Donner: UTXO-Based Virtual Channels Across Multiple Hops

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 are a promising solution to the scalability issues of current decentralized cryptocurrencies. They allow arbitrarily many payments between any two users connected through a path of intermediate payment channels while minimizing interaction with the blockchain only to open and close those channels. Yet, compromised intermediaries may make payments unreliable, slower, expensive, and privacy-invasive. Virtual channels mitigate these issues by allowing the two endpoints of a path to create a channel over the intermediaries such that after the channel is constructed, the intermediaries are no longer involved in payments. Unfortunately, existing UTXO-based virtual channel constructions are either limited to a single intermediary or only recursively build a virtual channel over multiple intermediaries. While the former single-hop channels are overly restrictive, the latter recursive constructions introduce issues such as forced closure and virtual griefing attacks.

This work presents Donner, the first virtual channel construction over multiple intermediaries in a single round of communication. We formally define the security and privacy in the Universal Composability framework and show that Donner is a realization thereof. Our experimental evaluation shows that Donner reduces the on-chain number of transactions for disputes from linear in the path length to a single one. Moreover, Donner reduces the storage overhead from logarithmic in the path length to constant. Donner is an efficient virtual channel construction that is backward compatible with the prominent, 50K channels strong Lightning network.

1 Introduction

The permissionless nature of the consensus algorithm that governs cryptocurrencies today such as Bitcoin heavily limits their transaction throughput to few transactions per second. This contrasts with centralized approaches such as Visa where a single server copes with peaks of 64000 transactions per second. This scalability issue hinders thus permissionless cryptocurrencies from serving a growing base of payments.

Payment channels. Among the different research efforts, payment channels have emerged as one of the most promising scalability solutions, with the Ligthning Network [21] being the most popular realization in Bitcoin. A payment channel enables arbitrarily many payments between two users while requiring to commit only two transactions to the ledger: one to open and one to close the channel. Alice and Bob can create a payment channel by a single on-chain transaction that locks bitcoins (also called *collateral*) into a multi-signature address controlled by both users. After that, they can pay each other off-chain (i.e., without interacting with the blockchain) an arbitrary number of times by exchanging authenticated messages that represent an update of their share of coins in the multi-signature address. After they are done with it, any of the two users can close the channel by submitting a transaction on-chain with the last authenticated distribution of bitcoins.

Interestingly, it is possible to leverage a path of opened payment channels with enough capacity between two users to perform a payment between them, thereby creating a payment channel network (PCN). Assume that Alice wants to pay to Bob and assume that they do not share a direct payment channel, but rather they are connected by a path of channels going through k intermediaries $I_1, \dots, I_k$ (i.e., through the channels (Alice, I_1), …, (I_k,Bob)). A payment from Alice to Bob requires to update each channel on the path: the challenge is to ensure atomicity, that is, either all channels are updated consistently or none of them are. The most popular solution is based on Hashed Timelock Contracts (HTLCs): channel updates are conditioned to the receiver revealing the preimage of a certain hash and locked for an amount of time that grows linearly from the sender to the receiver (in the Lightning Network, one day per hop), which requires users including the intermediaries to be online. This approach has multiple disadvantages, including (i) making the payments less reliable (e.g., some intermediary might go offline); (ii) increasing the latency of each payment; (iii) augmenting the transaction costs (e.g., intermediaries typically charge a fee for the forwarding service); and (iv) leaking sensitive data to intermediaries, e.g., partial information about who pays what to whom.

Note that solving these problems while retaining a decentralized off-chain network is very hard. In fact, most off-chain protocols suffer from similar problems. For instance, payment-channel hubs [14, 24] assume a star topology, in which a central hub is connected to all users, and all parties involved in the payment (i.e., sender, receiver, and hub) to be online. Atomic multiparty payments (e.g., Sprites [20], AMCU [12], and payment trees [16]) assume all parties to be online and make them interact with each other, consequently learning the identities of all the users involved in a payment.

Virtual channels. Virtual channels [10] constitute the most promising solution to the aforementioned problems of off-chain protocols. To explain their basic functionality, let us initially assume that Alice and Bob are connected by a single intermediary Ingrid. First, Ingrid must collaborate with Alice and Bob to coordinate the update of both payment channels to fund the virtual channel, i.e., a collateral is locked for this purpose. However, once the virtual channel is created, Alice and Bob can pay each other arbitrarily many times without the involvement of Ingrid. Finally, when Alice and Bob are done with the virtual channel, they can close it, that is, they establish collaboratively with Ingrid their balance as an off-chain update of their payment channels with Ingrid. The assumption of a single intermediary is suitable in certain settings (e.g., a hub topology where a user needs to have a payment channel with the hub and can then create a virtual channel with the many other users connected to the same hub), but restrictive in a decentralized setting. Recursive constructions [11] solve this issue allowing for creating virtual channels on top of other virtual channels (or a pair composed of a virtual and a payment channel), supporting arbitrary many intermediaries. Dziembowski et al [9] further extended the expressiveness of virtual channels proposing the notion of multi-party virtual channels, where a set of n participants build an n-party channel recursively from their pair-wise payment/virtual channels.

In this manner, virtual channels overcome the drawbacks of PCNs: (i) payments no longer rely on Ingrid being online; (ii) the latency is reduced to a direct payment between two users; (iii) Ingrid does not charge a fee for each payment (perhaps only once to create and close the virtual channel); (iv) Ingrid does not learn the amount of each single payment.

While all these constructions rely on the Turing-complete scripting capabilities of Ethereum, Aumayr et al. [3] have later shown, perhaps surprisingly, that it is possible to realize a Bitcoin-compatible virtual channel construction through carefully crafted cryptographic protocols in the Unspent Transaction Output (UTXO) model, albeit this protocol supports only one intermediary. Jourenko et al. [15] have recently introduced the first Bitcoin-compatible recursive construction over multiple intermediaries, named Lightweight Virtual Payment Channel (LVPC).

Drawbacks of UTXO-based recursive virtual channels. In this work, we observe that the recursive construction underlying LVPC [15] suffers from three fundamental drawbacks, which fundamentally undermine its security, privacy, and efficiency. Indeed, we conjecture that these limitations are not specific to this individual construction but are inherent to the recursive paradigm when deployed in the context of UTXO-based transactions.

Table 1: Comparison to other virtual channel schemes.

	GSCN [11]	MPVC [9]	LVPC [15]	Donner
Scripting req.	Ethereum	Ethereum	Bitcoin	Bitcoin
multi-hop	recursive	recursive	recursive	one round
Virtual griefing attack	yes	yes	yes	no
Forced closure attack	no	no	yes	no
Path privacy	no	no	no	yes

- Since virtual channels in the UTXO-based model are funded on top of each other, the end-point of a top-level virtual channel has the possibility to perform what we call a *forced closure attack*: by closing on-chain the virtual channel, the attacker forces the on-chain closure of all underlying virtual and payment channels, as the respective owners would otherwise lose money[1]. This attack makes the payment channel network volatile and forces honest users to perform additional on-chain transactions (paying the associated fees) to re-establish their channels;
- Virtual channels at different recursion layers have associated an increasing (also called staggered) timelock to ensure that the virtual channels can be closed in order, which opens the door to *virtual griefing attacks*. Similar to griefing in payment-channel networks [12], a user controlling two nodes can open a virtual channel to itself at the top recursion layer and keep it open until right before the timelock expires, effectively locking the funds used as collateral at each virtual channel in the underlying recursion layers.
- Recursive virtual channel constructions do not achieve the *path privacy* property guaranteed in the Lightning Network (intuitively, each intermediary does not learn more information than the one exposed by its own payment channels), as intermediaries have to perform transactions with other parties as well: for instance, if the sender constructs the virtual channel recursively by integrating the payment channels one by one from left to right, all intermediaries have to transact with the sender, which clearly reveals its identity to all other users in the path.
- The number of opening and offloading transactions grows with the number of intermediaries, which translates into a significant price to pay in terms of number of transactions and associated fees.

Hence, it is an open question, of theoretical and practical relevance, whether secure, privacy-preserving, and practical virtual channels across multiple hops are possible in UTXO-based cryptocurrencies featuring only a limited scripting language like Bitcoin.

[1] Ethereum-based constructions do not necessarily suffer from this attack, since it is possible to program in the smart contracts logical conditions that avoid financial losses without involving the closure of a channel.

Contribution. In this work, we give a positive answer, by advocating a radical paradigm shift in the design of virtual channels over multiple intermediaries, dispensing for the first time from a recursive construction. In particular,

- We present Donner, the first virtual channel construction that supports the creation of virtual channels over multiple intermediaries in one round of communication. As a result, Donner prevents virtual griefing and forced closure attacks, besides being more efficient and privacy-preserving than the state-of-the-art LVPC [15]. Donner relies only on digital signatures and a timelock functionality, without requiring HTLCs: Donner is thus compatible with Bitcoin, and also with cryptocurrencies that do not support HTLCs, such as Ripple and Stellar.
- We conduct a formal security and privacy analysis of Donner in the Universal Composability framework.
- We conduct an experimental evaluation, measuring the on-chain and off-chain overhead of Donner and demonstrating that Donner requires significantly less transactions than LVPC. Most notably, Donner requires only one on-chain transaction to offload the virtual channel in case of a dispute, while honest channels can perform off-chain updates without (the risk of) losing funds, compared to a linear number of on-chain transactions in LVPC. Furthermore, the storage overhead per payment channel is reduced from linear or logarithmic in LVPC, depending on how the VC is constructed, to constant.

We summarize the comparison between Donner and other virtual channel constructions in Table 1.

2 Background and notation

In this section we overview the background and present the notation used throughout the paper.

2.1 UTXO based blockchains

In this work, we focus on blockchains that follow the *unspent transaction output* (UTXO) model, such as Bitcoin. We adopt the notation for UTXO-based blockchains from [2], which we shortly review next. In UTXO-based blockchains, the units of currency, i.e., the *coins*, exist in *outputs* of transactions. We define such an output as a tuple $\theta := (\mathsf{cash}, \phi)$; $\theta.\mathsf{cash}$ contains the amount of coins stored in this output, while $\mathsf{tx}.\phi$ defines the condition under which the coins can be spent. The latter is done by encoding such a condition in the scripting language supported by the underlying blockchain. This can range from simple ownership, specifying which public key can spend the output, to more complex conditions such as time-locks, multi-signatures, or logical boolean functions.

Coins can be spent with *transactions*, resulting in the change of ownership of the coins. A transaction maps a list of outputs to a list of new outputs. For better readability, we denote the former outputs as transaction *inputs*. Formally, we define a transaction body as a tuple $\mathsf{tx} := (\mathsf{id}, \mathsf{input}, \mathsf{output})$. The identifier $\mathsf{tx.id} \in \{0,1\}^*$ is assigned as the hash of the other attributes, $\mathsf{tx.id} := \mathcal{H}(\mathsf{tx.input}, \mathsf{tx.output})$. We model $\mathcal{H}$ as a random oracle. The attribute $\mathsf{tx.input}$ is a non-empty list of the identifiers of the transaction's inputs and $\mathsf{tx.output} := (\theta_1, \dots, \theta_n)$ a non-empty list of new outputs. To prove that the spending conditions of the inputs are known, we introduce full transactions, which contain in addition to the transaction body also a witness. We define a full transaction $\overline{\mathsf{tx}} := (\mathsf{id}, \mathsf{input}, \mathsf{output}, \mathsf{witness})$ or for convenience also $\overline{\mathsf{tx}} := (\mathsf{tx}, \mathsf{witness})$. Valid transactions can be recorded on the public ledger $\mathcal{L}$ called blockchain. A transaction is valid if and only if (i) all its inputs are not spent by other transaction on $\mathcal{L}$; (ii) provides a valid witness for the spending condition ϕ of every input; and (iii) the sum of coins in the outputs is equal (or smaller) than the sum of coins in the inputs.

In practice, transactions are aggregated in blocks. These blocks are not immediately accepted to the blockchain but only after they are accepted by the participants of a distributed consensus mechanism (e.g., proof-of-work as in Bitcon). We model this delay through Δ, which we define as an upper bound on the time from when a transaction is broadcast to when it is accepted to $\mathcal{L}$.

To visualize how transactions are used in a protocol along with complex spending conditions in a more readable way, we use transaction charts. The arrows indicate that the charts are to be read from left to right. Rounded rectangles represent transactions, with incoming arrows being their inputs. The boxes within the transactions are the outputs and the value in them represents the amount of output coins. The arrow(s) going from these output boxes show the conditions under which an output can be spent.

More specifically, below an arrow we write who can spend the coins. This is usually a signature that verifies w.r.t. one or more public keys, which we denote as $\mathsf{OneSig}(\mathsf{pk})$ or $\mathsf{MultiSig}(\mathsf{pk}_1, \mathsf{pk}_2, \dots)$. Above the arrow, we write additional conditions for how an output can be spent. This could be any script supported by the scripting language of the underlying blockchain, but in this paper we only use relative and absolute time-locks. For the former, we write $\mathsf{RelTime}(t)$ or simply $+t$, which signifies that the output can be spent only if at least t rounds have passed since the transaction holding this output was accepted on $\mathcal{L}$. Similarly, we write $\mathsf{AbsTime}(t)$ or simply $\geq t$ for absolute time-locks, which means that the transaction can be spent only if the blockchain is at least t blocks long. A condition can be a disjunction of subconditions $\phi = \phi_1 \vee \dots \vee \phi_n$, which we denote as a diamond shape in the output box, with an outgoing arrow for each subcondition. A conjunction of subconditions is simply written as $\phi = \phi_1 \wedge \dots \wedge \phi_n$. We illustrate this notation in Figure 1.

2.2 Payment channels

Two users can utilize a payment channel in order to perform arbitrarily many payments, while putting only two transactions on the ledger. On a high level, there are three operations in a payment channel operation: *open*, *update* and *close*. First,

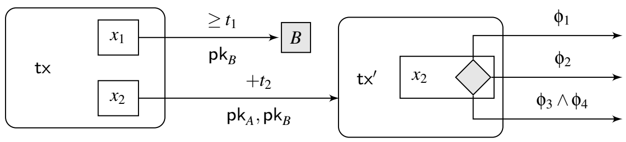

Figure 1: (Left) Transaction tx has two outputs, one of value x_1 that can be spent by B (indicated by the gray box) with a transaction signed w.r.t. pk_B at (or after) round t_1, and one of value x_2 that can be spent by a transaction signed w.r.t. pk_A and pk_B but only if at least t_2 rounds passed 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.

to open a channel, both users have to lock up some money in a shared output (i.e., an output that is spendable if both users give their signature) in a transaction called the *funding transaction* or tx^{f}. From this output, they can create a new transaction called *state* that assigns each of them a balance. Once the funding transaction is on the ledger, the users can exchange arbitrarily many new states (balance updates) in a peer-to-peer way, thereby realizing the update phase of the channel. Once they are done, they can close the channel by posting the final state to the ledger.

In this work, we use payment channels in a black-box manner and refer the reader to [2, 18, 19] for more details. We assume that there is a transaction $\mathsf{tx}^{\mathsf{state}}$ off-chain, which contains the outputs representing the most recent payment channel state. For simplicity, we assume that this is the only state that the users can publish and abstract away from how the dishonest behavior is handled. In practice, of course it is possible that a dishonest user publishes a stale state of the channel and current constructions come with a way to handle this cases (e.g., through a punishment mechanism that gives all the coins to the honest user [2]).

We model payment channels as tuples: $\overline{\gamma} := (\mathsf{id}, \mathsf{users}, \mathsf{cash}, \mathsf{st})$. The attribute $\overline{\gamma}.\mathsf{id} \in \{0,1\}^*$ uniquely identifies a channel; $\overline{\gamma}.\mathsf{users} \in \mathcal{P}^2$ identifies the two parties involved in the channel out of the set of all parties $\mathcal{P}$. Moreover, $\overline{\gamma}.\mathsf{cash} \in \mathbb{R}_{\leq 0}$ denotes the total monetary capacity of the channel and the current state is stored as a vector of outputs of $\mathsf{tx}^{\mathsf{state}}$: $\overline{\gamma}.\mathsf{st} := (\theta_1, \ldots \theta_n)$. In this work, we use channels in paths from a sender to a receiver. For simplicity, we say that $\overline{\gamma}.\mathsf{left} \in \overline{\gamma}.\mathsf{users}$ refers to the user closer to the sender, while $\overline{\gamma}.\mathsf{right} \in \overline{\gamma}.\mathsf{users}$ refers to the user closer to the receiver. The balance of both users can always be inferred from the current state $\overline{\gamma}.\mathsf{st}$. For convenience, we say that $\overline{\gamma}.\mathsf{balance}(U)$ gives the coins owned by $U \in \overline{\gamma}.\mathsf{users}$ in this channel's latest state $\overline{\gamma}.\mathsf{st}$. Finally, we define a channel skeleton γ for a channel $\overline{\gamma}$, as $\gamma := (\overline{\gamma}.\mathsf{id}, \overline{\gamma}.\mathsf{users})$.

2.3 Payment channel networks

A payment-channel network (PCN) [18] is a graph where the nodes represent the users and the edges represent the payment channels. The Lightning Network [21] is the state-of-the-art in both payment channels and PCNs for Bitcoin, and the largest PCN in terms of coins locked within its channels.

In a PCN, any two users that are connected to each other via a path of channels can perform what is called a *multi-hop payment* (MHP). Assume that there is a sender U_0 who wants to pay α coins to a receiver U_n, but they do not have a direct channel. Instead, they are connected by a path of channels going through intermediaries $\{U_i\}_{i\in[1,n-1]}$, i.e., U_0 and U_1 have a channel $\overline{\gamma_0}$, U_1 and U_2 have a channel $\overline{\gamma_1}$ and so on. A MHP allows to transfer coins from U_0 to U_n through $\{U_i\}_{i\in[1,n-1]}$ in a secure way, that is, ensuring that no intermediary is at risk of losing money. The Lightning Network [19, 21] adopts a two-round MHP protocol, where coins are locked along the path from left to right and, after reaching the receiver, unlocked from right to left. Besides requiring a two-round communication, this protocol introduces a collateral (intuitively, the time coins have to be locked on the channels) that is linear on the size of the path, which opens the door to denial-of-service attacks, also called griefing attacks in the literature. More recently, Blitz [4] improves on that by requiring only one round of communication and a constant collateral.

2.4 Multi-hop virtual channels

A multi-hop virtual channel (VC) allows two users U_0 and U_n connected via a path of payment channels over some intermediaries $\{U_i\}_{i\in[1,n-1]}$ to open and maintain a direct channel between them without the need of (i) involving the intermediaries in payments and (ii) performing any on-chain operation. This type of channels is called virtual because their funding output is not included in the blockchain but rather built using underlying payment channels, which we call base (or payment) channels from now on. In a sense, a VC can be seen as a payment channel "on top of" other base channels.

On a high level, there are four operations in a VC: *open*, *update*, *close*, and *offload*. First, U_0 and U_n, along with the intermediaries, open a VC by somehow funding off-chain a shared output. Depending on where the funding inputs for such shared output come from, we say that a VC is either *(fully) rooted* or *decoupled from its underlying base channels*, depending on whether the VC spends inputs that are (directly or indirectly) coming from the underlying base channels or from somewhere else, respectively.

After it is opened, U_0 and U_n can update the balance in their VC arbitrarily many times without any interaction with the intermediaries. Finally, after U_0 and U_n are done using the VC, in the optimistic case they can close it by establishing the last balance in the VC into the underlying base channels. Otherwise, the VC can be offloaded, transforming the VC into a payment channel, putting the funding transaction on-chain.

3 Key idea

We identify three main challenges in the design of multi-hop VCs for UTXO-based cryptocurrencies, as described next.

C1: Secure, efficient, and decentralized opening. As previously mentioned, a VC has to be funded off-chain. Since U_0 and U_n do not share a base channel, the intermediaries have to be involved in the opening phase, allocating a *collateral* (i.e., locking coins on the VC). This operation has to fulfill a number of requirements: if the VC is offloaded or closed, nobody should be at risk of losing its money or having its money locked indefinitely (security); the opening should require as little transactions and collateral as possible (efficiency); the intermediaries should transact only with the end-points of their channels, avoiding third-parties that may constitute bottlenecks or multiparty computation protocols that may reveal the identity of the involved users.

C2: Guaranteed closure. In the pessimistic case (i.e., in case of a dispute on the balance or unresponsiveness), the VC endpoints need a way to secure their balances. There are three possible options: a party can by itself offload the VC, a party can initiate an offload sequence and the others have to react in order not to lose coins, or a party is guaranteed to receive compensation if, by some agreed upon time, the VC is not offloaded or honestly closed. Once a channel is offloaded, both endpoints can close the VC with standard payment channel techniques.

Note that the guaranteed closure property should hold for intermediaries as well in order to prevent their balance from being indefinitely locked. In practice, it is often desirable to guarantee a duration validity for the virtual channel, after which the intermediaries can get back their collateral: this makes fee models potentially fairer, as the intermediaries can charge exactly for the the amount they lock and for how long. Additionally, this guarantees to the endpoints that no intermediary offloads the VC during such time.

C3: Atomic closure. In the optimistic case (i.e., all parties collaboratively close the VC), all base channels must be atomically updated to reflect the final balance of the VC. This is challenging as some of the malicious intermediaries might try to maximize their profit at the expense of the other parties.

Recursive, rooted solutions. A possible approach to solve these challenges is the recursive one, i.e., to let the sender construct a VC over a single intermediary (such as [3, 15]), subsequently construct a new VC funded on top of the previous VC and a contiguous base channel, or alternatively on top of two VCs, and so on and so forth, until the receiver is reached. Unfortunately, this approach suffers from three fundamental drawbacks:

1. Forced Closure Attack In the UTXO setting, a fully rooted, recursive channel is funded by outputs coming from all base channels. The way existing constructions allow an endpoint to initiate the offloading of the VC (cf. C2) is to give this party a transaction that transfers all the money on the VC to itself, unless within a certain time the intermediaries and the other endpoint claim their balance on-chain. This way all parties are incentivized to take part in the close. However, this also means that an end-point can force all rational parties (i.e., those who do not want to lose money) to close their channels on-chain. The fundamental problem is that the funding of the VC is rooted into the base channels, which ultimately forces their closure in the pessimistic case. We present the forced closure attack in more detail in Appendix A.

2. Virtual Griefing Attack The time it takes to offload a VC grows linearly in the number of layers, which coincides with the number of intermediaries between the endpoints. This means that an attacker can establish a virtual channel with itself, locking the the collateral of the intermediaries for an amount of time linear in the number of intermediaries. We present the forced closure attack in more detail in Appendix B.

3. Lack of Scalability The number of on-chain transactions to handle disputes among participants grows linearly with the number of layers, which translates not only in off- and on-chain storage consumption but also in on-chain fees. We show the linear grow of on-chain transactions in Section 6.2.

3.1 Our solution

We now give a high-level overview of our protocol, highlighting how we solve the aforementioned challenges in the design of VCs and, in particular, how we prevent the forced closure and virtual griefing attacks as well as the lack of scalability inherent to recursive constructions. For a more detailed protocol description and security analysis, we defer the reader to Section 4 and Appendix E, respectively.

Solving C1 and C2. To prevent forced closure attacks, we decouple the funding of the VC from the one of the base channels, letting the VC be funded only by the sender. The funding transaction, let us call it $\mathsf{tx}^{\mathsf{vc}}$, originating from the sender, can either spend an on-chain or an off-chain output.

Naturally, this brings two questions: how to allow intermediaries to allocate collateral (C1), which is necessary as sender and receiver do not share a channel and therefore someone has to borrow the money on behalf of the other endpoint, and how to guarantee that the channel can be closed or offloaded by the receiver (C2), which is crucial to ensure that the receiver's money is not locked forever.

We address both questions by letting (i) the sender insert an additional output for every base channel into the funding transaction $\mathsf{tx}^{\mathsf{vc}}$, (ii) a collateral (corresponding to the funding amount) be paid in each base channel by the left endpoint to the right one, and (iii) this collateral be refunded if and only if the $\mathsf{tx}^{\mathsf{vc}}$ transaction goes on-chain before a certain timeout T. In other words, this mechanism synchronizes the collateral on each base channel in a way that the sender is incentivized to close or offload the channel before its validity, otherwise the receiver gets the entire funding.

Interestingly enough, a similar synchronization mechanism is used in the recently proposed Blitz MHP protocol [4], where a transaction close in spirit to $\mathsf{tx}^{\mathsf{vc}}$ serves the completely different purpose of allowing the sender to withdraw the multi-hop payment in an atomic way along the payment channels in case the payment does not reach the receiver.

Solving C3. We now describe how to close the VC honestly, that is without offloading or going on-chain at all and update all underlying channels accordingly. For doing that, we rely on an incentive-based approach. The first key observation is that every intermediary can update the collateral on the right channel to a value that is smaller than the previous one without fear of losing money, as it can get the previous higher collateral from the left channel. The second one is that the final balance of the receiver in the VC cannot be higher than the funding initially provided by the sender, which corresponds to the collateral. Since the receiver U_n knows that if it does not initiate an honest closing, the sender will offload the VC, it can go ahead and update the collateral in its channel with U_{n-1} to the smaller amount that the receiver holds in the VC (technically, this proceeds by replacing the transactions reflecting the old balance in the base channels with fresh ones reflecting the new one). The neighbor U_{n-1} can safely update the right channel too, as it is still guaranteed the full amount from its left channel. If U_{n-1} does not continue, the sender will offload and trigger the refund on the payments, therefore U_{n-1} is incentivized to continue, and so on and so forth until the sender U_0 is reached and the base channels are all updated according to the final balance of the receiver in the VC. After this is done, the payment will simply go through after the timeout T expires.

