSE}, \overline{\gamma_i}.\mathsf{id})$
- $m_{10} := (\texttt{ssid}_L, \texttt{POST}, \overline{\mathsf{tx}^{\mathsf{r}}_i})$
- $m_{11} := (\texttt{sid}, \texttt{pid}, \texttt{REVOKED})$
- $m_{12} := (\texttt{ssid}_L, \texttt{POST}, \overline{\mathsf{tx}^{\mathsf{p}}_{i-1}})$
- $m_{13} := (\texttt{sid}, \texttt{pid}, \texttt{FORCE-PAY})$

U_i honest, revoke

Real world: In every round τ, U_i performs `RESPOND`, which provides a decision on whether or not to do the following. If yes, U_i performs `REVOKE`, which results in message m_9 to $\mathcal{F}_{Channel}$ in round τ. If the channel that is sent in m_9 is closed, U_i sends m_{10} to $\mathcal{G}_{Ledger}$ in round $\tau + t_{\mathsf{c}} + \Delta$. Finally, if the transaction sent in m_{10} appears on $\mathcal{L}$ in $\tau + t_{\mathsf{c}} + 2\Delta$, U_i sends m_{11} to $\mathcal{E}$. The ensemble is $\texttt{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_{11}[\tau + t_{\mathsf{c}} + 2\Delta]\} \cup \mathsf{obsSet}(m_9, \mathcal{F}_{Channel}, \tau) \cup \mathsf{obsSet}(m_{10}, \mathcal{G}_{Ledger}, \tau + t_{\mathsf{c}} + \Delta)$

Ideal world: In every round τ, $\mathcal{F}_{Pay}$ performs `RESPOND`, which provides a decision on whether or not to do the following. If yes, $\mathcal{F}_{Pay}$ instructs the simulator to perform `REVOKE`, which results in the message m_9 to $\mathcal{F}_{Channel}$ in round τ. If the channel that is sent in m_9 is closed, the simulator sends m_{10} to $\mathcal{G}_{Ledger}$ in round $\tau + t_{\mathsf{c}} + \Delta$. Finally, if the transaction sent in m_{10} appears on $\mathcal{L}$, $\mathcal{F}_{Pay}$ sends m_{11} to $\mathcal{E}$. The ensemble is $\texttt{EXEC}_{\mathcal{F}_{Pay},\mathcal{S},\mathcal{E}} := \{m_{11}[\tau + t_{\mathsf{c}} + 2\Delta]\} \cup \mathsf{obsSet}(m_9, \mathcal{F}_{Channel}, \tau) \cup \mathsf{obsSet}(m_{10}, \mathcal{G}_{Ledger}, \tau + t_{\mathsf{c}} + \Delta)$

U_i honest, force-pay

Real world: In every round τ, U_i performs `RESPOND`, which provides a decision on whether or not to do the following. If yes, U_i performs `FORCE_PAY`, which results in the messages m_{12} to $\mathcal{G}_{Ledger}$ in round τ and, if the transaction sent in m_{12} appears on $\mathcal{L}$, the message m_{13} to $\mathcal{E}$ in round $\tau + \Delta$. The ensemble is $\texttt{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_{13}[\tau + \Delta]\} \cup \mathsf{obsSet}(m_{12}, \mathcal{G}_{Ledger}, \tau)$

Ideal world: In every round τ, $\mathcal{F}_{Pay}$ performs `RESPOND`, which provides a decision on whether or not to do the

Table 7: Explanation of the sequence names used in Lemma 3 and where they can be found.

	Real World	Ideal World	Output	Description
		U_i honest		
`RESPOND`	Prot.Respond	IF.Respond	n/a	Checks every round if response in order.
`REVOKE`	Prot.Respond.1	IF.Respond.Revoke Sim.Respond.1	m_9, m_{10}, m_{11}	Carries out the revokation.
`FORCE_PAY`	Prot.Respond.2	IF.Respond.Revoke Sim.Respond.2	m_{12}, m_{13}	Carries out the force-pay.

following. If yes, $\mathcal{F}_{Pay}$ instructs the simulator to perform `FORCE_PAY`, which results in the messages m_{12} to $\mathcal{G}_{Ledger}$ in round τ and, if the transaction sent in m_{12} appears on $\mathcal{L}$, the message m_{13} to $\mathcal{E}$ in round $\tau+\Delta$. The ensemble is $\texttt{EXEC}_{\mathcal{F}_{Pay},\mathcal{S},\mathcal{E}} := \{m_{13}[\tau+\Delta]\} \cup \mathsf{obsSet}(m_{12}, \mathcal{G}_{Ledger}, \tau)$

$\square$

Theorem 2. *(formal) Let Σ be a EUF-CMA secure signature scheme. Then, for any ledger delay $\Delta \in \mathbb{N}$, the protocol Π UC-realizes the ideal functionality $\mathcal{F}_{Pay}$.*

This theorem follows directly from Lemma 1, 2 and Lemma 3.

