Breaking And Fixing UTXO-Based Virtual Channels

Abstract—**Payment channel networks (PCNs) mitigate the scalability issues of current decentralized cryptocurrencies. They allow for arbitrarily many payments between users connected through a path of intermediate payment channels while requiring interacting with the blockchain only to open and close the channels. Unfortunately, PCNs are (i) tailored to payments, excluding more complex smart contract functionalities, such as the oracle-enabling Discreet Log Contracts and (ii) their need for active participation from intermediaries may make payments unreliable, slower, expensive, and privacy-invasive. Virtual channels are among the most promising techniques to mitigate these issues, by allowing the two endpoints of a path to create a direct channel over the intermediaries without any interaction with the blockchain. After such a virtual channel is constructed, (i) the endpoints can use this direct channel for applications other than payments and (ii) the intermediaries are no longer involved in updates.**

In this work, we first introduce the Domino attack, a new attack that leverages virtual channels to destruct the PCN itself and is inherent to the design adopted by the existing UTXO-based virtual channels. We then demonstrate its severity by a quantitative analysis on a snapshot of the Lightning Network (LN), the most widely deployed PCN at present. We finally discuss other serious drawbacks of existing virtual channel designs, such as the support for only a single intermediary, a latency and blockchain overhead linear in the path length, or a non-constant storage overhead per user.

We then present Donner, the first virtual channel construction that overcomes the shortcomings above, by relying on a novel design paradigm. We formally define and prove security and privacy properties in the Universal Composability framework. Our evaluation shows that Donner is efficient, reduces the on-chain number of transactions for disputes from linear in the path length to a single one, which is the key to prevent Domino attacks, and reduces the storage overhead from logarithmic in the path length to constant. Donner is Bitcoin-compatible and can be easily integrated in the LN.

1. Introduction

The permissionless nature of the consensus algorithm that governs most of today's cryptocurrencies heavily limits their transaction throughput (e.g., a few transactions per second in Bitcoin). Payment channels (PCs) have emerged as one of the most promising scalability solutions, with the Lightning Network [30] being the most popular realization thereof in Bitcoin. A PC enables arbitrarily many payments between two users while requiring to commit only two transactions to the ledger: one to open and another to close the channel. Aside from payments, several applications proposed so far benefit from the scalability gains of 2-party PCs [5, 9, 15]. Recent work [3] has further shown how to lift any operation supported by the underlying blockchain to the off-chain setting, thereby further expanding the class of supported off-chain applications.

Creating a PC between any pair of users (i.e., a clique) is, however, economically infeasible since PCs are funded on-chain and users must lock coins for each PC. Creating PCs on-demand with any potential partner a user wishes to interact with is infeasible too, since (i) it is costly (on-chain fees for both the opening and the closing transaction), (ii) it requires waiting for transaction confirmation on the blockchain (around 1h in Bitcoin), and (iii) it is in contrast to the goal of achieving scalability, as it requires on-chain transactions for opening and closing each channel, which again negatively impacts the blockchain throughput. Therefore, single PCs are instead linked together to form a payment channel network (PCN), leveraging paths of PCs to connect two users rather than opening a PC between them. We can classify the interactions of users within a PCN in synchronization protocols and virtual channels (VCs).

Synchronization protocols [1, 19, 26, 27, 28, 30] allow a sender to pay a receiver when they are connected by a path of PCs, atomically updating the balance of all PCs along the path. Although some of these synchronization protocols are deployed in practice (e.g., for multi-hop payments in the Lightning Network), there are several drawbacks: (i, *online assumption*) they require users in the path to be online; (ii, *reliability*) each intermediate user must participate, making the payment less reliable; (iii, *cost*) each intermediate charges a fee per synchronization round; (iv, *latency*) the latency of the application suffers from the number of intermediaries (e.g., in the Lightning Network up to one day latency per channel); (v, *privacy*) intermediaries are aware of every single operation; and (vi, *efficiency*) they can handle only a limited number of simultaneous payments (e.g., 483 in the Lightning Network) [8]. Finally, and perhaps more importantly, current synchronization protocols are tailored to payments. Supporting 2-party applications (as the ones mentioned before) would require thus to come up with a synchronization protocol for each application. Apart from being a burden, it is not trivial to design such protocols tailored to applications beyond payments, as exemplified by the recent quest in the Bitcoin community about the realization of Discreet Log Contracts across multiple hops [13].

Virtual channels [17] constitute the most promising solution to perform repeated transactions or other, non-payment, applications (e.g., [5, 9, 15]) between any pair of users connected by a path of PCs. In essence, a virtual channel (VC) is akin to a PC, but instead of opening it by one on-chain transaction, it is opened off-chain using funds from existing PCs. To explain the basic functionality of VCs, let us initially assume that Alice and Bob are connected by a single intermediary Ingrid. First, Ingrid must collaborate with Alice and Bob to execute a synchronization protocol that coordinates the update of both PCs to fund the virtual channel, i.e., a collateral is locked at both PCs, i.e., (Alice, Ingrid) as well as (Ingrid, Bob), for this purpose. This collateral ensures that honest users do not lose funds; a user gets her collateral back only if she does not deviate from the protocol, else it used to compensate honest users (cf. Section 2.4). However, once the VC is created, Alice and Bob can use the VC without the involvement of Ingrid.

In this manner, VCs overcome the aforementioned drawbacks of synchronization protocols: (i) Ingrid is no longer required to be online; (ii) the reliability of the channel does not depend on Ingrid; (iii) Ingrid does not charge a fee for each usage of the channel (perhaps only once to create and close the VC); (iv) the latency does not depend on Ingrid; (v) Ingrid does not learn each single VC update; (vi) they can host one or even multiple VCs instead of payments which in turn can handle 483 payments or potentially more VCs, bypassing the limitation on the number of payments.

Moreover, applications built on top of PCs can be smoothly lifted to VCs, since a VC is essentially a PC "on top of other PCs", which constitutes a crucial improvement over synchronization protocols.[1] For instance, VCs support Discreet Log Contracts [15], an application that has received increased attention lately and that intuitively allows for bets based on attestations from an oracle on real world events.

As compared to PCs, VCs offer the same advantages while requiring no on-chain transaction for their setup, thereby dispensing from the associated blockchain delays, on-chain fees, and on-chain footprints. This makes it possible to keep VCs short-lived, to frequently close, open, or extend them based on current needs, and to avoid on-chain leakages. For a more detailed discussion see Appendix A.

State of the art on VCs Dziembowski et al. [17] proposed the first construction of VCs over a single intermediary. Recursive constructions [18] followed up allowing for creating VCs on top of other VCs (or a pair composed of a VC and a PC), thereby supporting arbitrarily many intermediaries. Dziembowski et al. [16, 29] further extended the expressiveness of VCs proposing the notion of multi-party VCs, where a set of n participants build an n-party channel recursively from their pair-wise payment/virtual channels. Unfortunately, all the aforementioned constructions rely on the expressiveness of Turing-complete scripting languages such as that of Ethereum and are based on the account model instead of Unspent Transaction Output (UTXO) model; thus, they are incompatible with many of the cryptocurrencies available today, including Bitcoin itself. Aumayr et al. [2] have later shown how to design a Bitcoin-compatible VC through a carefully crafted cryptographic protocol in the UTXO model, supporting however only one intermediary.

TABLE 1: Comparison to other multi-hop VC schemes.

	LVPC [21]	Elmo [24]	Donner
Scripting requirements	Bitcoin	Bitcoin + ANYPREVOUT	Bitcoin
Multi-hop	✓	✓	✓
Secure against Domino attack	✗	✗	✓
Path privacy	✗	✗	yes
Time-based fee model	✓	✗	✓
Unlimited lifetime	✗	✓	✓
Storage Overhead per party	$\Theta(n)$*	$\Theta(n^3)$	$\Theta(1)$
Off-chain closing	✓	✗	✓
Offload: txs on-chain	$\Theta(n)$	$\Theta(n)$	1
Offload: time delay	$\Theta(n)$*	$\Theta(n)$*	1

* by synchronizing all channels, this time can be only $\Theta(\log(n))$.

Jourenko et al. [21] have recently introduced the first Bitcoin-compatible construction over multiple intermediaries, called Lightweight Virtual Payment Channels (LVPC), where a VC over one hop is applied recursively to achieve a VC between two users separated by a path of any length. More recently, Kiayias and Litos have introduced Elmo [24], a VC construction that does not rely on creating intermediate VCs, by instead relying on scripting functionalty not present in Bitcoin, i.e., the opcode ANYPREVOUT.

All of these UTXO-based VC constructions share one common design pattern: The VC is funded from all underlying PCs. We refer to this design pattern as *rooted VCs* and illustrate it on a high level in Figure 1(a.1). Because VCs are, unlike PCs, not funded on-chain, they rely on an operation called *offloading*, which transforms a VC to a PC. This is important for honest users so they can enforce their balance in case the other user misbehaves: first transforming the VC to a PC by putting the VC funding on-chain, and second using the means provided by the PC to enforce their balance. Rooted designs enable both endpoints to offload the VC, but because they are funded by all underlying PCs, every underlying PC has to be closed on-chain (see Figure 1(a.2)).

Main idea of this work We show that rooted VCs are by design prone to severe drawbacks including the *Domino attack* (see Section 3), a new attack in which (i) a malicious intermediary of a VC or (ii) an attacker establishing a VC with itself over a number of honest PCs can close the whole path of underlying PCs and bring them on-chain. Not only are all existing UTXO-based VC constructions affected by this attack, but this attack is so severe that it can potentially shut down the underlying PCN, as we show in Section 3.3. As a result, we argue that none of the existing UTXO-based VC constructions should be deployed in practice. Furthermore, the rooted design allows adversaries to learn the identity of participants other than their direct neighbors, thereby breaking what we call *path privacy* (see Section 3.4). Given these security and privacy shortcomings, we introduce a paradigm shift towards the design of *non-rooted* VCs, based on two fundamental ingredients.

First, instead of being rooted, the VC is funded independently from the underlying PCs, by one of the VC endpoints. The underlying PCs are used to lock up some funds (or collateral) that are paid to the honest VC endpoint if the

1. VCs expose all the functionalities of a PC and can be used interchangeably as a building block for off-chain applications, see Section 5.

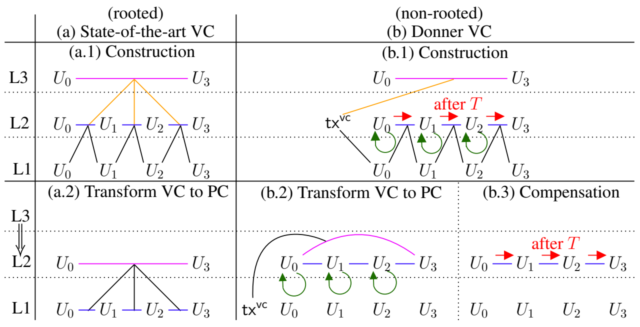


Figure 1: Conceptual comparison of (a) state-of-the-art VCs (rooted) and (b) our scheme (non-rooted) on layers L1 (blockchain), L2 (payment channels) and L3 (virtual channels). Note that the VC in (a.1) is funded by all the underlying channels In (b.1), the VC is funded only by U_0, indirectly via a transaction $\mathsf{tx}^{\mathsf{vc}}$. Additionally in (b.1), a payment is set up from U_0 to U_3, whose outcome depends on whether or not the VC is offloaded. Offloading, i.e., the act of forcefully transforming a VC (L3) to a PC (L2) in (a.2), requires that all the underlying PCs (L2) are put on-chain (L1). In (b.2), offloading the VC keeps the PCs open, posting only $\mathsf{tx}^{\mathsf{vc}}$ on-chain (L1). Since offloading enables U_3 to receive their funds, the payment is refunded then. However, since in (b), only U_0 can offload, U_3 is compensated (b.3) after a timeout T via a payment that is executed iff U_0 has not offloaded the VC (i.e., (b.2) did not happen).

other VC endpoint misbehaves. We illustrate this concept on a high level in Figure 1(b.1). In contrast to rooted designs, VCs can be offloaded without having to close the underlying PCs, which is the key to prevent Domino attacks. Since the VC is only funded by one endpoint, only this funding endpoint has the means of transforming the VC to a PC (offload). Subsequently, the other one cannot get their money via offloading in case of misbehavior. This issue is solved by compensating the non-funding endpoint in case the funding endpoint has not transformed the PC to a VC within a channel lifetime T, see Figure 1(b.2) and (b.3).

This lifetime T is the second crucial aspect where we depart from the state of the art. Current solutions provide unlimited lifetime without guaranteeing however that the VC will remain open, as an intermediary node could initiate the offloading. Instead, our design ensures that the VC is open until time T, which can be repeatedly prolonged if all involved parties agree. This allows intermediaries to charge fees based on the lifetime of the VC, which corresponds to the time they have to lock up their funds, something that is not possible in current VC solutions with unlimited lifetime [24]. The improvements over existing UTXO-based multi-hop VC constructions are summarized in Table 1. We compare with single-hop constructions and with those relying on Turing-complete smart contracts in Table 5.

Our contributions can be summarized as follows:

- We introduce the Domino attack, which allows the adversary to close arbitrarily many PCs of honest users, thereby destructing the underlying PCN. We argue that any rooted construction, in particular, all existing UTXO-based VC constructions are prone to this attack. We show the severity of this attack in a quantitative analysis; given current BTC transaction fees, it suffices for an attacker to spend 1 BTC to close every channel in the current LN.
- We present Donner, a new VC protocol that departs from the rooted paradigm by funding the VC from outside of the underlying PC path. In addition to being secure against the Domino attack, it significantly improves in terms of efficiency and interoperability over state-of-the-art VC protocols (see Table 1).
- We introduce the notion of *atomic modification*, a novel subroutine allowing parties to atomically change the value or timeout of a synchronization protocol, which we consider a contribution of independent interest. Atomic modification, non-rooted funding, and the *pay-or-revoke* paradigm [1] are the core building blocks of Donner.
- We conduct a formal security and privacy analysis of Donner in the Universal Composability framework.
- We conduct experimental evaluations to quantify the severity of the Domino attack and demonstrate that Donner requires significantly less transactions than state-of-the-art VCs; Donner decreases the on-chain costs for offloading VCs from linear in the path length to a single one and the storage overhead per PC from linear or logarithmic in LVPC [21] (depending on how the VC is constructed) or cubic in Elmo [24] to constant.

2. Background and notation

2.1. UTXO based blockchains

We adopt the notation for UTXO-based blockchains from [3], 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 and $\theta.\phi$ defines the condition under which the coins can be spent. The latter is done by encoding such a condition in the scripting language of the underlying blockchain. This can range from simple ownership, specifying which public key can spend the output, to more complex conditions (e.g., timelocks, 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 $\mathsf{tx.output} := (\theta_1, ..., \theta_n)$ 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 list. 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, with a delay of Δ. A transaction is valid if and only if (i) all its inputs exist and are not spent by other transaction on $\mathcal{L}$; (ii) it 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.

There are several conditions under which coins can be spent. Usually they consist of a signature that verifies w.r.t. one or more public keys, which we denote as $\mathsf{OneSig}(\mathsf{pk})$ or $\mathsf{MultiSig}(\mathsf{pk}_1, \mathsf{pk}_2, ...)$. Additional conditions 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 $\phi = \phi_1 \vee ... \vee \phi_n$. A conjunction of subconditions is simply written as $\phi = \phi_1 \wedge ... \wedge \phi_n$.

To visualize how transactions are used in a protocol, we use transaction charts. 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. Outgoing arrows show how outputs can be spent. Transactions that are on-chain have a double border (see, e.g., Figure 12 in Appendix D.1).