Putting it together. We now describe the protocol in slight more detail. The sender U_0 chooses a lifetime T for the VC and creates on its own the transaction $\mathsf{tx}^{\mathsf{vc}}$ (see Figure 4), which acts both as a funding for the VC and as a condition for the refund of the collateral. In particular, $\mathsf{tx}^{\mathsf{vc}}$ includes one output to fund the VC $\theta_{\mathsf{vc}} := (\alpha, \mathsf{MultiSig}(U_0, U_n))$ and one $\theta_{\varepsilon_i} := (\varepsilon, \mathsf{OneSig}(U_i) + \mathsf{RelTime}(t_c + \Delta))$ for each $i \in [0, n-1]$. The relative timelock ensures that there is enough time for everyone to close the channel, before refunding the collateral.

At this point, from left to right, the endpoints of each base channel set up the collateral (see Figure 2). The amount of the collateral is the full channel capacity α. While intermediaries might charge a fee for providing a base channel for a VC, we omit it for easing the presentation and refer to Section 4.4 for a discussion on this point. Technically, to set up the collateral, the left user U_i in the channel between U_i and U_{i+1} creates a new state where α coins from its balance are locked in an output $(\alpha, (\mathsf{MultiSig}(U_i, U_{i+1}) \wedge \mathsf{RelTime}(\Delta)) \vee (\mathsf{OneSig}(U_{i+1}) \wedge \mathsf{AbsTime}(T)))$.

This complex looking script simply stipulates that either both channel users collaboratively spend the output with a relative timelock of Δ, or after the absolute time T expires,

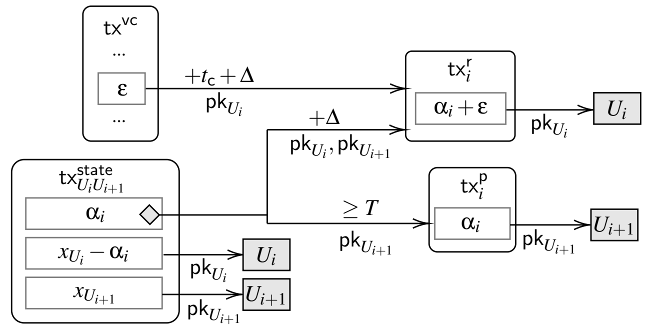

Figure 2: Contract to setup the collateral in each channel for users U_i and U_{i+1}

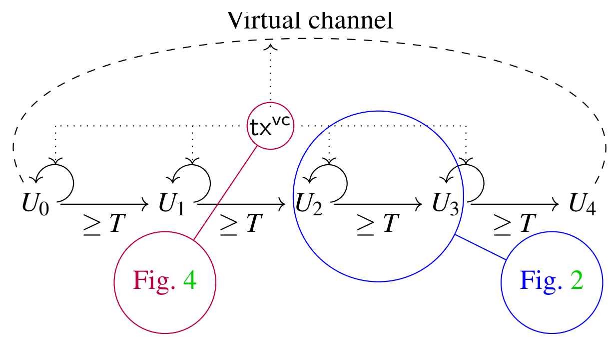

Figure 3: Illustration of the Donner virtual channel construction: The contract in each channel is the same as in Blitz (Figure 2), but references $\mathsf{tx}^{\mathsf{vc}}$ (Figure 4) instead of $\mathsf{tx}^{\mathsf{er}}$.

U_{i+1} can spend it on its own. Then, they create a transaction $\mathsf{tx}^{\mathsf{r}}_i$ that spends from this output and from the output θ_{ε_i} in $\mathsf{tx}^{\mathsf{vc}}$ designated for this channel back to the user U_i and U_{i+1} gives its signature to U_i. This way the refund $\mathsf{tx}^{\mathsf{r}}_i$ is the only way that U_i can on its own spend the collateral. Once the collateral has been set up in the last base channel with the receiver, the VC is open and can be used for payments. The whole scheme is illustrated in Figure 3.

We point out that the receiver is guaranteed to get either its money from the collateral or if the VC is offloaded. The intermediaries do not lose money, as the $\mathsf{tx}^{\mathsf{vc}}$ ensures that all

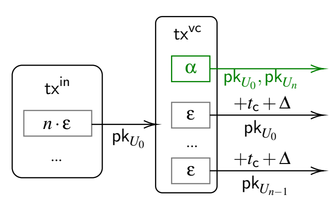

Figure 4: Transaction $\mathsf{tx}^{\mathsf{vc}}$ spending from an output under U_0's control, funding the virtual channel (green output) and linking to the collateral in each channel.

collateral is refunded atomically or not. Finally, the sender can in any case post $\mathsf{tx}^{\mathsf{vc}}$ to avoid losing money. To honestly close, the base channels are sequentially updated from right to left, as previously explained, such that the amount of the collateral is reduced to the receiver's final balance in the VC.

Channel life time. We point out, that our VC has a limited life time T, chosen by the sender, during which the VC is active. After this time, the VC is either honestly closed or offloaded, otherwise the sender will lose its collateral. Having a life time allows for a fairer fee model for the intermediaries. The intermediaries now know exactly how many coins are locked for how long and can charge an according fee for this service. Moreover, the sender and the receiver are sure that no intermediary can prematurely close the channel. We note that, similarly to the way the channel is closed honestly by updating the collateral amount in each base channel from right to left, the life time of the channel could be prolonged by sequentially updating the time in the collateral from right to left (the intermediaries might then charge some more fees).

Unidirectionally funded. Similar to current payment channels in the Lightning Network, our VCs are only funded by U_0, whom we call the sending endpoint or sender. User U_n is the receiving endpoint or receiver and the intermediaries are $\{U_i\}_{i\in[1,n-1]}$. Even though the VC is only funded by U_0, once some money has been moved, they can use the channel also in the other direction. Moreover, if they want to have a channel funded from both endpoints, they can simply construct another channel from U_n to U_0.

Privacy. There occur no on-chain transactions in the optimistic case throughout the protocol. Any two users connected in the payment channel network can open a VC, and apart from their open and close balance, the amount of the individual transaction remains known only to them, even in the pessimistic case. Moreover, in the optimistic case, the sender, the receiver and the path through which the channel is routed remains hidden, as the keys used in the $\mathsf{tx}^{\mathsf{vc}}$ can be freshly generated and unlinkable to previous transactions.

4 Protocol

In this section we present our security and privacy goals of our construction, we list the required assumptions and prerequisites, and finally detail our construction.

4.1 Security and privacy goals

We informally define three security and three privacy goals for our virtual channel construction. We mark security goals with an S and privacy goals with a P. For formal definitions and proofs, we refer the reader to Appendix F.

(S1) Balance security. Honest intermediaries do not lose any coins when participating in the virtual channel construction.

(S2) Endpoint security. No malicious users can steal the sender's rightful balance of the virtual channel. Additionally, the receiver is always guaranteed to get at least its rightful balance of the virtual channel.

(S3) Reliability. No malicious intermediary or colluding malicious intermediaries can force two honest endpoints of a virtual channel to close or offload the virtual channel before the lifespan T of the virtual channel expires.

(P1) Endpoint anonymity. In case of an optimistic virtual channel execution, malicious intermediaries cannot distinguish if their left (right) user is the sending (receiving) endpoint or merely an honest intermediary connected to the sending (receiving) endpoint via other, non-compromised users.

(P2) Path privacy. In case of an optimistic virtual channel execution, malicious intermediaries do not learn any identifiable information about the other intermediaries, except for their direct neighbors.

(P3) Value privacy. The users on the path learn only about the initial and the final balance of the virtual channel, not the value of the individual payments.

The careful reader may have noticed that, while the security properties and P3 hold always, the properties P1 and P2 hold only for the optimistic case. Indeed, like in any other off-chain protocol (e.g., the Lightning Network), the channels have to go on-chain in order to resolve disputes in the worst case. This means that anyone observing the blockchain can reconstruct the path. Note, however, that this happens rarely, as the optimistic case is less costly for the participants. Designing off-chain protocols that achieve privacy even in case of disputes is an interesting future question.

4.2 Assumptions and prerequisites

Digital signatures. A digital signature scheme is a tuple of algorithms $\Sigma := (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Vrfy})$. On a high level, $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(\lambda)$ is a PPT algorithm that on input a security parameter λ generates a keypair $(\mathsf{pk}, \mathsf{sk})$. The public key pk is publicly known, while the secret key sk is only known to the user who generated that keypair. $\sigma \leftarrow \mathsf{Sign}(\mathsf{sk}, m)$ is a PPT algorithm that on input a secret key sk and a message $m \in \{0,1\}^*$ generates a signature σ of m. Finally, $\{0,1\} \leftarrow \mathsf{Vrfy}(\mathsf{pk}, \sigma, m)$ is a DPT algorithm that on input a public key pk, a message m and a signature σ outputs 1 iff the signature is a valid authentication tag for m w.r.t. pk. We use a EUF-CMA secure [13] signature scheme Σ as a black-box throughout this work.

Stealth addresses. In order to hide the underlying path, we use stealth addresses [25] for the outputs in the transaction $\mathsf{tx}^{\mathsf{vc}}$. On a high level, every user U controls two private keys a and b. The respective public keys A and B are publicly known. A sender can use these public keys controlled by U to create a new public key P and a value R. The user U and only the user U knowing a and b can use R, P together with a and b to construct the private key p. In particular, also the sender is unaware of p. This new one-time public key is unlinkable to U by anyone observing only R and P [25].

Onion routing. Like in the Lightning Network, we rely on onion routing [6] techniques like Sphinx [8] to allow users communicate anonymously with each other. This allows users to route the virtual channel correctly through the desired path, while ensuring that intermediaries remain oblivious to the path except for their direct neighbors. On a high level, an *onion* is a layered encryption of routing information and a payload. Each user in turn can peel off one layer, revealing the next user on the path, the payload meant for the current user and another onion, which is designated for the next user. For simplicity, we use onion routing by calling the following two functions: $\mathsf{onion} \leftarrow \mathsf{CreateRoutingInfo}(\{U_i\}_{i\in[1,n]}, \{\mathsf{msg}_i\}_{i\in[1,n]})$ generates an onion using the public keys of users $\{U_i\}_{i\in[1,n]}$ on the path, while $\mathsf{GetRoutingInfo}(\mathsf{onion}_i, U_i)$ returns $(U_{i+1}, \mathsf{msg}_i, \mathsf{onion}_{i+1})$ when called by the correct user U_i, or $\perp$ otherwise.

Ledger and channels. We use the a ledger (or blockchain) and a PCN (both introduced in Section 2) as black-boxes in our construction. The ledger keeps a record of all transactions and balances and is append-only. The PCN supports opening, updating and closing of payment channels. For simplicity, we assume the payment channels involved in VCs to be already open. We interact with ledger and PCN through the following procedures.

$\mathsf{publishTx}(\overline{\mathsf{tx}})$**:** The transaction $\overline{\mathsf{tx}}$ is appended to the ledger after at most Δ time, if it is valid.

$\mathsf{updateChannel}(\overline{\gamma_i}, \mathsf{tx}_i^{\mathsf{state}})$**:** This procedure initiates an update in the channel $\overline{\gamma_i}$ to the state $\mathsf{tx}_i^{\mathsf{state}}$, when called by a user $\in \overline{\gamma_i}.\mathsf{users}$. The procedure terminates after at most t_{u} time and returns $(\mathsf{update{-}ok})$ in case of success and $(\mathsf{update{-}fail})$ in case of failure to both users.

$\mathsf{closeChannel}(\overline{\gamma_i})$**:** This procedure closes the channel $\overline{\gamma_i}$, when called by a user $\in \overline{\gamma_i}.\mathsf{users}$. The latest state transaction $\mathsf{tx}_i^{\mathsf{state}}$ appears on the ledger after at most t_{c} time.

Assumptions. In our construction, we assume that every user U has a public key pk_U as well as a public key pair (A,B). The former is used to receive transactions, the latter is used for stealth address generation. Additionally, we say that honest participants of the protocol stay online for the duration of the protocol. A path finding algorithm to identify a payment path can be called by $\mathsf{pathList} \leftarrow \mathsf{GenPath}(U_0, U_n)$. This will return a path in the PCN from U_0 to U_n. Path finding algorithms are orthogonal to the problem tackled in this paper and we refer the reader to [22, 23] for more details. Finally, for simplicity, we assume fee to be a publicly known value representing the fee that every user charges. Note that in practice, every user can charge an individual fee.

4.3 Detailed construction and pseudocode

Let us assume U_0 and U_n, connected via U_i for $i \in [1, n-1]$, wish to open a bidirectional virtual channel with capacity α fully funded by U_0. We consider the different phases Open, Update, Close and Respond. We show the used macros in Figure 5, the procedure for updating individual payment channels for the open and close phase in Figure 6, and the whole protocol in Figure 7.

Open. The sender U_0 starts by creating a transaction $\mathsf{tx}^{\mathsf{vc}}$ that contains an output θ_{vc} spendable under the condition $\mathsf{MultiSig}(U_0, U_n)$ holding α coins and n outputs θ_{ε_i} spendable under the condition $\mathsf{OneSig}(U_i) + \mathsf{RelTime}(t_{\mathsf{c}} + \Delta)$ holding ε coins, one for every user U_i for $i \in [0, n-1]$.

Spending from θ_{vc}, U_0 and U_n create commitment transactions for the virtual channel with $\overline{\gamma_{\mathsf{vc}}} := \mathsf{preCreate}(\mathsf{tx}^{\mathsf{vc}}, 0, U_0, U_n)$. This function pre-creates the virtual channel $\overline{\gamma_{\mathsf{vc}}}$, exchanging the initial state transactions with the other user in $\overline{\gamma_{\mathsf{vc}}}.\mathsf{users} := (U_0, U_n)$ based on the output identified by 0 of the funding transaction tx^{f} that remains off-chain for now. It finally returns $\overline{\gamma_{\mathsf{vc}}}$.

Sender U_0 presents its neighbor U_1 with $\mathsf{tx}^{\mathsf{vc}}$ and an update of their channel to a state, where α coins of U_0 are spendable under the condition $\phi = (\mathsf{OneSig}(U_1) \wedge \mathsf{AbsTime}(T)) \vee (\mathsf{MultiSig}(U_0, U_1) \wedge \mathsf{RelTime}(\Delta))$.

Before actually updating the channel, U_1 gives U_0 its signature for $\mathsf{tx}_0^{\mathsf{r}}$. $\mathsf{tx}_0^{\mathsf{r}}$ takes as inputs the output holding α of the aforementioned proposed state update and the output holding ε under U_0's control of $\mathsf{tx}^{\mathsf{vc}}$. After receiving the signature, they perform this update and revoke their previous state. We show this procedure in Figure 6 and illustrate the contract in Figure 2.

In the same fashion, U_1 continues this procedure with its neighbor U_2 and this continues with its neighbor until the receiver has successfully updated its channel with its left neighbor U_{n-1}. In case of a dispute, the honest participants merely go idle and watch the blockchain to observe if $\mathsf{tx}^{\mathsf{vc}}$ is published. After the successful update, the receiver U_n forwards the transaction $\mathsf{tx}^{\mathsf{vc}}$ presented to it by U_{n-1} to U_0, who checks if it is the same transaction U_0 initially created.

Update. At this point the virtual channel $\overline{\gamma_{\mathsf{vc}}}$ is considered to be open and ready to be used. An update can be performed by creating a new state $\mathsf{tx}_i^{\mathsf{state}}$ and calling $\mathsf{preUpdate}(\overline{\gamma_{\mathsf{vc}}}, \mathsf{tx}_i^{\mathsf{state}})$. This function pre-updates the virtual channel $\overline{\gamma_{\mathsf{vc}}}$, updating the latest state transaction to $\mathsf{tx}_i^{\mathsf{state}}$. In case of a dispute, the users wait until the virtual channel is offloaded. At this time, the VC is closed.

In the beginning, the whole balance lies with U_0, but once the balance is redistributed, the channel is usable in both directions. Should they wish to construct a channel where they both hold some balance initially, they can start the construction in the other direction for a second time. When they have rebalanced the money inside the virtual channel and definitely before time T, they proceed to the next phase, the closing phase.

Close. Let us say the final balance of the virtual channel is $\alpha - \alpha'$ belonging to U_0 and α' to U_n. To close the virtual channel, U_n starts the following update process with its left neighbor U_{n-1}. U_n presents a state, where (instead of α)

only α' coins from U_{i-1} are spendable under the condition $\phi = (\mathsf{OneSig}(U_n) \wedge \mathsf{AbsTime}(T)) \vee (\mathsf{MultiSig}(U_{n-1}, U_n) \wedge \mathsf{RelTime}(\Delta))$. For this new state, U_n creates a transaction $\mathsf{tx}^{\mathsf{r}}_{n-1}$ spending this output and the output of $\mathsf{tx}^{\mathsf{vc}}$ belonging to U_{n-1} and gives its signature for this new $\mathsf{tx}^{\mathsf{r}}_{n-1}$ to U_{n-1}. After U_{n-1} checks that the new state and new $\mathsf{tx}^{\mathsf{r}}_{n-1}$ are correct, they update their channel to this new state and revoke the previous one. We show this procedure in Figure 6.

User U_{n-1} continues this process with its left neighbor U_{n-2} and so on, until the sender U_0 is reached. U_0 checks that the balance in the state update is actually the balance that U_0 owes U_n in the virtual channel, α'. If it is not the same, or no such request reaches the sender, U_0 simply publishes $\mathsf{tx}^{\mathsf{vc}}$ on-chain and claims $\mathsf{tx}^{\mathsf{r}}_0$ before the timeout T expires.

If such a request reaches U_0 with the correct balance, the sender has two options. First, the sender can wait until time T, at which the money α' automatically moves from left to right to the receiver, as the absolute timelock of the outputs holding α' in every channel on the path runs out. Alternatively, the sender can ask U_1 to update to a state, identical to the current one, but for the α' coins, which belong to U_1 in this new state. This procedure then continues from left to right until the last channel is reached: again, if the intermediaries do not collaborate, the money moves anyway from left to right after time T.

Respond. To react in an appropriate way, we require the participants to monitor the ledger if $\mathsf{tx}^{\mathsf{vc}}$ is published. In case it is published and its outputs are spendable before T, the intermediary users need to claim the money they staked in their right channel. They can either do this off-chain if their right neighbor is cooperating or in the worst case, forcefully on-chain via $\mathsf{tx}^{\mathsf{r}}_i$. Similarly, after time T has expired without $\mathsf{tx}^{\mathsf{vc}}$ being published on-chain, the intermediaries can claim the money from their left channel. Again, this can happen honestly off-chain or forcefully via $\mathsf{tx}^{\mathsf{p}}_i$.

We show the pseudocode for the full protocol in Figure 7. Note that for better readability we simplify the protocol, e.g., we omit ids required for routing several virtual channels through one node concurrently. For the formal protocol description in the Universal Composability framework, we defer to Appendix E.5.

4.4 Discussion

We now discuss a few points regarding the practical deployment of our virtual channel construction.

Fees for the intermediaries. To incentivize intermediaries to participate in the protocol, a fee is typically paid to them. For simplicity, let us assume that this is the same amount fee for everyone. In practice, a different fee can be charged per hop. During the open phase, when forwarding the contract holding α coins from left to right, the intermediaries simply deduct the amount fee. The sender starts with the amount $\alpha_0 := \alpha + \mathsf{fee} \cdot (n-1)$. An intermediary U_i forwards $\alpha_i := \alpha_{i-1} - \mathsf{fee}$. Similarly, in the closing phase, when the contract is forwarded from right to left, each intermediary adds fee to the total amount.

Outputs of $\mathsf{tx}^{\mathsf{vc}}$. Aside from the output funding the virtual channel, the transaction $\mathsf{tx}^{\mathsf{vc}}$ needs to hold outputs for every user on the path except the receiver. The amount of coins in an output cannot be zero and we argue that this can be a small amount, such that losing this for the sender would be insignificant compared to the on-chain fees that need to be paid anyway. Note however, that due most Bitcoin node implementations only accept transactions with outputs that are larger than a dust, which is 546 satoshis, to prevent cluttering the blockchain. Therefore, in practice one needs to put at least 546 satoshis in these outputs.

Macros (see Appendix D and [4]:)

`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 $\bot$ **`genTxVc`**$(U_0, \mathsf{channelList}, \mathsf{tx}^{\mathsf{in}})$**:** Generates $\mathsf{tx}^{\mathsf{vc}}$ 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 $\mathsf{tx}^{\mathsf{state}}_i$ from the given parameters. **`checkTxVc`**$(U_i, a, b, \mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_i)$**:** Checks if $\mathsf{tx}^{\mathsf{vc}}$ 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. **`genPay`**$(\mathsf{tx}^{\mathsf{state}}_i)$**.** Returns $\mathsf{tx}^{\mathsf{p}}_i$, which takes $\mathsf{tx}^{\mathsf{state}}_i.\mathsf{output}[0]$ as input and creates a single $\mathsf{output} := (\alpha_i, \mathsf{OneSig}(U_{i+1}))$. **`genRef`**$(\mathsf{tx}^{\mathsf{state}}_i, \mathsf{tx}^{\mathsf{vc}}, \theta_{\varepsilon_i})$**.** Return $\mathsf{tx}^{\mathsf{r}}_i$, which takes as input $\mathsf{tx}^{\mathsf{state}}_i.\mathsf{output}[0]$ and $\theta_{\varepsilon_i} \in \mathsf{tx}^{\mathsf{vc}}.\mathsf{output}$. 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 $\mathsf{tx}^{\mathsf{state}}_i$.

Figure 5: Subprocedures used in the protocol

5 Security analysis

We first argue informally why the security and privacy goals outlined in Section 4.1 hold in our construction and then overview the main soundness result. A formal security modelling and analysis is given in Appendix E and Appendix F.

5.1 Informal security analysis

Balance security. When the virtual channel is opened, every intermediary establishes the contract shown in Figure 2 first with their left neighbor and then with their right neighbor.

$\texttt{2pSetup}(\overline{\gamma_i}, \mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_{i+1}, \theta_{\varepsilon_i}, \alpha_i, T)$: (see [4])

U_i

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

U_{i+1} upon $(\mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_{i+1}, \mathsf{tx}^{\mathsf{state}}, \mathsf{tx}_i^{\mathsf{r}})$ from U_i

4. Check that $\texttt{checkTxVc}(U_{i+1}, U_{i+1}.a, U_{i+1}.b, \mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_{i+1}) \neq \perp$, 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 α_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 $\theta_{\varepsilon_x} \in \mathsf{tx}^{\mathsf{vc}}.\mathsf{output}$ it holds that $\mathsf{tx}_i^{\mathsf{r}} := \texttt{genRef}(\mathsf{tx}_i^{\mathsf{state}}, \theta_{\varepsilon_x})$. If one of these previous checks failed, return $\perp$.
7. $\mathsf{tx}_i^{\mathsf{p}} := \texttt{genPay}(\mathsf{tx}_i^{\mathsf{state}})$
8. Send $(\sigma_{U_{i+1}}(\mathsf{tx}_i^{\mathsf{r}}))$ to U_{i+1}

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 $\perp$.
10. $\mathsf{updateChannel}(\overline{\gamma_i}, \mathsf{tx}_i^{\mathsf{state}})$
11. If, after t_{u} time has expired, the message $(\mathsf{update{-}ok})$ is returned, return $\top$. Else return $\perp$.

U_{i+1}: Upon $(\mathsf{update{-}ok})$, return $(\mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_{i+2}, U_{i+2}, \theta_{\varepsilon_{i+1}}, \alpha_i, T)$. Else, upon $(\mathsf{update{-}fail})$, return $\perp$

$\texttt{2pTeardown}(\overline{\gamma_i}, \mathsf{tx}^{\mathsf{vc}}, \alpha_i')$:

U_i

1. $\mathsf{tx}_{i-1}^{\mathsf{state}'} := \texttt{genState}(\alpha_i', T, \overline{\gamma_{i-1}})$ //T is known from $\texttt{2pSetup}$
2. $\mathsf{tx}_{i-1}^{\mathsf{r}'} := \texttt{genRef}(\mathsf{tx}_{i-1}^{\mathsf{state}'}, \theta_{\varepsilon_{i-1}})$ //$\theta_{\varepsilon_{i-1}}$ known as θ_{ε_x} from $\texttt{2pSetup}$
3. Send $(\mathsf{tx}_{i-1}^{\mathsf{state}'}, \mathsf{tx}_{i-1}^{\mathsf{r}'}, \sigma_{U_i}(\mathsf{tx}_{i-1}^{\mathsf{r}'}))$ to U_{i-1}

U_{i-1} upon $(\mathsf{tx}_{i-1}^{\mathsf{state}'}, \mathsf{tx}_{i-1}^{\mathsf{r}'}, \sigma_{U_i}(\mathsf{tx}_{i-1}^{\mathsf{r}'}))$

1. Extract α_i' from $\mathsf{tx}_{i-1}^{\mathsf{state}'}$ and check that $\alpha_i' < \alpha_i$ and $\mathsf{tx}_{i-1}^{\mathsf{state}'} = \texttt{genState}(\alpha_i', T, \overline{\gamma_{i-1}})$ //α_i and T from $\texttt{2pSetup}$
2. If $U_{i-1} = U_0$, ensure that $\alpha_i' \leq x + n \cdot \mathsf{fee}$ where x is the final balance of U_n in the virtual channel.
3. Check that $\mathsf{tx}_{i-1}^{\mathsf{r}'} = \texttt{genRef}(\mathsf{tx}_{i-1}^{\mathsf{state}'}, \theta_{\varepsilon_{i-1}})$ //$\theta_{\varepsilon_{i-1}}$ from $\texttt{2pSetup}$
4. Check that $\sigma_{U_i}(\mathsf{tx}_{i-1}^{\mathsf{r}'})$ is a correct signature of U_i for $\mathsf{tx}_{i-1}^{\mathsf{r}'}$
5. $\mathsf{updateChannel}(\overline{\gamma_{i-1}}, \mathsf{tx}_{i-1}^{\mathsf{state}'})$
6. If, after t_{u} time has expired, the message $(\mathsf{update{-}ok})$ is returned, replace variables $\mathsf{tx}_{i-1}^{\mathsf{state}}$ and $\mathsf{tx}_{i-1}^{\mathsf{r}}$ with $\mathsf{tx}_{i-1}^{\mathsf{state}'}$ and $\mathsf{tx}_{i-1}^{\mathsf{r}'}$, respectively. Return $(\top, \alpha_i')$.
7. Else, return $\perp$.