K Discussion on security and privacy goals

So far, in Section 4.1 we have informally stated what are our security and privacy goals in this work. Additionally, in Appendix J.4 we have described the ideal functionality $\mathcal{F}_{Pay}$ that formally defines the security and privacy guarantees achieved by Blitz. In this section, we aim to show how $\mathcal{F}_{Pay}$ indeed has the security and privacy goals that we intuitively want to achieve. For that, we first formalize each intuitive security and privacy goal into a cryptographic game and then show that $\mathcal{F}_{Pay}$ fulfills such definition.

K.1 Assumptions

For the theorems in this section, we have the following assumptions: (i) we assume that stealth addresses achieve unlinkability and (ii) we assume that the routing scheme we use (i.e., Sphinx extended with a per-hop payload) is a secure onion routing process.

Unlinkability of stealth addresses Consider the following game. The challenger computes two pair of stealth addresses (A_0, B_0) and (A_1, B_1). Moreover, the challenger picks a bit b and computes $P_b, R_b \leftarrow \mathsf{GenPk}(A_b, B_b)$. Finally, the challenger sends the tuples (A_0, B_0), (A_1, B_1) and P_b, R_b to the adversary.

Additionally, the adversary has access to an oracle that upon being queried, it returns P^*_b, R^*_b to the adversary.

We say that they adversary wins the game if it correctly guesses the bit b chosen by the challenger.

Definition 2 (Unlinkability of Stealth Addresses)**.** We say that a stealth addresses scheme achieves unlinkability if for all PPT adversary $\mathcal{A}$, the adversary wins the aforementioned game with probability at most $1/2+\varepsilon$, where ε denotes a negligible value.

Secure onion routing process We say that an onion routing process is secure, if it realizes the ideal functionality defined in [8]. Sphinx [11], for instance, is a realization of this. We use it in Blitz, extended with a per-hop payload (see also Section 4.2).

K.2 Balance security

Given a path $\mathsf{channelList} := \gamma_1, \dots, \gamma_n$ and given a user U such that $\gamma_i.\mathsf{right} = U$ and $\gamma_{i+1}.\mathsf{left} = U$, we say that the balance of U in the path is $\mathbf{PathBalance}(U) := \gamma_i.\mathsf{balance}(U) + \gamma_{i+1}.\mathsf{balance}(U)$. Intuitively then, we say that a payment protocol achieves *balance security* if the $\mathbf{PathBalance}(U)$ for each honest user U does not decrease.

Formally, consider the following game. The adversary selects a $\mathsf{channelList}$, a transaction $\mathsf{tx}^{\mathsf{in}}$, a payment amount α and a timeout T such that the output $\mathsf{tx}^{\mathsf{in}}.\mathsf{output}[0]$ holds at least $n \cdot \varepsilon$ coins, where n is the length of the path defined in $\mathsf{channelList}$. The adversary sends the tuple $(\mathsf{channelList}, \mathsf{tx}^{\mathsf{in}}, \alpha, T)$ to the challenger.

The challenger sets `sid` and `pid` to two random identifiers. Then, the challenger simulates a payment from the setup phase on input $(\texttt{sid}, \texttt{pid}, \texttt{SETUP}, \mathsf{channelList}, \mathsf{tx}^{\mathsf{in}}, \alpha, T, \overline{\gamma_0})$. The challenger runs the Pay phase. Every time that a corrupted user U_i needs to be contacted, the challenger forwards the query to the attacker and waits for the corresponding answer, thereby giving the attacker the opportunity to stop payments and trigger refunds or let them be successful.

We say that the adversary wins the game if there exists an honest intermediate user U, such that $\mathbf{PathBalance}(U)$ is lower after the payment operation.

Definition 3 (Balance security)**.** We say that a payment protocol achieves balance security if for every PPT adversary $\mathcal{A}$, the adversary wins the aforementioned game with negligible probability.