2.2. Payment channels

Two users can utilize a payment channel (PC) 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 PC operation: *open*, *update* and *close*. First, 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 new transactions called *state* or tx^{s} which assign each of them a balance. Once the funding transaction is on the ledger, the users can exchange arbitrarily many new states (balance updates) in an off-chain manner, 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 PCs in a black-box manner and refer the reader to [3, 26, 27] for more details. We abstract away from the implementation details and instead model the state of the channel as the outputs contained in a transaction tx^{s}, which is kept off-chain. 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, it is possible that a dishonest user publishes a stale state of the channel and current constructions come with a way to handle this case (e.g., through a punishment mechanism that compensates the honest user [3]). We illustrate this abstraction in Figure 2.

2.3. Payment channel networks

A payment-channel network (PCN) [26] is a graph where the nodes represent the users and the edges represent the PCs. The Lightning Network [30] is the state-of-the-art in both PCs and PCNs for Bitcoin, and the largest PCN in terms of coins locked within its channel fundings, currently having around 81k channels, 19k active nodes and a total capacity of 3k BTC (around 130M USD).

In a PCN, any two users connected by 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 $\{U_i\}_{i\in[1,n-1]}$, such that any pair of neighbors U_j and U_{j+1} have a channel $\overline{\gamma_j}$, for $j \in [0, n-1]$. A mechanism synchronizing all channels on the path is required for a payment, such that each channel is updated to represent the fact that α coins moved from left to right. We give an example in Figure 13 in Appendix D.2.

There exist many different MHP schemes. In Blitz [1], the PCs on the path are synchronized by connecting them to a single event, a transaction called $\mathsf{tx}^{\mathsf{er}}$, which is created and can be posted by the sender. If posted within some predefined time, the payment in each channel is refunded. Else, the payment is automatically successful.

2.4. State-of-the-art virtual channels

A virtual channel (VC) allows two users to establish a direct channel, without putting any transaction on-chain. Indeed, the fundamental difference between a PC and a VC is that in a VC, the funding transaction tx^{f} does not go on-chain in the honest case. To still ensure that users do not lose their funds in case of dispute this requires a new operation: In addition to the three operations *open*, *update* and *close* of PCs, we need the operation *offload*, which allows a user of the VC to put the funding transaction tx^{f} on-chain, transforming the VC into a PC.

To realize the operation *offload*, VCs are constructed over a path of PCs connecting the two users or endpoints of the VC. To *open* the VC, the intermediaries controlling the *underlying* PCs or base channels on the path iteratively lock up some collateral, taking it from the respective PCs,

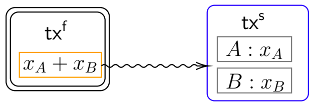


Figure 2: We abstract PCs using a squiggly line to hide details that are not needed in this work. $P : x_P$ indicates that user P owns x_P coins in the state tx^{s}. The box containing $x_A + x_B$ indicates the shared output of A and B.

and the funding of the VC and the collateral is connected in a construction-specific way. The *update* of the VC requires no interaction of the intermediaries. Finally, to *close* the VC, the final balance of the VC has to be mapped into the base channels so that in the end both VC endpoints receive the latest balance of the VC and the intermediaries do not lose coins. Note that with the exception of *offload*, which requires *at least* one on-chain transaction (i.e., the funding), all other operations require no on-chain transaction.

We depict the behaviour of a VC between users U_0 and U_2 via U_1 according to state-of-the-art constructions in Figure 3. Note that the funding transaction tx^{f} of the VC spends outputs coming from both underlying PCs, something we call a *rooted* VC. The transaction $\mathsf{tx}^{\mathsf{punish}}$ abstracts a punishment mechanism that is in place to ensure that the endpoints can unilaterally offload the VC. In a nutshell, it forces the intermediary U_1 to close the PC in which the offloading endpoint is not involved, thereby making tx^{f} spendable. This concept shown for one intermediary can be used to construct a tree-like structure over a path of arbitrary intermediaries to get VCs of arbitrary length. We show this concept in Figures 4, 14 and 15.

3. The Domino attack

3.1. Key ideas of the attack

Observation 1: Balance security for VC endpoints Independently of its inner workings, any VC construction must ensure that honest VC endpoints Alice and Bob can cash out the coins they hold in the VC (i.e., get their coins on-chain). As discussed in Section 2.4, VCs are akin to payment channels (PCs), with the difference of having their funding transaction off-chain. This means that both endpoints can no longer directly claim their latest balance as in a PC. Instead, the VC funding transaction first needs to be put on-chain through the operation offload.

Existing VC schemes achieve balance security by allowing both endpoints to perform the offload operation, i.e., the possibility to start a protocol (possibly involving the intermediaries) that will lead to the VC funding transaction

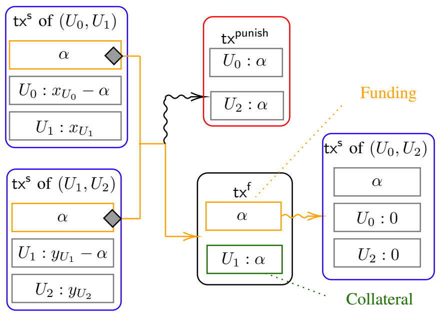


Figure 3: Illustration of a VC construction over a single intermediary. The squiggly line indicates an abstraction of details irrelevant to this work.

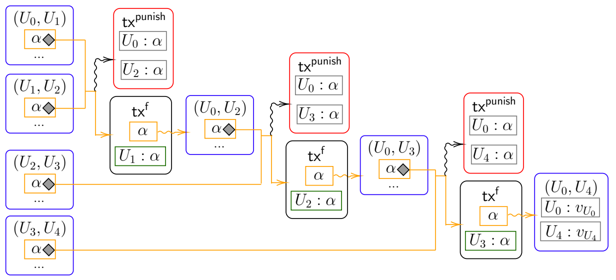


Figure 4: Illustration of a rooted VC via multiple hops. The yellow lines indicate how the VC is rooted. We abbreviate "tx^{s} of (A, B)" with "(A, B)". The yellow visualize the flow of the funding and connect to all PCs that need to be closed in the case the rightmost VC is offloaded.

going on-chain (or get compensated otherwise through a punishment mechanism). We note that in some of the existing VC schemes [24], even the intermediaries are given the possibility to initiate a forceful offloading sequence.

Observation 2: VC funding transaction is rooted in all underlying base channels To enable the offload operation, the VC funding takes as inputs (either directly or indirectly, via intermediate transactions) outputs of each of the underlying base channels. We denote such a VC as being *rooted* in the base channels.[2] We again point to Figure 4 for an illustration of what we mean by rooted. To ensure that the funding of the VC between U_0 and U_4 actually ends up on-chain when initiated by an endpoint, existing constructions employ a *punishment* mechanism that forces intermediaries to close their channels, thereby enabling the VC funding to go on-chain too. At a first glance, this seems the most natural approach since it allows both endpoints to offload the VC and the intermediaries to unlock their collateral. However, a rooted funding implies that it can be posted on-chain *if and only if all underlying PCs are closed*. This feature is the source of the Domino attack, as shown next.

3.2. Attack description

The *Domino attack* follows directly from the two observations mentioned above and can proceed in the following three phases: (i) an adversary controlling two nodes opens two PCs encasing a path of victim channels; (ii) the adversary opens a VC to herself via these victim paths; and (iii) she initiates the offloading of the VC.

The effect of this attack is to force the closure of every channel on this path, i.e., the two the attacker created and the channels on the victim path. Anyone not closing their channel risks losing their money. In stark contrast to payment protocols in PCNs such as Lightning or Blitz where closing one channel in the payment path still allows channels in the rest of the path to remain open, in current VC constructions there is no way that honest nodes can settle

2. By base channel we mean either a PC or a VC that was used for opening a VC, to capture the fact that VCs can be constructed recursively.

their channels honestly off-chain and keep them open. They are forced to close every channel, as the VC funding can only exist on-chain if all base channels are closed.

Example Assume an attacker controlling nodes U_0 and U_4 who wants to perform a Domino attack on the victim path U_1, U_2 and U_3, see Figure 4. If not already opened, the attacker opens the channels (U_0, U_1) and (U_3, U_4). Then, she constructs a VC between her own nodes U_0 and U_4 recursively, as, e.g., established in the LVPC protocol [21]. After the attacker is done with this step, the transaction structure among different users is as in Figure 4. At this point, the attacker can proceed to unilaterally force the closure of all underlying channels, that is, the PCs (U_0, U_1), (U_1, U_2), (U_2, U_3) and (U_3, U_4) as well as the intermediate VCs (U_0, U_2), (U_0, U_3) and the offloading of (U_0, U_4).

First, U_4 closes the PC (U_3, U_4), which she can do on her own. In the rooted VC example of Figure 4 (e.g., this could be LVPC), the output in the state of (U_3, U_4) which is used to fund the VC (U_0, U_4) goes to U_4, *unless* it is first consumed by the VC. This means that an honest U_3 will lose money in the channel (U_3, U_4) to U_4 by means of $\mathsf{tx}^{\mathsf{punish}}$ (dubbed *Punish* transaction in the LVPC protocol), unless she closes the channel (U_0, U_3) and claims its money by posting tx^f, i.e., the transaction funding the VC (i.e., offloading) (U_0, U_4), dubbed *Merge* transaction in LVPC.

However, to post tx^f for (U_0, U_4), U_3 first needs close (U_0, U_3). U_3 initiates the offloading by first closing (U_2, U_3). This triggers a similar response from U_2, who is now at risk of losing the coins in (U_2, U_3), unless she offloads (U_0, U_3) by putting the corresponding tx^f. But to do that, U_2 first needs to close (U_0, U_2). This is done, finally, by closing (U_1, U_2), which forces U_1 to close also (U_0, U_1).

In the end, all channels are closed (as shown in Figure 9 in Appendix C). Let us clarify that by closing the underlying channels we mean that at least two transactions per channel have to be put on-chain, one for closing the channel and another one to spend the collateral locked for the VC. Due to the fact that LVPC first splits the channel into two subchannels before using one of them to fund the VC, closing the initial channel simultaneously spawns a new channel (i.e., the remaining subchannel) that has a capacity reduced by the amount put in the collateral funding the VC. The Domino attack works regardless of how the recursion was applied, as well as on Elmo [24]. In LVPC some (U_3 in the example above) and in Elmo all intermediaries can carry out this attack. The Domino attack can also be launched if the attacker controls only one of the endpoints, assuming the other one agrees to open a VC with her over the victim path. We remark that LVPC and Elmo are modelled in the UC framework, however, their ideal functionality explicitly allow for the Domino attack.

3.3. Quantitative analysis of the Domino attack

To quantify the severity of the Domino attack, we perform the following simulation. We take a current (January 2022) snapshot of the Lightning Network [25]. We assume that an adversary has a certain budget to spend on fees for establishing channels over the network. We take a current average fee of 0.000037 BTC (2.13 USD) per transaction [6]. As in the example before, the adversary, knowing the (public) topology of the network, proceeds to open two PCs to encase a path of victim channels. Then, the adversary constructs VCs of a length of up to $n \in [2, 11]$ to herself, i.e., the adversary is the first and last node. We set the maximum VC length to n to 11, as this is the diameter of the LN snapshot, which means that with this length, every pair of nodes can reach one another. Finally, the adversary forces the closure of all underlying channels. We measure the number of channels that the adversary can force to be closed (excluding the ones where the adversary is a part of) and the money in terms of on-chain fees the adversary causes honest users to pay.

To carry out this attack, the adversary needs to post 3 on-chain transactions per VC with the associated fees, two for establishing the two PCs, and one to close one of these channels. Further, for the VC itself, a certain minimum amount is needed to open it, similar to LN payments. These are coins that the adversary needs to lock in addition. However, since this amount is presumably not only very small, but also the adversary gets it back, we omit it in our simulation and say instead that the adversary performs this attack in sequence. Finally, we note that the effect of this attack is likely to be even more severe in reality, since in existing VC constructions, not only does the channel need to be closed, but subsequent transactions making up the rooted funding of the VC need to be posted as well.

We present our results in Figure 5. Using only 1 BTC for fees, the adversary can close 81k honest channels, which is slightly more than all channels in our LN snapshot (80.9k), and cause a cost of at least 6 BTC to the involved nodes. Since the topology of all public PCs is known, the adversary can also target specific channels to close them. Budgets in the order of 0.2, 0.5, and 1 BTC are not unrealistic, as there are 1501, 799, and 453 nodes, respectively, holding this money within the LN, assuming equal balance distribution in the channels, i.e., 0.5% of nodes in the LN have enough balance to shut down the whole network. If we consider all Bitcoin addresses (even outside the LN), there exist 815k addresses owning 1 BTC or more [7].

The effectiveness of this attack increases with the length of the path underlying the VC, i.e., the longer the path

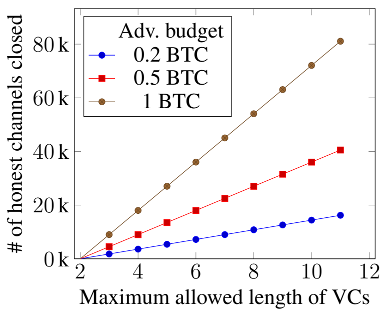


Figure 5: Simulated effect of the Domino attack.

between the two VC endpoints in the PCN, the fewer coins an attacker has to spend to forcefully close base channels. This is especially problematic, since the attacker as an endpoint can actively choose the length of the underlying path. Note that in our analysis we assumed an hypothetical countermeasure based on imposing an upper bound on the length of the VC (i.e., 11). Without this restriction, attackers could shut down hundreds of channels in one stroke and the attack would be even more devastating. We note, however, that it is unclear if such a countermeasure can even be deployed, as the length of the VC is not necessarily leaked.

To conclude, this attack is too severe for the adaption of current VC solutions in PCNs such as the LN. In order to make VCs usable in practice, it is essential to prevent the Domino attack.

3.4. More drawbacks of current VC constructions

Unlimited lifetime Existing VC constructions such as Elmo [24] offer VCs with an a priori unlimited lifetime. On a high level, unlimited lifetime of a VC means that if every party agrees (including endpoints and intermediaries), the VC can remain open potentially forever. While existing work highlights unlimited lifetime as a desirable feature for both PCs and VCs, we view it as a drawback in the context of VCs. Indeed, there is an important difference between VCs and PCs: in a VC funds are locked up not only by the endpoints, but also by the intermediaries of the underlying path. Without a lifetime, intermediaries could have their collateral locked up forever, unless they decide to go on-chain, which however forces them to close their PCs. Related to that, intermediaries should charge a fee proportional to the collateral and the time this collateral is locked (analogously to the LN): without a lifetime, the second parameter cannot be estimated nor enforced without closing the base PCs.

We therefore propose a new approach: instead of having an a priori unlimited lifetime, we fix a certain lifetime at the point of creation. When this lifetime expires, users have the option to prolong it for another fixed lifetime if everyone agrees or to close it. Prolonging it means that the VC remains active and any applications hosted on top of can be kept on being used smoothly. In addition, every intermediary can charge a lifetime-based fee every time they prolong the VC. While all agree, they can repeat this process indefinitely. If one party wants to stop it, the party can unlock their funds *without having to close any channel on-chain*. We explain this concept in more detail in Section 4.