U_i: Upon $(\mathsf{update{-}ok})$, replace variables $\mathsf{tx}_{i-1}^{\mathsf{state}}$, $\mathsf{tx}_{i-1}^{\mathsf{r}}$ and $\mathsf{tx}_{i-1}^{\mathsf{p}}$ with $\mathsf{tx}_{i-1}^{\mathsf{state}'}$, $\mathsf{tx}_{i-1}^{\mathsf{r}'}$ and $\mathsf{tx}_{i-1}^{\mathsf{p}'} := \texttt{genPay}(\mathsf{tx}_{i-1}^{\mathsf{state}'})$, respectively.

Figure 6: Protocol for 2-party channel update.

Open

U_0 upon receiving $(\texttt{setup}, \mathsf{channelList}, \mathsf{tx}^{\mathsf{in}}, \alpha, T)$

1. If $\texttt{checkChannels}(\mathsf{channelList}, U_0) = \perp$, abort.
2. Let $n := |\mathsf{channelList}|$. If $\texttt{checkT}(n, T) = \perp$, abort.
3. If $\texttt{checkTxIn}(\mathsf{tx}^{\mathsf{in}}, n, U_0) = \perp$, abort.
4. $(\mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}) := \texttt{genTxVc}(U_0, \mathsf{channelList}, \mathsf{tx}^{\mathsf{in}})$
5. $\overline{\gamma_{\mathsf{vc}}}$:= preCreate $(\mathsf{tx}^{\mathsf{vc}}, 0, U_0, U_n)$ together with U_n
6. $\alpha_0 := \alpha + \mathsf{fee} \cdot (n-1)$
7. $(\mathsf{sk}_{\widetilde{U_0}}, \theta_{\varepsilon_0}, R_0, U_1, \mathsf{onion}_1) := \texttt{checkTxVc}(U_0, U_0.a, U_0.b, \mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion})$
8. $\texttt{2pSetup}(\overline{\gamma_0}, \mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_1, U_1, \theta_{\varepsilon_0}, \alpha_0, T)$

U_{i+1} upon receiving $(\mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_{i+2}, U_{i+2}, \theta_{\varepsilon_{i+1}}, \alpha_i, T)$

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

Finalize

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

Update

Either user $U_i \in \overline{\gamma_{\mathsf{vc}}}.\mathsf{users}$ can update the virtual channel $\overline{\gamma_{\mathsf{vc}}}$ by creating a new state $\mathsf{tx}_i^{\mathsf{state}}$ and calling $\mathsf{preUpdate}(\overline{\gamma_{\mathsf{vc}}}, \mathsf{tx}_i^{\mathsf{state}})$.

Close

U_n: Let α' be the final balance of U_n in the virtual channel. Execute $\texttt{2pTeardown}(\overline{\gamma_i}, \mathsf{tx}^{\mathsf{vc}}, \alpha')$

U_{i-1} upon $(\top, \alpha_i')$

1. If $U_{i-1} = U_0$, go idle.
2. Else, let $\alpha_{i-1}' := \alpha_i' + \mathsf{fee}$
3. $\texttt{2pTeardown}(\overline{\gamma_{i-2}}, \mathsf{tx}^{\mathsf{vc}}, \alpha_{i-1}')$

Emergency-Offload

U_0: If U_0 has not successfully performed $\texttt{2pTeardown}$ until $T - t_{\mathsf{c}} - 3\Delta$, $\mathsf{publishTx}(\mathsf{tx}^{\mathsf{vc}}, \sigma_{U_0'}(\mathsf{tx}^{\mathsf{vc}}))$.

Respond (executed by U_i for $i \in [0, n]$ in every round)

1. If $\tau_x < T - t_{\mathsf{c}} - 2\Delta$ and $\mathsf{tx}^{\mathsf{vc}}$ 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 Δ 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 $\tau_x > T$, $\overline{\gamma_{i-1}}$ is closed and $\mathsf{tx}^{\mathsf{vc}}$ and $\mathsf{tx}_{i-1}^{\mathsf{state}}$ is on the blockchain, but not $\mathsf{tx}_{i-1}^{\mathsf{r}}$, $\mathsf{publishTx}(\mathsf{tx}_{i-1}^{\mathsf{p}}, (\sigma_{U_i}(\mathsf{tx}_{i-1}^{\mathsf{p}})))$.

Figure 7: Pseudocode of the protocol.

Note that the refund transactions in both contracts are dependent on the same transaction $\mathsf{tx}^{\mathsf{vc}}$ and that the time-locks on its outputs are the same. This means that if an honest intermediary's left neighbor can refund their collateral, the intermediary itself can refund as well. If the intermediary pays the collateral to its right user, this means that $\mathsf{tx}^{\mathsf{vc}}$ was not published before T and the intermediary itself will be paid as well. In the close phase, the contract is updated to hold a smaller value. Since this is done from right to left, an honest intermediary is safe. In the refund case, nothing changes. In the case where the collateral moves from left to right, the intermediary will receive more money.

Endpoint security. An honest sender can always enforce the virtual channel that holds its correct balance by posting $\mathsf{tx}^{\mathsf{vc}}$ and thereby offloading the virtual channel. By doing so, the refunding of the collateral along the path is triggered, including the one of the sender itself. This means, that in case of a dispute or someone not cooperating, the sender can always use the offloading before T to ensure its balance. An honest receiver will get its rightful balance either when the channel is offloaded or, if it is not, after time T through the collateral, which is moved from left to right along the path.

Reliability. Only the sender is able to offload the virtual channel. This means that if sender and receiver are honest, no one can force them to offload the virtual channel before T.

Endpoint anonymity and path privacy. We require $\mathsf{tx}^{\mathsf{vc}}$ to use an input that is unlinkable to the sender. The addresses used within the virtual channel are fresh and unlinkable addresses of the sender and the receiver as well. Further, the outputs of $\mathsf{tx}^{\mathsf{vc}}$ are constructed as stealth addresses. This means that an intermediary observing $\mathsf{tx}^{\mathsf{vc}}$ will learn nothing about the sender and the receiver. These properties hold only in the optimistic case. In the pessimistic case, it might be possible to link (parts of) the path to $\mathsf{tx}^{\mathsf{vc}}$ and also link the virtual channel to sender/receiver, which is also what happens in any other off-chain protocol, including the Lightning Network.

Value privacy. Similar to how payments between users of a payment channel are known only to those users, also virtual channel updates are only known by the endpoints.

5.2 Security model

We rely on the synchronous, global universal composability (GUC) framework [7] to model the Donner protocol. We make use of some preliminary functionalities commonly used in the literature [2, 3, 4, 9, 10]. The global ledger $\mathcal{L}$ is maintained by the functionality $\mathcal{G}_{Ledger}$, which is parameterized by a signature scheme Σ and a blockchain delay Δ, i.e., an upper bound on number of rounds it takes for a valid transaction to appear on $\mathcal{L}$, after it is posted. The notion of time (or computational rounds) is modelled by $\mathcal{G}_{clock}$ and the communication by $\mathcal{F}_{GDC}$. Finally, a functionality $\mathcal{F}_{Channel}$ handles the creation, update and closure of payment channels as well as the preparation and update of the virtual channels.

We define an ideal functionality $\mathcal{F}_{VC}$ that models the idealized behavior of our VC protocol, stipulating input/output behavior, impact on the ledger as well as possible attacks by adversaries. In the ideal world, $\mathcal{F}_{VC}$ is a trusted third party. Additionally, we formally define the real world hybrid protocol Π and show that Π *emulates* (or realizes) $\mathcal{F}_{VC}$. For this, we describe a simulator $\mathcal{S}$ that translates any attack of any adversary on the protocol Π into an attack on $\mathcal{F}_{VC}$.

To show that the protocol Π realizes $\mathcal{F}_{VC}$, we need to show that no PPT *environment* $\mathcal{E}$ can distinguish between interacting with the real world and interacting with the ideal world with a probability non-negligibly greater than $\frac{1}{2}$. This implies, that any attack that is possible on the protocol is also possible on the ideal functionality. Intuitively, it suffices to output the same messages in the same rounds and add the same transaction to the ledger in the same rounds in both the real and the ideal world. We refer to Appendix E for the preliminaries, Appendix E.4 for the ideal functionality, Appendix E.5 for the formal protocol and Appendix E.6 for the simulator and the formal proof of the following theorem.

Theorem 1. *Let Σ be an EUF-CMA secure signature scheme. Then, for functionalities $\mathcal{G}_{Ledger}$, $\mathcal{G}_{clock}$, $\mathcal{F}_{GDC}$, $\mathcal{F}_{Channel}$ and for any ledger delay $\Delta \in \mathbb{N}$, the protocol Π UC-realizes the ideal functionality $\mathcal{F}_{VC}$.*

6 Evaluation and comparison

6.1 Communication overhead

We implemented a small proof-of-concept that creates the raw Bitcoin transactions necessary for the VC construction [1]. We use the library `python-bitcoin-utils` and Bitcoin Script to build the transactions and tested their compatibility with Bitcoin by deploying them on the testnet. We show the results for the operations *Open*, *Update*, *Close*, *Offload* in Table 2. For transactions that go on-chain, we provide additionally the expected cost in USD at the time of writing. For this evaluation we assume generalized channels [2] as the underlying payment channel scheme, but note that this could be also done with Lightning channels (see Section 6.3).

For opening a virtual channel, each of the n underlying payment channels needs to exchange 4 transactions: $\mathsf{tx}^{\mathsf{vc}}$, $\mathsf{tx}^{\mathsf{r}}_i$ and two transaction for updating the state. Since $\mathsf{tx}^{\mathsf{vc}}$ has an output for every intermediary and the sender, its size increases with the number of channels on the path n and is $192 + 34 \cdot n$ bytes. $\mathsf{tx}^{\mathsf{r}}_i$ has a size of 306 bytes, and a channel update to a state holding this contract is 742 bytes. Notice that $\mathsf{tx}^{\mathsf{p}}_i$ does not need to be exchanged, since the left user of a channel can generate it independently. This totals to $1240 + 34 \cdot n$ bytes of off-chain commincation per channel for the open phase. Additionally to that, we require to exchange the initial state of the virtual channel, which is 2 transactions or 695 bytes. All in all, this totals to $4 \cdot n + 2$ transactions or $34 \cdot n^2 + 1240 \cdot n + 695$ bytes for the whole path.

For honestly closing a virtual channel, the payment needs

Table 2: Communication overhead of Donner for the whole path (not per party) for the different operations, assuming a virtual channel across n channels. In the pessimistic offload, $k \in [0,n]$ is the number of channels where there is a dispute.

	# txs	size (bytes)	on-chain cost (USD)
Open	$4 \cdot n+2$	$34 \cdot n^2 + 1240 \cdot n + 695$	0
Update	2	695	0
Close	$3 \cdot n$	$1048 \cdot n$	0
Offload (Optimistic)	1	$192+34$	$0.62+0.04 \cdot n$
Offload (Pessimistic)	$3k+1$	$1048 \cdot k + 192 + 34 \cdot n$	$1.38 \cdot k + 0.62 + 0.04 \cdot n$

to be updated from right to left. However, $\mathsf{tx}^{\mathsf{vc}}$ does not need to be exchanged anymore, so we only need to exchange 3 transactions or 1048 bytes for each of the $n-1$ underlying channels. To update a virtual channel, the two endpoints of that virtual channel need to exchange 2 transactions with 695 bytes, the same as a payment channel update.

Finally, for offloading, only the transaction $\mathsf{tx}^{\mathsf{vc}}$ needs to be posted on-chain and nothing per channel. This means $192+34 \cdot n$ bytes and costs $0.62+0.04 \cdot n$ USD. Note that if individual users on the path do not collaborate, regardless if the virtual channel is offloaded or successfully closed, these channels may need to be closed as well. We argue that this is also the case during the normal payment channel execution, e.g., when routing multi-hop payments. However, for every channel that does need to be closed, the three transactions exchanged in the close phase need to be posted additionally. If there are k channels with such a dispute, this results in a total of $3k+1$ transactions or $1048 \cdot k + 192 + 34 \cdot n$ bytes, which costs $1.38 \cdot k + 0.62 + 0.04 \cdot n$ USD for the whole path. We mark this as the *pessimistic* case in Table 2.

6.2 Efficiency comparison

In this section we compare our construction to LVPC [15] (cf. Table 3). As already mentioned, LVPC is rooted and needs to be constructed recursively. We construct a VC with LVPC by applying the different operations (open, update, close, offload) over one intermediary recursively. We compare the number of off-chain and on-chain transactions to Donner and conclude, that Donner is more efficient in every case.

Note that there are many different ways to combine virtual channels with normal payment channels or even other virtual channels, e.g., see Figure 8 or Figure 9. One can easily see, that every combination leads to the same minimum number of virtual channels required for a path of n base channels: One for each of the $n-1$ intermediaries. Before presenting the differences for each operation, we point out that in recursive solutions the storage overhead per intermediary is linear in the number of layers on top of a user, which in turn is in the worst case (Figure 8) linear in the path length and in the best case (Figure 9) logarithmic in the path length.

In the open phase across the whole path, Donner requires $4 \cdot n+2$ off-chain transactions for the whole path. In LVPC, 7 off-chain transactions per virtual channel are needed, so $7 \cdot (n-1)$.

Table 3: Comparison between recursive constructions and Donner for a VC over from U_0 to U_n via $n-1$ hops. [1]In the pessimistic offload in Donner, $k \in [0,n]$ is the number of channels where there is a dispute.

		# txs	off-chain
Open	LVPC [15]	$7 \cdot (n-1)$	✓
	Donner	$4 \cdot n+2$	✓
Update	LVPC [15]	2	✓
	Donner	2	✓
Close	LVPC [15]	$4 \cdot (n-1)$	✓
	Donner	$3 \cdot n$	✓
Offload (Optimistic)	LVPC [15]	$5 \cdot n-5$	✗
	Donner	1	✗
Offload (Pessimistic)	LVPC [15]	$5 \cdot n-5$	✗
	Donner[1]	$3 \cdot k+1$	✗

Similar, for closing, we need to exchange 4 transactions per virtual channel in LVPC, so $4 \cdot (n-1)$. Donner requires the close operation per underlying channel, so $3 \cdot n$ transactions. The update phase is the same for both.

The interesting case again is the offload case. As we already pointed out, a fully rooted, recursive VC construction requires to close *all underlying channels*. This means in LVPC, we require 2 transactions per underlying channel, of which we have n payment channels and $n-2$ virtual channels (all but the topmost one). Additionally, we need to publish $n-1$ funding transactions of the virtual channels including the topmost one. This results in $2 \cdot (2n-2)+n-1 = 5 \cdot n-5$ transactions that have to be posted on-chain along with the fact that all involved channels have to be closed in the case of a dispute . In Donner, only 1 transaction has to be posted on-chain. For the pessimistic offload, there need to be $3 \cdot k+1$ transactions posted in Donner, where k is the number of channels where there is a dispute.

Application scenario: Bootstrapping of new nodes. According to a recent Lightning Network (LN) snapshot, the average number of channels per node is 7.8. This means that, on average, the bootstrapping of a newly created node in the LN costs (rounding up) 8 transactions posted on-chain, i.e., one funding transaction per channel. Additional 8 transactions need to be posted on-chain when such channels are closed. VCs can reduce the on-chain bootstrapping cost of a new node in the LN. In particular, given that the LN is a connected component and assuming that each channel has enough capacity in both directions, one can open only one payment channel holding all the funds of the user and leverage it then to open a virtual channel to the other 7 nodes, thereby minimizing the overhead on-chain.

The results of our back-of-the-envelop calculations are shown in Table 4. Here we assume that there exists 4 intermediate channels to create each VC since the average shortest path length in our snapshot of the LN is 3.4, and take the results from Table 3 to count the number of transactions. These results show that VCs effectively move the on-chain overhead to the off-chain setting for bootstrapping, making the PCNs an attractive and cheap layer-2 solution: A user can use a single but expensive on-chain operation to put all its funds over a single channel to a well-connected node and then create many and cheap virtual channels to any frequent counterparties over the PCN topology. By doing that, the user additionally gains in liveness and privacy guarantees as VCs in Donner are not susceptible to the corresponding attacks by the intermediaries.

6.3 Compatibility with Lightning channels

To simplify the formalization of this work, we built our virtual channel construction on top of generalized payment channels (GC) [2], which have one symmetric channel state. However, it is also possible to construct Donner on top of LN channels, which have two asymmetric channel states. The (one-hop) BCVC [3] constructions rely on GCs as well, while the recursive LVPC [15] relies on simple channels that have only one state, but each update reduces the limited lifetime of the channel.

As LN channels are the only ones deployed in practice, it is interesting to investigate the effect of building VCs on top of LN channels. We point out that building Donner on top of LN channel is not difficult, as the collateralization in the underlying base channels is similar to a MHP. In fact, the only two differences for implementing Donner on top of LN channels instead of GCs is that (i) for each of the two asymmetric states per channel we now need to create a tx_i^r transaction, so two instead of one, and (ii) a punishment mechanism has to be introduced per output instead of per state (e.g., similar to how HTLCs are handled in LN).

The LVPC construction is not as straightforward to implement on top of LN channels. Similarly to Donner, we need to introduce a punishment mechanism (ii). However, the more difficult part is handling the two asymmetric states (i). Since the VC needs to be able to be posted regardless of which of the two states are posted, there needs to be a unique funding transaction (called Merge in [15]) for each possible combination of states in the underlying channels. This implies that in a LVPC like construction which is built on top of LN channels, the storage overhead per party is *exponential* in the layers of VCs that are constructed over this party. In fact, using channels with duplicated states this exponential growth is present in every rooted, recursive VC construction. This follows from the evaluation in [2].

To give an example, a VC that is rooted in two channels with two states each, needs to be copied four times: Once for each of the ways of closing these two channels. On each of these four combinations we have to build a copy of the VC, and for each VC we need again two commitment transactions, so eight. This growth continues for each layer. In addition, for each of these exponentially many copies of the VC, commitment transactions need to be exchanged for an update, so there is an exponential communication overhead too. Note that the storage overhead for Donner on top of LN channels is *constant* as is the communication overhead for updates.

Table 4: Bootstrapping cost comparison

	no VCs		LVPC [15]		**Donner**	
on-/off-chain	on	off	on	off	on	off
connecting to the network	8	16	1	147	1	126
disconnecting honestly	8	0	1	84	1	84
disconnecting forcefully	8	0	120	0	8	0

7 Conclusion

Payment channel networks (PCN) such as the Lightning network have emerged as successful layer-2 solutions for cryptocurrencies. However, the current path-based protocols require active participation from all intermediaries, and every employed intermediary introduces a single-point-of-failure, which can make layer-2 payments unreliable, slow and expensive. While virtual channels are already envisioned towards mitigating these problems, the existing constructions are either not compatible with Bitcoin offering only limited scripting capabilities, or restricted to single-hop virtual channels, or vulnerable to forced closure and griefing attacks.

In this work, we have presented the first Bitcoin-compatible, secure, privacy-preserving and efficient multi-hop virtual channel construction. Donner is a provably secure, efficient single-round (or non-recursive), decoupled, multi-hop virtual channel construction. Our performance analysis demonstrated Donner is efficient during the disputes and in terms of storage: it only requires a single on-chain transaction during a dispute (as compared to linear in the path length for the state-of-the-art); moreover, the storage overhead also becomes independent of the path length.

Overall, Donner offers an easy-to-adopt, scalable virtual channel construction for PCNs such as the Lightning Network. We find that Donner makes PCNs attractive layer-2 solutions: the users can put all of their cryptocurrency over a single channel to a well-connected node and then create cheap virtual channels to any frequent counterparty over the strongly connected PCN topology. Unlike the underlying PCNs, the virtual channels are not susceptible to liveness and privacy attacks by the intermediaries.

Acknowledgements. 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) 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] Donner virtual channel evaluation of the communication overhead, 2021. https://github.com/donner-vc/overhead.

[2] Lukas Aumayr, Oguzhan Ersoy, Andreas Erwig, Sebastian Faust, Kristina Hostáková, Matteo Maffei, Pedro Moreno-Sanchez, and Siavash Riahi. Generalized bitcoin-compatible channels. Cryptology ePrint Archive, Report 2020/476, 2020. https://eprint.iacr.org/2020/476.

[3] Lukas Aumayr, Oguzhan Ersoy, Andreas Erwig, Sebastian Faust, Kristina Hostáková, Matteo Maffei, Pedro Moreno-Sanchez, and Siavash Riahi. Bitcoin-Compatible Virtual Channels. In *IEEE Symposium on Security and Privacy*, 2021.

[4] Lukas Aumayr, Pedro Moreno-Sanchez, Aniket Kate, and Matteo Maffei. Blitz: Secure Multi-Hop Payments Without Two-Phase Commits. In *USENIX Security Symposium*, 2021.

[5] Christian Badertscher, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. Bitcoin as a transaction ledger: A composable treatment. In *Advances in Cryptology CRYPTO*, volume 10401, pages 324–356, 2017.

[6] Jan Camenisch and Anna Lysyanskaya. A formal treatment of onion routing. In *Advances in Cryptology CRYPTO*, pages 169–187, 2005.

[7] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. Universally composable security with global setup. In *Theory of Cryptography TCC*, volume 4392, pages 61–85, 2007.

[8] G. Danezis and I. Goldberg. Sphinx: A compact and provably secure mix format. In *2009 30th IEEE Symposium on Security and Privacy*, pages 269–282, 2009.

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

[10] Stefan Dziembowski, Lisa Eckey, Sebastian Faust, and Daniel Malinowski. Perun: Virtual payment hubs over cryptocurrencies. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 106–123. IEEE, 2019.

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

[12] Christoph Egger, Pedro Moreno-Sanchez, and Matteo Maffei. Atomic multi-channel updates with constant collateral in bitcoin-compatible payment-channel networks. In *Computer and Communications Security CCS*, page 801–815, 2019.

[13] Shafi Goldwasser, Silvio Micali, and Ronald L Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on computing*, 17(2):281–308, 1988.

[14] Ethan Heilman, Leen Alshenibr, Foteini Baldimtsi, Alessandra Scafuro, and Sharon Goldberg. Tumblebit: An untrusted bitcoin-compatible anonymous payment hub. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017.

[15] Maxim Jourenko, Mario Larangeira, and Keisuke Tanaka. Lightweight Virtual Payment Channels. In *19th International Conference on Cryptology and Network Security (CANS)*, 2020.

[16] Maxim Jourenko, Mario Larangeira, and Keisuke Tanaka. Payment Trees: Low Collateral Payments for Payment Channel Networks. In *Financial Cryptography and Data Security*, 2021.

[17] Jonathan Katz, Ueli Maurer, Björn Tackmann, and Vassilis Zikas. Universally composable synchronous computation. In Amit Sahai, editor, *Theory of Cryptography TCC*, volume 7785, pages 477–498, 2013.

[18] Giulio Malavolta, Pedro Moreno-Sanchez, Aniket Kate, Matteo Maffei, and Srivatsan Ravi. Concurrency and privacy with payment-channel networks. In *Computer and Communications Security CCS*, page 455–471, 2017.

[19] Giulio Malavolta, Pedro Moreno-Sanchez, Clara Schneidewind, Aniket Kate, and Matteo Maffei. Anonymous multi-hop locks for blockchain scalability and interoperability. In *Network and Distributed System Security Symposium, NDSS*, 2019.

[20] Andrew Miller, Iddo Bentov, Surya Bakshi, Ranjit Kumaresan, and Patrick McCorry. Sprites and state channels: Payment networks that go faster than lightning. In Ian Goldberg and Tyler Moore, editors, *Financial Cryptography and Data Security - 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18-22, 2019, Revised Selected Papers*, volume 11598 of *Lecture Notes in Computer Science*, pages 508–526. Springer, 2019.

[21] Joseph Poon and Thaddeus Dryja. The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments, January 2016. Draft version 0.5.9.2, available at https://lightning.network/lightning-network-paper.pdf.

[22] Stefanie Roos, Pedro Moreno-Sanchez, Aniket Kate, and Ian Goldberg. Settling payments fast and private: Efficient decentralized routing for path-based transactions. In *Network and Distributed System Security Symposium NDSS*, 2018.

[23] 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 *Networked Systems Design and Implementation NSDI*, pages 777–796, 2020.

[24] E. Tairi, P. Moreno-Sanchez, and M. Maffei. A2l: Anonymous atomic locks for scalability in payment channel hubs. In *2021 2021 IEEE Symposium on Security and Privacy (SP)*, pages 1919–1936, Los Alamitos, CA, USA, may 2021. IEEE Computer Society.

[25] Nicolas Van Saberhagen. Cryptonote v 2.0 (2013). *URL: https://cryptonote. org/whitepaper. pdf. White Paper. Accessed*, pages 04–13, 2018.

A Forced closure attack

We now illustrate the *forced closure attack* in more detail. Assume a virtual channel between users U_0 and U_4 recursively constructed using the LVPC protocol [15] in the fashion of Figure 8. A malicious user U_0 can unilaterally force the closure of all underlying channels, that is, (U_0,U_1), (U_1,U_2), (U_2,U_3) and (U_3,U_4) as well as the virtual channels (U_0,U_2), (U_0,U_3) and the offloading of (U_0,U_4).