Theorem 3 (Blitz achieves balance security)**.** *Blitz payments achieve balance security as defined in Definition 3.*

Proof. Assume that an adversary exists, can win the balance privacy game. This means, that after the balance security game, there exists an honest intermediate user U, such **PathBalance**(U) is lower after the payment. An intermediary U has coins locked up in its right channel when $\mathcal{F}_{Pay}$ (if right neighbor is honest) or the simulator (if right neighbor dishonest) updates this right channel to its new state. However, both $\mathcal{F}_{Pay}$ and the simulator do this only, after successfully updating also their left channel using the same $\mathsf{tx}^{\mathsf{er}}$ to fund the refund transactions in both channels.

Assume now that U has less channel balance after the payment. This would imply, that U lost its fund in the right channel without gaining any in the left. Consider two cases: (i) The left neighbor refunded in time, which implies that $\mathsf{tx}^{\mathsf{er}}$ was posted in time, which triggers also the refund in $\mathcal{F}_{Pay}$ of U in its right channel and no balance is lost. (ii) The right neighbor claimed the collateral of U's right channel. Since for an honest U, $\mathcal{F}_{Pay}$ would have automatically refunded before T if possible, this means that also in U's left channel no refund occurred. Therefore, U can claim the money put by its left neighbor and will not lose balance. This lead to the conclusion, that no such honest U exists with a lower channel balance, or if its the sender, a lower channel balance and an unsuccessful payment. □

K.3 Sender privacy

Intuitively, we say that a payment protocol achieves *sender privacy* if an adversary controlling an intermediary node cannot distinguish the case where the sender is its left neighbor in the path from the case where the sender is separated for one (or more) intermediaries.

A bit more formally, consider the following game. The adversary controls node U^* and selects two paths $\mathsf{channelList}_0$ and $\mathsf{channelList}_1$ that differ on the number of intermediary nodes between the sender and the adversary. In particular, the path $\mathsf{channelList}_0$ is formed by U_1, U^*, U_2, U_3 whereas the path $\mathsf{channelList}_1$ contains the users U_0, U_1, U^*, U_2. Note that we force both queries to have the same path length to avoid a trivial distinguishability attack based on path length. Additionally, the adversary picks transaction $\mathsf{tx}^{\mathsf{in}}$, a payment amount α as well as a timeout T such that the output $\mathsf{tx}^{\mathsf{in}}.\mathsf{output}[0]$ holds at least $n \cdot \varepsilon$ coins, where n is the length of the path defined in $\mathsf{channelList}_b$. Finally, the adversary sends two queries $(\mathsf{channelList}_0, \mathsf{tx}^{\mathsf{in}}, \alpha, T)$ and $(\mathsf{channelList}_1, \mathsf{tx}^{\mathsf{in}}, \alpha + \mathsf{fee}, T)$ to the challenger. The challenger sets $\mathtt{sid}$ and $\mathtt{pid}$ to two random identifiers. Moreover, the challenger picks a bit b at random and simulates setup and open of the Pay phase on input $(\mathtt{sid}, \mathtt{pid}, \mathtt{SETUP}, \mathsf{channelList}_b, \mathsf{tx}^{\mathsf{in}}, \alpha, T, \overline{\gamma_0})$. Every time that the corrupted user U^* needs to be contacted, the challenger forwards the query to the attacker and waits for the corresponding answer.

We say that the adversary wins the game if it correctly guesses the bit b chosen by the challenger.

Definition 4 (Sender privacy)**.** We say that a payment protocol achieves sender privacy if for every PPT adversary $\mathcal{A}$, the adversary wins the aforementioned game with probability at most $1/2 + \varepsilon$, where ε denotes a negligible value.

Theorem 4 (Blitz achieves sender privacy)**.** *Blitz payments achieve sender privacy as defined in Definition 4.*