Recursiveness The last issue we point out comes from how the VC funding is rooted in the underlying channels. In current VC constructions, the VC funding is built by recursively combining two channels at a time, forming a tree with the VC funding transaction being the root of the tree and the underyling channels being the leaves. This has two negative implications. First, in addition to closing all PCs (which requires at least one on-chain transaction per channel), i.e., the leaves of the tree, a linear number of transactions needs to go on-chain in order to offload a channel, i.e., the non-leaf nodes of the tree. Second, depending on how the recursiveness has been applied, the time it takes to offload a VC is also either linear (in case of an unbalanced tree, cf. Figure 14 in Appendix E.1) or logarithmic (in case of a balanced tree, cf. Figure 15 in Appendix E.1) in the number of underlying channels. Our construction does not suffer from this issue as elaborated in the next section.

Lack of path privacy State-of-the-art VC constructions create the rooted funding by connecting outputs of pairs of channels in a recursive way. However, this requires interaction of some intermediaries with more than their direct neighbors on the path. In our construction, intermediaries on the path only learn about their direct neighbors in the honest case, exactly as in the Lightning Network.

4. Donner: Key ideas

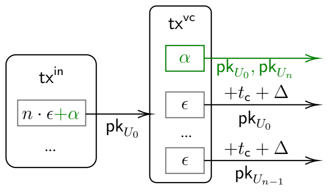
Getting rid of the Domino attack We recall the causes for the Domino attack: (i) the VC funding has to be enforceable on-chain by offloading and (ii) the VC funding is rooted in all underlying PCs. To prevent the attack, we need to get rid of either one. In this work, we choose to remove (ii): investigating whether it is possible to design a construction that removes (i) is interesting and left as future work.

More specifically, *we detach the VC funding transaction from the underlying PCs*. The idea is to fund (off-chain) the VC by an output that is under the control of Alice, but not bound to any channel. This approach has the positive side-effect that intermediaries are no longer able to force a VC to be offloaded. However, this also introduces some new issues. More specifically, if only Alice has the power to offload the VC, we need an alternative mechanism to ensure balance security for Bob.

Adding a compensation for Bob We solve this issue by setting up a conditional collateral payment on the underlying PCN from Alice to Bob through the intermediaries. The condition ruling such a collateral payment is as follows: *if the VC is offloaded before some pre-defined time T (i.e., its lifetime), the collateral payment is refunded. Otherwise, if Alice does not offload the VC before T, Bob receives the collateral payment.*

The amount of this payment is set to be the full capacity of the VC, which means that even in the extreme case of Bob owning all coins in the VC, he is fully compensated. In other words, Bob is safe as he will get his money (or more) in either case (i.e., Alice offloading the VC or not). Alice is also safe, since she can offload the VC before T by herself. The challenging part now is to connect the funding transaction to the collateral payment in a way that these two cases are mutually exclusive. One natural approach would be to root the output of Alice again to all the channels used for the collateral payment, but this approach would enable the Domino attack, as we explain in Section 3.

Leveraging Blitz It turns out that the payment scheme Blitz [1] encodes exactly the functionality we need for this. Blitz relies on a special $\mathsf{tx}^{\mathsf{er}}$ transaction that is used to synchronize the outcome of a payment in a way that if



$\mathsf{tx}^{\mathsf{in}}$
$n \cdot \epsilon + \alpha$
pk_{U_0}
$\mathsf{tx}^{\mathsf{vc}}$
α
$\mathsf{pk}_{U_0}, \mathsf{pk}_{U_n}$
ϵ
$+t_c + \Delta$
pk_{U_0}
ϵ
$+t_c + \Delta$
$\mathsf{pk}_{U_{n-1}}$

Figure 6: Transaction $\mathsf{tx}^{\mathsf{vc}}$ spending from an output under U_0's control, funding the VC (green output, difference from $\mathsf{tx}^{\mathsf{er}}$ in [1]) and linking to the collateral in each channel.

$\mathsf{tx}^{\mathsf{er}}$ is published, then the payment is refunded, otherwise the payment is successful. We make use of this scheme, changing only the transaction used for synchronization to fit our needs, i.e., augmenting it to hold the funding of our VC. We denote this transaction as $\mathsf{tx}^{\mathsf{vc}}$ and show it in Figure 6, highlighting the difference to $\mathsf{tx}^{\mathsf{er}}$.

The protocol discussed so far can be summarized as follows. Alice constructs off-chain the transaction $\mathsf{tx}^{\mathsf{vc}}$, which can be seen as the $\mathsf{tx}^{\mathsf{er}}$ transaction from Blitz augmented with an output funding the VC. Using $\mathsf{tx}^{\mathsf{vc}}$, she sets up a Blitz payment of the full VC capacity to Bob through the intermediaries with timeout T, where T is the lifetime of the VC. After the payment has been set up, the VC can be used within the lifetime T. We emphasize that if $\mathsf{tx}^{\mathsf{vc}}$ goes on-chain, the intermediaries can keep their channels open and resolve the payment off-chain, as done in Blitz or similarly in HTLC-based payments. *Once the VC is opened, both Alice and Bob can update the VC balance just as they would do with a PC.*

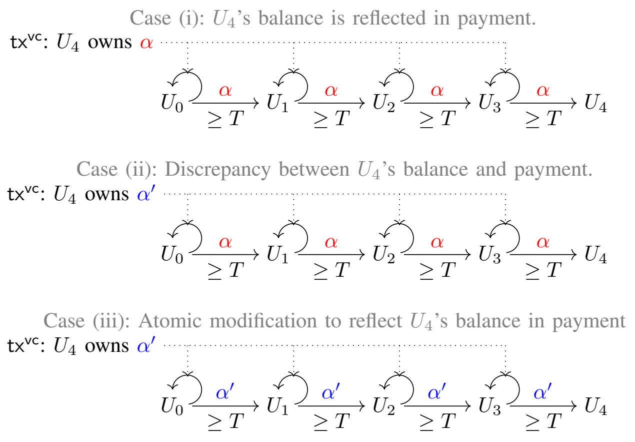
Adding an honest closure We described how two parties can establish a VC off-chain and perform updates on it while securing their balance. However, we still need to address some remaining challenges. So far, Alice would be forced to close the VC on-chain before T, defeating the whole purpose of the VC which is to be operated completely off-chain. Therefore, we need to provide a secure way to close the VC off-chain.

We address this challenge as follows. When Alice and Bob wish to close their VC, they will have some final balance in the VC. In this final balance Bob owns either (i) the full capacity of the VC α or (ii) a smaller amount $0 \leq \alpha' < \alpha$. We observe that for case (i), we do not need to take any action, since after T, Bob's rightful balance is paid to him via the collateral payment, off-chain. For (ii) the idea is to update the payment atomically in a way that, like (i), the correct balance of Bob is reflected in the payment amount, while ensuring that neither the VC endpoints nor the intermediaries are at risk of losing coins. This is challenging as it does not suffice to generate a new payment: one has to modify the contract in each channel atomically to reflect the new balance, tying it to the *same* transaction $\mathsf{tx}^{\mathsf{vc}}$. We illustrate this problem in Figure 7.

We thus introduce a new protocol, called *atomic modification*, which given a collateral payment of value α tied to transaction $\mathsf{tx}^{\mathsf{vc}}$ and a timeout T, allows for updating the collateral payment to a value α' such that $0 \leq \alpha' < \alpha$. By updating a payment we mean updating a PC off-chain, copying the contract that locks the money for the Blitz payment as it was in the previous state, but with the reduced amount. The technical problem is how to do that in a way that a user is not at risk of receiving α' on his left and having to pay α on his right, if for instance the right user refuses to update.

It turns out that we can safely update a payment to a smaller amount from right to left, since a user is always fine to reduce the amount she locks in a collateral. Indeed, if we proceed from right to left, each intermediary U_i is sure to not lose money. The atomicity of the payment ensures that in both the left (U_{i-1}, U_i), having locked α, and the right channel (U_i, U_{i+1}), having locked α', the payment is either refunded or succeeds. In the first case, the balance of the user who reduced the amount in the channel on her right is 0, in the second case it is $\alpha - \alpha' > 0$. After accepting this update in (U_i, U_{i+1}), U_i can safely update (U_{i-1}, U_i) to α' in the same way. We can incentivize the participation of intermediary users with fees. Additionally intermediaries know that if they do not forward this update, Alice will simply publish $\mathsf{tx}^{\mathsf{vc}}$. In fact, Alice is incentivized to do so if the correct updates do not reach her (paying more money than she owes otherwise), thereby ensuring the atomicity of the atomic modification. If all the channels are updated, they can simply go idle as in case (i), or they can finalize this payment instantly by using the fast track functionality [1]. This can also be done in (i).

Fair unlimited lifetime Interestingly, we can use the aforementioned *atomic modification* operation also for updating the lifetime. In particular, besides updating the contracts in each channel to a smaller amount, as shown in Figure 7, we can in fact update the timeout T in each channel. Before the initial timeout T expires, the VC endpoints can run an atomic modification update from receiver to sender. If everyone agrees, they can update to the time $T' > T$, and intermediaries would charge a fee for this. Intuitively,



Case (i): U_4's balance is reflected in payment.
$\mathsf{tx}^{\mathsf{vc}}$: U_4 owns α
U_0 α $\geq T$ U_1 α $\geq T$ U_2 α $\geq T$ U_3 α $\geq T$ U_4
Case (ii): Discrepancy between U_4's balance and payment.
$\mathsf{tx}^{\mathsf{vc}}$: U_4 owns α'
U_0 α $\geq T$ U_1 α $\geq T$ U_2 α $\geq T$ U_3 α $\geq T$ U_4
Case (iii): Atomic modification to reflect U_4's balance in payment
$\mathsf{tx}^{\mathsf{vc}}$: U_4 owns α'
U_0 α' $\geq T$ U_1 α' $\geq T$ U_2 α' $\geq T$ U_3 α' $\geq T$ U_4

Figure 7: Atomic modification: Safely modify the contract tied to a transaction $\mathsf{tx}^{\mathsf{vc}}$ in each channel atomically. Note that $\mathsf{tx}^{\mathsf{vc}}$ is the same transaction in all three cases.

users are incentivized to agree as they are fine to to pay their money later (at T') to their right while receiving it earlier (at T) on their left. This solves the problem of the a priori unlimited lifetime of prior VC constructions. The endpoints have the guarantee that the VC remains virtual until a pre-defined timeout, while the intermediaries have a guarantee that they can unlock their collateral after at most a pre-defined timeout without going on-chain and they can prolong it if everyone agrees for as long as they wish. Since the time for which the VC is prolonged is known, intermediaries can adopt a fee model that is based on time, which is not possible in existing solutions.

Resolving the other issues By detaching the funding transaction from the underlying channels, we additionally get rid of the other issues presented in Section 3.4. Since the funding can be published independently from the channels and the collateral outcome depends on the funding, we give back the possibility to intermediaries to resolve their channels honestly. Additionally, as the funding is not constructed by combining the outputs of the underlying channels in sequence, we eliminate the additional linear on-chain transactions (needing only one) and reduce the linear (or logarithmic) time delay for publishing the funding transaction to a constant. Further, as we discuss in Section 6 and Appendix F.6, Donner achieves a better level of privacy. For an illustration of the full construction as well as the offload operation, we defer the reader to Figures 10 and 11 in Appendix C.

5. Donner: Protocol description

5.1. Security and privacy goals

We informally define three security and three privacy goals for our VC construction. For formal definitions of these properties and proofs, see Appendix F.6. We mark security goals with an *S* and privacy goals with a *P*. Side channel attacks (e.g., *probing* and *balance discovery*) constitute a significant privacy threat for PCNs [22]. Here, we rule out side channels from the attacker model to reason about the leakage induced by the design of the VC construction itself.

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

(S2) Endpoint security No user can steal the sender's balance of the VC. Additionally, the receiver is always guaranteed to get at least its VC balance.

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

(P1) Endpoint anonymity In an optimistic VC execution, 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 an optimistic VC execution, 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 VC, not the value of the individual payments.

The careful reader may have noticed that 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 open question.

5.2. 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 [20] signature scheme Σ as a black-box throughout this work.

Payment channel notation 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 $\mathsf{tx}^{\mathsf{state}}$: $\overline{\gamma}.\mathsf{st} := (\theta_1, ...\theta_n)$. 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 $\gamma := (\overline{\gamma}.\mathsf{id}, \overline{\gamma}.\mathsf{users})$.

Ledger and channels We use the 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 PCs. We assume the PCs 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 posted on-chain after at most Δ time (the blockchain delay), 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.

$\mathsf{preCreate}(\mathsf{tx}^{\mathsf{vc}}, \mathsf{index}, U_0, U_n)$**:** Pre-creates the VC $\overline{\gamma_{\mathsf{vc}}}$, 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 index of the funding transaction $\mathsf{tx}^{\mathsf{vc}}$ that remains off-chain for now. It finally returns $\overline{\gamma_{\mathsf{vc}}}$.

Blitz We leverage the Blitz construction to synchronize the collateral for the VC. Following the pseudo-code definition in [1], we reuse the operations Setup, Open, Finalize and Respond. There is a difference in Setup, where we create the initial state of the VC along with $\mathsf{tx}^{\mathsf{vc}}$, which is the same as $\mathsf{tx}^{\mathsf{er}}$ from Blitz but with one additional output holding α coins funding the VC. This new transaction can be seen in Figure 6, with the difference to Blitz highlighted.

Assumptions In our construction, we assume that every user U has a public key pk_U to receive transactions. Additionally, we assume that honest parties stay online for the duration of the protocol, like in the Lightning Network. 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 [31, 32] for more details. Finally, we assume fee to be a publicly known value charged by every user. Note that in practice, every user can charge an individual fee.

5.3. 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 VC with capacity α fully funded by U_0. We consider the different phases OpenVC, UpdateVC, CloseVC, ProlongVC and Respond. We show the used macros in Figure 8(a), the procedure for updating individual PCs for the close or prolong VC phase in Figure 8(b), and the whole protocol in Figure 8(c).

OpenVC The sender U_0 starts by creating a transaction $\mathsf{tx}^{\mathsf{vc}}$ that contains an output θ_{vc} holding α coins spendable under the condition $\mathsf{MultiSig}(U_0, U_n)$ and n outputs θ_{ϵ_i} holding ϵ coins each spendable under the condition $\mathsf{OneSig}(U_i) + \mathsf{RelTime}(t_{\mathsf{c}} + \Delta)$, 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 VC with $\overline{\gamma_{\mathsf{vc}}} := \mathsf{preCreate}(\mathsf{tx}^{\mathsf{vc}}, 0, U_0, U_n)$. This function pre-creates the VC $\overline{\gamma_{\mathsf{vc}}}$, exchanging the initial state transactions with the other user in $\overline{\gamma_{\mathsf{vc}}}.\mathsf{users} := (U_0, U_n)$ based on the output with index 0 of the funding transaction $\mathsf{tx}^{\mathsf{vc}}$ that remains off-chain for now. It finally returns $\overline{\gamma_{\mathsf{vc}}}$.

Then, each pair of users from U_0 to U_n performs `2pSetup` of [1], which we briefly summarize as follows. 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 θ_{ϵ_0} of $\mathsf{tx}^{\mathsf{vc}}$ holding ϵ under U_0's control. After receiving the signature, they perform this update and revoke their previous state. 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, honest parties just watch the blockchain to observe if $\mathsf{tx}^{\mathsf{vc}}$ is published.

UpdateVC At this point the VC $\overline{\gamma_{\mathsf{vc}}}$ 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 $\mathsf{updateChannel}(\overline{\gamma_{\mathsf{vc}}}, \mathsf{tx}_i^{\mathsf{state}})$. This function updates the VC $\overline{\gamma_{\mathsf{vc}}}$, changing the latest state transaction to $\mathsf{tx}_i^{\mathsf{state}}$ and revoking the previous one. In case of a dispute, the users wait until the VC 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, as we discuss in Appendix B. When they have rebalanced the money inside the VC and definitely before time T, they proceed to the next phase, the closing phase.