First, U_0 closes the channel (U_0,U_1), which it can do on its own. In LVPC, the output in the state of (U_0,U_1) which is used to fund the virtual channel (U_0,U_2) goes to U_0, *unless* it is first consumed by the virtual channel. This means that an honest U_1 will lose money in the channel (U_0,U_1) to U_0 by means of the *Punish* transaction, unless it closes the channel (U_1,U_2) and claims its money by posting the transaction funding the VC (i.e., offloading) (U_0,U_2), dubbed *Merge* transaction.

This process is repeated: Now that the funding of (U_0,U_2) is on-chain, U_0 can close the channel (U_0,U_2), which again puts the output that is used to fund (U_0,U_3) on-chain. Same as before, now U_2 has to close (U_0,U_3) with the Merge transaction, otherwise it will lose the money to U_0 via the Punish transaction. So U_2 is forced to close the channel (U_2,U_3) and then publish the funding transaction of (U_0,U_3).

Now, one more time, U_0 repeats this process by closing (U_0,U_3), which forces U_3 to close (U_3,U_4) and offload (U_0,U_4). One can see that, similarly, U_4 can force the closure of all these channels. In fact U_4 only needs to close (U_3,U_4) to achieve the same effect. In an analogous fashion, this attack can be applied also when the VC is constructed by different underlying channels, like in Figure 9.

Let us clarify that by closing the underlying payment channels we mean that at least one transaction per payment channel has to be put on-chain. Due to the fact that LVPC first splits the channel before funding the VC, closing the initial channel simultaneously spawns a new channel that has a capacity reduced by the amount put in the collateral funding the VC.

While we show this attack only for LVPC, we conjecture that it is a more general problem with all Bitcoin compatible virtual channels that are rooted in the underlying payment channels. The intuition for this is that at least one user has to have a way of offloading the virtual channel to avoid locking funds forever. However, to offload a fully rooted VC, all channels in which the VC is rooted need to be closed. So if the VC is rooted in all underlying channels, all of them need to be closed.

B Virtual griefing attack

Here, we present the *virtual griefing attack* in more detail. We already detailed the offloading/closing sequence that can be initiated by an endpoint of a virtual channel constructed using LVPC [15] in Appendix A. Note that the closure is done in a sequential way, i.e., U_0 starts by closing its channel with U_1, who continues to close its channel with U_2 and so on. Each step takes a certain time to be executed. Additionally, the next user that has to react needs enough time to take the appropriate action in order not to lose coins.

In LVPC, this fact is represented by setting the timelock of each Merge transaction to be the maximum of the timelock of the two underlying channel *plus* a time Δ, i.e., the blockchain delay. This implies, that the timelock on the outermost VC is linear in the number of layers beneath it. The number of layers is linear in the path length in case of Figure 8 and logarithmic in case of Figure 9.

For the virtual griefing attack, however, a sender sets up a VC with itself following Figure 8. It follows that the collateral of the intermediaries is locked for a time linear in the path length. As for the forced closure attack, we conjecture that this is a more general problem that is affecting all virtual channels that are constructed recursively and tear down this structure in a sequential way. In particular, it affects also the Ethereum based construction [11]. The intuition here is that the different layers need to be closed sequentially, and each closure takes a certain time.

C Example graphs for recursive VC

In this section, we show in Figure 8 and Figure 9 two example graphs that illustrate the different ways that one could recursively create a multi-hop VC using VC with a single intermediary as a building block.

D Extended macros

In this section, we give extended pseudo-code for the used subprocedures used in our protocol, both in the pseudocode definition given in Section 4 and in the formal model in Appendix E.4, Appendix E.5 and Appendix E.6.

<u>Subprocedures</u>

<u>`checkTxIn`$(\mathsf{tx}^{\mathsf{in}}, n, U_0, \alpha)$:</u>

1. Check that $\mathsf{tx}^{\mathsf{in}}$ is a transaction on the ledger $\mathcal{L}$.
2. If $\mathsf{tx}^{\mathsf{in}}.\mathsf{output}[0].\mathsf{cash} \geq n \cdot \varepsilon + \alpha$ 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

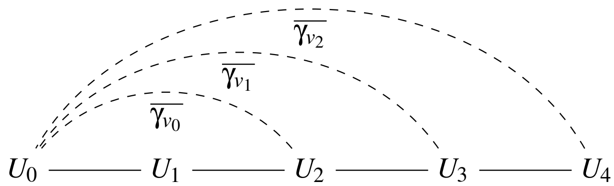

Figure 8: Recursive virtual channel: Example A

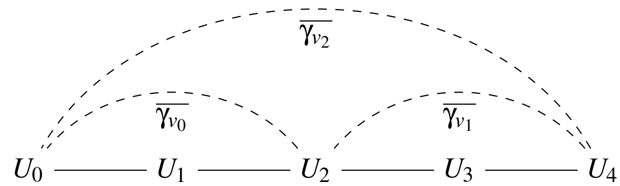

Figure 9: Recursive virtual channel: Example B

fund $\mathsf{tx}^{\mathsf{vc}}$), the sender will pay any superfluous coins back to a fresh address of itself.

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

Check that 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.

$\texttt{checkT}(n, T)$:

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

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

1. Let $\mathsf{outputList} := \emptyset$ and $\mathsf{rList} := \emptyset$
2. For every channel γ_i in 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 nodeList be a list, where $\mathcal{P}$ is sorted from sender to receiver. Let $n := |\mathcal{P}|$.
4. Shuffle outputList and rList.
5. Let $\mathsf{tx}^{\mathsf{vc}} := (\mathsf{tx}^{\mathsf{in}}.\mathsf{output}[0], \mathsf{outputList})$
6. Create a list $[\mathsf{msg}_i]_{i \in [0,n]}$, where $\mathsf{msg}_i := \mathcal{H}(\mathsf{tx}^{\mathsf{vc}})$
7. $\mathsf{onion} \leftarrow \mathsf{CreateRoutingInfo}(\mathsf{nodeList}, [\mathsf{msg}_i]_{i \in [0,n]})$
8. Return $(\mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion})$

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

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 $\mathsf{tx}_i^{\mathsf{state}}$.

$\texttt{checkTxVc}(U_i, a, b, \mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_i)$:

1. $x := \mathsf{GetRoutingInfo}(\mathsf{onion}_i, U_i)$. If $x = \bot$, return $\bot$. If U_i is the receiver and $x = \mathcal{H}(\mathsf{tx}^{\mathsf{vc}})$, return $(\top, \top, \top, \top, \top)$. Else, if $x \neq (U_{i+1}, \mathcal{H}(\mathsf{tx}^{\mathsf{vc}}), \mathsf{onion}_{i+1})$, return $\bot$.
2. For all outputs $(\mathsf{cash}, \phi) \in \mathsf{tx}^{\mathsf{vc}}.\mathsf{output}$ except output with index 0 it must hold that:
   - $\mathsf{cash} = \varepsilon$
   - $\phi = \mathsf{OneSig}(\mathsf{pk}_x) \wedge \mathsf{RelTime}(t_{\mathsf{c}} + \Delta)$ for some identity pk_x
3. For exactly one output $\theta_{\varepsilon_i} := (\varepsilon, \mathsf{OneSig}(\widetilde{U}_i) \wedge \mathsf{RelTime}(t_{\mathsf{c}} + \Delta)) \in \mathsf{tx}^{\mathsf{vc}}.\mathsf{output}$ 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 $\bot$
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

$\texttt{createMaps}(U_0, \mathsf{nodeList}, \mathsf{tx}^{\mathsf{in}}, \alpha)$:

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. $\mathsf{rList} = \mathsf{rMap}.\texttt{values().shuffle()}$
3. Let $\theta_{\mathsf{vc}} := (\alpha, \mathsf{MultiSig}(U_0, U_n))$
4. $\mathsf{tx}^{\mathsf{vc}} := (\mathsf{tx}^{\mathsf{in}}.\mathsf{output}[0], [\theta_{\mathsf{vc}}, \mathsf{outputMap}.\texttt{values()}$ $\texttt{.shuffle()}])$
5. Create a map stealthMap that stores for every user U_i that is a key in outputMap the corresponding output of $\mathsf{tx}^{\mathsf{vc}}$ corresponding to $\mathsf{outputMap}(U_i)$
6. Create two empty lists $\emptyset$ named msgList, userList
7. For every $U_i \in \mathsf{nodeList}$ from U_n to U_0 (in descending order):
   - Append $[\mathcal{H}(\mathsf{tx}^{\mathsf{vc}})]$ to msgList
   - Prepend $[U_i]$ to userList.
   - $\mathsf{onion}_i := \mathsf{CreateRoutingInfo}(\mathsf{userList}, \mathsf{msg})$
   - $\mathsf{onions}(U_i) := \mathsf{onion}_i$
8. Return $(\mathsf{tx}^{\mathsf{vc}}, \mathsf{onions}, \mathsf{rMap}, \mathsf{rList}, \mathsf{stealthMap})$

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

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 $\vec{\theta}_i$.

genNewState$(\overline{\gamma_i}, \alpha'_i, T)$**:**

1. Let $\vec{\theta}_i := \overline{\gamma_i}.\mathsf{st}$.
2. Let $\alpha_i := \vec{\theta}_i[0].\mathsf{cash}$
3. Set $\theta_0 := (\alpha'_i, \vec{\theta}_i[0].\phi)$
4. Set $\theta_1 := (\vec{\theta}[1].\mathsf{cash} + \alpha_i - \alpha'_i, \vec{\theta}_i[1].\phi)$
5. Set $\theta_2 := \vec{\theta}_i[2]$
6. Return vector $\vec{\theta}'_i := (\theta_0, \theta_1, \theta_2)$

genRefTx$(\theta, \theta_{\varepsilon_i}, U_i)$**:**

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

genPayTx(θ, U_{i+1})**:**

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

[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.

E UC modeling

For our formal security analysis, we utilize the global UC framework (GUC) [7]. In contrast to the standard Universal Composability (UC) framework, the GUC allows for a global setup, which in turn we use for modelling the blockchain as a global ledger. In this section, we go over some preliminaries and notation before presenting the ideal functionality. Note that our formal model follows closely [2, 3, 4, 9, 10, 11].

E.1 Preliminaries, communication model and threat model

A protocol Π is executed between a set of parties $\mathcal{P}$ and runs in the presence of an adversary $\mathcal{A}$, who receives as input a security parameter $\lambda \in \mathbb{N}$ along with an auxiliary input $z \in \{0,1\}^*$. $\mathcal{A}$ can corrupt any party $P_i \in \mathcal{P}$ at the beginning of the protocol execution, i.e., a static corruption model. Corrupting a party P_i means that $\mathcal{A}$ takes control over P_i and learns its internal state. The parties and the adversary $\mathcal{A}$ take their input from the *environment* $\mathcal{E}$, a special entity which represents everything external to the protocol execution. Additionally, $\mathcal{E}$ observes the messages that are output by the parties of the protocol.

In our model, we assume a synchronous communication network with computation happening in rounds, which allows for a more natural arguing about time. This abstraction of computational rounds is formalized in the ideal functionality $\mathcal{G}_{clock}$ [17], which represents a global clock, that proceeds to the next round if all honest parties indicate that they are ready to do so. This means that every entity always knows the given round.

Further, we assume that parties communicate via authenticated channels with guaranteed delivery after precisely one round. This is modeled by the ideal functionality $\mathcal{F}_{GDC}$: If a party P sends a message to party Q in round t, then Q receives that message in the beginning of round $t+1$ and knows that the message was sent by P. Note that the adversary $\mathcal{A}$ is capable of reading the content of every message that is sent and can reorder messages that are sent in the same round, but cannot drop, modify or delay messages. For a formal definition of $\mathcal{F}_{GDC}$ we refer to [9].

In contrast to this communication between parties of $\mathcal{P}$ which takes one round, all other communication, that involves for instance the adversary $\mathcal{A}$ or the environment $\mathcal{E}$, takes zero rounds. Further, every computation that a party executes locally takes zero rounds as well.

E.2 Ledger and channels

We use the global ideal functionality $\mathcal{G}_{Ledger}$ to model a UTXO based blockchain, parameterized by Δ, an upper bound on the number of rounds it takes for a valid transaction to be accepted (the blockchain delay) and a signature scheme Σ. $\mathcal{G}_{Ledger}$ communicates with a fixed set of parties $\mathcal{P}$. The environment $\mathcal{E}$ first initializes $\mathcal{G}_{Ledger}$ by setting up a key pair $(\mathsf{sk}_P, \mathsf{pk}_P)$ for every party $P \in \mathcal{P}$ and registers it to the ledger by sending $(\mathtt{sid}, \mathtt{REGISTER}, \mathsf{pk}_p)$ to $\mathcal{G}_{Ledger}$. Then, $\mathcal{E}$ sets the initial state of $\mathcal{L}$, a publicly accessible set of all published transactions. Any party $P \in \mathcal{P}$ can always post a transaction on $\mathcal{L}$ via $(\mathtt{sid}, \mathtt{POST}, \overline{\mathtt{tx}})$. If a transaction is valid, it will be appear on $\mathcal{L}$ after at most Δ round, the exact number is chosen by the adversary. Recall that a transaction is valid, if all its inputs exist and are unspent, there is a correct witness for each input and a unique id.

We point out that this model is simplified: We fix the set of users instead of allowing them to join or leave dynamically. Further, transactions are in reality bundled in blocks, which are submitted by parties and $\mathcal{A}$. For a more accurate formalization, we refer to works such as [5]. To increase readability, we opted for these simplifications.

Channels are handles by the functionality $\mathcal{F}_{Channel}$ [3], which is an extension of [2] and builds on top of $\mathcal{G}_{Ledger}$. $\mathcal{F}_{Channel}$ allows to to create, update and close a payment channel between two users, as well as handling channels (pre-create and pre-update) that are funded off-chain, i.e., a virtual channel. We define t_{u} as an upper bound on rounds it takes to update and t_{c} as an upper bound on rounds it takes to close a channel (regardless of whether or not there is cooperation). 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. Finally, t_{o} is an upper bound it takes to pre-create a channel.

We assume that for our constructions, all parties in the protocol have been registered with $\mathcal{L}$, and all relevant channels between them are already open. We present an API along with an explanation of $\mathcal{F}_{Channel}$ and $\mathcal{G}_{Ledger}$ below. For increased readability, we hide the calls to $\mathcal{G}_{clock}$ and $\mathcal{F}_{GDC}$ in our notation. Instead of explicitly calling these functionalities, we write $(\mathsf{msg}) \overset{t}{\hookrightarrow} X$ to denote sending message (msg) to X in round t and $(\mathsf{msg}) \overset{t}{\hookleftarrow} X$ to denote receiving message (msg) from X at time t. The sending/receiving entity as well as X are either a party $P \in \mathcal{P}$, the environment $\mathcal{E}$, the simulator $\mathcal{S}$ or another ideal functionality.

Interface of $\mathcal{F}_{Channel}(T,k)$ [2]

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, $(\mathtt{sid},\mathtt{UPDATED},\mathsf{id},\vec{\theta}) \xhookrightarrow{\leq\tau+t_{\mathsf{u}}} \overline{\gamma}.\mathsf{users}$ 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 $\mathcal{G}_{Ledger}$).

- $(\mathtt{sid},\mathtt{PRE\text{-}CREATE},\overline{\gamma},\mathsf{tx}^{\mathsf{f}},i,t_{\mathsf{ofl}}) \overset{\tau}{\hookleftarrow} P$:
  Does the same as CREATE, with the following difference. Instead of the an input for the funding transaction, the funding transaction tx^{f} along with an index i, defining which output of tx^{f} is used to fund the channel. The parameter t_{ofl} defines the maximum number of rounds it takes to put tx^{f} on-chain. If successfully invoked by both users of the channel, $\mathcal{F}_{Channel}$ returns $(\mathtt{sid},\mathtt{PRE\text{-}CREATED},\overline{\gamma}.\mathsf{id})$ after at most t_{o} rounds.
- $(\mathtt{sid},\mathtt{PRE\text{-}UPDATE},\mathsf{id},\vec{\theta}) \overset{\tau}{\hookleftarrow} P$:
  Does the same as UPDATE for a pre-created channel, however, in case of a dispute, $\mathcal{F}_{Channel}$ waits for tx^{f} to appear on the ledger within t_{ofl} rounds. If it does, the channel is closed.
- Additionally, $\mathcal{F}_{Channel}$ checks every round if the tx^{f} of a pre-created channel is put on the ledger. If it is, the pre-created channel is handled just as a normal channel from that time forward.

Interface of $\mathcal{G}_{Ledger}(\Delta,\Sigma)$ [2]

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.

E.3 The UC-security definition

Closely following [4], we define 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, $\phi_{\mathcal{F}_{VC}}$ denotes the ideal protocol of ideal functionality $\mathcal{F}_{VC}$, where the dummy users simply forward their input to $\mathcal{F}_{VC}$. It has access to the same functionalities $\mathcal{F}_{\mathsf{prelim}}$. The output of $\phi_{\mathcal{F}_{VC}}$ on input λ and z when interacting with $\mathcal{E}$ and a simulator $\mathcal{S}$ is denoted as $\mathtt{EXEC}^{\mathcal{F}_{\mathsf{prelim}}}_{\phi_{\mathcal{F}_{VC}},\mathcal{S},\mathcal{E}}(\lambda,z)$.

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

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

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

where $\approx^c$ denotes computational indistinguishability.

E.4 Ideal functionality

In this section we explain our ideal functionality (IF) $\mathcal{F}_{VC}$ in prose. Note that the IF is capable of outputting an ERROR message, e.g., when a transaction does not appear on the ledger after instructing the simulator. We remark that the only protocols that realize this IF that are of interest to us are the ones that *never* output `ERROR`. The cases where `ERROR` is output are not meaningful to us and any guarantees are lost. We use the extended macros defined in Appendix D. The IF is split into different parts: (i) Open-VC, (ii) Finalize-Open, (iii) Update-VC, (iv) Close-VC, (v) Emergency-Offload and (vi) Respond. We remark the similarity of (i), (ii) and (vi) to the IF in [4].

Open-VC. This part starts with the setup phase, in which the sender U_0 invokes the IF to open a VC. In it, $\mathcal{F}_{VC}$ takes care of creating all necessary object, such as $\mathsf{tx}^{\mathsf{vc}}$, the onions, the stealth addresses, etc. and calls `PRE-CREATE` of $\mathcal{F}_{Channel}$ to set up the VC with U_n. Afterwards, $\mathcal{F}_{VC}$ continues to do the following. If the next neighbor on the path is honest, it takes care of creating the objects and updating the channel with that neighbor, which is captured in the subprocedure Open. If the next neighbor is instead dishonest, $\mathcal{F}_{VC}$ instructs the simulator $\mathcal{S}$ to simulate the view of the attacker. Additionally, $\mathcal{F}_{VC}$ exposes the functionality to the simulator, which was asked to continue the open phase with a legitimate request, the simulator can perform Check to see if an id is already in use and Register to register the channel that was updated with the adversary. If the subsequent neighbor is again honest, the IF will continue handling the opening, else the simulator will do it. This continues until the receiver U_n is reached and all channels along with their created objects are stored in the IF for each channel that contains at least one honest user. If U_n is honest, but not U_0, the last step of the Open-VC phase is actually to instruct $\mathcal{S}$ to send a confirmation to U_0. At this point, the Finalize-Open starts.

Finalize-Open. If U_0 is honest, the IF will either know that U_n completed the opening within a certain round if U_n is also honest. Or, if U_n is dishonest, $\mathcal{F}_{VC}$ expects a confirmation from U_n via $\mathcal{S}$. If an incorrect or no confirmation was received in the correct round, the IF instructs the simulator to publish $\mathsf{tx}^{\mathsf{vc}}$, offloading the VC.

Update-VC. While the VC is open, the two endpoints can use `PRE-UPDATE` of $\mathcal{F}_{Channel}$ to update the VC. The IF simply forwards these messages.

Close-VC. This phase is similar to the Open-VC phase, but it is initiated by U_n, conducted from right to left and the requires fewer objects to be created. Similar to the Open-VC phase, the IF distinguishes if the left neighbor is honest or not. If it is, then $\mathcal{F}_{VC}$ takes care of updating the channel, reducing the collateral to U_n's final balance in the VC plus its according fee. If it is dishonest, it instructs $\mathcal{S}$ to simulate the view of the adversary. If the simulator is invoked by the adversary to continue the closing with a legitimate request, the IF continues with the closure, until the sender is reached.

Emergency-Offload. If the sender of a payment is honest, the IF will expect the Close-VC request to be concluded for that payment in a certain round. If it is not, $\mathcal{F}_{VC}$ instructs $\mathcal{S}$ to offload the VC.

Respond. This phase is executed in every round and in it, $\mathcal{F}_{VC}$ observes if a transaction $\mathsf{tx}^{\mathsf{vc}}$ is posted on the ledger $\mathcal{L}$, which is used in channels that have an honest user and are registered as pending in the IF. If it is published early enough to refund the collateral, $\mathcal{F}_{VC}$ closes the channels and instructs the simulator to publish the refund transaction. Else, if the lifetime of the VC T has already expired and the neighbor closes the channel, $\mathcal{F}_{VC}$ instructs the simulator to publish the payment transaction.

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

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 $(\texttt{pid}, U_i)$ to prevent duplicate ids to avoid loops in payments.

Φ : A map, storing for a given key $(\texttt{pid}, U_0)$ of an id $\texttt{pid}$ and a user U_0, a tuple $(\tau_{\mathsf{f}}, \mathsf{tx}^{\mathsf{vc}}, U_n)$, where τ_{f} is the round in which the payment confirmation is expected from the receiver, the transaction $\mathsf{tx}^{\mathsf{vc}}$ and the receiver U_n. The map is initially empty and read write access is written as $\Phi(\texttt{pid}, U_0)$. $\Phi.\texttt{keyList}()$ returns a set of all keys.

Γ : A set of tuples $(\texttt{pid}, \overline{\gamma_i}, \vec{\theta}_i, \mathsf{tx}^{\mathsf{vc}}, T, \theta_{\varepsilon_i}, R_i)$ for channels with opened payment construction, containing a payment id $\texttt{pid}$, the channel $\overline{\gamma_i}$, the state the payment builds upon $\vec{\theta}_i$, the time T, the output used in the refund by $\overline{\gamma_i}.\mathsf{left}$ and value R_i to reconstruct the secrect key of the stealth address used. It is initially empty.

Ψ : A set of tuples $(\texttt{pid}, \mathsf{tx}^{\mathsf{vc}})$ containing payments, that have been opened and where the receiver is honest.

t_{u}, $t_{\mathsf{c}}, t_{\mathsf{o}}$: Time it takes at most to update, close or (pre-)open a channel.

<u>Init (executed at initialization in round t_{init}.)</u>

Send $(\texttt{sid}, \texttt{init}) \xhookrightarrow{t_{\mathsf{init}}} \mathcal{S}$ and upon $(\texttt{sid}, \texttt{init-ok}, t_{\mathsf{u}}, t_{\mathsf{c}}, t_{\mathsf{o}}) \xhookleftarrow{t_{\mathsf{init}}} \mathcal{S}$ set t_{u}, t_{c}, t_{o} accordingly.

<u>Open-VC</u>

Let τ be the current round.
Setup:

1. Upon $(\texttt{sid}, \texttt{pid}, \texttt{SETUP}, \mathsf{channelList}, \mathsf{tx}^{\mathsf{in}}, \alpha, T, \overline{\gamma_0}) \xhookleftarrow{\tau} U_0$, if $(\texttt{pid}, U_0) \in \mathsf{idSet}$ go idle. $\mathsf{idSet} := \mathsf{idSet} \cup \{(\texttt{pid}, U_0)\}$
2. Let $x := \texttt{checkChannels}(\mathsf{channelList}, U_0)$. 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 $U_1 := \overline{\gamma_0}.\mathsf{right}$ (corresponding to the channel skeleton γ_0 in $\mathsf{channelList}$), go idle. Let $\mathsf{nodeList}$ be a list of all the users on the path sorted from U_0 to U_n.
3. Let $n := |\mathsf{channelList}|$. If $\texttt{checkT}(n, T) = \bot$, go idle.

4. If $\texttt{checkTxIn}(\mathsf{tx}^{\mathsf{in}}, n, U_0, \alpha) = \bot$, go idle.
5. $(\mathsf{tx}^{\mathsf{vc}}, \mathsf{onions}, \mathsf{rMap}, \mathsf{rList}, \mathsf{stealthMap}) := \texttt{createMaps}(U_0, \mathsf{nodeList}, \mathsf{tx}^{\mathsf{in}}, \alpha)$.
6. Send $(\texttt{sid}, \texttt{pid}, \texttt{pre-create-vc}, \overline{\gamma_{\mathsf{vc}}}, \mathsf{tx}^{\mathsf{vc}}, T) \xhookrightarrow{\tau} \mathcal{S}$ and wait 1 round.
7. Send $(\texttt{ssid}_C, \texttt{PRE-CREATE}, \overline{\gamma_{\mathsf{vc}}}, \mathsf{tx}^{\mathsf{vc}}, 0, T - \tau) \xhookrightarrow{\tau+1} \mathcal{F}_{Channel}$
8. If not $(\texttt{ssid}_C, \texttt{PRE-CREATED}, \overline{\gamma_{\mathsf{vc}}}.\mathsf{id}) \xhookleftarrow{\tau+1+t_o} \mathcal{F}_{Channel}$, go idle.
9. Set $\alpha_0 := \alpha + \mathsf{fee} \cdot (n-1)$.
10. Set $\Phi(\texttt{pid}, U_0) := (\tau_{\mathsf{f}} := \tau + n \cdot (2 + t_{\mathsf{u}}) + 2 + t_{\mathsf{o}}, \mathsf{tx}^{\mathsf{vc}}, U_n)$.
11. If U_1 honest, execute **Open**$(\texttt{pid}, \mathsf{nodeList}, \mathsf{tx}^{\mathsf{vc}}, \mathsf{onions}, \mathsf{rMap}, \mathsf{rList}, \mathsf{stealthMap}, \alpha_0, T, \overline{\gamma_0})$.
12. Else, let $\mathsf{onion}_1 := \mathsf{onions}(U_1)$ and $\theta_{\varepsilon_0} := \mathsf{stealthMap}(U_0)$. Send $(\texttt{sid}, \texttt{pid}, \texttt{open}, \mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_1, \alpha_0, T, \bot, \overline{\gamma_0}, \bot, \theta_{\varepsilon_0}) \xhookrightarrow{\tau+1+t_o} \mathcal{S}$.