Proof. The message $(\mathtt{sid}, \mathtt{pid}, \mathtt{open}, \mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_{i+1}, \alpha_i, T, \overline{\gamma_i}, \theta_{\varepsilon_i})$ that $\mathcal{F}_{Pay}$ sends to the simulator in the Open phase, is leaked to the adversary. By looking at $\overline{\gamma_i}$ and opening onion_{i+1}, U^* knows its neighbors U_1 and U_2. We know that U^* cannot learn any additional information about the path from T and $\overline{\gamma_i}$. Since the amount to be sent was increased fee for the path $\mathsf{channelList}_1$, the amount α_i for U_i is identical for both cases. This leaves $\mathsf{tx}^{\mathsf{er}}$, rList and onion_{i+1}. Let us assume, that there exists an adversary that can break sender privacy. There are two possible cases.
1. The adversary finds out by looking at $\mathsf{tx}^{\mathsf{er}}$ and rList: We defined that the output, that serves as input for $\mathsf{tx}^{\mathsf{er}}$, has never been used and is unlinkable to the sender and check this in $\mathtt{checkTxIn}$. Looking at the outputs of $\mathsf{tx}^{\mathsf{er}}$, the adversary knows to whom all but one output belongs. Since our adversary breaks the sender privacy, it needs to be able to reconstruct, to whom this final output of $\mathsf{tx}^{\mathsf{er}}$ belongs observing rList. This contradicts our assumption of unlinkable stealth addresses.
2. The adversary finds out by looking at onion_{i+1}: The adversary controlling U^* is able to open onion_{i+1} revealing U_2, a message m and onion_{i+2}. Since our adversary breaks the sender privacy, he has to be able to open onion_{i+2} to reveal if U_2 is the receiver or not, thereby learning who is the sender. This contradicts our assumption of secure anonymous communication networks.

These two cases lead to the conclusion, that a PPT adversary that can win the sender privacy game with a probability non-negligibly better than 1/2, can also break our assumptions of unlinkability of stealth addresses or secure anonymous communication networks. Note that the both receiver privacy and its proof are analogous to the sender privacy. □

K.4 Path privacy

Intuitively, we say that a payment protocol achieves *path privacy* if an adversary controlling an intermediary node does not know what other nodes are part of the path other than its own neighbors.

A bit more formally, consider the following game. The adversary controls node U^* and selects two paths $\mathsf{channelList}_0$ and $\mathsf{channelList}_1$ that differ on the nodes other than the adversary neighbors. In particular, the path $\mathsf{channelList}_0$ is formed by U_0, U_1, U^*, U_2, U_3 whereas the path $\mathsf{channelList}_1$ contains the users $U_0', U_1, U^*, U_2, U_3'$. Note that we force both queries to have the same path length to avoid a trivial distinguishability attack based on path length. Further note that we force that in both paths, the adversary has the same neighbors as otherwise there exists a trivial distinguishability attack based on what neighbors are used in each case.

Additionally, the adversary picks transaction $\mathsf{tx}^{\mathsf{in}}$, a payment amount α as well as a timeout T such that the output $\mathsf{tx}^{\mathsf{in}}.\mathsf{output}[0]$ holds at least $n \cdot \varepsilon$ coins. Finally, the adversary sends two queries $(\mathsf{channelList}_0, \mathsf{tx}^{\mathsf{in}}, \alpha, T)$ and $(\mathsf{channelList}_1, \mathsf{tx}^{\mathsf{in}}, \alpha, T)$ to the challenger.

The challenger sets `sid` and `pid` to two random identifiers. Moreover, the challenger picks a bit b at random and simulates the setup and open phases on input $(\mathtt{sid}, \mathtt{pid}, \mathtt{SETUP}, \mathsf{channelList}_b, \mathsf{tx}^{\mathsf{in}}, \alpha, T, \overline{\gamma_0})$. Every time that the corrupted user U^* needs to be contacted, the challenger forwards the query to the attacker and waits for the corresponding answer.

We say that the adversary wins the game if it correctly guesses the bit b chosen by the challenger.

Definition 5 (Path privacy)**.** We say that a payment protocol achieves path privacy if for every PPT adversary $\mathcal{A}$, the adversary wins the aforementioned game with probability at most $1/2 + \varepsilon$, where ε denotes a negligible value.

Theorem 5 (Blitz achieves path privacy)**.** *Blitz payments achieve path privacy as defined in Definition 5.*

Proof. As this proof is analogous to the proof for sender privacy, we only sketch it here. Again, the simulator leaks the same message $(\mathtt{sid}, \mathtt{pid}, \mathtt{open}, \mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_{i+1}, \alpha_i, T, \overline{\gamma_i}, \theta_{\varepsilon_i})$ to the adversary. Again, the adversary can find out the correct bit b by looking at (i) $\mathsf{tx}^{\mathsf{er}}$ and rList or (ii) at onion_{i+1}. If there exists an adversary that breaks the path privacy of Blitz, then it also can be used to break (i) unlinkability of stealth addresses or (ii) secure anonymous communication networks. □