CloseVC/ProlongVC (Atomic modification) For closing the VC, assume its final balance is $\alpha - \alpha'$ belonging to U_0 and α' to U_n (and $T' = T$). For prolonging the lifetime, assume the new time is $T' > T$ (and $\alpha' := \alpha$). In either case, pairs of users from perform the new functionality `2pModify` from right to left, which we outline as follows. 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 $\phi = (\mathsf{OneSig}(U_n) \wedge \mathsf{AbsTime}(T)) \vee (\mathsf{MultiSig}(U_{n-1}, U_n) \wedge \mathsf{RelTime}(\Delta))$ (closing) or the time in this condition is changed to T' (prolong). For this new state, U_n creates a transaction $\mathsf{tx}_{n-1}^{\mathsf{r}}$ spending this output and the output of $\mathsf{tx}^{\mathsf{vc}}$ belonging to U_{n-1} and gives its signature for this new $\mathsf{tx}_{n-1}^{\mathsf{r}}$ to U_{n-1}. After U_{n-1} checks that the new state and new $\mathsf{tx}_{n-1}^{\mathsf{r}}$ are correct, they update their channel to this new state and revoke the previous one (cf. Figure 8(b)).

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 VC, 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 $\mathsf{tx}_0^{\mathsf{r}}$ before the currently active timeout T expires. In the case where the correct request reaches the sender, they can either continue using the VC until T' (prolong) or in the case of closing, they wait until T expires, at which the money α' automatically moves from left to right to the receiver, or they perform the fast-track mechanism of [1] to immediately unlock their funds (cf. Appendix B).

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 $\mathsf{tx}_i^{\mathsf{r}}$. 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 $\mathsf{tx}_i^{\mathsf{p}}$.

We show the pseudocode for the protocol in Figure 8(c) and the protocol without relying on the API of [1] in Appendix E. For better readability we simplify the protocol, e.g., we omit ids required for routing VCs concurrently. For the formal protocol description in the UC framework, we defer to Appendix F.4.

6. Security analysis

6.1. Informal security analysis

Balance security When the VC is opened, a Blitz [1] collateral payment is simultaneously opened from sender to receiver. A Blitz payment provides balance security to the intermediaries. An intermediary is merely involved in a payment, the outcome of which is atomically determined by whether or not $\mathsf{tx}^{\mathsf{vc}}$ is posted. For both of these outcomes, the intermediary does not lose money. As already argued in Section 4 the *atomic modification* operation does not put an intermediary at risk.

Endpoint security An honest sender can always enforce the VC that holds its correct balance by posting $\mathsf{tx}^{\mathsf{vc}}$ and thereby offloading the VC. 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 VC. This means that if sender and receiver are honest, no one can force them to offload the VC before T.

Endpoint anonymity and path privacy $\mathsf{tx}^{\mathsf{vc}}$ is constructed, as in Blitz, based on fresh and stealth addresses and the endpoints of the VC rely on fresh addresses too. Hence, an intermediary observing $\mathsf{tx}^{\mathsf{vc}}$ learns no meaningful information about the sender, the receiver, and the path. This holds 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 VC to sender/receiver, like in any other off-chain protocol, including the Lightning Network.

Value privacy Similarly to how payments between users of a payment channel (PC) are known only to those users, also VC updates are only known to the endpoints. There occur no on-chain transactions in the optimistic case throughout the protocol. Any two users connected in the PC network can open a VC, and apart from their open and close balance, the amount and nature of the individual updates remains known only to them, even in the pessimistic case.

6.2. Security model

We rely on the synchronous, global universal composability (GUC) framework [11] to model the Donner protocol.

(a) Macros: **`genState`**$(\alpha_i, T, \overline{\gamma_i})$**:** Generates and returns a new channel state carrying transaction $\mathsf{tx}_i^{\mathsf{state}}$ from the given parameters. **`genPay`**$(\mathsf{tx}_i^{\mathsf{state}})$ Returns $\mathsf{tx}_i^{\mathsf{p}}$, which takes $\mathsf{tx}_i^{\mathsf{state}}.\mathsf{output}[0]$ as input and creates a single $\mathsf{output} := (\alpha_i, \mathsf{OneSig}(U_{i+1}))$. **`genRef`**$(\mathsf{tx}_i^{\mathsf{state}}, \mathsf{tx}^{\mathsf{vc}}, \theta_{\epsilon_i})$ Return $\mathsf{tx}_i^{\mathsf{r}}$, which takes as input $\mathsf{tx}_i^{\mathsf{state}}.\mathsf{output}[0]$ and $\theta_{\epsilon_i} \in \mathsf{tx}^{\mathsf{vc}}.\mathsf{output}$. The calling user U_i makes sure that this output belongs to an address under U_i's control. It creates a single output $\mathsf{tx}_i^{\mathsf{r}}.\mathsf{output} := (\alpha_i + \epsilon, \mathsf{OneSig}(U_i))$, where α_i, U_i, U_{i+1} are taken from $\mathsf{tx}_i^{\mathsf{state}}$.

(b) 2-party operation: <u>`2pModify`$(\overline{\gamma_i}, \mathsf{tx}^{\mathsf{vc}}, \alpha_i', T')$</u>

Let T be the timeout, α_i the amount and $\theta_{\epsilon_{i-1}}$ be the output used for the two party contract set up between U_{i-1} and U_i, known from `2pSetup` executed in the Open [1] phase.

<u>U_i:</u> $\mathsf{tx}_{i-1}^{\mathsf{state}'} :=$ `genState`$(\alpha_i', T', \overline{\gamma_{i-1}})$, $\mathsf{tx}_{i-1}^{\mathsf{r}'} :=$ `genRef`$(\mathsf{tx}_{i-1}^{\mathsf{state}'}, \theta_{\epsilon_{i-1}})$, then 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} //$\theta_{\epsilon_{i-1}}$ known as θ_{ϵ_x} from `2pSetup`

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

1) Extract α_i' and T' from $\mathsf{tx}_{i-1}^{\mathsf{state}'}$. Check that $\alpha_i' \leq \alpha_i$, $T' \geq T$ and $\mathsf{tx}_{i-1}^{\mathsf{state}'} =$ `genState`$(\alpha_i', T', \overline{\gamma_{i-1}})$.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. Check that $\sigma_{U_i}(\mathsf{tx}_{i-1}^{\mathsf{r}'})$ is a correct signature of U_i for $\mathsf{tx}_{i-1}^{\mathsf{r}'}$. Check that $\mathsf{tx}_{i-1}^{\mathsf{r}'} =$ `genRef`$(\mathsf{tx}_{i-1}^{\mathsf{state}'}, \theta_{\epsilon_{i-1}})$ //α_i, T and $\theta_{\epsilon_{i-1}}$ from `2pSetup`
2) $\mathsf{updateChannel}(\overline{\gamma_{i-1}}, \mathsf{tx}_{i-1}^{\mathsf{state}'})$
3) 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 $(\top, \alpha_i', T')$. Else, return $\bot$.

<u>U_i:</u> 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 $\mathsf{tx}_{i-1}^{\mathsf{p}'} :=$ `genPay`$(\mathsf{tx}_{i-1}^{\mathsf{state}'})$, respectively.

(c) Protocol: <u>OpenVC</u>

<u>(i) Setup*, as in [1], except:</u>

- Create $\mathsf{tx}^{\mathsf{vc}}$ instead of $\mathsf{tx}^{\mathsf{er}}$ as shown in Figure 6
- $\overline{\gamma_{\mathsf{vc}}} := \mathsf{preCreate}(\mathsf{tx}^{\mathsf{vc}}, 0, U_0, U_n)$ together with U_n after creating $\mathsf{tx}^{\mathsf{vc}}$, to create the VC commitment transactions.

<u>(ii) Open and Finalize as in [1]</u>

<u>UpdateVC</u>

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

<u>CloseVC/ProlongVC (atomic modification)</u>

<u>(i) InitClose/InitProlong</u>

<u>U_n:</u> Let α_i' be the final balance of U_n in the virtual channel and $T' = T$ (Close) or let $T' > T$ be the new lifetime of the VC and leave $\alpha_i' = \alpha_i$ (Prolong). Execute `2pModify`$(\overline{\gamma_i}, \mathsf{tx}^{\mathsf{vc}}, \alpha', T')$

<u>U_{i-1} upon $(\top, \alpha_i', T')$:</u> If $U_{i-1} \neq U_0$, let $\alpha_{i-1}' := \alpha_i' + \mathsf{fee}$. Execute `2pModify`$(\overline{\gamma_{i-2}}, \mathsf{tx}^{\mathsf{vc}}, \alpha_{i-1}', T')$

<u>(ii) Emergency-Offload</u>

<u>U_0:</u> If U_0 has not successfully performed `2pModify` with the correct value α' (plus fee for each hop) until $T - t_{\mathsf{c}} - 3\Delta$, $\mathsf{publishTx}(\mathsf{tx}^{\mathsf{vc}}, \sigma_{U_0'}(\mathsf{tx}^{\mathsf{vc}}))$. Else, update $T := T'$

<u>Respond (executed in every round) as in [1]</u>

Figure 8: (a) macros, (b) 2-party operation, (c) protocol

We make use of some preliminary functionalities commonly used in the literature [1, 2, 3, 16, 17]. 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 PCs as well as the preparation and update of the VCs.

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 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 and add the same transaction to the ledger in both the real and the ideal world in the same rounds. We refer to Appendix F for the preliminaries, the ideal functionality, the formal protocol, the simulator, the formal proof of Theorem 1 and the formalization of the security and privacy goals of Section 5.1 as well as the proof that $\mathcal{F}_{VC}$ has these properties.

Theorem 1. *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 $\mathcal{F}_{VC}$.*

7. Evaluation and comparison

Communication overhead We implemented a small proof-of-concept that creates the raw Bitcoin transactions necessary for Donner [14]. 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 [3] as the underlying payment channel (PC) scheme, but note that Donner is also compatible with Lightning channels (as we discuss at the end of this section).

For opening a virtual channel (VC), each of the n underlying PCs needs to exchange 4 transactions: $\mathsf{tx}^{\mathsf{vc}}$, $\mathsf{tx}_i^{\mathsf{r}}$ and two transactions 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. $\mathsf{tx}_i^{\mathsf{r}}$ has a size of 306 bytes, and a channel update to a state holding this contract is 742 bytes. $\mathsf{tx}_i^{\mathsf{p}}$ does

TABLE 2: Communication overhead of Donner for the whole path (not per party) for the different operations, assuming a VC across n channels. In the pessimistic offload, $k \in [0, n]$ is the number of channels where there is a dispute. Only in the Offload case transactions are posted on-chain.

	# 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	$192 + 34$	$0.25 + 0.04 \cdot n$
Offload (Pessimistic)	$3k + 1$	$1048 \cdot k + 192 + 34 \cdot n$	$1.36 \cdot k + 0.25 + 0.04 \cdot n$

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 communication per channel for the open phase. Then, we require to exchange the initial state of the VC, which is 2 transactions or 695 bytes. This totals $4\cdot n+2$ transactions or $34 \cdot n^2 + 1240 \cdot n + 695$ bytes for the path.

For honestly closing a VC, the payment needs 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 underlying channels. To update a VC, the two endpoints need to exchange 2 transactions with 695 bytes, the same as a PC 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 $0.25 + 0.04 \cdot n$ USD. Note that if individual users on the path do not collaborate, regardless if the VC 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 PC 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 $1.36 \cdot k + 0.25 + 0.04 \cdot n$ USD for the whole path. We mark this as the *pessimistic* case in Table 2.

Efficiency comparison We compare our construction to LVPC [21] and Elmo [24] (cf. Table 3), the only current UTXO-based VC solutions over multiple hops. As already mentioned, LVPC and Elmo have rooted VC funding transactions. We evaluate, in particular, the off-chain and on-chain costs of the core VC operations (open, update, close, and offload), concluding that Donner is better in each case.

LVPC is constructed recursively; there are different ways of doing the recursion. Each combination leads to the same minimum number of VCs required for a path of n base channels: One for each of the $n - 1$ intermediaries. 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 linear (Figure 14) and in the best case logarithmic (Appendix E.1) 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 VC are needed, so $7 \cdot (n - 1)$. Similar, for closing, we need to store 4 transactions per VC in LVPC, so $4\cdot(n-1)$. Elmo requires to store $n-2+\chi_{i=2}+\chi_{i=n-1}+(i-2+\chi_{i=2})(n-i-1+\chi_{i=n-1}) \in \Theta(n^2)$ (where

TABLE 3: Comparison of LVPC, Elmo and Donner for a VC over from U_0 to U_n.[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 [21]	$7 \cdot (n-1)$	✓
	Elmo [24]	$\Theta(n^3)$	✓
	Donner	$4 \cdot n + 2$	✓
Update	LVPC [21]	2	✓
	Elmo [24]	2	✓
	Donner	2	✓
Close	LVPC [21]	$4 \cdot (n-1)$	✓
	Elmo [24]	$3 \cdot n + 3$	✗
	Donner	$3 \cdot n$	✓
Offload (Optimistic)	LVPC [21]	$5 \cdot (n-1)$	✗
	Elmo [24]	$3 \cdot n + 1$	✗
	Donner	1	✗
Offload (Pessimistic)	LVPC [21]	$5 \cdot (n-1)$	✗
	Elmo [24]	$3 \cdot n + 1$	✗
	Donner[1]	$3 \cdot k + 1$	✗

χ_P is 1 if P is true and 0 otherwise) for the i^{th} intermediary (and 1 for the endpoints), resulting in a storage overhead of $\Theta(n^3)$ for the whole path. Closing honestly (i.e., off-chain) is not defined for Elmo, so it needs to be closed on-chain, resulting in 2 transactions per channel (n) for closing plus 1 transaction per user ($n+1$) plus 2 transactions to close the VC or $3 \cdot n + 3$ on-chain transactions. Donner requires the close operation per underlying channel, so $3 \cdot n$ transactions. The update phase is the same in all constructions.

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 PCs and $n - 2$ VCs (all but the topmost one). Additionally, we need to publish $n-1$ funding transactions of the VCs 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 Elmo, we need $3 \cdot n + 1$, i.e., the number of transactions to close minus the 2 transactions required to put the VC state on-chain. 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. We show an application example in Appendix A.1, where we analyze how Donner can be used to connect a node better to a network via VCs, compared to no VCs and LVPC.

Compatibility with Lightning channels To simplify the formalization of this work, we built our VC construction on top of generalized payment channels (GC) [3], 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 [2] constructions rely on GCs as well, while the recursive LVPC [21] relies on simple channels that have only one state, but each update reduces the limited lifetime of the channel. (Elmo [24] needs the opcode `ANYPREVOUT` that is not supported in Bitcoin and thus not either in the LN.)

As LN channels are the only ones deployed in practice so far, 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 $\mathsf{tx}^{\mathsf{r}}_i$ 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 [21]) 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 [3]. 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.

8. Conclusion

Payment channel networks (PCNs) have emerged as successful scaling solutions for cryptocurrencies. However, path-based protocols are tailored to payments, excluding novel and interesting non-payment applications such as Discreet Log Contracts, while creating direct PCs on-demand is expensive, slow and infeasible on a large scale. VCs are among the most promising solutions. We show that all existing UTXO-based constructions are vulnerable to the Domino attack, which fundamentally undermines the underlying PCN itself.

Hence we introduce a new VC design, the first one to be secure against the Domino attack, besides the only one achieving path privacy and a time-based fee model. Our performance analysis demonstrates that Donner is more efficient: It only requires a single on-chain transaction to solve disputes, as opposed to a number that is linear in the path length, and the storage overhead is constant too, as opposed to linear.