Continue: //Continue after a dishonest user

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

Check: //Sim. can check that id was not yet used

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

VC-Open: //Mark VC as opened

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

Register: //Sim. can register a channel

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

Open$(\texttt{pid}, \mathsf{nodeList}, \mathsf{tx}^{\mathsf{vc}}, \mathsf{onions}, \mathsf{rMap}, \mathsf{rList}, \mathsf{stealthMap}, \alpha_{i-1}, T, \overline{\gamma_{i-1}})$:
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 $\bot$, 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 $\vec{\theta}_{i-1} = \bot$, go idle. Else, wait 1 round.
7. $(\texttt{sid}, \texttt{pid}, \texttt{OPEN}, \mathsf{tx}^{\mathsf{vc}}, \theta_{\varepsilon_i}, R_i, U_{i-1}, U_{i+1}, \alpha_{i-1}, T) \xhookrightarrow{\tau+1} U_i$
8. If not $(\texttt{sid}, \texttt{pid}, \texttt{OPEN-ACCEPT}, \overline{\gamma_i}) \xhookleftarrow{\tau+1} U_i$, 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. $\Gamma := \Gamma \cup (\texttt{pid}, \overline{\gamma_i}, \vec{\theta}_i, \mathsf{tx}^{\mathsf{vc}}, T, \theta_{\varepsilon_i}, R_i)$
12. If $U_i = U_n$:
    - $\Psi := \Psi \cup \{(\texttt{pid}, \mathsf{tx}^{\mathsf{vc}})\}$
    - $(\texttt{sid}, \texttt{pid}, \texttt{VC-OPENED}, \mathsf{tx}^{\mathsf{vc}}, T, \alpha_{i-1}) \xhookrightarrow{\tau+2+t_{\mathsf{u}}} U_i$
    - If U_0 is dishonest, send $(\texttt{sid}, \texttt{pid}, \texttt{finalize}, \mathsf{tx}^{\mathsf{vc}}) \xhookrightarrow{\tau+2+t_{\mathsf{u}}} \mathcal{S}$
13. Else:
    - $(\texttt{sid}, \texttt{pid}, \texttt{OPENED}) \xhookrightarrow{\tau+2+t_{\mathsf{u}}} U_i$
    - If U_{i+1} honest, execute **Open**$(\texttt{pid}, \mathsf{nodeList}, \mathsf{tx}^{\mathsf{vc}}, \mathsf{onions}, \mathsf{rMap}, \mathsf{rList}, \mathsf{stealthMap}, \alpha_{i-1} - \mathsf{fee}, \overline{\gamma_i})$
    - Else, send $(\texttt{sid}, \texttt{pid}, \texttt{open}, \mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_{i+1}, \alpha_{i-1} - \mathsf{fee}, T, \overline{\gamma_{i-1}}, \overline{\gamma_i}, \theta_{\varepsilon_{i-1}}, \theta_{\varepsilon_i}) \xhookrightarrow{\tau} \mathcal{S}$, where $\mathsf{onion}_{i+1} := \mathsf{onions}(U_{i+1})$ and $\theta_{\varepsilon_{i-1}} := \mathsf{stealthMap} U_{i-1}$

Finalize-Open (executed at every round)

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

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

Update-VC

While VC is open, the sending and the receiving endpoint can update the VC using $\texttt{PRE-UPDATE}$ of $\mathcal{F}_{Channel}$ just as they would a ledger channel.

Close-VC

Let τ be the current round.
Start:

1. Upon $(\texttt{sid}, \texttt{pid}, \texttt{SHUTDOWN}, \alpha'_{n-1}) \xhookleftarrow{\tau} U_n$, for parameter $\texttt{pid}$, fetch entry $(\texttt{pid}, \overline{\gamma_{n-1}}, \vec{\theta}_i, \mathsf{tx}^{\mathsf{vc}}, T, \theta_{\varepsilon_i}, R_i)$ from Γ, s.t. $\overline{\gamma_{n-1}}.\mathsf{right} = U_n$. If there is no such entry, go idle.
2. Let $U_{n-1} := \overline{\gamma_{n-1}}.\mathsf{left}$.
3. If U_n is not the endpoint in VC $\texttt{pid}$, go idle.
4. If U_{n-1} honest, execute **Close**$(\texttt{pid}, \overline{\gamma_{n-1}}, \alpha'_{n-1})$
5. Else, send $(\texttt{sid}, \texttt{pid}, \texttt{close}, \alpha'_{n-1}, \overline{\gamma_{n-1}}) \xhookrightarrow{\tau} \mathcal{S}$

Continue-Close: //Continue after a dishonest user

1. Upon $(\texttt{sid}, \texttt{pid}, \texttt{continue-close}, \overline{\gamma_{i-1}}, \alpha'_{i-1}) \xhookleftarrow{\tau} \mathcal{S}$
2. **Close**$(\texttt{pid}, \overline{\gamma_{i-1}}, \alpha'_{i-1})$.

Close$(\texttt{pid}, \overline{\gamma_i}, \alpha'_i)$: Let τ be the current round and $U_i := \overline{\gamma_i}.\mathsf{left}$

1. For the parameters $\texttt{pid}$ and $\overline{\gamma_i}$, fetch entry $(\texttt{pid}, \overline{\gamma_i}, \vec{\theta}_i, \mathsf{tx}^{\mathsf{vc}}, T, \theta_{\varepsilon_i}, R_i)$ from Γ. If there is no entry where the parameters $\texttt{pid}$ and $\overline{\gamma_i}$ match, go idle.

2. If $\overline{\gamma_i}$.st $\neq \vec{\theta}_i$, go idle.
3. Let $\alpha_i := \vec{\theta}_i[0]$.cash. If not $0 \leq \alpha_i' \leq \alpha_i$, go idle.
4. $\vec{\theta}_i' :=$ genNewState$(\overline{\gamma_i}, \alpha_i', T)$. If $\vec{\theta}_i' = \perp$, go idle. Else, wait 1 round.
5. $(\mathtt{sid}, \mathtt{pid}, \mathtt{CLOSE}, \alpha_i') \xrightarrow{\tau+1} U_i$
6. If not $(\mathtt{sid}, \mathtt{pid}, \mathtt{CLOSE\text{-}ACCEPT}) \xleftarrow{\tau+1} U_i$, go idle.
7. $(\mathtt{ssid}_C, \mathtt{UPDATE}, \overline{\gamma_i}.\mathsf{id}, \vec{\theta}_i') \xhookrightarrow{\tau+1} \mathcal{F}_{Channel}$
8. If not $(\mathtt{ssid}_C, \mathtt{UPDATED}, \overline{\gamma_i}.\mathsf{id}) \xleftarrow{\tau+1+t_u} \mathcal{F}_{Channel}$, go idle.
9. $\Gamma := \Gamma \setminus (\mathtt{pid}, \overline{\gamma_i}, \vec{\theta}_i, \mathsf{tx}^{\mathsf{vc}}, T, \theta_{\varepsilon_i}, R_i)$
10. $\Gamma := \Gamma \cup (\mathtt{pid}, \overline{\gamma_i}, \vec{\theta}_i', \mathsf{tx}^{\mathsf{vc}}, T, \theta_{\varepsilon_i}, R_i)$
11. If $U_i = U_0$:
    - $(\mathtt{sid}, \mathtt{pid}, \mathtt{VC\text{-}CLOSED}) \xrightarrow{\tau+1+t_u} U_i$
    - Remove key and value for key $(\mathtt{pid}, U_0)$ from Φ.
12. Else:
    - Retrieve $\overline{\gamma_{i-1}}$ from Γ matching pid and s.t. $\overline{\gamma_{i-1}}$.right $= U_i$
    - $(\mathtt{sid}, \mathtt{pid}, \mathtt{CLOSED}) \xrightarrow{\tau+1+t_u} U_i$
    - If U_{i-1} honest, execute **Close**$(\mathtt{pid}, \alpha_i' + \mathsf{fee}, \overline{\gamma_{i-1}})$
    - Else, send $(\mathtt{sid}, \mathtt{pid}, \mathtt{close}, \alpha_i' + \mathsf{fee}, \overline{\gamma_{i-1}}) \xhookrightarrow{\tau} \mathcal{S}$.

Replace: //Update the state currently saved by the IF

1. Upon $(\mathtt{sid}, \mathtt{pid}, \mathtt{replace}, \overline{\gamma_i}, \vec{\theta}_i') \xhookleftarrow{\tau} \mathcal{S}$, let $U_i := \overline{\gamma_i}$.left
2. For parameters pid and $\overline{\gamma_i}$, fetch entry $(\mathtt{pid}, \overline{\gamma_i}, \vec{\theta}_i, \mathsf{tx}^{\mathsf{vc}}, T, \theta_{\varepsilon_i}, R) \in \Gamma$
3. $\Gamma := \Gamma \setminus \{(\mathtt{pid}, \overline{\gamma_i}, \vec{\theta}_i, \mathsf{tx}^{\mathsf{vc}}, T, \theta_{\varepsilon_i}, R)\}$
4. $\Gamma := \Gamma \cup \{(\mathtt{pid}, \overline{\gamma_i}, \vec{\theta}_i', \mathsf{tx}^{\mathsf{vc}}, T, \theta_{\varepsilon_i}, R)\}$
5. If $U_i = U_0$, remove key and value for key $(\mathtt{pid}, U_0)$ from Φ.

Emergency-Offload (executed at every round)

Let τ be the current round. For every $(\mathtt{pid}, U_0) \in \Phi.\mathtt{keyList}()$ do the following:

1. For pid and a channel $\overline{\gamma_0}$ where $\overline{\gamma_0}$.left $= U_0$, fetch entry $(\mathtt{pid}, \overline{\gamma_0}, \vec{\theta}_0, \mathsf{tx}^{\mathsf{vc}}, T, \theta_{\varepsilon_0}, R_0) \in \Gamma$
2. If $\tau < T - t_c - 3\Delta$, continue with next loop iteration.
3. Else, let $(\tau_f, \mathsf{tx}^{\mathsf{vc}}, U_n) = \Phi(\mathtt{pid}, U_0)$. Send $(\mathtt{sid}, \mathtt{pid}, \mathtt{offload}, \mathsf{tx}^{\mathsf{vc}}, U_0) \xhookrightarrow{\tau} \mathcal{S}$. $\mathsf{tx}^{\mathsf{vc}}$ must be on $\mathcal{L}$ in round $\tau' \leq \tau + \Delta$. Otherwise, output $(\mathtt{sid}, \mathtt{ERROR}) \xhookrightarrow{t_1} U_0$.
4. Remove key and value for key $(\mathtt{pid}, U_0)$ from Φ.

Respond (executed at the end of every round)

Let t be the current round. For every element $(\mathtt{pid}, \overline{\gamma_i}, \vec{\theta}_i, \mathsf{tx}^{\mathsf{vc}}, T, \theta_{\varepsilon_i}, R_i) \in \Gamma$, check if $\overline{\gamma_i}$.st $= \vec{\theta}_i$ and $\mathsf{tx}^{\mathsf{vc}}$ is on $\mathcal{L}$. If yes, do the following:

Revoke: If γ_i.left honest and $t < T - t_c - 2\Delta$ do the following.
- Set $\Gamma := \Gamma \setminus \{(\mathtt{pid}, \overline{\gamma_i}, \vec{\theta}_i, \mathsf{tx}^{\mathsf{vc}}, T, \theta_{\varepsilon_i}, R_i)\}$.
- $(\mathtt{ssid}_C, \mathtt{CLOSE}, \overline{\gamma_i}.\mathsf{id}) \xhookrightarrow{t} \mathcal{F}_{Channel}$
- At time $t + t_c$, a transaction tx with tx.output $= \overline{\gamma_i}$.st has to be on $\mathcal{L}$. If not, do the following. If $(\mathtt{ssid}_C, \mathtt{PUNISHED}, \overline{\gamma_i}.\mathsf{id}) \xleftarrow{\tau<T} \mathcal{F}_{Channel}$, go idle. Else, send $(\mathtt{sid}, \mathtt{ERROR}) \xhookrightarrow{T} \gamma_i$.users.
- Wait for Δ rounds and send $(\mathtt{sid}, \mathtt{pid}, \mathtt{post\text{-}refund}, \overline{\gamma_i}, \theta_{\varepsilon_i}, R_i) \xhookrightarrow{t'<T-\Delta} \mathcal{S}$
- At time $t'' < T$, check whether a transaction tx′ appears on $\mathcal{L}$ with tx′.input $= [\theta_{\varepsilon_i}, tx.\mathsf{output}[0]]$ and tx′.output $= [(\mathsf{tx.output}[0].\mathsf{cash} + \theta_{\varepsilon_i}.\mathsf{cash}, \mathsf{OneSig}(U_i))]$. If it does, send $(\mathtt{sid}, \mathtt{pid}, \mathtt{REVOKED}) \xhookrightarrow{t''} \overline{\gamma_i}$.left. If not, send $(\mathtt{sid}, \mathtt{ERROR}) \xhookrightarrow{T} \gamma_i$.users.

Force-Pay: Else, if a transaction tx with tx.output $= \overline{\gamma_i}$.st is on-chain and 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 $\Gamma := \Gamma \setminus \{(\mathtt{pid}, \overline{\gamma_i}, \vec{\theta}_i, \mathsf{tx}^{\mathsf{vc}}, T, \theta_{\varepsilon_i}, R_i)\}$.
- Send $(\mathtt{sid}, \mathtt{pid}, \mathtt{post\text{-}pay}, \overline{\gamma_i}) \xhookrightarrow{t} \mathcal{S}$
- In round $t + \Delta$ transaction tx′ with tx′.input $= [\mathsf{tx.output}[0]]$ and tx′.output $= (\mathsf{tx.output}[0].\mathsf{cash}, \mathsf{OneSig}(U_{i+1}))$ must have appeared on $\mathcal{L}$. If yes, $(\mathtt{sid}, \mathtt{pid}, \mathtt{FORCE\text{-}PAY}) \xrightarrow{t+\Delta} \overline{\gamma_i}$.right. Otherwise, $(\mathtt{sid}, \mathtt{ERROR}) \xrightarrow{t+\Delta} \gamma_i$.users.

E.5 Protocol

In this section we give the formal protocol Π along with a short description of it. We note that for simplicity, we assume that users do not update or close the channels involved with virtual channels[2] Also, a user knows if it is an endpoint (sender/receiver) or an intermediary of a VC as well as its direct neighbors on the path. Following, the simulator simulating an honest user knows that also.

The protocol is similar to the simplified pseudo-code presented in Section 4. The main differences lie in having VC ids that allow handling multiple different VCs, the notion of time and the environment $\mathcal{E}$. Briefly, the protocol starts with $\mathcal{E}$ invoking U_0 to set up the initial objects and pre-create the VC with U_n. Then U_0 asks its neighbor U_1 to exchange the necessary transactions and update their channel to hold the collateral. This is continued until the receiver U_n is reached. In the finalize phase, U_n sends a confirmation to U_0, indicating that the VC is open. In the Update VC phase, the channel can be used. The Close VC phase updates the collateral from right to left to hold U_n's final balance in the VC. The Respond phase is there, for users to react to $\mathsf{tx}^{\mathsf{vc}}$ being posted on the ledger, and triggers either a refund or claim of the collateral. We point to the similarities of Open VC, Finalize and Respond with the formal protocol description in [4].

Protocol Π

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

[2] In reality, they can take part in multiple VCs, update, close or use their channels in some other fashion while a VC is open. For this, they recreate the output used for the collateral and $\mathsf{tx}_i^{\mathsf{r}}$, but we omit this for readability.

pidSet : A set storing every payment id `pid` that a user has participated in to prevent duplicates.

paySet : A map storing tuples $(\texttt{pid}, \tau_f, U_n)$ 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 $\mathsf{tx}^{\mathsf{vc}}$ and T in a tuple $(\mathsf{tx}^{\mathsf{vc}}, T)$.

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`.

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`.

Open VC

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

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

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 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 $\texttt{checkT}(n, T) = \bot$, abort.
4. If $\texttt{checkTxIn}(\mathsf{tx}^{\mathsf{in}}, n, U_0, \alpha) = \bot$, abort
5. $(\mathsf{tx}^{\mathsf{vc}}, \mathsf{onions}, \mathsf{rMap}, \mathsf{rList}, \mathsf{stealthMap}) := \texttt{createMaps}(U_0, \mathsf{nodeList}, \mathsf{tx}^{\mathsf{in}}, \alpha)$.
6. $(\mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_0) := \texttt{genTxVc}(U_0, \mathsf{channelList}, \mathsf{tx}^{\mathsf{in}})$
7. $\mathsf{paySet} := \mathsf{paySet} \cup \{(\texttt{pid}, \tau_f := \tau + n \cdot (2 + t_u) + 2 + t_o, U_n)\}$
8. $(\mathsf{sk}_{\widetilde{U_0}}, \theta_{\varepsilon_0}, R_0, U_1, \mathsf{onion}_1) := \texttt{checkTxVc}(U_0, U_0.a, U_0.b, \mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_0)$
9. Set $\mathsf{local}(\texttt{pid}) := (\mathsf{tx}^{\mathsf{vc}}, T)$.
10. Send $(\texttt{sid}, \texttt{pid}, \texttt{pre-create-vc}, \overline{\gamma_{\mathsf{vc}}}, \mathsf{tx}^{\mathsf{vc}}, T) \overset{\tau_0}{\hookrightarrow} U_n$ and wait 1 round.
11. Send $(\texttt{ssid}_C, \texttt{PRE-CREATE}, \overline{\gamma_{\mathsf{vc}}}, \mathsf{tx}^{\mathsf{vc}}, 0, T - \tau_0) \xrightarrow{\tau_0+1} \mathcal{F}_{Channel}$
12. If not $(\texttt{ssid}_C, \texttt{PRE-CREATED}, \overline{\gamma_{\mathsf{vc}}}.\mathsf{id}) \xleftarrow{\tau_0+1+t_o} \mathcal{F}_{Channel}$, go idle.
13. Set $\alpha_0 := \alpha + \mathsf{fee} \cdot (n-1)$ and compute:
    - $\vec{\theta}_0 := \texttt{genStateOutputs}(\overline{\gamma_0}, \alpha_0, T)$
    - $\mathsf{tx}^{\mathsf{r}}_0 := \texttt{genRefTx}(\vec{\theta}_0, \theta_{\varepsilon_0}, U_0)$
14. Set $\mathsf{right}(\texttt{pid}) := (\overline{\gamma_0}, \vec{\theta}_0, \mathsf{tx}^{\mathsf{r}}_0, \mathsf{sk}_{\widetilde{U_0}})$.
15. Send $(\texttt{sid}, \texttt{pid}, \texttt{open-req}, \mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_1, \vec{\theta}_0, \mathsf{tx}^{\mathsf{r}}_0) \xrightarrow{\tau_0+1+t_o} U_1$.

U_n upon $(\texttt{sid}, \texttt{pid}, \texttt{pre-create-vc}, \overline{\gamma_{\mathsf{vc}}}, \mathsf{tx}^{\mathsf{vc}}, T) \overset{\tau}{\hookleftarrow} U_0$

1. $(\texttt{ssid}_C, \texttt{PRE-CREATE}, \overline{\gamma_{\mathsf{vc}}}, \mathsf{tx}^{\mathsf{vc}}, 0, T - \tau) \overset{\tau}{\hookrightarrow} \mathcal{F}_{Channel}$
2. If not $(\texttt{ssid}_C, \texttt{PRE-CREATED}, \overline{\gamma_{\mathsf{vc}}}.\mathsf{id}) \xleftarrow{\tau+t_o} \mathcal{F}_{Channel}$, mark VC as unusable.

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

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

1. Perform the following checks:
    - Verify that $\texttt{pid} \notin \mathsf{pidSet}$. Add `pid` to pidSet
    - Let $x := \texttt{checkTxVc}(U_{i+1}, U_{i+1}.a, U_{i+1}.b, \mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_{i+1})$. 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 $\vec{\theta}_i = \texttt{genStateOutputs}(\overline{\gamma_i}, \alpha_i, T)$.
    - Check that $\mathsf{tx}^{\mathsf{r}}_i := \texttt{genRefTx}(\vec{\theta}_i, \theta_{\varepsilon_x}, U_i)$, where θ_{ε_x} is an output of $\mathsf{tx}^{\mathsf{vc}}$, s.t. $\theta_{\varepsilon_x} \neq \theta_{\varepsilon_{i+1}}$.
2. If one or more of the previous checks fail, abort. Otherwise, send $(\texttt{sid}, \texttt{pid}, \texttt{OPEN}, \mathsf{tx}^{\mathsf{vc}}, \theta_{\varepsilon_{i+1}}, R_i, U_i, U_{i+2}, \alpha_i, T) \overset{\tau_x}{\hookrightarrow} \mathcal{E}$.
3. If $(\texttt{sid}, \texttt{pid}, \texttt{OPEN-ACCEPT}, \overline{\gamma_{i+1}}) \overset{\tau_x}{\hookleftarrow} \mathcal{E}$, 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_i upon $(\texttt{sid}, \texttt{pid}, \texttt{open-ok}, \sigma_{U_{i+1}}(\mathsf{tx}^{\mathsf{r}}_i)) \xleftarrow{\tau_i+2} U_{i+1}$

(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 $\mathsf{rightSig}(\texttt{pid}) := \sigma_{U_{i+1}}(\mathsf{tx}^{\mathsf{r}}_i)$ and $(\texttt{ssid}_C, \texttt{UPDATE}, \overline{\gamma_i}.\mathsf{id}, \vec{\theta}_i) \xrightarrow{\tau_i+2} \mathcal{F}_{Channel}$.

U_{i+1} upon $(\texttt{ssid}_C, \texttt{UPDATED}, \overline{\gamma_i}.\mathsf{id}, \vec{\theta}_i) \xleftarrow{\tau_x+1+t_u} \mathcal{F}_{Channel}$

6. Define $\tau_{(i+1)} := \tau_x + 1 + t_u$.
7. If U_{i+1} is not the receiver, using the values of step 1:
    - Send $(\texttt{sid}, \texttt{pid}, \texttt{OPENED}) \overset{\tau_{i+1}}{\hookrightarrow} \mathcal{E}$.
    - $(\mathsf{sk}_{\widetilde{U_{i+1}}}, \theta_{\varepsilon_{i+1}}, R_{i+1}, U_{i+2}, \mathsf{onion}_{i+2}) := \texttt{checkTxVc}(U_{i+1}, U_{i+1}.a, U_{i+1}.b, \mathsf{tx}^{\mathsf{er}}_i, \mathsf{rList}, \mathsf{onion}_{i+1})$
    - $\vec{\theta}_{i+1} := \texttt{genStateOutputs}(\overline{\gamma_{i+1}}, \alpha_i - \mathsf{fee}, T)$
    - $\mathsf{tx}^{\mathsf{r}}_{i+1} := \texttt{genRefTx}(\vec{\theta}_{i+1}, \theta_{\varepsilon_{i+1}}, U_{i+1})$
    - Set $\mathsf{right}(\texttt{pid}) := (\overline{\gamma_{i+1}}, \vec{\theta}_{i+1}, \mathsf{tx}^{\mathsf{r}}_{i+1}, \mathsf{sk}_{\widetilde{U_{i+1}}})$
    - Send $(\texttt{sid}, \texttt{pid}, \texttt{open-req}, \mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_{i+2}, \vec{\theta}_{i+1}, \mathsf{tx}^{\mathsf{r}}_{i+1}) \xrightarrow{\tau_{i+1}} U_{i+2}$.
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 $(\mathtt{sid}, \mathtt{pid}, \mathtt{finalize}, \mathsf{tx}^{\mathsf{vc}}, \sigma_{U_n}(\mathsf{tx}^{\mathsf{vc}})) \xhookrightarrow{\tau_{i+1}} U_0$. Send the message $(\mathtt{sid}, \mathtt{pid}, \mathtt{VC\text{-}OPENED}, \mathsf{tx}^{\mathsf{vc}}, T, \alpha_i) \xhookrightarrow{\tau_{i+1}} \mathcal{E}$.

Finalize

U_0 in every round τ

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

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

Update VC

While VC is open, the sending and the receiving endpoint can update the VC using PRE-UPDATE of $\mathcal{F}_{Channel}$ just as they would a ledger channel.