Overall, Donner offers an easy-to-adopt, LN-compatible VC construction enabling new applications such as Discreet Log Contracts without the need to create a direct PC or fast and direct micropayments, among others. Unlike the underlying PCNs, the VCs are not susceptible to liveness and privacy attacks by the intermediaries and do not require fees per payment.

References

[1] L. Aumayr, P. Moreno-Sanchez, A. Kate, and M. Maffei. “Blitz: Secure Multi-Hop Payments Without Two-Phase Commits”. In: *USENIX Security*. 2021.

[2] L. Aumayr et al. “Bitcoin-Compatible Virtual Channels”. In: *IEEE Security and Privacy*. 2021.

[3] L. Aumayr et al. “Generalized Channels from Limited Blockchain Scripts and Adaptor Signatures”. In: *Asiacrypt*. 2021.

[4] C. Badertscher, U. Maurer, D. Tschudi, and V. Zikas. “Bitcoin as a Transaction Ledger: A Composable Treatment”. In: *Advances in Cryptology CRYPTO*. Vol. 10401. 2017, pp. 324–356.

[5] I. Bentov and R. Kumaresan. “How to Use Bitcoin to Design Fair Protocols”. In: *CRYPTO*. 2014.

[6] *Bitcoin Avg. Transaction Fee historical chart*. https://bitinfocharts.com/comparison/bitcoin-transactionfees.html. Jan. 2022.

[7] *Bitcoin Rich List*. https://bitinfocharts.com/top-100-richest-bitcoin-addresses.html. Jan. 2022.

[8] *BOLT #2: Peer Protocol for Channel Management*. https://github.com/lightning/bolts/blob/master/02-peer-protocol.md#rationale-7. Mar. 2022.

[9] C. Burchert, C. Decker, and R. Wattenhofer. “Scalable Funding of Bitcoin Micropayment Channel Networks”. In: *Stabilization, Safety, and Security of Distributed Systems*. 2017, pp. 361–377.

[10] J. Camenisch and A. Lysyanskaya. “A Formal Treatment of Onion Routing”. In: *Advances in Cryptology CRYPTO*. 2005, pp. 169–187.

[11] R. Canetti, Y. Dodis, R. Pass, and S. Walfish. “Universally Composable Security with Global Setup”. In: *TCC*. Vol. 4392. 2007, pp. 61–85.

[12] G. Danezis and I. Goldberg. “Sphinx: A Compact and Provably Secure Mix Format”. In: *IEEE Security and Privacy*. 2009.

[13] *DLC over Lightning*. Available at https://mailmanlists.org/pipermail/dlc-dev/2021-November/000091.html. [dlc-dev] Mailing List, Nov. 2021.

[14] *Donner VC evaluation of the communication overhead*. https://github.com/donner-vc/overhead. 2021.

[15] T. Dryja. *Discreet Log Contracts*. Available at https://adiabat.github.io/dlc.pdf. 2017.

[16] S. Dziembowski, L. Eckey, S. Faust, J. Hesse, and K. Hostáková. “Multi-party Virtual State Channels”. In: *Eurocrypt*. 2019, pp. 625–656.

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

[18] S. Dziembowski, S. Faust, and K. Hostáková. “General State Channel Networks”. In: *Computer and Communications Security, CCS*. 2018, pp. 949–966.

[19] C. Egger, P. Moreno-Sanchez, and M. Maffei. “Atomic Multi-Channel Updates with Constant Collateral in Bitcoin-Compatible Payment-Channel Networks”. In: *ACM CCS*. 2019, 801–815.

[20] S. Goldwasser, S. Micali, and R. L. Rivest. “A digital signature scheme secure against adaptive chosen-message attacks”. In: *SIAM Journal on computing* 17.2 (1988), pp. 281–308.

[21] M. Jourenko, M. Larangeira, and K. Tanaka. “Lightweight Virtual Payment Channels”. In: *19th International Conference on Cryptology and Network Security (CANS)*. 2020.

[22] G. Kappos et al. “An Empirical Analysis of Privacy in the Lightning Network”. In: *Financial Cryptography and Data Security*. 2021, pp. 167–186.

[23] J. Katz, U. Maurer, B. Tackmann, and V. Zikas. “Universally Composable Synchronous Computation”. In: *Theory of Cryptography TCC*. Ed. by A. Sahai. Vol. 7785. 2013, pp. 477–498.

[24] A. Kiayias and O. S. T. Litos. *Elmo: Recursive Virtual Payment Channels for Bitcoin*. Cryptology ePrint Archive. https://eprint.iacr.org/2021/747. 2021.

[25] *LN Snapshot*. https://ln.fiatjaf.com/. Jan. 2022.

[26] G. Malavolta, P. Moreno-Sanchez, A. Kate, M. Maffei, and S. Ravi. “Concurrency and Privacy with Payment-Channel Networks”. In: *ACM CCS*. 2017.

[27] G. Malavolta, P. Moreno-Sanchez, C. Schneidewind, A. Kate, and M. Maffei. “Anonymous Multi-Hop Locks for Blockchain Scalability and Interoperability”. In: *NDSS Symposium*. 2019.

[28] A. Miller, I. Bentov, R. Kumaresan, and P. McCorry. “Sprites: Payment Channels that Go Faster than Lightning”. In: *CoRR* abs/1702.05812 (2017). URL: http://arxiv.org/abs/1702.05812.

[29] *Perun Network*. https://perun.network/. 2020.

[30] J. Poon and T. Dryja. *The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments*. Draft version 0.5.9.2, available at https://lightning.network/lightning-network-paper.pdf. Jan. 2016.

[31] S. Roos, P. Moreno-Sanchez, A. Kate, and I. Goldberg. “Settling Payments Fast and Private: Efficient Decentralized Routing for Path-Based Transactions”. In: *NDSS Symposium*. 2018.

[32] V. Sivaraman et al. “High Throughput Cryptocurrency Routing in Payment Channel Networks”. In: *NSDI*. 2020, pp. 777–796.

[33] N. Van Saberhagen. “Cryptonote v 2.0 (2013)”. In: *URL: https://web.archive.org/web/20201028121818/https://cryptonote.org/whitepaper.pdf. White Paper.* ().

Appendix A. When to use virtual channels

In the state of the art on off-chain protocols, we can distinguish between generic 2-party applications and simple payments. The former require a direct channel between the parties and therefore it is interesting to compare VCs and direct PCs in this setting. In the latter, PCNs have already been shown to offer improvements over constructing a direct channel and therefore it is worth to compare VC against

PCN payments. In the following, we highlight a few use cases of VCs in these two settings.

VCs vs PCs for 2-party applications Imagine that two arbitrary users that do not share a PC or a VC decide to execute a 2-party application between them. The first disadvantage of using a PC over a VC is that over their lifespan they would pay twice as many fees per on-chain transaction (i.e., to open and close the channel). At the current average Bitcoin transaction cost of 4100 satoshi (or 0.000041 BTC or 1.73 USD), the overall cost would be 8200 satoshi (3.46 USD).

Since VCs are currently not being used in practice, there is no fee model for them. To put the cost of opening a VC into perspective, we can compare it to payments over the PCN. Say Alice and Bob are connected by a path of payment channels that has 3 hops (we take the average shortest distance of a current LN snapshot). Taking the current average fees of the LN, and, say, an average transaction amount of $50,000$ satoshi (21.10 USD), Alice and Bob could perform 1115 payments in the LN for the same fee of 8200 satoshi (3.46 USD). This means that in this example, the fees paid to intermediaries for operating a VC, i.e., opening and closing, is cheaper in terms of fees, if these VC operating fees are less than the fees of 1115 LN payments.

More generally, we can compare the cost of VC versus PC as follows. We introduce x as a factor by which VCs are more expensive than PCN payments. A VC channel is cheaper if $l \cdot (\mathsf{BF} + \mathsf{RF} \cdot a) \cdot x < 2 \cdot \mathsf{TF}$ holds, where l is the number of hops in the path between the two VC endpoints. Further BF and RF are the two types of fees charged in PCN implementation such as the LN, where BF is a base fee charged by intermediaries for forwarding payments and RF a relative fee based on the payment amount. We compare this to the transactions fee on-chain TF, paid twice in the lifespan of a PC. For instance, taking the concrete values from the example above we can write the following: $3 \cdot (1 + 0.000029 \cdot a) \cdot x < 8200$.

Secondly, creating direct PCs on-demand for applications such as Discreet Log Contracts instead of VCs is again not scalable. Doing so would incur a continuous on-chain transaction load for opening and closing channels. This is against the purpose of PCs and PCNs, which aim at reducing the on-chain load.

Finally, and perhaps still more importantly, it is not possible to open a short-lived PC, since it requires to wait for the confirmation of the funding transaction on the blockchain, which is around 1 hour in Bitcoin. So for applications that are time-critical, direct PCs are not an option. Applications such as betting on a sports event, say, half an hour before they end are simply impossible with direct PCs.

VCs vs PCN payments Due to the limited transaction size in Bitcoin, current Lightning channels are limited to hold 483 concurrent payments, which becomes especially critical in a micropayment setting. VCs can be used to overcome this issue. Simply, instead of a payment, an output can be used to collateralize a VC, which in turn can be used to again hold 483 payments or further VCs, effectively helping to mitigate this limitation.

In terms of fees, VCs are more desirable than payments over a PCN in the context of micropayments. This is due to the fact that in a PCN, the intermediaries charge a fee for every payment, while for a VC, the fee is charged only once. We can therefore say that a VC is cheaper, if the (simplified) inequality $l \cdot (\mathsf{BF} + \mathsf{RF} \cdot a) \cdot x < l \cdot (n \cdot \mathsf{BF} + \mathsf{RF} \cdot a)$ holds, where similar to above we use the base fee BF and relative fee RF of the LN. a is the sum of the amounts of all micropayments, n the number of micropayments and x again the factor by which a VC is more expensive than a payment. We stress that for any given x there is a number of payments n, such that the use of VC becomes cheaper than payments over the PCN, because the base fee BF is paid for each of the n micropayments in the PCN setting and only once in the VC setting.

Offline users Routing multi-hop payments (MHPs) through the network requires active participation from the intermediaries. However, users may want to go offline and then cannot route MHPs. To still lend their capacity in a productive way and generate some fees, they can allow other nodes to build a VC over them, using watchtowers to ensure their balance.

A.1. Application scenario: Bootstrapping

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. We exclude Elmo here, as it does not provide functionality to close virtual channels off-chain. 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.

Appendix B. Extended comparison and discussion

Some of the existing VC schemes either rely on a Turing-complete scripting language or are limited to a single intermediary. In Table 5 we include all VC schemes.

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

Unidirectionally funded Similar to current PCs 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.

Choosing the lifetime The lifetime T is chosen by the two endpoints of the VC, depending on how long they plan to use the VC. They propose this to the intermediaries who can, based on this time and the amount they need to lock as a collateral, charge a fee. Note that this T has to be larger than the time it takes to settle the Blitz contracts, $T \geq 3\Delta + t_{\mathsf{c}}$, where Δ is an upper bound on the time it takes for a valid transaction to appear on the ledger (i.e., modelling the block delay as mentioned in Section 2) and t_{c} is the time it takes to close a channel. In order to prolong the lifetime, intermediaries can do so if they agree. Moreover, they can charge a fee based on the time and amount.

Properties inherited from Blitz The fee mechanism of Blitz can be reused here as well, i.e., the intermediary nodes forward fewer coins than they receive. Additionally, the outputs ϵ of $\mathsf{tx}^{\mathsf{vc}}$ represent a small number. Since they cannot be 0, they are the smallest possible value, one dust (546 satoshi), i.e., something that is insignificant in value to the sender. If a VC is closed (honestly) before the lifetime expires, parties do not need to wait until the lifetime expires to unlock their money. They can unlock it right away by using the *fast track* mechanism of Blitz. We refer the reader for these details to [1]. Finally, reusing the same stealth address and onion routing mechanism as in [1] allows us to achieve our desired privacy properties.

Appendix C. Operation examples

To illustrate the different operations for different schemes as examples, we provide the following figures. For rooted VCs, we show the construction in Figure 4 in Section 2. We further show the offload operation in Figure 9, which coincides with the outcome of the Domino attack example in Section 3. For Donner, we show the full construction in Figure 10 and the offload operation in Figure 11

TABLE 4: Bootstrapping cost comparison

	no VCs		LVPC [21]		**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

Appendix D. Extended background

D.1. Transaction graphs

In this section we give a more in-depth explanation and example (Figure 12) of our transaction graph notation. Rounded rectangles represent transactions, if they have a single border it means they are off-chain, with a double border on-chain. Incoming arrows to a transaction represent its inputs. The boxes within transactions denote outputs, the outgoing arrows define how 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 $\mathsf{MultiSig}(\mathsf{pk}_1, \mathsf{pk}_2, ...)$. 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 the blockchain. 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 $\phi = \phi_1 \vee ... \vee \phi_n$, 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 $\phi = \phi_1 \wedge ... \wedge \phi_n$.

D.2. Synchronization example

A multi-hop payment (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. A mechanism synchronizing all channels on the path is required for a payment, such that each channel is updated to represent the fact that α coins moved from left to right. We give an example of what we mean in Figure 13.

Appendix E. Extended macros, prerequisites and protocol

In this section, discuss the prerequisites *stealth addresses* and *onion routing*. We give extended pseudo-code for the used subprocedures used in our protocol, both in the pseudocode definition given in Section 5 and in the formal model in Appendix F.3, Appendix F.4 and Appendix F.5. To be transparent about the similarities to [1] and highlight the novelties of this work, we mark the latter in green . Further, we spell out the full protocol pseudocode, including the

TABLE 5: Comparison to other virtual channel schemes. We denote *dispute* as the case where a party needs to enforce their VC funds or be compensated. In the UTXO case, this means offloading. * by synchronizing all channels, this time can be reduced to $\Theta(log(n))$. † for single-hop constructions n is constant, however, since the action/storage overhead/time delay is per user, we write $\Theta(n)$. ‡ This depends on using indirect/direct dispute.

	Perun [17]	GSCN [18]	MPVC [16]	BCVC-V/BCVC-NV [2]	LVPC [21]	Elmo [24]	Donner
Scripting req.	Ethereum	Ethereum	Ethereum	Bitcoin	Bitcoin	Bitcoin + ANYPREVOUT	Bitcoin
Multi-hop	no	yes	yes	no	yes	yes	yes
Domino attack	no	no	no	yes	yes	yes	no
Path privacy	no	no	no	no	no	no	yes
Storage Overhead per party	$\Theta(n)^\dagger$	$\Theta(n)^*$	$\Theta(n)^*/\Theta(1)^\ddagger$	$\Theta(n)^\dagger$	$\Theta(n)^*$	$\Theta(n^3)$	$\Theta(1)$
Time-based fee model	yes	yes	yes	no/yes	no	no	yes
Unlimited lifetime	no	no	no	yes/no	no	yes	yes
Off-chain closing	yes	yes	yes	yes	yes	no	yes
Dispute: txs on-chain	1	1	1	$\Theta(n)^\dagger$	$\Theta(n)$	$\Theta(n)$	1
Dispute: time delay	$\Theta(n)^\dagger$	$\Theta(n)^*$	$\Theta(n)/1^\ddagger$	$\Theta(n)^\dagger$	$\Theta(n)^*$	$\Theta(n)^*$	1

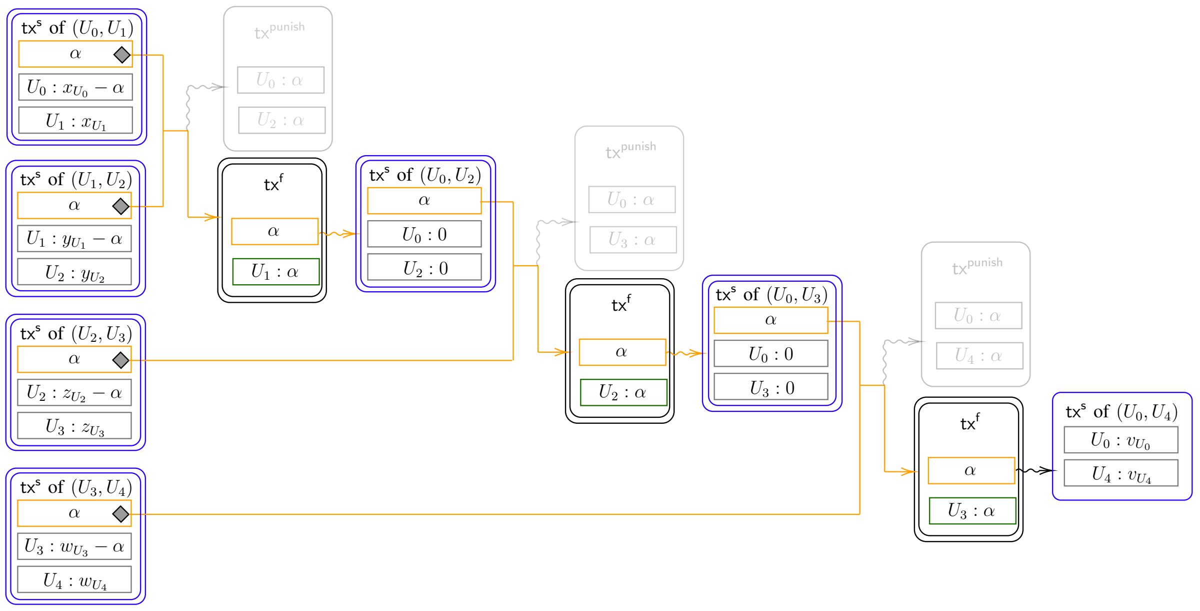


Figure 9: Illustration showing the transactions that go on-chain in case of offloading, an operation that can be forced by a malicious enduser in the Domino attack, forcing all underlying channels to be closed.

parts taken from. For the protocol see Figure 16, for the two party protocols used therein see Figure 17. To be transparent about the similarities to [1] and highlight the novelties of this work, we mark the latter in green color.

Subprocedures

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

1) Check that $\mathsf{tx}^{\mathsf{in}}$ is a transaction on the ledger $\mathcal{L}$.
2) If $\mathsf{tx}^{\mathsf{in}}.\mathsf{output}[0].\mathsf{cash} \geq n \cdot \epsilon + \alpha$ 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 fund $\mathsf{tx}^{\mathsf{vc}}$), the sender will pay any superfluous coins back to a fresh address of itself.

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

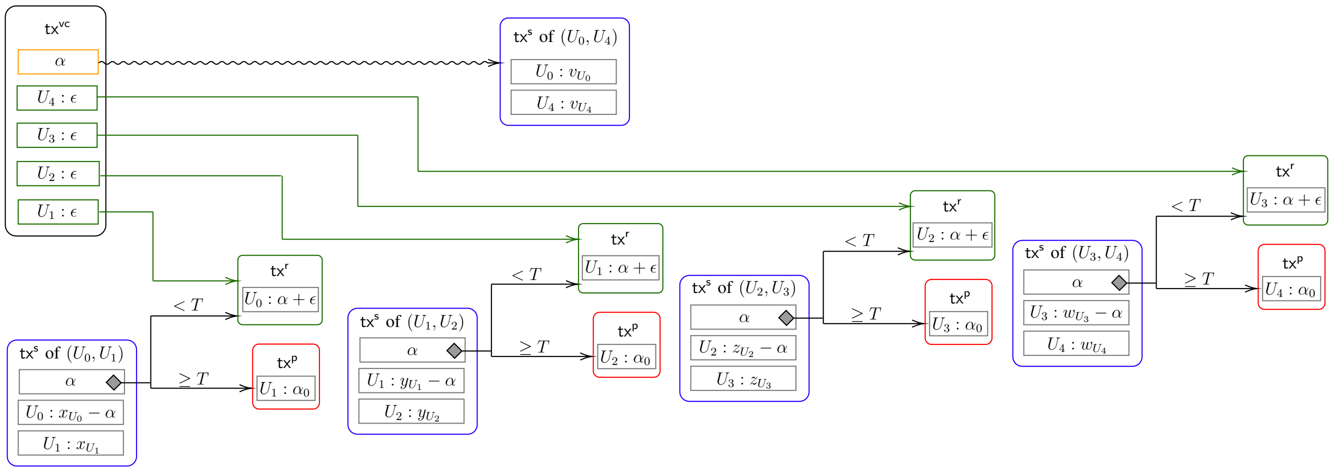
Check that channelList forms a valid path from U_0 via some intermediaries to a receiver U_n and that no users are in the path twice. If not, return $\bot$. Else, return U_n.

`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 $\bot$.

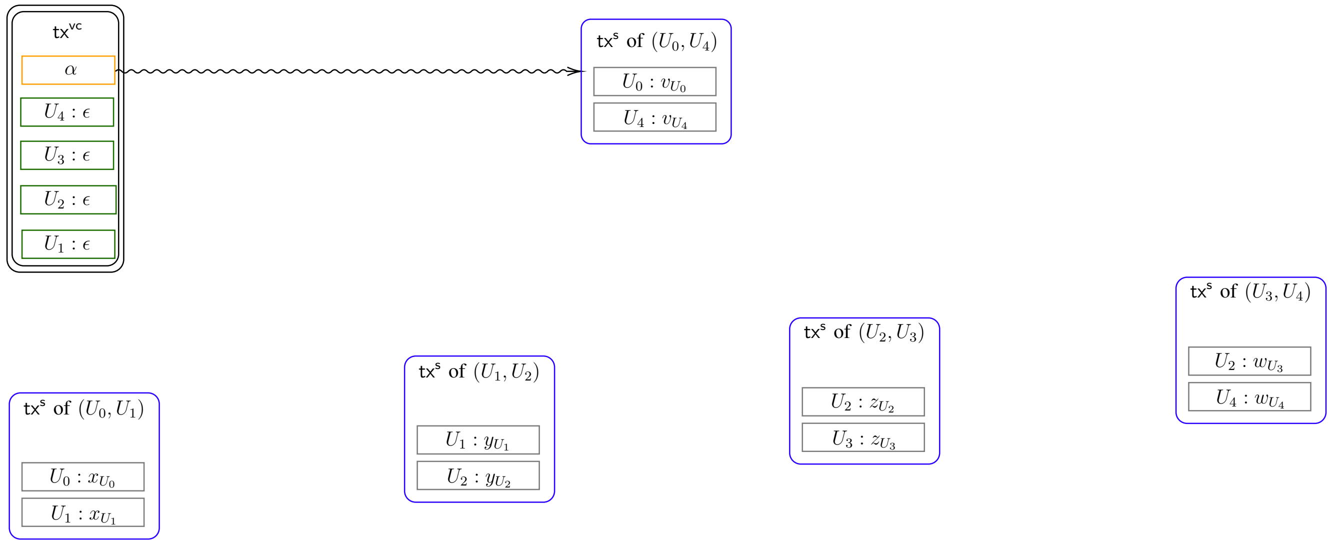
`genTxVc`$(U_0, \mathsf{channelList}, \mathsf{tx}^{\mathsf{in}})$:

1) Let $\mathsf{outputList} := \emptyset$ and $\mathsf{rList} := \emptyset$
2) For every channel γ_i in channelList:
   - $(\mathsf{pk}_{\widetilde{U_i}}, R_i) \leftarrow \mathsf{GenPk}(\gamma_i.\mathsf{left}.A, \gamma_i.\mathsf{left}.B)$
   - $\mathsf{outputList} := \mathsf{outputList} \cup (\epsilon, \mathsf{OneSig}(\mathsf{pk}_{\widetilde{U_i}}) \wedge \mathsf{RelTime}(t_{\mathsf{c}} + \Delta))$
   - $\mathsf{rList} := \mathsf{rList} \cup R_i$
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 $(\mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion})$



$\mathsf{tx}^{\mathsf{vc}}$
α
$U_4 : \epsilon$
$U_3 : \epsilon$
$U_2 : \epsilon$
$U_1 : \epsilon$
tx^{s} of (U_0, U_4)
$U_0 : v_{U_0}$
$U_4 : v_{U_4}$
tx^{r}
$U_0 : \alpha + \epsilon$
$U_1 : \alpha + \epsilon$
$U_2 : \alpha + \epsilon$
$U_3 : \alpha + \epsilon$
tx^{p}
$U_1 : \alpha_0$
$U_2 : \alpha_0$
$U_3 : \alpha_0$
$U_4 : \alpha_0$
$< T$
$\geq T$
tx^{s} of (U_0, U_1)
$U_0 : x_{U_0} - \alpha$
$U_1 : x_{U_1}$
tx^{s} of (U_1, U_2)
$U_1 : y_{U_1} - \alpha$
$U_2 : y_{U_2}$
tx^{s} of (U_2, U_3)
$U_2 : z_{U_2} - \alpha$
$U_3 : z_{U_3}$
tx^{s} of (U_3, U_4)
$U_3 : w_{U_3} - \alpha$
$U_4 : w_{U_4}$

Figure 10: Illustration of a Donner VC of U_0 and U_4 via U_1, U_2 and U_3.



$\mathsf{tx}^{\mathsf{vc}}$
α
$U_4 : \epsilon$
$U_3 : \epsilon$
$U_2 : \epsilon$
$U_1 : \epsilon$
tx^{s} of (U_0, U_4)
$U_0 : v_{U_0}$
$U_4 : v_{U_4}$
tx^{s} of (U_3, U_4)
$U_2 : w_{U_3}$
$U_4 : w_{U_4}$
tx^{s} of (U_2, U_3)
$U_2 : z_{U_2}$
$U_3 : z_{U_3}$
tx^{s} of (U_1, U_2)
$U_1 : y_{U_1}$
$U_2 : y_{U_2}$
tx^{s} of (U_0, U_1)
$U_0 : x_{U_0}$
$U_1 : x_{U_1}$

Figure 11: Illustration of the offload operation for a Donner VC. Note that the underlying PCs remain open and only one transaction goes on-chain: $\mathsf{tx}^{\mathsf{vc}}$.

```
genState(α_i, T, γ̄_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 $\mathsf{tx}_i^{\mathsf{state}}$.

checkTxVc(U_i, a, b, tx^vc, rList, onion_i):
1) $x := \mathsf{GetRoutingInfo}(\mathsf{onion}_i, U_i)$. If $x = \perp$, return $\perp$. If $U_i$ is the receiver and $x = \mathcal{H}(\mathsf{tx}^{\mathsf{vc}})$ , return $(\top, \top, \top, \top, \top)$. Else, if $x \neq (U_{i+1}, \mathcal{H}(\mathsf{tx}^{\mathsf{vc}}), \mathsf{onion}_{i+1})$, return $\perp$.
2) For all outputs $(\mathsf{cash}, \phi) \in \mathsf{tx}^{\mathsf{vc}}.\mathsf{output}$ except output with index 0 it must hold that:
   • $\mathsf{cash} = \epsilon$
   • $\phi = \mathsf{OneSig}(\mathsf{pk}_x) \wedge \mathsf{RelTime}(t_{\mathsf{c}} + \Delta)$ for some identity $\mathsf{pk}_x$
3) For exactly one output $\theta_{\epsilon_i} := (\epsilon, \mathsf{OneSig}(\widetilde{U_i}) \wedge \mathsf{RelTime}(t_{\mathsf{c}} + \Delta)) \in \mathsf{tx}^{\mathsf{vc}}.\mathsf{output}$ 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 $\perp$
5) Return $(\mathsf{sk}_{\widetilde{U_i}}, \theta_{\epsilon_i}, R_i, U_{i+1}, \mathsf{onion}_{i+1})$
```

Subprocedures used exclusively in UC model

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

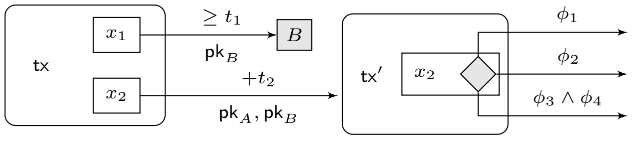


Figure 12: (Left) Transaction tx has two outputs, one of value x_1 that can be spent by B (indicated by the gray box) with a transaction signed w.r.t. pk_B at (or after) round t_1, and one of value x_2 that can be spent by a transaction signed w.r.t. pk_A and pk_B but only if at least t_2 rounds passed 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.

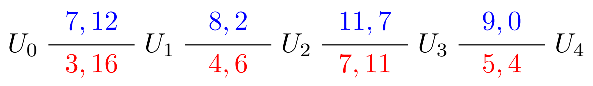


Figure 13: Example of a MHP in a PCN. Here, U_0 pays 4 coins (disregarding any fees) to U_4, via U_1, U_2 and U_3. The lines represent payment channels. We write balances as (x, y), where x is the balance of the user on the right, and y the balance of the user on the left. Above we write the channel balances before and below after the payment. In an MHP, this change of balance should happen atomically in every channel (or not at all).

```
1) For every $U_i \in \mathsf{nodeList} \setminus U_n$ do:
   • $(\mathsf{pk}_{\widetilde{U_i}}, R_i) \leftarrow \mathsf{GenPk}(U_i.A, U_i.B)$
   • $\mathsf{outputMap}(U_i) := (\epsilon, \mathsf{OneSig}(\mathsf{pk}_{\widetilde{U_i}}) \wedge \mathsf{RelTime}(t_c + \Delta))$
   • $\mathsf{rMap}(U_i) := R_i$
2) $\mathsf{rList} = \mathsf{rMap}.\mathtt{values}().\mathtt{shuffle}()$
3) Let $\theta_{\mathsf{vc}} := (\alpha, \mathsf{MultiSig}(U_0, U_n))$
4) $\mathsf{tx}^{\mathsf{vc}} := (\mathsf{tx}^{\mathsf{in}}.\mathsf{output}[0], [\theta_{\mathsf{vc}}, \mathsf{outputMap}.\mathtt{values}().\mathtt{shuffle}()])$
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 $\mathsf{msgList}, \mathsf{userList}$
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 $(\mathsf{tx}^{\mathsf{vc}}, \mathsf{onions}, \mathsf{rMap}, \mathsf{rList}, \mathsf{stealthMap})$
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$(\theta, \theta_{\epsilon_i}, U_i)$:
1) Create a transaction $\mathsf{tx}^{\mathsf{r}}_i$ with $\mathsf{tx}^{\mathsf{r}}_i.\mathsf{input} := [\theta, \theta_{\epsilon_i}]$ and $\mathsf{tx}^{\mathsf{r}}_i.\mathsf{output} := (\theta.\mathsf{cash} + \theta_{\epsilon_i}.\mathsf{cash}, \mathsf{OneSig}(U_i))$.
2) Return $\mathsf{tx}^{\mathsf{r}}_i$
genPayTx$(\theta, U_{i+1})$:
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 $\mathsf{tx}^{\mathsf{p}}_i$
```

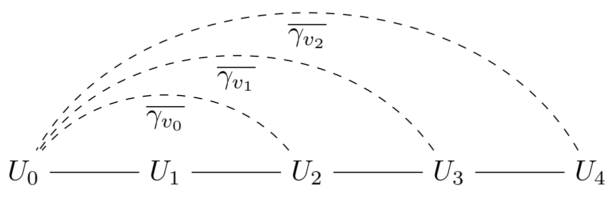
[a]. Possibly other outputs $\{\theta'_j\}_{j \geq 0}$ could also be present in this state. They, along with the off-chain objects there (e.g., other payments) would have to be recreated in the new state while adapting the index of the output these objects are referring to. For simplicity, we say this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

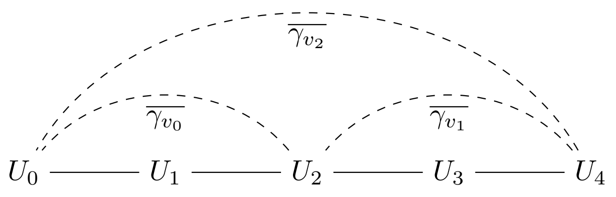
E.1. Example graphs for recursive VC

In this section, we show in Figure 14 and Figure 15 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.