Close VC

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

U_n upon $(\mathtt{sid}, \mathtt{pid}, \mathtt{SHUTDOWN}, \alpha'_{n-1}) \xleftarrow{\tau_n} \mathcal{E}$

1. If $\mathtt{pid} \notin \mathsf{pidSet}$, abort.
2. If U_n is not the receiving endpoint in the VC, abort.
3. Retrieve $(\overline{\gamma_{n-1}}, \vec{\theta}_{n-1}, \mathsf{tx}^{\mathsf{r}}_{n-1}) := \mathsf{left}(\mathtt{pid})$
4. Extract $\theta_{\varepsilon_{n-1}} \in \mathsf{tx}^{\mathsf{r}}_{n-1}.\mathsf{input}$
5. Extract T from $\vec{\theta}_{n-1}[0].\phi$
6. Let $\alpha_i := \vec{\theta}_{n-1}[0].\mathsf{cash}$. If not $0 \le \alpha'_i \le \alpha_i$, abort. Compute:
   - $\vec{\theta}'_{n-1} := \mathtt{genNewState}(\overline{\gamma_{n-1}}, \alpha'_{n-1}, T)$
   - $\mathsf{tx}^{\mathsf{r}'}_{n-1} := \mathtt{genRefTx}(\vec{\theta}'_{n-1}, \theta_{\varepsilon_{n-1}}, U_{n-1})$
7. Create the signature $\sigma_{U_n}(\mathsf{tx}^{\mathsf{r}'}_{n-1})$
8. Send $(\mathtt{sid}, \mathtt{pid}, \mathtt{close\text{-}req}, \vec{\theta}'_{n-1}, \mathsf{tx}^{\mathsf{r}'}_{n-1}, \sigma_{U_n}(\mathsf{tx}^{\mathsf{r}'}_{n-1})) \xhookrightarrow{\tau_0} U_{n-1}$.

Close: In every round, every node $U_i \in \mathcal{P}$ does the following. We denote τ_x as the current round.

U_i upon $(\mathtt{sid}, \mathtt{pid}, \mathtt{close\text{-}req}, \vec{\theta}'_i, \mathsf{tx}^{\mathsf{r}'}_i, \sigma_{U_{i+1}}(\mathsf{tx}^{\mathsf{r}'}_i)) \xleftarrow{\tau_x} U_{i+1}$

1. If $\mathtt{pid} \notin \mathsf{pidSet}$, abort.
2. Retrieve $(\overline{\gamma_i}, \vec{\theta}_i, \mathsf{tx}^{\mathsf{r}}_i, \mathsf{sk}_{\widetilde{U}_i}) := \mathsf{right}(\mathtt{pid})$
3. If $\overline{\gamma_i}.\mathsf{right} \ne U_{i+1}$, abort. If $\vec{\theta}_i[0] \notin \mathsf{tx}^{\mathsf{r}}_i.\mathsf{input}$, abort.
4. Extract $\theta_{\varepsilon_i} \in \mathsf{tx}^{\mathsf{r}}_i.\mathsf{input}$
5. Extract T from $\vec{\theta}_i[0].\phi$ and $\alpha_i := \vec{\theta}_i[0].\mathsf{cash}$
6. Extract T' from $\vec{\theta}'_i[0].\phi$ and $\alpha'_i := \vec{\theta}'_i[0].\mathsf{cash}$
7. If $T' \ne T$, abort. If not $0 \le \alpha'_i \le \alpha_i$, abort.
8. If $\vec{\theta}'_i \ne \mathtt{genNewState}(\overline{\gamma_i}, \alpha'_i, T)$, abort.
9. If $\mathsf{tx}^{\mathsf{r}'}_i \ne \mathtt{genRefTx}(\vec{\theta}'_i, \theta_{\varepsilon_i}, U_i)$, abort.
10. If $\sigma_{U_{i+1}}(\mathsf{tx}^{\mathsf{r}'}_i)$ is not a valid signature for $\mathsf{tx}^{\mathsf{r}'}_i$, abort.
11. Send $(\mathtt{sid}, \mathtt{pid}, \mathtt{CLOSE}, \alpha'_i) \xhookrightarrow{\tau_x} \mathcal{E}$
12. If not $(\mathtt{sid}, \mathtt{pid}, \mathtt{CLOSE\text{-}ACCEPT}) \xleftarrow{\tau_x} \mathcal{E}$, abort.
13. $(\mathtt{ssid}_C, \mathtt{UPDATE}, \overline{\gamma_i}.\mathsf{id}, \vec{\theta}'_i) \xhookrightarrow{\tau_x} \mathcal{F}_{Channel}$.

U_{i+1} upon $(\mathtt{ssid}_C, \mathtt{UPDATED}, \overline{\gamma_i}.\mathsf{id}, \vec{\theta}'_i) \xleftarrow{\tau_i+1+t_{\mathsf{u}}} \mathcal{F}_{Channel}$

14. Set $\mathsf{left}(\mathtt{pid}) := (\overline{\gamma_i}, \vec{\theta}'_i, \mathsf{tx}^{\mathsf{r}'}_i)$

U_i upon $(\mathtt{ssid}_C, \mathtt{UPDATED}, \overline{\gamma_i}.\mathsf{id}, \vec{\theta}'_i) \xleftarrow{\tau_x+t_{\mathsf{u}}} \mathcal{F}_{Channel}$

15. Let $\tau_i := \tau_x + t_{\mathsf{u}}$
16. Set $\mathsf{rightSig}(\mathtt{pid}) := \sigma_{U_{i+1}}(\mathsf{tx}^{\mathsf{r}'}_i)$ and set $\mathsf{right}(\mathtt{pid}) := (\overline{\gamma_i}, \vec{\theta}'_i, \mathsf{tx}^{\mathsf{r}'}_i, \mathsf{sk}_{\widetilde{U}_i})$.
17. If U_i is not the sending endpoint:
   - Retrieve $(\overline{\gamma_{i-1}}, \vec{\theta}_{i-1}, \mathsf{tx}^{\mathsf{r}}_{i-1}) := \mathsf{left}(\mathtt{pid})$
   - Extract $\theta_{\varepsilon_{i-1}} \in \mathsf{tx}^{\mathsf{r}}_{i-1}.\mathsf{input}$
   - $\vec{\theta}'_{i-1} := \mathtt{genNewState}(\overline{\gamma_{i-1}}, \alpha'_i + \mathsf{fee}, T)$
   - $\mathsf{tx}^{\mathsf{r}'}_{i-1} := \mathtt{genRefTx}(\vec{\theta}'_{i-1}, \theta_{\varepsilon_{i-1}}, U_{i-1})$
   - Create the signature $\sigma_{U_i}(\mathsf{tx}^{\mathsf{r}'}_{i-1})$
   - Send $(\mathtt{sid}, \mathtt{pid}, \mathtt{close\text{-}req}, \vec{\theta}'_{i-1}, \mathsf{tx}^{\mathsf{r}'}_{i-1}, \sigma_{U_i}(\mathsf{tx}^{\mathsf{r}'}_{i-1})) \xhookrightarrow{\tau_i} U_{i-1}$.
   - $(\mathtt{sid}, \mathtt{pid}, \mathtt{CLOSED}) \xhookrightarrow{\tau_i} \mathcal{E}$
18. If U_i is the sending endpoint:
   - $(\mathtt{sid}, \mathtt{pid}, \mathtt{VC\text{-}CLOSED}) \xhookrightarrow{\tau_i} \mathcal{E}$

Emergency-Offload

U_0 in every round τ

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

1. Let $(\mathsf{tx}^{\mathsf{vc}}, T) := \mathsf{local}(\mathtt{pid})$.
2. If $\tau < T - t_{\mathsf{c}} - 3\Delta$, continue with next loop iteration.
3. Remove $(\mathtt{pid}, \tau_{\mathsf{f}}, U_n)$ from paySet.
4. Sign $\mathsf{tx}^{\mathsf{vc}}$ yielding $\sigma_{U_0}(\mathsf{tx}^{\mathsf{vc}})$ and set $\overline{\mathsf{tx}^{\mathsf{vc}}} := (\mathsf{tx}^{\mathsf{vc}}, (\sigma_{U_0}(\mathsf{tx}^{\mathsf{vc}})))$. Send $(\mathtt{ssid}_L, \mathtt{POST}, \overline{\mathsf{tx}^{\mathsf{vc}}}) \xhookrightarrow{\tau} \mathcal{G}_{Ledger}$.

Respond

U_i at the end of every round

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 $(\mathsf{tx}^{\mathsf{vc}}, T) := \mathsf{local}(\mathtt{pid})$ and do the following. If $t < T - t_{\mathsf{c}} - 2\Delta$, $\mathsf{tx}^{\mathsf{vc}}$ is on the ledger $\mathcal{L}$ and $\overline{\gamma_i}.\mathsf{st} = \vec{\theta}_i$, do the following:

- Remove the entry for $\texttt{pid}$ from right, send $(\texttt{ssid}_C, \texttt{CLOSE}, \overline{\gamma_i}.\mathsf{id}) \overset{t}{\hookrightarrow} \mathcal{F}_{Channel}$.
- If a transaction tx with $\mathsf{tx.output} = \vec{\theta}_i$ is on $\mathcal{L}$ in round $t_1 \leq t + t_c$ wait Δ rounds.
- Sign tx_i^r yielding $\sigma_{U_i}(\mathsf{tx}_i^\mathsf{r})$ and use $\mathsf{sk}_{\widetilde{U}_i}$ to sign tx_i^r yielding $\sigma_{\widetilde{U}_i}(\mathsf{tx}_i^\mathsf{r})$
- Set $\overline{\mathsf{tx}_i^\mathsf{r}} := (\mathsf{tx}_i^\mathsf{r}, (\sigma_{U_i}(\mathsf{tx}_i^\mathsf{r}), \mathsf{rightSig}(\texttt{pid}), \sigma_{\widetilde{U}_i}(\mathsf{tx}_i^\mathsf{r})))$ and send $(\texttt{ssid}_L, \texttt{POST}, \overline{\mathsf{tx}_i^\mathsf{r}}) \xhookrightarrow{t_1+\Delta} \mathcal{G}_{Ledger}$. When it appears on $\mathcal{L}$ in round $t_2 < T$, send $(\texttt{sid}, \texttt{pid}, \texttt{REVOKED}) \overset{t_2}{\hookrightarrow} \mathcal{E}$

2. For every $\texttt{pid}$ in $\mathsf{left}.\texttt{keyList}()$, let $(\overline{\gamma_{i-1}}, \vec{\theta}_{i-1}, \mathsf{tx}_{i-1}^\mathsf{r}) := \mathsf{left}(\texttt{pid})$, let $(\mathsf{tx}^\mathsf{vc}, T) := \mathsf{local}(\texttt{pid})$ and do the following. If $t \geq T$ and a transaction tx with $\mathsf{tx.output} = \vec{\theta}_{i-1}$ is on the ledger $\mathcal{L}$, but not $\mathsf{tx}_{i-1}^\mathsf{r}$, do the following:
   - Remove the entry for $\texttt{pid}$ from left and create $\mathsf{tx}_{i-1}^\mathsf{p} := \texttt{genPayTx}(\overline{\gamma_{i-1}}.\mathsf{st}, U_i)$.
   - Sign $\mathsf{tx}_{i-1}^\mathsf{p}$ yielding $\sigma_{U_i}(\mathsf{tx}_{i-1}^\mathsf{p})$.
   - Set $\overline{\mathsf{tx}_{i-1}^\mathsf{p}} := (\mathsf{tx}_{i-1}^\mathsf{p}, \sigma_{U_i}(\mathsf{tx}_{i-1}^\mathsf{p}))$ and send $(\texttt{ssid}_L, \texttt{POST}, \overline{\mathsf{tx}_{i-1}^\mathsf{p}}) \overset{t}{\hookrightarrow} \mathcal{G}_{Ledger}$.
   - If it appears on $\mathcal{L}$ in round $t_1 \leq t + \Delta$, send $(\texttt{sid}, \texttt{pid}, \texttt{FORCE-PAY}) \overset{t_1}{\hookrightarrow} \mathcal{E}$

E.6 Simulation

In this section we provide the code for the simulator $\mathcal{S}$, which can simulate the protocol in the ideal world, and give the proof that the protocol (see Appendix E.5) UC-realizes the ideal functionality $\mathcal{F}_{VC}$ shown in Appendix E.4.

Simulator

Local variables:

left	A map, storing the channel $\overline{\gamma_{i-1}}$ and output $\theta_{\varepsilon_{i-1}}$ for a given keypair consisting of a payment id $\texttt{pid}$ and a user U_i, or $(\bot, \bot)$ if U_i is the sending endpoint.
right	A map, storing the transaction tx_i^r for a given keypair consisting of a payment id $\texttt{pid}$ and a user U_i.
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 $\texttt{pid}$ and a user U_i.

Simulator for init phase

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

Simulator for Open-VC phase

<ins>Pre-create VC</ins>

1. Upon $(\texttt{sid}, \texttt{pid}, \texttt{pre-create-vc}, \overline{\gamma_\mathsf{vc}}, \mathsf{tx}^\mathsf{vc}, T) \overset{\tau}{\hookleftarrow} U_0$ if U_0 dishonest, go to step (3).
2. Upon $(\texttt{sid}, \texttt{pid}, \texttt{pre-create-vc}, \overline{\gamma_\mathsf{vc}}, \mathsf{tx}^\mathsf{vc}, T) \overset{\tau}{\hookleftarrow} \mathcal{F}_{VC}$ if U_0 honest, do the following. If U_n honest go to step (3). If U_n dishonest, send $(\texttt{sid}, \texttt{pid}, \texttt{pre-create-vc}, \overline{\gamma_\mathsf{vc}}, \mathsf{tx}^\mathsf{vc}, T) \overset{\tau}{\hookrightarrow} U_n$ and go idle.
3. $(\texttt{ssid}_C, \texttt{PRE-CREATE}, \overline{\gamma_\mathsf{vc}}, \mathsf{tx}^\mathsf{vc}, 0, T - \tau) \overset{\tau}{\hookrightarrow} \mathcal{F}_{Channel}$.
4. If not $(\texttt{ssid}_C, \texttt{PRE-CREATED}, \overline{\gamma_\mathsf{vc}}.\mathsf{id}) \xhookleftarrow{\tau+t_\mathsf{o}} \mathcal{F}_{Channel}$, mark VC as unusable.

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

1. Upon $(\texttt{sid}, \texttt{pid}, \texttt{open}, \mathsf{tx}^\mathsf{vc}, \mathsf{rList}, \mathsf{onion}_{i+1}, \alpha_i, T, \overline{\gamma_{i-1}}, \overline{\gamma_i}, \theta_{\varepsilon_{i-1}}, \theta_{\varepsilon_i}) \overset{\tau}{\hookleftarrow} \mathcal{F}_{VC}$ or upon being called by the simulator $\mathcal{S}$ itself in round τ with parameters $(\texttt{pid}, \mathsf{tx}^\mathsf{vc}, \mathsf{rList}, \mathsf{onion}_{i+1}, \alpha_i, T, \overline{\gamma_{i-1}}, \overline{\gamma_i}, \theta_{\varepsilon_{i-1}}, \theta_{\varepsilon_i})$.
2. Let $U_i := \overline{\gamma_i}.\mathsf{left}$ and $U_{i+1} := \overline{\gamma_i}.\mathsf{right}$.
3. $\vec{\theta}_i := \texttt{genStateOutputs}(\overline{\gamma_i}, \alpha_i, T)$
4. $\mathsf{tx}_i^\mathsf{r} := \texttt{genRefTx}(\vec{\theta}_i, \theta_{\varepsilon_i}, U_i)$
5. $(\texttt{sid}, \texttt{pid}, \texttt{open-req}, \mathsf{tx}^\mathsf{vc}, \mathsf{rList}, \mathsf{onion}_{i+1}, \vec{\theta}_i, \mathsf{tx}_i^\mathsf{r}) \overset{\tau}{\hookrightarrow} U_{i+1}$
6. Upon $(\texttt{sid}, \texttt{pid}, \texttt{open-ok}, \sigma_{U_{i+1}}(\mathsf{tx}_i^\mathsf{r})) \xhookleftarrow{\tau+2} U_{i+1}$, check that $\sigma_{U_{i+1}}(\mathsf{tx}_i^\mathsf{r})$ is a valid signature for tx_i^r. If not, go idle.
7. Set $\mathsf{rightSig}(\texttt{pid}, U_i) := \sigma_{U_{i+1}}(\mathsf{tx}_i^\mathsf{r})$, $\mathsf{right}(\texttt{pid}, U_i) := \mathsf{tx}_i^\mathsf{r}$
8. Send $(\texttt{ssid}_C, \texttt{UPDATE}, \overline{\gamma_i}.\mathsf{id}, \vec{\theta}_i) \xhookrightarrow{\tau+2} \mathcal{F}_{Channel}$.
9. If not $(\texttt{ssid}_C, \texttt{UPDATED}, \overline{\gamma_i}.\mathsf{id}, \vec{\theta}_i) \xhookleftarrow{\tau+2+t_\mathsf{u}} \mathcal{F}_{Channel}$, go idle.
10. Set $\mathsf{left}(\texttt{pid}, U_i) := (\overline{\gamma_{i-1}}, \theta_{\varepsilon_{i-1}})$
11. Send$(\texttt{sid}, \texttt{pid}, \texttt{register}, \overline{\gamma_i}, \vec{\theta}_i, \mathsf{tx}^\mathsf{vc}, T, \theta_{\varepsilon_i}, R) \overset{\tau}{\hookrightarrow} \mathcal{F}_{VC}$.

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

1. Upon $(\texttt{sid}, \texttt{pid}, \texttt{open-req}, \mathsf{tx}^\mathsf{vc}, \mathsf{rList}, \mathsf{onion}_i, \vec{\theta}_{i-1}, \mathsf{tx}_{i-1}^\mathsf{r}) \overset{\tau}{\hookleftarrow} U_{i-1}$. Let $\alpha_{i-1} := \vec{\theta}_{i-1}[0].\mathsf{cash}$ and extract T from $\vec{\theta}_{i-1}[0].\phi$ (the parameter of $\mathsf{AbsTime}()$). Let $\overline{\gamma_{i-1}}$ be the channel between U_{i-1} and U_i
2. Let $x := \texttt{checkTxVc}(U_i, U_i.a, U_i.b, \mathsf{tx}^\mathsf{vc}, \mathsf{rList}, \mathsf{onion}_i)$. 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 $\vec{\theta}_{i-1} = \texttt{genStateOutputs}(\overline{\gamma_{i-1}}, \alpha_{i-1}, T)$ and $\mathsf{tx}_i^\mathsf{r} := \texttt{genRefTx}(\vec{\theta}_{i-1}, \theta_{\varepsilon_{i-1}}, U_i)$, where $\theta_{\varepsilon_{i-1}} \in \mathsf{tx}^\mathsf{vc}$ and $\theta_{\varepsilon_{i-1}} \neq \theta_{\varepsilon_i}$.
4. $(\texttt{sid}, \texttt{pid}, \texttt{check-id}, \mathsf{tx}^\mathsf{vc}, \theta_{\varepsilon_i}, R_i, U_{i-1}, U_i, U_{i+1}, \alpha_i, T) \overset{\tau}{\hookrightarrow} \mathcal{F}_{VC}$
5. If not $(\texttt{sid}, \texttt{pid}, \texttt{ok}, \overline{\gamma_i}) \overset{\tau}{\hookleftarrow} \mathcal{F}_{VC}$, go idle. Let $U_{i+1} := \overline{\gamma_i}.\mathsf{right}$.
6. Sign $\mathsf{tx}_{i-1}^\mathsf{r}$ on behalf of U_i yielding $\sigma_{U_i}(\mathsf{tx}_{i-1}^\mathsf{r})$ and $(\texttt{sid}, \texttt{pid}, \texttt{open-ok}, \sigma_{U_i}(\mathsf{tx}_{i-1}^\mathsf{r})) \overset{\tau}{\hookrightarrow} U_{i-1}$.
7. Upon $(\texttt{ssid}_C, \texttt{UPDATED}, \overline{\gamma_{i-1}}.\mathsf{id}, \vec{\theta}_{i-1}) \xhookleftarrow{\tau+1+t_\mathsf{u}} \mathcal{F}_{Channel}$, send $(\texttt{sid}, \texttt{pid}, \texttt{register}, \overline{\gamma_{i-1}}, \vec{\theta}_{i-1}, \mathsf{tx}^\mathsf{vc}, T, \bot, \bot) \overset{\tau}{\hookrightarrow} \mathcal{F}_{VC}$. Otherwise, go idle.
8. Set $\mathsf{left}(\texttt{pid}, U_i) := (\overline{\gamma_{i-1}}, \theta_{\varepsilon_{i-1}})$.

9. 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 $(\mathtt{sid},\mathtt{pid},\mathtt{vc\text{-}open},\mathsf{tx}^{\mathsf{vc}}) \xhookrightarrow{\tau+1+t_u} \mathcal{F}_{VC}$. If U_0 is dishonest, create signature $\sigma_{U_n}(\mathsf{tx}^{\mathsf{vc}})$ on behalf of U_n and send $(\mathtt{sid},\mathtt{pid},\mathtt{finalize},\mathsf{tx}^{\mathsf{vc}},\sigma_{U_n}(\mathsf{tx}^{\mathsf{vc}})) \xhookrightarrow{\tau+1+t_u} U_0$. In both cases, send via $\mathcal{F}_{VC}$ to the dummy user U_n the message $(\mathtt{sid},\mathtt{pid},\mathtt{VC\text{-}OPENED},\mathsf{tx}^{\mathsf{vc}},T,\alpha_{i-1}) \xhookrightarrow{\tau+1+t_u} U_n$. Go Idle.
10. Send via $\mathcal{F}_{VC}$ to the dummy user U_i the message $(\mathtt{sid},\mathtt{pid},\mathtt{OPENED}) \xhookrightarrow{\tau+1+t_u} U_i$.
11. If U_{i+1} honest, call $\mathsf{process}(\mathtt{sid},\mathtt{pid},\mathsf{tx}^{\mathsf{vc}},\overline{\gamma_{i-1}},\overline{\gamma_i},R_i,\mathsf{onion}_i,\alpha_i,T)$.
12. If U_{i+1} dishonest, go to step Simulator U_i honest, U_{i+1} dishonest step 1 with parameters $(\mathtt{pid},\mathsf{tx}^{\mathsf{vc}},\mathsf{rList},\mathsf{onion}_{i+1},\alpha_{i-1}-\mathsf{fee},T,\overline{\gamma_{i-1}},\overline{\gamma_i},\theta_{\varepsilon_{i-1}},\theta_{\varepsilon_i})$.

$\underline{\mathsf{process}(\mathtt{sid},\mathtt{pid},\mathsf{tx}^{\mathsf{vc}},\overline{\gamma_{i-1}},\overline{\gamma_i},R_i,\mathsf{onion}_i,\alpha_{i-1},T)}$

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:
   - $x := \mathtt{checkTxVc}(U_{i+1},U_{i+1}.a,U_{i+1}.b,\mathsf{tx}^{\mathsf{vc}},\mathsf{rList},\mathsf{onion}_{i+1})$:
     - If $x = \perp$, append U_{i+1} and then $\perp$ 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 $(\mathtt{sid},\mathtt{pid},\mathtt{continue},\mathsf{nodeList},\mathsf{tx}^{\mathsf{vc}},\mathsf{onions},\mathsf{rMap},\mathsf{rList},\mathsf{stealthMap},\alpha_{i-1},T,\overline{\gamma_{i-1}}) \xhookrightarrow{\tau} \mathcal{F}_{VC}$

[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 and emergency-offload phase

a) Publishing $\mathsf{tx}^{\mathsf{vc}}$

Upon receiving a message $(\mathtt{sid},\mathtt{pid},\mathtt{offload},\mathsf{tx}^{\mathsf{vc}},U_0) \xhookleftarrow{\tau} \mathcal{F}_{VC}$ and U_0 honest, sign $\mathsf{tx}^{\mathsf{vc}}$ on behalf of U_0 yielding $\sigma_{U_0}(\mathsf{tx}^{\mathsf{vc}})$. Set $\overline{\mathsf{tx}^{\mathsf{vc}}} := (\mathsf{tx}^{\mathsf{vc}},\sigma_{U_0}(\mathsf{tx}^{\mathsf{vc}}))$ and send $(\mathtt{ssid}_L,\mathtt{POST},\overline{\mathsf{tx}^{\mathsf{vc}}}) \xhookrightarrow{\tau} \mathcal{G}_{Ledger}$.

b) Case U_n honest, U_0 dishonest

Upon message $(\mathtt{sid},\mathtt{pid},\mathtt{finalize},\mathsf{tx}^{\mathsf{vc}}) \xhookleftarrow{\tau} \mathcal{F}_{VC}$, sign $\mathsf{tx}^{\mathsf{vc}}$ on behalf of U_n yielding $\sigma_{U_n}(\mathsf{tx}^{\mathsf{vc}})$. Send $(\mathtt{sid},\mathtt{pid},\mathtt{finalize},\mathsf{tx}^{\mathsf{vc}},\sigma_{U_n}(\mathsf{tx}^{\mathsf{vc}})) \xhookrightarrow{\tau} U_0$.

c) Case U_n dishonest, U_0 honest

Upon message $(\mathtt{sid},\mathtt{pid},\mathtt{finalize},\mathsf{tx}^{\mathsf{vc}},\sigma_{U_n}(\mathsf{tx}^{\mathsf{vc}})) \xhookleftarrow{\tau} U_n$, send $(\mathtt{sid},\mathtt{pid},\mathtt{confirmed},\mathsf{tx}^{\mathsf{vc}},\sigma_{U_n}(\mathsf{tx}^{\mathsf{vc}})) \xhookrightarrow{\tau} \mathcal{F}_{VC}$.

Simulator for Close-VC phase

a) Case U_i is honest, U_{i-1} dishonest

1. Upon $(\mathtt{sid},\mathtt{pid},\mathtt{close},\alpha'_{i-1},\overline{\gamma_{i-1}}) \xhookleftarrow{\tau} \mathcal{F}_{VC}$ or upon being called by the simulator $\mathcal{S}$ itself in round τ with parameters $(\mathtt{pid},\alpha'_{i-1},\overline{\gamma_{i-1}})$.
2. Retrieve $(\overline{\gamma_{i-1}},\theta_{\varepsilon_{i-1}}) := \mathsf{left}(\mathtt{pid},U_i)$.
3. Extract T from $\overline{\gamma_{i-1}}.\mathsf{st}[0]$.
4. Let $U_i := \overline{\gamma_i}.\mathsf{left}$ and $U_{i+1} := \overline{\gamma_i}.\mathsf{right}$.
5. $\vec{\theta}'_{i-1} := \mathtt{genNewState}(\overline{\gamma_{i-1}},\alpha'_i,T)$
6. $\mathsf{tx}^{\mathsf{r}}_i := \mathtt{genRefTx}(\vec{\theta}'_{i-1},\theta_{\varepsilon_{i-1}},U_{i-1})$
7. Create the signature $\sigma_{U_i}(\mathsf{tx}^{\mathsf{r}'}_{i-1})$ on U_i's behalf.
8. Send $(\mathtt{sid},\mathtt{pid},\mathtt{close\text{-}req},\vec{\theta}'_{i-1},\mathsf{tx}^{\mathsf{r}'}_{i-1},\sigma_{U_i}(\mathsf{tx}^{\mathsf{r}'}_{i-1})) \xhookrightarrow{\tau} U_{i-1}$.
9. If $(\mathtt{ssid}_C,\mathtt{UPDATED},\overline{\gamma_{i-1}}.\mathsf{id},\vec{\theta}'_{i-1}) \xhookleftarrow{\tau+1+t_u} \mathcal{F}_{Channel}$, send $(\mathtt{sid},\mathtt{pid},\mathtt{replace},\overline{\gamma_{i-1}},\vec{\theta}'_{i-1}) \xhookrightarrow{\tau+1+t_u} \mathcal{F}_{VC}$.

b) Case U_i is honest, U_{i+1} dishonest

1. Upon $(\mathtt{sid},\mathtt{pid},\mathtt{close\text{-}req},\vec{\theta}'_i,\mathsf{tx}^{\mathsf{r}'}_i,\sigma_{U_{i+1}}(\mathsf{tx}^{\mathsf{r}'}_i)) \xhookleftarrow{\tau} U_{i+1}$, let $\overline{\gamma_i}$ the channel between U_i and U_{i+1}.
2. Let $\mathsf{tx}^{\mathsf{r}}_i := \mathsf{right}(\mathtt{pid},U_i)$. If no such entry exists, go idle.
3. Let $\vec{\theta}_i := \overline{\gamma_i}.\mathsf{st}$ and check that $\vec{\theta}_i[0] \in \mathsf{tx}^{\mathsf{r}}_i.\mathsf{input}$. If not, go idle.
4. Extract $\theta_{\varepsilon_i} \in \mathsf{tx}^{\mathsf{r}}_i.\mathsf{input}$
5. Extract T from $\vec{\theta}_i[0].\phi$ and $\alpha_i := \vec{\theta}_i[0].\mathsf{cash}$
6. Extract T' from $\vec{\theta}'_i[0].\phi$ and $\alpha'_i := \vec{\theta}'_i[0].\mathsf{cash}$
7. If $T' \neq T$, abort. If not $0 \leq \alpha'_i \leq \alpha_i$, abort.
8. If $\vec{\theta}'_i \neq \mathtt{genNewState}(\overline{\gamma_i},\alpha'_i,T)$, abort.
9. If $\mathsf{tx}^{\mathsf{r}'}_i \neq \mathtt{genRefTx}(\vec{\theta}'_i,\theta_{\varepsilon_i},U_i)$, abort.
10. If $\sigma_{U_{i+1}}(\mathsf{tx}^{\mathsf{r}'}_i)$ is not a valid signature for $\mathsf{tx}^{\mathsf{r}'}_i$, abort.
11. Via $\mathcal{F}_{VC}$ to the dummy user U_i send $(\mathtt{sid},\mathtt{pid},\mathtt{CLOSE},\alpha'_i) \xhookrightarrow{\tau} U_i$ and expect the answer $(\mathtt{sid},\mathtt{pid},\mathtt{CLOSE\text{-}ACCEPT}) \xhookleftarrow{\tau} U_i$, otherwise go idle.
12. Send $(\mathtt{ssid}_C,\mathtt{UPDATE},\overline{\gamma_i}.\mathsf{id},\vec{\theta}'_i) \xhookrightarrow{\tau} \mathcal{F}_{Channel}$.
13. Expect $(\mathtt{ssid}_C,\mathtt{UPDATED},\overline{\gamma_i}.\mathsf{id},\vec{\theta}'_i) \xhookleftarrow{\tau+t_u} \mathcal{F}_{Channel}$, else go idle.
14. Send $(\mathtt{sid},\mathtt{pid},\mathtt{replace},\overline{\gamma_i},\vec{\theta}'_i) \xhookrightarrow{\tau} \mathcal{F}_{VC}$.
15. Set $\mathsf{right}(\mathtt{pid},U_i) := \mathsf{tx}^{\mathsf{r}'}_i$
16. Retrieve $(\overline{\gamma_{i-1}},\theta_{\varepsilon_{i-1}}) := \mathsf{left}(\mathtt{pid},U_i)$.

17. If $U_i = U_0$, send via $\mathcal{F}_{VC}$ to the dummy user U_i the message $(\mathtt{sid}, \mathtt{pid}, \mathtt{VC\text{-}CLOSED}) \xhookrightarrow{\tau+t_u} U_i$. Go idle.
18. Send via $\mathcal{F}_{VC}$ to the dummy user U_i the message $(\mathtt{sid}, \mathtt{pid}, \mathtt{CLOSED}) \xhookrightarrow{\tau+t_u} U_i$.
19. If U_{i-1} honest, send $(\mathtt{sid}, \mathtt{pid}, \mathtt{continue\text{-}close}, \overline{\gamma_{i-1}}, \alpha_i' + \mathsf{fee}) \xhookrightarrow{\tau+t_u} \mathcal{F}_{VC}$
20. If dishonest, go to step Simulator U_i honest, U_{i+1} dishonest step 1 with parameters $(\mathtt{pid}, \alpha_i' + \mathsf{fee}, \overline{\gamma_{i-1}})$.

Simulator for respond phase

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

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

Proof.

We proceed to show that for any environment $\mathcal{E}$ an interaction with $\phi_{\mathcal{F}_{VC}}$ (the ideal protocol of ideal functionality $\mathcal{F}_{VC}$) via the dummy parties and $\mathcal{S}$ (ideal world) is indistinguishable from an interaction with Π and an adversary $\mathcal{A}$. More formally, we show that the execution ensembles $\mathtt{EXEC}_{\mathcal{F}_{VC},\mathcal{S},\mathcal{E}}$ and $\mathtt{EXEC}_{\Pi,\mathcal{A},\mathcal{E}}$ are indistinguishable for the environment $\mathcal{E}$.

We use the notation $m[\tau]$ to denote that a message m is observed by $\mathcal{E}$ at round τ. We interact with other ideal functionalities. These functionalities might in turn interact with the environment or parties under adversarial control, either by sending messages or by impacting public variables, i.e., the ledger $\mathcal{L}$. To capture this impact, we define a function $\mathsf{obsSet}(m, \mathcal{F}, \tau)$, returning a set of all by $\mathcal{E}$ observable actions which are triggered by calling $\mathcal{F}$ with message m in round τ.

In this proof, we do a case-by-case analysis of each corruption setting. We start with the view of the environment in the real world and follow with the view in the ideal world, simulated by $\mathcal{S}$. Due to the similarities of the Open-VC, the Finalize well as the Respond phase and the Pay, Finalize and Respond phase in [4], parts of the corresponding proofs are taken verbatim from there.

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

Proof. We compare the execution ensembles for the open phase in the real and the ideal world. In Table 5 we match the sequence of the Open-VC phase of the ideal and the real world and point to which code is executed. We divide this phase in setup and open. For readability, we define the following messages:

- $m_0 := (\mathtt{sid}, \mathtt{pid}, \mathtt{pre\text{-}create\text{-}vc}, \overline{\gamma_{\mathsf{vc}}}, \mathsf{tx}^{\mathsf{vc}}, T)$
- $m_1 := (\mathtt{sid}, \mathtt{pid}, \mathtt{PRE\text{-}CREATE}, \overline{\gamma_{\mathsf{vc}}}, \mathsf{tx}^{\mathsf{vc}}, 0, T - \tau)$
- $m_2 := (\mathtt{sid}, \mathtt{pid}, \mathtt{PRE\text{-}CREATED}, \overline{\gamma_{\mathsf{vc}}}.\mathsf{id})$
- $m_3 := (\mathtt{sid}, \mathtt{pid}, \mathtt{open\text{-}req}, \mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_{i+1}, \vec{\theta}_i, \mathsf{tx}_i^{\mathsf{r}})$
- $m_4 := (\mathtt{sid}, \mathtt{pid}, \mathtt{OPEN}, \mathsf{tx}^{\mathsf{vc}}, \theta_{\varepsilon_{i+1}}, R_i, U_i, U_{i+2}, \alpha_i, T)$
- $m_5 := (\mathtt{sid}, \mathtt{pid}, \mathtt{OPEN\text{-}ACCEPT}, \overline{\gamma_{i+1}})$
- $m_6 := (\mathtt{sid}, \mathtt{pid}, \mathtt{open\text{-}ok}, \sigma_{U_{i+1}}(\mathsf{tx}_i^{\mathsf{r}}))$
- $m_7 := (\mathtt{ssid}_C, \mathtt{UPDATE}, \overline{\gamma_i}.\mathsf{id}, \vec{\theta}_i)$
- $m_8 := (\mathtt{ssid}_C, \mathtt{UPDATED}, \overline{\gamma_i}.\mathsf{id}, \vec{\theta}_i)$
- $m_9 := (\mathtt{sid}, \mathtt{pid}, \mathtt{OPENED})$ or, if sent by the receiver, $m_9 := (\mathtt{sid}, \mathtt{pid}, \mathtt{VC\text{-}OPENED}, \mathsf{tx}^{\mathsf{vc}}, T, \alpha_i)$

Setup.

Real world: An honest U_0 performs $\mathtt{SETUP}$ in τ_0 to set up the initial objects and to pre-create the VC with U_n. In round τ_0, U_0 sends m_0 to U_n (which $\mathcal{E}$ sees in round $\tau_0 + 1$ only if U_n is corrupted) and then, after waiting 1 round, m_1 to $\mathcal{F}_{Channel}$. Note that an honest U_n receiving m_0 in some round τ, sends also a message m_1 to $\mathcal{F}_{Channel}$. If $\mathcal{F}_{Channel}$ received two valid messages m_1 from U_0 and U_n, it returns m_2. Depending on the corruption setting, the ensemble

- $\mathtt{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_0[\tau_0 + 1]\} \cup \mathsf{obsSet}(m_1, \mathcal{F}_{Channel}, \tau_0 + 1)$ for U_0 honest, U_n corrupted
- $\mathtt{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \mathsf{obsSet}(m_1, \mathcal{F}_{Channel}, \tau_0 + 1) \cup \mathsf{obsSet}(m_1, \mathcal{F}_{Channel}, \tau_0 + 1)$ for U_0 honest, U_n honest, where m_1 is sent by each user.
- $\mathtt{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \mathsf{obsSet}(m_1, \mathcal{F}_{Channel}, \tau)$ for U_0 corrupted, U_n honest

Ideal world: For an honest U_0, $\mathcal{F}_{VC}$ performs $\mathtt{SETUP}$ in τ_0 to set up the initial objects and to pre-create the VC. In round τ_0, $\mathcal{F}_{VC}$ asks $\mathcal{S}$ to send m_0 to a dishonest U_n (who receives it in round $\tau_0 + 1$), or, if U_n is honest send m_1 to $\mathcal{F}_{Channel}$ in $\tau_0 + 1$ on behalf of U_n. In both cases, $\mathcal{F}_{VC}$ sends m_1 to $\mathcal{F}_{Channel}$ in $\tau_0 + 1$. If U_0 is dishonest and U_n

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.OpenVC.Setup 1-15	IF.OpenVC.Setup 1-6, Sim.OpenVC.PrecreateVC 1-4, IF.OpenVC.Setup 7-10,12, Sim.OpenVC.a 1-5	IF.OpenVC.Setup 1-6, Sim.OpenVC.PrecreateVC 1-4, IF.OpenVC.Setup 7-11	Sim.OpenVC.PrecreateVC 1-4	m_0, $2 \cdot m_1$, m_3	Pre-Creates VC, performs setup and contacts next user
`CREATE_STATE`	Prot.OpenVC.Open 6-8	IF.OpenVC.Open 12,13 , Sim.OpenVC.a 1-5	IF.OpenVC.Open 12, 13	Sim.OpenVC.b 8-12	m_9, m_3	Upon m_8, sends message m_9 to $\mathcal{E}$. Then, ceates the objects to send in m_3 and sends it to next user (or finalize).
`CHECK_STATE`	Prot.OpenVC.Open 1-4	n/a	IF.OpenVC.Open 1-8	Sim.OpenVC.b 1-4 IF.Check Sim.OpenVC.b 5-7 IF.Register	m_4, m_6	Checks if objects in m_3 are correct, sends m_4 to $\mathcal{E}$ and on m_5, sends m_6 to U_i
`CHECK_SIG`	Prot.OpenVC.Open 5	Sim.OpenVC.a 6-11	IF.OpenVC.Open 9-11	n/a	m_7	Checks if signature of tx_i^r is correct

honest, $\mathcal{S}$ waits for a message m_0 from U_0 in some round τ and sends m_1 to $\mathcal{F}_{Channel}$. If $\mathcal{F}_{Channel}$ received two valid messages m_1 from U_0 and U_n, it returns m_2. Depending on the corruption setting, the ensemble

- $\mathtt{EXEC}_{\mathcal{F}_{VC},\mathcal{S},\mathcal{E}} := \{m_0[\tau_0 + 1]\} \cup \mathsf{obsSet}(m_1, \mathcal{F}_{Channel}, \tau_0 + 1)$ for U_0 honest, U_n corrupted
- $\mathtt{EXEC}_{\mathcal{F}_{VC},\mathcal{S},\mathcal{E}} := \mathsf{obsSet}(m_1, \mathcal{F}_{Channel}, \tau_0 + 1) \cup \mathsf{obsSet}(m_1, \mathcal{F}_{Channel}, \tau_0 + 1)$ for U_0 honest, U_n honest, where m_1 is sent for each user.
- $\mathtt{EXEC}_{\mathcal{F}_{VC},\mathcal{S},\mathcal{E}} := \mathsf{obsSet}(m_1, \mathcal{F}_{Channel}, \tau)$ for U_0 corrupted, U_n honest

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

Real world: After U_i performs either `SETUP` or `CREATE_STATE`, it sends m_3 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_3 in round $\tau + 1$. Iff U_{i+1} replies with a correct message m_6 in $\tau + 2$, U_i will perform `CHECK_SIG` and call $\mathcal{F}_{Channel}$ with message m_7 in the same round. The ensemble is $\mathtt{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_3[\tau + 1]\} \cup \mathsf{obsSet}(m_7, \mathcal{F}_{Channel}, \tau + 2)$

Ideal world: After $\mathcal{F}_{VC}$ performs either `SETUP` or simulator performs `CREATE_STATE`, the simulator sends m_3 to U_{i+1} in the current round τ. $\mathcal{E}$ will see m_3 in round $\tau + 1$. Iff U_{i+1} replies with a correct message m_6 in $\tau + 2$, the simulator will perform `CHECK_SIG` and call $\mathcal{F}_{Channel}$ with message m_7 in the same round. The ensemble is $\mathtt{EXEC}_{\mathcal{F}_{VC},\mathcal{S},\mathcal{E}} := \{m_3[\tau + 1]\} \cup \mathsf{obsSet}(m_7, \mathcal{F}_{Channel}, \tau + 2)$

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

Real world: After U_i performs either `SETUP` or `CREATE_STATE`, it sends m_3 to U_{i+1} in the current round τ. U_{i+1} performs `CHECK_STATE` and sends m_4 to $\mathcal{E}$ in round $\tau + 1$. Iff $\mathcal{E}$ replies with m_5, U_{i+1}, U_{i+1} replies with m_6. U_i receives this in round $\tau + 2$, performs `CHECK_SIG` and sends m_7 to $\mathcal{F}_{Channel}$. U_{i+1} expects the message m_8 in round $\tau + 2 + t_{\mathsf{u}}$ and will then send m_9 to $\mathcal{E}$. Afterwards it continues with either `CREATE_STATE` or `FINALIZE`. The ensemble is $\mathtt{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_4[\tau + 1], m_9[\tau + 2 + t_{\mathsf{u}}]\} \cup \mathsf{obsSet}(m_7, \mathcal{F}_{Channel}, \tau + 2)$

Ideal world: After $\mathcal{F}_{VC}$ 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, $\mathcal{F}_{VC}$ perform the procedure Open. This behaves exactly like `CREATE_STATE`, `CHECK_STATE` and `CHECK_SIG`. However, since every object is created by $\mathcal{F}_{VC}$, the checks are omitted. The procedure Open outputs the messages m_4 in round $\tau + 1$ and iff $\mathcal{E}$ replies with m_5, calls $\mathcal{F}_{Channel}$ with m_7 in $\tau + 2$. Finally, if m_8 is received in round $\tau + 2 + t_{\mathsf{u}}$, outputs m_9 to $\mathcal{E}$. The ensemble is $\mathtt{EXEC}_{\mathcal{F}_{VC},\mathcal{S},\mathcal{E}} := \{m_4[\tau + 1], m_9[\tau + 2 + t_{\mathsf{u}}]\} \cup \mathsf{obsSet}(m_7, \mathcal{F}_{Channel}, \tau + 2)$

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

Real world: After U_{i+1} receives the message m_3 from U_i, it performs `CHECK_STATE` and sends m_4 to $\mathcal{E}$ in the current round τ. Iff $\mathcal{E}$ replies with m_5, U_{i+1} sends m_6 to U_i. If U_{i+1} receives the message m_8 from $\mathcal{F}_{Channel}$ in round $\tau + 1 + t_{\mathsf{u}}$, it sends m_9 to $\mathcal{E}$. The ensemble is $\mathtt{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_4[\tau], m_6[\tau + 1], m_9[\tau + 1 + t_{\mathsf{u}}]\}$

Ideal world: After the simulator receives m_3 from U_i, it performs `CHECK_STATE` together with $\mathcal{F}_{VC}$ and $\mathcal{F}_{VC}$ sends m_4 to $\mathcal{E}$. Iff $\mathcal{E}$ replies with m_5, $\mathcal{F}_{VC}$ asks the simulator to send m_6 to U_i. All of this happens in the current round τ. If the simulator receives m_8 in round $\tau + 1 + t_{\mathsf{u}}$, it sends m_9 to $\mathcal{E}$. The ensemble is $\mathtt{EXEC}_{\mathcal{F}_{VC},\mathcal{S},\mathcal{E}} := \{m_4[\tau], m_6[\tau + 1], m_9[\tau + 1 + t_{\mathsf{u}}]\}$

Note that we do not care about the case were both U_i and U_{i+1} are corrupted, because the environment is communicating with itself, which is trivially the same in the ideal and the real world. We see that for the setup and open phase in all three corruption 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 $\mathcal{F}_{VC}$.*

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.

- $m_{10} := (\texttt{sid}, \texttt{pid}, \texttt{finalize}, \mathsf{tx}^{\mathsf{vc}}, \sigma_{U_n}(\mathsf{tx}^{\mathsf{vc}}))$
- $m_{11} := (\texttt{ssid}_L, \texttt{POST}, \overline{\mathsf{tx}^{\mathsf{vc}}})$

1. U_n honest, U_0 corrupted.

Real world: After performing `FINALIZE` in the current round τ, U_n sends m_{10} to U_0, which $\mathcal{E}$ sees in $\tau+1$. The ensemble is $\texttt{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_{10}[\tau+1]\}$

Ideal world: After either $\mathcal{F}_{VC}$ or the simulator performs `FINALIZE` in the current round τ, the simulator sends m_{10} to U_0, which $\mathcal{E}$ sees in $\tau+1$. The ensemble is $\texttt{EXEC}_{\mathcal{F}_{VC},\mathcal{S},\mathcal{E}} := \{m_{10}[\tau+1]\}$

2. U_n honest, U_0 honest.

Real world: After performing `FINALIZE` in the current round τ, U_n sends m_{10} to U_0. In the meantime, U_0 performs `CHECK_FINALIZE` and should it not receive a correct message m_{10} in the correct round, will send m_{11} to $\mathcal{G}_{Ledger}$ in round τ'. The ensemble is $\texttt{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \mathsf{obsSet}(m_{11}, \mathcal{G}_{Ledger}, \tau')$

Ideal world: Either $\mathcal{F}_{VC}$ or the simulator performs `FINALIZE` in the current round τ. In the meantime, $\mathcal{F}_{VC}$ performs `CHECK_FINALIZE` and will, if the checks in `FINALIZE` failed or it was performed in a incorrect round τ', $\mathcal{F}_{VC}$ will instruct the simulator to send m_{11} to $\mathcal{G}_{Ledger}$ in rounds τ'. The ensemble is $\texttt{EXEC}_{\mathcal{F}_{VC},\mathcal{S},\mathcal{E}} := \mathsf{obsSet}(m_{11}, \mathcal{G}_{Ledger}, \tau')$

3. U_n corrupted, U_0 honest.

Real world: U_0 performs `CHECK_FINALIZE` and should it not receive a correct message m_{10} in the correct round, will send m_{11} to $\mathcal{G}_{Ledger}$ in round τ'. The ensemble is $\texttt{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \mathsf{obsSet}(m_{11}, \mathcal{G}_{Ledger}, \tau')$

Ideal world: The simulator and $\mathcal{F}_{VC}$ perform `CHECK_FINALIZE` and should the simulator not receive a correct message m_{10} in the correct round, $\mathcal{F}_{VC}$ will instruct the simulator to send m_{11} to $\mathcal{G}_{Ledger}$ in round τ'. The ensemble is $\texttt{EXEC}_{\mathcal{F}_{VC},\mathcal{S},\mathcal{E}} := \mathsf{obsSet}(m_{11}, \mathcal{G}_{Ledger}, \tau')$ □

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

Proof. Trivially, this the update phase is the same, as the pre-update messages are simply forwarded to $\mathcal{F}_{Channel}$ in both the real and the ideal world. □

Lemma 4. *Let Σ be a EUF-CMA secure signature scheme. Then, the Close phase of protocol Π GUC-emulates the Close phase of functionality $\mathcal{F}_{VC}$.*

Proof. Again, we consider the execution ensembles of the interaction between users U_{i+1} and U_i for three different cases. We match the sequences and where they are used in the ideal and real world in Table 7. We define the following messages.

- $m_{12} := (\texttt{sid}, \texttt{pid}, \texttt{close-req}, \vec{\theta}'_{i-1}, \mathsf{tx}^{\mathsf{r}'}_{i-1}, \sigma_{U_i}(\mathsf{tx}^{\mathsf{r}'}_{i-1}))$
- $m_{13} := (\texttt{sid}, \texttt{pid}, \texttt{CLOSE}, \alpha'_i)$
- $m_{14} := (\texttt{sid}, \texttt{pid}, \texttt{CLOSE-ACCEPT})$
- $m_{15} := (\texttt{ssid}_C, \texttt{UPDATE}, \overline{\gamma_i}.\mathsf{id}, \vec{\theta}'_i)$
- $m_{16} := (\texttt{ssid}_C, \texttt{UPDATED}, \overline{\gamma_i}.\mathsf{id}, \vec{\theta}'_i)$
- $m_{17} := (\texttt{sid}, \texttt{pid}, \texttt{CLOSED})$ or, if sent by the sender, $m_{17} := (\texttt{sid}, \texttt{pid}, \texttt{VC-CLOSED})$

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

Real world: After U_{i+1} performs either `SHUTDOWN` or `PROCEED_CLOSE`, it sends m_{12} to U_i in the current round τ. The environment $\mathcal{E}$ controls $\mathcal{A}$ and therefore U_i and will see m_{12} in round $\tau+1$. The ensemble is $\texttt{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_{12}[\tau+1]\}$

Ideal world: After $\mathcal{F}_{VC}$ performs either `SHUTDOWN` or simulator performs `PROCEED_CLOSE`, the simulator sends m_{12} to U_i in the current round τ. $\mathcal{E}$ will see m_{12} in round $\tau+1$. The ensemble is $\texttt{EXEC}_{\mathcal{F}_{VC},\mathcal{S},\mathcal{E}} := \{m_{12}[\tau+1]\}$

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

Real world: After U_{i+1} performs either `SHUTDOWN` or `PROCEED_CLOSE`, it sends m_{12} to U_i in the current round τ. U_i receives this message in $\tau+1$ and carries out `CLOSE`, sending m_{13} to $\mathcal{E}$ in $\tau+1$ and, upon m_{14} in $\tau+1$, sends m_{15} in $\tau+1$ to $\mathcal{F}_{Channel}$. After a successful update (m_{16} is received), U_i sends m_{17} to $\mathcal{E}$ in $\tau+1+t_{\mathsf{u}}$ and continues with U_{i-1}, if it exists. The ensemble is $\texttt{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_{13}[\tau+1], m_{17}[\tau+1+t_{\mathsf{u}}]\} \cup \mathsf{obsSet}(m_{15}, \tau+1, \mathcal{F}_{Channel})$.