Stealth addresses In order to hide the underlying path, we use stealth addresses [33] 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 [33].

Onion routing Like in the Lightning Network, we rely on onion routing [10] techniques like Sphinx [12] to allow users communicate anonymously with each other. This allows users to route the VC 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 the procedure $\mathsf{GetRoutingInfo}(\mathsf{onion}_i, U_i)$ returns the tuple $(U_{i+1}, \mathsf{msg}_i, \mathsf{onion}_{i+1})$ when called by the correct user U_i, or $\perp$ otherwise.


$\overline{\gamma_{v_2}}$
$\overline{\gamma_{v_1}}$
$\overline{\gamma_{v_0}}$
U_0
U_1
U_2
U_3
U_4
Figure 14: Recursive virtual channel: Example A


$\overline{\gamma_{v_2}}$
$\overline{\gamma_{v_0}}$
$\overline{\gamma_{v_1}}$
U_0
U_1
U_2
U_3
U_4
Figure 15: Recursive virtual channel: Example B

Appendix F. UC modeling

For our formal security analysis, we utilize the global UC framework (GUC) [11]. 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 [1, 2, 3, 16, 17, 18].

F.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 $\mathcal{G}_{clock}$ [23], 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 [16].

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.

F.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 $(\texttt{sid}, \texttt{REGISTER}, \mathsf{pk}_p)$ 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 $(\texttt{sid}, \texttt{POST}, \overline{\mathsf{tx}})$. 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 [4]. To increase readability, we opted for these simplifications.

Channels are handles by the functionality $\mathcal{F}_{Channel}$ [2], which is an extension of [3] 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

```
OpenVC

Setup

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

1) Let $n := |\mathsf{channelList}|$. If $\mathtt{checkTxIn}(\mathsf{tx}^{\mathsf{in}}, n, U_0) = \bot$ or $\mathtt{checkChannels}(\mathsf{channelList}, U_0) = \bot$ or $\mathtt{checkT}(n, T) = \bot$, abort. Else, let $\alpha_0 := \alpha + \mathsf{fee} \cdot (n-1)$
2) $(\mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}) := \mathtt{genTxVc}(U_0, \mathsf{channelList}, \mathsf{tx}^{\mathsf{in}})$
3) $\overline{\gamma_{\mathsf{vc}}} := \mathsf{preCreate}$ $(\mathsf{tx}^{\mathsf{vc}}, 0, U_0, U_n)$ together with $U_n$
4) $(\mathsf{sk}_{\widetilde{U_0}}, \theta_{\epsilon_0}, R_0, U_1, \mathsf{onion}_1) :=$
   $\mathtt{checkTxVc}(U_0, U_0.a, U_0.b, \mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion})$
5) $\mathtt{2pSetup}(\overline{\gamma_0}, \mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_1, U_1, \theta_{\epsilon_0}, \alpha_0, T)$

Open

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

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

Finalize

$U_0$: Upon $(\mathtt{confirm}, \sigma_{U_n}(\mathsf{tx}^{\mathsf{vc}})) \hookleftarrow U_n$, 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 $T - t_{\mathsf{c}} - 3\Delta$, $\mathsf{publishTx}(\mathsf{tx}^{\mathsf{vc}}, \sigma_{U'_0}(\mathsf{tx}^{\mathsf{vc}}))$.

UpdateVC

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 $\mathsf{preUpdate}(\overline{\gamma_{\mathsf{vc}}}, \mathsf{tx}_i^{\mathsf{state}})$.

CloseVC/ProlongVC (atomic modification)

InitClose/InitProlong

$U_n$: Let $\alpha'_i$ be the final balance of $U_n$ in the virtual channel and $T' = T$ (Close) or let $T' > T$ be the new lifetime of the VC and leave $\alpha'_i = \alpha_i$ (Prolong). Execute $\mathtt{2pModify}(\overline{\gamma_i}, \mathsf{tx}^{\mathsf{vc}}, \alpha', T')$
$U_{i-1}$ upon $(\top, \alpha'_i, T')$: If $U_{i-1} \neq U_0$, let $\alpha'_{i-1} := \alpha'_i + \mathsf{fee}$ and $\mathtt{2pModify}(\overline{\gamma_{i-2}}, \mathsf{tx}^{\mathsf{vc}}, \alpha'_{i-1}, T')$

Emergency-Offload

$U_0$: If $U_0$ has not successfully performed $\mathtt{2pModify}$ with the correct value $\alpha'$ (plus fee for each hop) until $T - t_{\mathsf{c}} - 3\Delta$, $\mathsf{publishTx}(\mathsf{tx}^{\mathsf{vc}}, \sigma_{U'_0}(\mathsf{tx}^{\mathsf{vc}}))$. Else, update $T := T'$

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 $t_{\mathsf{c}}$ rounds, wait $\Delta$ rounds. Let $\sigma_{\widetilde{U_i}}(\mathsf{tx}_i^{\mathsf{r}})$ be a signature using the secret key $\mathsf{sk}_{\widetilde{U_i}}$. $\mathsf{publishTx}(\mathsf{tx}_i^{\mathsf{r}}, (\sigma_{\widetilde{U_i}}(\mathsf{tx}_i^{\mathsf{r}}), \sigma_{U_i}(\mathsf{tx}_i^{\mathsf{r}}), \sigma_{U_{i+1}}(\mathsf{tx}_i^{\mathsf{r}})))$.
2) If $\tau_x > T$, $\overline{\gamma_{i-1}}$ 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 16: Pseudocode of the protocol.

message (msg) to X in round t and (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.

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

$U_i$

1) $\mathsf{tx}_i^{\mathsf{state}} := \mathtt{genState}(\alpha_i, T, \overline{\gamma_i})$
2) $\mathsf{tx}_i^{\mathsf{r}} := \mathtt{genRef}(\mathsf{tx}_i^{\mathsf{state}}, \theta_{\epsilon_i})$
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 $\mathtt{checkTxVc}(U_{i+1}, U_{i+1}.a, U_{i+1}.b, \mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_{i+1}) \neq \bot$, but returns some values $(\mathsf{sk}_{\widetilde{U_{i+1}}}, \theta_{\epsilon_{i+1}}, R_{i+1}, U_{i+2}, \mathsf{onion}_{i+2})$
5) Extract $\alpha_i$ and $T$ from $\mathsf{tx}^{\mathsf{state}}$ and check $\mathsf{tx}_i^{\mathsf{state}} = \mathtt{genState}(\alpha_i, T, \overline{\gamma_i})$
6) Check that for one output $\theta_{\epsilon_x} \in \mathsf{tx}^{\mathsf{vc}}.\mathsf{output}$ it holds that $\mathsf{tx}_i^{\mathsf{r}} := \mathtt{genRef}(\mathsf{tx}_i^{\mathsf{state}}, \theta_{\epsilon_x})$. If one of these previous checks failed, return $\bot$.
7) $\mathsf{tx}_i^{\mathsf{p}} := \mathtt{genPay}(\mathsf{tx}_i^{\mathsf{state}})$
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 $\bot$.
10) $\mathsf{updateChannel}(\overline{\gamma_i}, \mathsf{tx}_i^{\mathsf{state}})$
11) If, after $t_{\mathsf{u}}$ time has expired, the message $(\mathsf{update{-}ok})$ is returned, return $\top$. Else return $\bot$.

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

$\mathtt{2pModify}(\overline{\gamma_i}, \mathsf{tx}^{\mathsf{vc}}, \alpha'_i, T')$

Let $T$ be the timeout, $\alpha_i$ the amount and $\theta_{\epsilon_{i-1}}$ be the output used for the two party contract set up between $U_{i-1}$ and $U_i$, known from $\mathtt{2pSetup}$ executed in the Open [1] phase.

$U_i$

1) $\mathsf{tx}_{i-1}^{\mathsf{state}'} := \mathtt{genState}(\alpha'_i, T', \overline{\gamma_{i-1}})$
2) $\mathsf{tx}_{i-1}^{\mathsf{r}'} := \mathtt{genRef}(\mathsf{tx}_{i-1}^{\mathsf{state}'}, \theta_{\epsilon_{i-1}})$ $//\theta_{\epsilon_{i-1}}$ known as $\theta_{\epsilon_x}$ from $\mathtt{2pSetup}$
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 $\alpha'_i$ and $T'$ from $\mathsf{tx}_{i-1}^{\mathsf{state}'}$ and check that $\alpha'_i \leq \alpha_i$, $T' \geq T$ and $\mathsf{tx}_{i-1}^{\mathsf{state}'} = \mathtt{genState}(\alpha'_i, T', \overline{\gamma_{i-1}})$ $//\alpha_i$ and $T$ from $\mathtt{2pSetup}$
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 $\mathsf{tx}_{i-1}^{\mathsf{r}'} = \mathtt{genRef}(\mathsf{tx}_{i-1}^{\mathsf{state}'}, \theta_{\epsilon_{i-1}})$ $//\theta_{\epsilon_{i-1}}$ from $\mathtt{2pSetup}$
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 $t_{\mathsf{u}}$ 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 $(\top, \alpha'_i, T')$.
7) Else, return $\bot$.

$U_i$: 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 $\mathsf{tx}_{i-1}^{\mathsf{p}'} := \mathtt{genPay}(\mathsf{tx}_{i-1}^{\mathsf{state}'})$, respectively.
```

Figure 17: Protocol for 2-party channel update.

Interface of $\mathcal{F}_{Channel}(T, k)$ [3]

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:

- $(\texttt{sid}, \texttt{CREATE}, \overline{\gamma}, \mathsf{tid}_P) \overset{\tau}{\hookleftarrow} P$:
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.
- $(\texttt{sid}, \texttt{UPDATE}, \mathsf{id}, \vec{\theta}) \overset{\tau}{\hookleftarrow} P$:
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, $(\texttt{sid}, \texttt{UPDATED}, \mathsf{id}, \vec{\theta}) \xhookrightarrow{\leq \tau + t_\mathsf{u}} \overline{\gamma}.\mathsf{users}$ is output.
- $(\texttt{sid}, \texttt{CLOSE}, \mathsf{id}) \overset{\tau}{\hookleftarrow} P$:
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 $\mathcal{G}_{Ledger}$).

- $(\texttt{sid}, \texttt{PRE-CREATE}, \overline{\gamma}, \mathsf{tx}^\mathsf{f}, i, t_\mathsf{ofl}) \overset{\tau}{\hookleftarrow} P$:
Does the same as $\texttt{CREATE}$, 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 $(\texttt{sid}, \texttt{PRE-CREATED}, \overline{\gamma}.\mathsf{id})$ after at most t_o rounds.
- $(\texttt{sid}, \texttt{PRE-UPDATE}, \mathsf{id}, \vec{\theta}) \overset{\tau}{\hookleftarrow} P$:
Does the same as $\texttt{UPDATE}$ 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 $\mathcal{G}_{Ledger}(\Delta, \Sigma)$ [3]

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 $P \in \mathcal{P}$:

- $(\texttt{sid}, \texttt{REGISTER}, \mathsf{pk}_P) \overset{\tau}{\hookleftarrow} P$:
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.
- $(\texttt{sid}, \texttt{POST}, \overline{\mathsf{tx}}) \overset{\tau}{\hookleftarrow} P$:
This function checks if $\overline{\mathsf{tx}}$ is a valid transaction and if yes, accepts it on $\mathcal{L}$ after at most Δ rounds.

F.2.1. The UC-security definition. Closely following [1], 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 $\texttt{EXEC}^{\mathcal{F}_\mathsf{prelim}}_{\Pi,\mathcal{A},\mathcal{E}}(\lambda, z)$. 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 $\texttt{EXEC}^{\mathcal{F}_\mathsf{prelim}}_{\phi_{\mathcal{F}_{VC}},\mathcal{S},\mathcal{E}}(\lambda, z)$.

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

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

where $\approx^c$ denotes computational indistinguishability.

F.3. 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 $\texttt{ERROR}$ 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 $\texttt{ERROR}$. The cases where $\texttt{ERROR}$ is output are not meaningful to us and any guarantees are lost. We use the extended macros defined in Appendix E. 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 [1]. To be transparent about the similarities to [1] and highlight the novelties of this work, we mark the latter in green in the ideal functionality, formal UC protocol and simulator.

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 tx^vc, the onions, the stealth addresses, etc. and calls $\texttt{PRE-CREATE}$ 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 $\texttt{PRE-UPDATE}$ 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 $(\texttt{pid}, \overline{\gamma_i}, \vec{\theta}_i, \mathsf{tx}^{\mathsf{vc}}, T, \theta_{\epsilon_i}, R_i)$ 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 $\overline{\gamma_i}$.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.

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

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.

Open-VC

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 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 $\texttt{checkT}(n, T) = \bot$, 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 $(\texttt{ssid}_C, \texttt{PRE-CREATE}, \overline{\gamma_{\mathsf{vc}}}, \mathsf{tx}^{\mathsf{vc}}, 0, T - \tau) \xrightarrow{\tau+1} \mathcal{F}_{Channel}$
8) If not $(\texttt{ssid}_C, \texttt{PRE-CREATED}, \overline{\gamma_{\mathsf{vc}}}.\mathsf{id}) \xleftarrow{\tau+1+t_{\mathsf{o}}} \mathcal{F}_{Channel}$, 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 $\theta_{\epsilon_0} := \mathsf{stealthMap}(U_0)$. Send $(\texttt{sid}, \texttt{pid}, \texttt{open}, \mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_1, \alpha_0, T, \bot, \overline{\gamma_0}, \bot, \theta_{\epsilon_0}) \xhookrightarrow{\tau+1+t_{\mathsf{o}}} \mathcal{S}$.