Ideal world: After $\mathcal{F}_{VC}$ performs either `SHUTDOWN` or is invoked by itself (in step Close.12) or by the simulator (in step b.19 and then IF.Continue-Close) in the current round τ, $\mathcal{F}_{VC}$ perform the procedure Close. This behaves exactly like `CLOSE` and `PROCEED_CLOSE`. However, since every object is created by $\mathcal{F}_{VC}$, the checks are omitted. The procedure Close outputs the messages m_{13} in round $\tau+1$ and iff $\mathcal{E}$ replies with m_{14}, calls $\mathcal{F}_{Channel}$ with m_{15} in $\tau+1$. Finally, if m_{16} is received in round $\tau+1+t_{\mathsf{u}}$, outputs m_{17} to $\mathcal{E}$ and continues for U_{i-1}, if it exists. The ensemble is $\texttt{EXEC}_{\mathcal{F}_{VC},\mathcal{S},\mathcal{E}} := \{m_{13}[\tau+1], m_{17}[\tau+1+t_{\mathsf{u}}]\} \cup \mathsf{obsSet}(m_{15}, \tau+1, \mathcal{F}_{Channel})$

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

Real world: After U_i receives the message m_{12} from U_{i+1} in round τ, it performs `CLOSE` and sends m_{13} to $\mathcal{E}$ in τ. Iff $\mathcal{E}$ replies with m_{14} in the same round, U_i sends

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.OpenVC.Open 8	IF.OpenVC.12 and Sim.Finalize.b or Sim.OpenVC.b 9	IF.OpenVC.12 or Sim.OpenVC.b 9, IF.VCOpen	n/a	m_{10}	Sends finalize message to U_0
`CHECK_FINALIZE`	Prot.Finalize 1-3	n/a	IF.Finalize 1,2,4 Sim.Finalize.a	Sim.Finalize.c IF.Finalize 1,3,4 Sim.Finalize.a	m_{11}	Checks if $\mathsf{tx}^{\mathsf{vc}}$ is the same, if not, publishes it to ledger with m_{11}.

Table 7: Explanation of the sequence names used in Lemma 4 and where they can be found.

	Real World	Ideal World			Output	Description
		U_{i+1} honest, U_i corrupted	U_{i+1} honest, U_i honest	U_{i+1} corrupted, U_i honest		
`SHUTDOWN`	Prot.CloseVC.Shutdown 1-8	IF.CloseVC.Start 1-3,5, Sim.CloseVC.a 1-8	IF.CloseVC.Start 1-4	n/a	m_{12}	Shutdown starts with U_n, creates objects, contacts next user
`CLOSE`	Prot.CloseVC.Close 1-14	n/a	IF.CloseVC.Close 1-10	Sim.CloseVC.b 1-12	m_{13}, m_{15}	Checks if objects in m_{12} are correct, sends m_{13} to $\mathcal{E}$ and on m_{14}, sends m_{15} to $\mathcal{F}_{Channel}$
`PROCEED_CLOSE`	Prot.CloseVC.Close 15-18	IF.CloseVC.Close 11,12, Sim.CloseVC.a	IF.CloseVC.Close 11,12	Sim.CloseVC.b 13,14, IF.CloseVC.Replace, Sim.CloseVC.B 14-20	m_{17}	On m_{16}, sends m_{17} to $\mathcal{E}$ and continues with next user (if exists).

m_{15} to $\mathcal{F}_{Channel}$ in τ. After receiving m_{16} in $\tau + t_{\mathsf{u}}$, performs `PROCEED_CLOSE`, sending m_{17} to $\mathcal{E}$ and continues with U_{i-1}, if it exists. The ensemble is $\mathtt{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_{13}[\tau], m_{17}[\tau + t_{\mathsf{u}}]\} \cup \mathsf{obsSet}(m_{15}, \tau, \mathcal{F}_{Channel})$.

Ideal world: After the $\mathcal{S}$ receives m_{12} from U_{i+1} in round τ, performs the steps `CLOSE`, sending m_{13} to $\mathcal{E}$ in τ. Iff $\mathcal{E}$ replies with m_{14} in the same round, $\mathcal{S}$ sends m_{15} to $\mathcal{F}_{Channel}$ in τ. After receiving m_{16} in $\tau + t_{\mathsf{u}}$, $\mathcal{S}$ performs `PROCEED_CLOSE` together with $\mathcal{F}_{VC}$, sending m_{17} to $\mathcal{E}$ and continues for U_{i-1}, if it exists. The ensemble is $\mathtt{EXEC}_{\mathcal{F}_{VC},\mathcal{S},\mathcal{E}} := \{m_{13}[\tau], m_{17}[\tau + t_{\mathsf{u}}]\} \cup \mathsf{obsSet}(m_{15}, \tau, \mathcal{F}_{Channel})$.

□

Lemma 5. *Let Σ be a EUF-CMA secure signature scheme. Then, the Emergency-Offload phase of protocol Π GUC-emulates the Emergency-Offload phase of functionality $\mathcal{F}_{VC}$.*

Proof. Again, we consider the execution ensembles, but this time only for an honest U_0. We use message $m_{11} := (\mathtt{ssid}_{L}, \mathtt{POST}, \overline{\mathsf{tx}^{\mathsf{vc}}})$ from before.

Real world: An honest U_0 checks every round and each of its VCs (with a certain `pid`), if the VC has already been closed, see Prot.EmergencyOffload 1-4. If it has not within a certain round τ, U_0 sends m_{11} to $\mathcal{G}_{Ledger}$ in τ. The ensemble is $\mathtt{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \mathsf{obsSet}(m_{11}, \mathcal{G}_{Ledger}, \tau)$.

Ideal world: $\mathcal{F}_{VC}$ checks every round and every VC (with a certain `pid`), if the VC has already been closed. If it has not within a certain round τ, $\mathcal{F}_{VC}$ instructs $\mathcal{S}$ to send m_{11} to $\mathcal{G}_{Ledger}$, see IF.EmergencyOffload 1-4 and Sim.Finalize.a. The ensemble is $\mathtt{EXEC}_{\mathcal{F}_{VC},\mathcal{S},\mathcal{E}} := \mathsf{obsSet}(m_{11}, \mathcal{G}_{Ledger}, \tau)$.

□

Lemma 6. *Let Σ be a EUF-CMA secure signature scheme. Then, the Respond phase of protocol Π GUC-emulates the Respond phase of functionality $\mathcal{F}_{VC}$.*

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 8. We define the following messages.

- $m_{18} := (\mathtt{ssid}_{C}, \mathtt{CLOSE}, \overline{\gamma_i}.\mathsf{id})$
- $m_{19} := (\mathtt{ssid}_{L}, \mathtt{POST}, \overline{\mathsf{tx}^{\mathsf{r}}_i})$
- $m_{20} := (\mathtt{sid}, \mathtt{pid}, \mathtt{REVOKED})$
- $m_{21} := (\mathtt{ssid}_{L}, \mathtt{POST}, \overline{\mathsf{tx}^{\mathsf{p}}_{i-1}})$
- $m_{22} := (\mathtt{sid}, \mathtt{pid}, \mathtt{FORCE\text{-}PAY})$

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_{18} to $\mathcal{F}_{Channel}$ in round τ. If the channel that is sent in m_{18} is closed, U_i sends m_{19} to $\mathcal{G}_{Ledger}$ in round $\tau + t_{\mathsf{c}} + \Delta$. Finally, if the transaction sent in m_{19} appears on $\mathcal{L}$ in $\tau + t_{\mathsf{c}} + 2\Delta$, U_i sends m_{20} to $\mathcal{E}$. The ensemble is $\mathtt{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_{20}[\tau + t_{\mathsf{c}} + 2\Delta]\} \cup \mathsf{obsSet}(m_{18}, \mathcal{F}_{Channel}, \tau) \cup \mathsf{obsSet}(m_{19}, \mathcal{G}_{Ledger}, \tau + t_{\mathsf{c}} + \Delta)$

Ideal world: In every round τ, $\mathcal{F}_{VC}$ performs `RESPOND`, which provides a decision on whether or not to do the following. If yes, $\mathcal{F}_{VC}$ instructs the simulator to perform `REVOKE`, which results in the message m_{18} to $\mathcal{F}_{Channel}$ in round τ. If the channel that is sent in m_{18} is closed, the simulator sends m_{19} to

Table 8: Explanation of the sequence names used in Lemma 6 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_{18}, m_{19}, m_{20}	Carries out the revokation.
FORCE_PAY	Prot.Respond.2	IF.Respond.Revoke Sim.Respond.2	m_{21}, m_{22}	Carries out the force-pay.

$\mathcal{G}_{Ledger}$ in round $\tau + t_c + \Delta$. Finally, if the transaction sent in m_{19} appears on $\mathcal{L}$, $\mathcal{F}_{VC}$ sends m_{20} to $\mathcal{E}$. The ensemble is $\mathtt{EXEC}_{\mathcal{F}_{VC},\mathcal{S},\mathcal{E}} := \{m_{20}[\tau + t_c + 2\Delta]\} \cup \mathsf{obsSet}(m_{18}, \mathcal{F}_{Channel}, \tau) \cup \mathsf{obsSet}(m_{19}, \mathcal{G}_{Ledger}, \tau + t_c + \Delta)$

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_{21} to $\mathcal{G}_{Ledger}$ in round τ and, if the transaction sent in m_{21} appears on $\mathcal{L}$, the message m_{22} to $\mathcal{E}$ in round $\tau + \Delta$. The ensemble is $\mathtt{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_{22}[\tau + \Delta]\} \cup \mathsf{obsSet}(m_{21}, \mathcal{G}_{Ledger}, \tau)$

Ideal world: In every round τ, $\mathcal{F}_{VC}$ performs RESPOND, which provides a decision on whether or not to do the following. If yes, $\mathcal{F}_{VC}$ instructs the simulator to perform FORCE_PAY, which results in the messages m_{21} to $\mathcal{G}_{Ledger}$ in round τ and, if the transaction sent in m_{21} appears on $\mathcal{L}$, the message m_{22} to $\mathcal{E}$ in round $\tau + \Delta$. The ensemble is $\mathtt{EXEC}_{\mathcal{F}_{VC},\mathcal{S},\mathcal{E}} := \{m_{22}[\tau + \Delta]\} \cup \mathsf{obsSet}(m_{21}, \mathcal{G}_{Ledger}, \tau)$

□

Thereom 1. *(again) Let Σ be an EUF-CMA secure signature scheme. Then, for functionalities $\mathcal{G}_{Ledger}$, $\mathcal{G}_{clock}$, $\mathcal{F}_{GDC}$, $\mathcal{F}_{Channel}$ and for any ledger delay $\Delta \in \mathbb{N}$, the protocol Π UC-realizes the ideal functionality $\mathcal{F}_{VC}$.*

This theorem follows directly from Lemma 1, 2, 3, 4, 5 and Lemma 6.

F Discussion on security and privacy goals

We state our security and privacy goals informally in Section 4.1. In this section we formally define these goals as cryptographic games on top of the ideal functionality $\mathcal{F}_{VC}$ described in Appendix E.4 and then show that $\mathcal{F}_{VC}$ fulfills each goal. Due to the same assumptions and similarities in some of the security and privacy goals, parts of this section are taken verbatim from [4].

F.1 Assumptions

For the theorems in this section, we have the following assumptions: (i) stealth addresses achieve unlinkability and (ii) the used routing scheme (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 the 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 [6]. Sphinx [8], for instance, is a realization of this. We use it in Donner, extended with a per-hop payload (see also Section 4.2).

F.2 Balance security

Given a path $\mathsf{channelList} := \gamma_0, \ldots, \gamma_{n-1}$ 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 virtual channel (VC) protocol achieves *balance security* if the $\mathbf{PathBalance}(U)$ for each honest intermediary U does not decrease..

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

The challenger sets $\texttt{sid}$ and $\texttt{pid}$ to two random identifiers. Then, the challenger simulates opening a VC from the OpenVC phase on input $(\texttt{sid}, \texttt{pid}, \texttt{SETUP}, \mathsf{channelList}, \mathsf{tx}^{\mathsf{in}}, \alpha, T, \overline{\gamma_0})$. Every time 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 opening and trigger the offload and thereby refunding the collateral or let them be successful. If the opening was successful, an attacker can instruct the simulator to either perform updates, honestly close the VC or do nothing. In the case of an honest closure, the queries to corrupted users are forwarded to the attacker, who again can let the closure be successful or force an offload.

We say that the adversary wins the game if there exists an honest intermediate user U, such that **PathBalance**(U) is lower after the VC execution.

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

Theorem 2 (Donner achieves balance security)**.** *Donner virtual channel executions achieve balance security as defined in Definition 3.*

Proof. Assume that an adversary exists, can win the balance security game. This means, that after the balance security game, there exists an honest intermediate user U, such **PathBalance**(U) is lower after the VC exeuction.

An intermediary U_i potentially has coins locked up in the state stored in $\mathcal{F}_{Channel}$ with its left neighbor U_{i-1} and its right neighbor U_{i+1}. Depending on if and where an adversary potentially disrupts the VC execution there are amount locked up differs. We analyze below all different cases and show that no honest intermediary U_i exists, such that **PathBalance**(U_i) is lower after the execution.

1. The adversary disrupts the VC execution before it reaches U_i. In this case, U_i has no coins locked up and therefore the balance does not change.

2. The adversary disrupts the VC execution after U_i and U_{i-1} have updated their channel for opening. In this case, U_{i-1} has a non-negative amount of coins locked up with U_i. Regardless of the outcome, the balance of U_i can only increase or stay the same, since the locked up coins come from U_{i-1}.

3. The adversary disrupts the VC execution after U_i and U_{i+1} have updated their channel for opening. In this case, U_{i-1} has a non-negative amount α_{i-1} of coins locked up with U_i. U_i has the same amount (minus a fee) α_i locked up with U_{i+1}.

4. The adversary disrupts the VC execution after U_i and U_{i+1} have updated their channel for closing. In this case, U_{i-1} has a non-negative amount α_{i-1} of coins locked up with U_i. U_i has the smaller amount α'_i locked up with U_{i+1}.

5. The adversary disrupts the VC execution after U_i and U_{i+1} have updated their channel for closing. In this case, U_{i-1} has a non-negative amount α'_{i-1} of coins locked up with U_i. U_i has the same amount (minus a fee) α'_i locked up with U_{i+1}.

To sum up, in all cases the money that U_i locks up U_{i+1} is always either the same or less than what U_{i-1} locks up with U_i. Now in each of these five cases, there are two possible things that can happen. Either $\mathsf{tx}^{\mathsf{vc}}$ is posted before $T-3\Delta-t_{\mathsf{c}}$ or it is not. In the former case, $\mathcal{F}_{VC}$ ensures with the Respond phase, that U_i is refunding itself, thereby keeping a neutral path balance. In the case that $\mathsf{tx}^{\mathsf{vc}}$ is not posted before $T-3\Delta-t_{\mathsf{c}}$, U_i always gets the collateral from U_{i-1} via the Respond phase of $\mathcal{F}_{VC}$, keeping either a neutral or positive path balance. □

F.3 Endpoint security

Intuitively, a VC protocol achieves endpoint security, if the endpoints can either enforce their VC balance on-chain, or, they are compensated with an amount that is at least as large as their VC balance within an agreed upon time. More concretely in our construction, we ensure that the sender can always enforce its VC balance on-chain. For the receiver, we ensure that either the sender puts the VC funding on-chain (allowing the receiver to enforce its balance) or, it gets the full capacity of the VC after the life time T. We extend our definition of **PathBalance**(U) for the sender U_0 and the receiver U_n. For each endpoint, this is the balance that it holds in the VC, if the VC is offloaded or 0, if the VC is not offloaded, plus its respective balance in its channel with its direct neighbor on the path.

Formally, consider the following game. The adversary selects a $\mathsf{channelList}$, a transaction $\mathsf{tx}^{\mathsf{in}}$, a virtual channel capacity α and a channel lifetime T, such that the output of $\mathsf{tx}^{\mathsf{in}}.\mathsf{output}[0]$ holds at least $\alpha + n\cdot\varepsilon$ 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 $\texttt{sid}$ and $\texttt{pid}$ to two random identifiers. Then, the challenger simulates opening a VC from the OpenVC phase on input $(\texttt{sid}, \texttt{pid}, \texttt{SETUP}, \mathsf{channelList}, \mathsf{tx}^{\mathsf{in}}, \alpha, T, \overline{\gamma_0})$. 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 opening and trigger the offload and thereby refunding the collateral or let them be successful. If the opening was successful, an attacker can instruct the simulator to either perform updates, honestly close the VC or do nothing. In the case of an honest closure, the queries to corrupted users are forwarded to the attacker, who again can let the closure be successful or force an offload.

Define x_{U_0} and x_{U_n} as the latest balance of the sender and receiver in the VC, respectively. We say that the adversary wins the game if for an honest sender **PathBalance**(U_0) is lower (by an amount greater than the combined fees $(n-$

$1) \cdot \mathsf{fee})$ after the VC execution or if for an honest receiver, **PathBalance**(U_n) is lower after T, compared balance with their respective neighbors before the VC execution plus x_{U_0} or x_{U_n}, respectively.

Definition 4 (Endpoint security)**.** We say that a VC protocol achieves endpoint security if for every PPT adversary $\mathcal{A}$, the adversary wins the aforementioned game with negligible probability.

Theorem 3 (Donner achieves endpoint security)**.** *Donner virtual channel executions achieve endpoint security as defined in Definition 5.*

Proof. For an honest sender, there are two possible scenarios. Either, $\mathcal{F}_{VC}$ has updated (or registered an update via $\mathcal{S}$ in) the channel between U_0 and U_1 to exactly the final balance α_i' ($= x_{U_0}$ minus fees) in the CloseVC phase before the round $T - 3\Delta - t_c$. Or, if not, $\mathcal{F}_{VC}$ has instructed the simulator to publish $\mathsf{tx}^{\mathsf{vc}}$, allowing the balance to be enforcable on-chain. In both cases, **PathBalance**(U_0) is not lower than its initial balance with U_1 plus x_{U_0} minus the sum of all fees $(n-1) \cdot \mathsf{fee}$.

For an honest receiver, there are also two possible scenarios. Either, the VC was offloaded, allowing U_n to enforce its balance on-chain, or it is not. If VC is not offloaded, U_n either gets the full VC capacity, if the channel with U_{n-1} was not updated in the CloseVC phase or, its actual balance if it was updated in the CloseVC phase. The **PathBalance**(U_n) is therefore not lower. □

F.4 Reliability

Intuitively, we say that a VC protocol achieves reliability, if after successfully opening the VC, no (colluding) malicious intermediaries can force two honest endpoints to close or offload the virtual channel before the lifespan T of the VC expires. Note that in this intuition we write before T, when technically the offloading process has to be initiated some time before, i.e., at time $T - 3\Delta - t_c$.

Formally, consider the following game. The adversary selects a $\mathsf{channelList}$, a transaction $\mathsf{tx}^{\mathsf{in}}$, a virtual channel capacity α and a channel lifetime T, such that the output of $\mathsf{tx}^{\mathsf{in}}.\mathtt{output}[0]$ holds at least $\alpha + n \cdot \varepsilon$ 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 opening a VC from the OpenVC phase on input $(\mathtt{sid}, \mathtt{pid}, \mathtt{SETUP}, \mathsf{channelList}, \mathsf{tx}^{\mathsf{in}}, \alpha, T, \overline{\gamma_0})$. 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 opening and trigger the offload and thereby refunding the collateral or let them be successful. If the opening was successful, an attacker can instruct the simulator to either perform updates, honestly close the VC or do nothing. In the case of an honest closure, the queries to corrupted users are forwarded to the attacker, who again can let the closure be successful or force an offload.

We say that the adversary wins the game if after successfully opening the VC, i.e., the OpenVC and Finalize phases are completed successfully, the VC is offloaded before $T - 3\Delta - t_c$.

Definition 5 (Reliability)**.** We say that a VC protocol achieves reliability if for every PPT adversary $\mathcal{A}$, the adversary wins the aforementioned game with negligible probability.

Theorem 4 (Donner achieves reliability)**.** *Donner virtual channel executions achieve reliability as defined in Definition 5.*

Proof. This follows directly from $\mathcal{F}_{VC}$. Note that after a successful OpenVC and Finalize phase, the only way for a VC to be offloaded is if the close phase is not reaching the sender until time $T - 3\Delta - t_c$. □

F.5 Endpoint anonymity

A VC protocol achieves endpoint anonymity, if it achieves sender anonymity and receiver anonymity. Intuitively, we say that a VC protocol achieves *sender anonymity* 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 by one (or more) intermediaries. For receiver anonymity, an intermediary has to be unable to distinguish that the right neighbor is the receiver from the case that the intermediary and the receiver are separated by 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 VC capacity α as well as a channel life time T such that the output $\mathsf{tx}^{\mathsf{in}}.\mathtt{output}[0]$ holds at least $\alpha + n \cdot \varepsilon$ 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 `sid` and `pid` to two random identifiers. Moreover, the challenger picks a bit b at random and simulates the OpenVC phase on input $(\mathtt{sid}, \mathtt{pid}, \mathtt{SETUP}, \mathsf{channelList}_b, \mathsf{tx}^{\mathsf{in}}, \alpha, T, \overline{\gamma_0})$, followed by the Finalize, Update and CloseVC phases. 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 6 (Sender anonymity)**.** We say that a VC protocol achieves sender anonymity 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 (Donner achieves sender anonymity)**.** *Donner virtual channel executions achieve sender anonymity as defined in Definition 6.*

Proof. The message $(\mathtt{sid}, \mathtt{pid}, \mathtt{open}, \mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_{i+1}, \alpha_i, T, \overline{\gamma_{i-1}}, \overline{\gamma_i}, \theta_{\varepsilon_{i-1}}, \theta_{\varepsilon_i})$ that $\mathcal{F}_{VC}$ sends to the simulator in the OpenVC phase, is leaked to the adversary. By looking at $\overline{\gamma_{i-1}}$, $\overline{\gamma_i}$ 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, $\overline{\gamma_{i-1}}$ 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 $\mathsf{tx}^{\mathsf{vc}}$, rList, $\theta_{\varepsilon_{i-1}}$, θ_{ε_i} and onion_{i+1}. Let us assume, that there exists an adversary that can break sender anonymity. There are two possible cases.

1. The adversary finds out by looking at $\mathsf{tx}^{\mathsf{vc}}$, rList, $\theta_{\varepsilon_{i-1}}$ and θ_{ε_i}. By design, the adversary knows that outputs $\theta_{\varepsilon_{i-1}}$ belongs to its left neighbor U_1 and θ_{ε_i} to itself. We defined that the output, that serves as input for $\mathsf{tx}^{\mathsf{vc}}$, has never been used and is unlinkable to the sender and check this in `checkTxIn`. Looking at the outputs of $\mathsf{tx}^{\mathsf{vc}}$, the adversary knows to whom all but one output belongs. Since our adversary breaks the sender anonymity, it needs to be able to reconstruct, to whom this final output of $\mathsf{tx}^{\mathsf{vc}}$ 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 anonymity, 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 anonymity 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 anonymity and its proof are analogous to the sender anonymity. □

F.6 Path privacy

Intuitively, we say that a VC 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 VC capacity α as well as a life time T such that the output $\mathsf{tx}^{\mathsf{in}}.\mathsf{output}[0]$ holds at least $\alpha + n\cdot\varepsilon$ 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 7 (Path privacy)**.** We say that a VC 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 6 (Donner achieves path privacy)**.** *Donner virtual channel executions achieve path privacy as defined in Definition 7.*

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

F.7 Value privacy

Intuitively, a VC protocol achieves value privacy, if no intermediaries gains information about the VC payments of two honest endpoints other than the opening and closing balances of each endpoint. In particular, no intermediary learns about number of transactions being exchanged and their amount. Formally, consider the following game. The adversary selects a $\mathsf{channelList}$, a transaction $\mathsf{tx}^{\mathsf{in}}$, a virtual channel capacity α and a channel lifetime T such that the output $\mathsf{tx}^{\mathsf{in}}.\mathsf{output}[0]$ holds at least $\alpha + n\cdot\varepsilon$ 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 random identifiers and simulates the opening of the virtual channel for the given parameters, forwarding queries that a corrupted intermediary would receive to the adversary. After the VC has been opened successfully, we denote the current round in the simulation as τ the challengers asks the adversary to select two list of payments p_0 and p_1 with a length in range $[0,k]$, containing VC payments between the endpoints and their order. k denotes the maximum number of transactions that are possible within the time period between τ and when the VC needs to be honestly closed. The adversary can select arbitrary payments in an arbitrary direction with an amount between 0 and the balance of the respective sending user at the time the payment is performed. Additionally, performing either list of payments has to result in the same end balance, to avoid trivial distinction by looking at the final balance. That is, U_0's final balance is $\alpha - \alpha'$ and U_n's final balance is α', with $0 \leq \alpha' \leq \alpha$. The adversary sends p_0 and p_1 to the challenger.

The challenger picks a random bit $b \in \{0,1\}$, and then performs the payments specified in p_b. After the payments, the challenger initiates the honest closing such, that if successful, the closing will be completed 1 round before $T - t_{\mathsf{c}} - 3\Delta$, forwarding queries to corrupted intermediaries again to the adversary. This gives the chance to the adversary, to let either VC close honestly or force to offload.

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

Definition 8 (Value privacy)**.** We say that a VC protocol achieves path value 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 7 (Donner achieves path privacy)**.** *Donner virtual channel executions achieve value privacy as defined in Definition 8.*

Proof. This property follows directly from $\mathcal{F}_{VC}$ and $\mathcal{F}_{Channel}$. The only information regarding the VC updates are sent by either VC endpoint to $\mathcal{F}_{VC}$ (in the Update phase) and forwarded to $\mathcal{F}_{Channel}$, other than that, the two simulations of the challenger are identical. The adversary sees only the messages that the challenger forwards to the corrupted intermediaries, which means that the adversary knows neither about the content nor the existence of these VC update messages in both scenarios. Additionally, the functionality $\mathcal{F}_{Channel}$ does not expose the internal state of a channel to anyone but the two users of it, in the case of the VC, the two endpoints.

The adversary has two options, either letting the VC close honestly or, forcing the VC to offload. In the former case, the adversary will see only the final balance α' being forwarded in the close request. In the latter case, the adversary will learn about the final balance in the VC, after it is offloaded and it is closed. It follows, that an adversary cannot guess b correctly with a probability better than $1/2 + \varepsilon$, where ε denotes a negligible value. $\square$