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

VC-Open: //Mark VC as opened

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

Register: //Sim. can register a channel

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

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 $\theta_{\epsilon_i} := \mathsf{stealthMap}(U_i)$
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) $(\texttt{sid}, \texttt{pid}, \texttt{OPEN}, \mathsf{tx}^{\mathsf{vc}}, \theta_{\epsilon_i}, R_i, U_{i-1}, U_{i+1}, \alpha_{i-1}, T) \xhookrightarrow{\tau+1} U_i$
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) $\Gamma := \Gamma \cup (\texttt{pid}, \overline{\gamma_i}, \vec{\theta}_i, \mathsf{tx}^{\mathsf{vc}}, T, \theta_{\epsilon_i}, R_i)$
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 $(\texttt{sid}, \texttt{pid}, \texttt{open}, \mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_{i+1}, \alpha_{i-1} - \mathsf{fee}, T, \overline{\gamma_{i-1}}, \overline{\gamma_i}, \theta_{\epsilon_{i-1}}, \theta_{\epsilon_i}) \overset{\tau}{\hookrightarrow} \mathcal{S}$, where $\mathsf{onion}_{i+1} := \mathsf{onions}(U_{i+1})$ and $\theta_{\epsilon_{i-1}} := \mathsf{stealthMap}U_{i-1}$

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

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

1) Let $(\tau_{\mathsf{f}}, \mathsf{tx}^{\mathsf{vc}}, U_n) = \Phi(\texttt{pid}, U_0)$. 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 $(\texttt{sid}, \texttt{pid}, \texttt{confirmed}, \mathsf{tx}^{\mathsf{er}}_x, \sigma_{U_n}(\mathsf{tx}^{\mathsf{er}}_x)) \overset{\tau_{\mathsf{f}}}{\hookleftarrow} \mathcal{S}$, such that $\mathsf{tx}^{\mathsf{er}}_x = \mathsf{tx}^{\mathsf{vc}}$ and $\sigma_{U_n}(\mathsf{tx}^{\mathsf{er}}_x)$ is U_n's valid signature of $\mathsf{tx}^{\mathsf{vc}}$, go idle.
4) Send $(\texttt{sid}, \texttt{pid}, \texttt{offload}, \mathsf{tx}^{\mathsf{vc}}, U_0) \overset{\tau_{\mathsf{f}}}{\hookrightarrow} \mathcal{S}$ 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 $(\texttt{sid}, \texttt{ERROR}) \overset{t_1}{\hookrightarrow} U_0$.

<u>Update-VC</u>

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.

<u>Close-VC</u>

Let τ be the current round.
Start:

1) Upon $(\texttt{sid}, \texttt{pid}, \texttt{SHUTDOWN}, \alpha'_{n-1}) \overset{\tau}{\hookleftarrow} U_n$, for parameter $\underline{\texttt{pid}}$, fetch entry $(\texttt{pid}, \overline{\gamma_{n-1}}, \vec{\theta}_i, \mathsf{tx}^{\mathsf{vc}}, T, \theta_{\epsilon_i}, R_i)$ 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 $(\texttt{sid}, \texttt{pid}, \texttt{close}, \alpha'_{n-1}, \overline{\gamma_{n-1}}) \overset{\tau}{\hookrightarrow} \mathcal{S}$

Continue-Close: //Continue after a dishonest user

1) Upon $(\texttt{sid}, \texttt{pid}, \texttt{continue-close}, \overline{\gamma_{i-1}}, \alpha'_{i-1}) \overset{\tau}{\hookleftarrow} \mathcal{S}$
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 $(\texttt{pid}, \overline{\gamma_i}, \vec{\theta}_i, \mathsf{tx}^{\mathsf{vc}}, T, \theta_{\epsilon_i}, R_i)$ from Γ. If there is no entry where the parameters $\texttt{pid}$ and $\overline{\gamma_i}$ match, go idle.
2) If $\overline{\gamma_i}.\mathsf{st} \neq \vec{\theta}_i$, go idle.
3) Let $\alpha_i := \vec{\theta}_i[0].\mathsf{cash}$. If not $0 \leq \alpha'_i \leq \alpha_i$, go idle.
4) $\vec{\theta}'_i := \texttt{genNewState}(\overline{\gamma_i}, \alpha'_i, T)$. If $\vec{\theta}'_i = \bot$, go idle. Else, wait 1 round.
5) $(\texttt{sid}, \texttt{pid}, \texttt{CLOSE}, \alpha'_i) \xhookrightarrow{\tau+1} U_i$
6) If not $(\texttt{sid}, \texttt{pid}, \texttt{CLOSE-ACCEPT}) \xhookleftarrow{\tau+1} U_i$, go idle.
7) $(\texttt{ssid}_C, \texttt{UPDATE}, \overline{\gamma_i}.\mathsf{id}, \vec{\theta}'_i) \xhookrightarrow{\tau+1} \mathcal{F}_{Channel}$
8) If not $(\texttt{ssid}_C, \texttt{UPDATED}, \overline{\gamma_i}.\mathsf{id}) \xhookleftarrow{\tau+1+t_{\mathsf{u}}} \mathcal{F}_{Channel}$, go idle.
9) $\Gamma := \Gamma \setminus (\texttt{pid}, \overline{\gamma_i}, \vec{\theta}_i, \mathsf{tx}^{\mathsf{vc}}, T, \theta_{\epsilon_i}, R_i)$
10) $\Gamma := \Gamma \cup (\texttt{pid}, \overline{\gamma_i}, \vec{\theta}'_i, \mathsf{tx}^{\mathsf{vc}}, T, \theta_{\epsilon_i}, R_i)$
11) If $U_i = U_0$:
    - $(\texttt{sid}, \texttt{pid}, \texttt{VC-CLOSED}) \xhookrightarrow{\tau+1+t_{\mathsf{u}}} U_i$
    - Remove key and value for key $(\texttt{pid}, U_0)$ from Φ.
12) Else:
    - Retrieve $\overline{\gamma_{i-1}}$ from Γ matching $\texttt{pid}$ and s.t. $\overline{\gamma_{i-1}}.\mathsf{right} = U_i$
    - $(\texttt{sid}, \texttt{pid}, \texttt{CLOSED}) \xhookrightarrow{\tau+1+t_{\mathsf{u}}} U_i$
    - If U_{i-1} honest, execute **Close**$(\texttt{pid}, \alpha'_i + \mathsf{fee}, \overline{\gamma_{i-1}})$
    - Else, send $(\texttt{sid}, \texttt{pid}, \texttt{close}, \alpha'_i + \mathsf{fee}, \overline{\gamma_{i-1}}) \overset{\tau}{\hookrightarrow} \mathcal{S}$.

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

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

<u>Emergency-Offload (executed at every round)</u>

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

1) For $\texttt{pid}$ and a channel $\overline{\gamma_0}$ where $\overline{\gamma_0}.\mathsf{left} = U_0$, fetch entry $(\texttt{pid}, \overline{\gamma_0}, \vec{\theta}_0, \mathsf{tx}^{\mathsf{vc}}, T, \theta_{\epsilon_0}, R_0) \in \Gamma$
2) If $\tau < T - t_{\mathsf{c}} - 3\Delta$, continue with next loop iteration.
3) Else, let $(\tau_{\mathsf{f}}, \mathsf{tx}^{\mathsf{vc}}, U_n) = \Phi(\texttt{pid}, U_0)$. Send $(\texttt{sid}, \texttt{pid}, \texttt{offload}, \mathsf{tx}^{\mathsf{vc}}, U_0) \overset{\tau}{\hookrightarrow} \mathcal{S}$. $\mathsf{tx}^{\mathsf{vc}}$ must be on $\mathcal{L}$ in round $\tau' \leq \tau + \Delta$. Otherwise, output $(\texttt{sid}, \texttt{ERROR}) \overset{t_1}{\hookrightarrow} U_0$.

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 $(\mathtt{pid}, \overline{\gamma_i}, \vec{\theta}_i, \mathsf{tx}^{\mathsf{vc}}, T, \theta_{\epsilon_i}, R_i) \in \Gamma$, check if $\overline{\gamma_i}.\mathsf{st} = \vec{\theta}_i$ and $\mathsf{tx}^{\mathsf{vc}}$ is on $\mathcal{L}$. If yes, do the following:

Revoke: If γ_i.left honest and $t < T - t_{\mathsf{c}} - 2\Delta$ do the following.

- Set $\Gamma := \Gamma \setminus \{(\mathtt{pid}, \overline{\gamma_i}, \vec{\theta}_i, \mathsf{tx}^{\mathsf{vc}}, T, \theta_{\epsilon_i}, R_i)\}$.
- $(\mathtt{ssid}_C, \mathtt{CLOSE}, \overline{\gamma_i}.\mathsf{id}) \overset{t}{\hookrightarrow} \mathcal{F}_{Channel}$
- At time $t + t_{\mathsf{c}}$, 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 $(\mathtt{sid}, \mathtt{ERROR}) \overset{T}{\hookrightarrow} \gamma_i$.users.
- Wait for Δ rounds, then $(\mathtt{sid}, \mathtt{pid}, \mathtt{post\text{-}refund}, \overline{\gamma_i}, \theta_{\epsilon_i}, R_i) \xhookrightarrow{t' < T - \Delta} \mathcal{S}$
- At time $t'' < T$, check whether a transaction tx$'$ appears on $\mathcal{L}$ with tx$'$.input $= [\theta_{\epsilon_i}, tx.\mathsf{output}[0]]$ and tx$'$.output $= [(\mathsf{tx.output}[0].\mathsf{cash} + \theta_{\epsilon_i}.\mathsf{cash}, \mathsf{OneSig}(U_i))]$. If it appears, send $(\mathtt{sid}, \mathtt{pid}, \mathtt{REVOKED}) \overset{t''}{\hookrightarrow} \overline{\gamma_i}$.left. If not, send $(\mathtt{sid}, \mathtt{ERROR}) \overset{T}{\hookrightarrow} \gamma_i$.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 $\Gamma := \Gamma \setminus \{(\mathtt{pid}, \overline{\gamma_i}, \vec{\theta}_i, \mathsf{tx}^{\mathsf{vc}}, T, \theta_{\epsilon_i}, R_i)\}$.
- Send $(\mathtt{sid}, \mathtt{pid}, \mathtt{post\text{-}pay}, \overline{\gamma_i}) \overset{t}{\hookrightarrow} \mathcal{S}$
- In round $t + \Delta$ transaction tx$'$ with tx$'$.input = [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, $(\mathtt{sid}, \mathtt{pid}, \mathtt{FORCE\text{-}PAY}) \xhookrightarrow{t+\Delta} \overline{\gamma_i}$.right. Otherwise, $(\mathtt{sid}, \mathtt{ERROR}) \xhookrightarrow{t+\Delta} \gamma_i$.users.

F.4. 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[3] 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 5. 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 [1].

3. 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 $\mathsf{tx}^{\mathsf{r}}_i$, but we omit this for readability.

Protocol Π

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

pidSet :	A set storing every payment id $\mathtt{pid}$ that a user has participated in to prevent duplicates.
paySet :	A map storing tuples $(\mathtt{pid}, \tau_{\mathsf{f}}, U_n)$ where $\mathtt{pid}$ 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 $\mathtt{pid}$ 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 $\mathtt{pid}$ 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 $\mathtt{pid}$.
right :	A map, sotring for a given $\mathtt{pid}$ 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 $\mathtt{pid}$.
rightSig :	A map, storing for a given $\mathtt{pid}$ the signature for $\mathsf{tx}^{\mathsf{r}}_i$ of the right neighbor $\sigma_{U_{i+1}}(\mathsf{tx}^{\mathsf{r}}_i)$ in the payment $\mathtt{pid}$.

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 $(\mathtt{sid}, \mathtt{pid}, \mathtt{SETUP}, \mathsf{channelList}, \mathsf{tx}^{\mathsf{in}}, \alpha, T, \overline{\gamma_0}) \xleftarrow{\tau_0} \mathcal{E}$

1) If $\mathtt{pid} \in$ pidSet, abort. Add $\mathtt{pid}$ to pidSet.
2) Let $x := \mathtt{checkChannels}(\mathsf{channelList}, U_0)$. 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 $U_1 := \overline{\gamma_0}$.right (corresponding to the channel skeleton γ_0 in channelList), go idle. Let nodeList be a list of all the users on the path sorted from U_0 to U_n.
3) Let $n := |\mathsf{channelList}|$. If $\mathtt{checkT}(n, T) = \bot$, abort.
4) If $\mathtt{checkTxIn}(\mathsf{tx}^{\mathsf{in}}, n, U_0, \alpha) = \bot$, abort
5) $(\mathsf{tx}^{\mathsf{vc}}, \mathsf{onions}, \mathsf{rMap}, \mathsf{rList}, \mathsf{stealthMap}) := \mathtt{createMaps}(U_0, \mathsf{nodeList}, \mathsf{tx}^{\mathsf{in}}, \alpha)$.
6) $(\mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_0) := \mathtt{genTxVc}(U_0, \mathsf{channelList}, \mathsf{tx}^{\mathsf{in}})$
7) paySet := paySet$\cup\{(\mathtt{pid}, \tau_{\mathsf{f}} := \tau + n \cdot (2 + t_{\mathsf{u}}) + 2 + t_{\mathsf{o}}, U_n)\}$
8) $(\mathsf{sk}_{\widetilde{U_0}}, \theta_{\epsilon_0}, R_0, U_1, \mathsf{onion}_1) := \mathtt{checkTxVc}(U_0, U_0.a, U_0.b, \mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_0)$
9) Set local$(\mathtt{pid}) := (\mathsf{tx}^{\mathsf{vc}}, T)$.
10) Send $(\mathtt{sid}, \mathtt{pid}, \mathtt{pre\text{-}create\text{-}vc}, \overline{\gamma_{\mathsf{vc}}}, \mathsf{tx}^{\mathsf{vc}}, T) \overset{\tau_0}{\hookrightarrow} U_n$, wait 1 round.
11) Send $(\mathtt{ssid}_C, \mathtt{PRE\text{-}CREATE}, \overline{\gamma_{\mathsf{vc}}}, \mathsf{tx}^{\mathsf{vc}}, 0, T - \tau_0) \xhookrightarrow{\tau_0 + 1} \mathcal{F}_{Channel}$
12) If not $(\mathtt{ssid}_C, \mathtt{PRE\text{-}CREATED}, \overline{\gamma_{\mathsf{vc}}}.\mathsf{id}) \xhookleftarrow{\tau_0 + 1 + t_{\mathsf{o}}} \mathcal{F}_{Channel}$, go idle.
13) Set $\alpha_0 := \alpha + \mathsf{fee} \cdot (n - 1)$ and compute:
    - $\vec{\theta}_0 := \mathtt{genStateOutputs}(\overline{\gamma_0}, \alpha_0, T)$
    - $\mathsf{tx}^{\mathsf{r}}_0 := \mathtt{genRefTx}(\vec{\theta}_0, \theta_{\epsilon_0}, U_0)$
14) Set right$(\mathtt{pid}) := (\overline{\gamma_0}, \vec{\theta}_0, \mathsf{tx}^{\mathsf{r}}_0, \mathsf{sk}_{\widetilde{U_0}})$.
15) Send $(\mathtt{sid}, \mathtt{pid}, \mathtt{open\text{-}req}, \mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_1, \vec{\theta}_0, \mathsf{tx}^{\mathsf{r}}_0) \xhookrightarrow{\tau_0 + 1 + t_{\mathsf{o}}} U_1$.

U_n upon $(\mathtt{sid}, \mathtt{pid}, \mathtt{pre\text{-}create\text{-}vc}, \overline{\gamma_{\mathsf{vc}}}, \mathsf{tx}^{\mathsf{vc}}, T) \xhookleftarrow{\tau} U_0$

1) $(\mathtt{ssid}_C, \mathtt{PRE\text{-}CREATE}, \overline{\gamma_{\mathsf{vc}}}, \mathsf{tx}^{\mathsf{vc}}, 0, T-\tau) \xhookrightarrow{\tau} \mathcal{F}_{Channel}$
2) If not $(\mathtt{ssid}_C, \mathtt{PRE\text{-}CREATED}, \overline{\gamma_{\mathsf{vc}}}.\mathsf{id}) \xhookleftarrow{\tau+t_o} \mathcal{F}_{Channel}$, 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} upon

$(\mathtt{sid}, \mathtt{pid}, \mathtt{open\text{-}req}, \mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_{i+1}, \vec{\theta}_i, \mathsf{tx}^{\mathsf{r}}_i) \xhookleftarrow{\tau_x} U_i$

1) Perform the following checks:
   - Verify that $\mathtt{pid} \notin \mathsf{pidSet}$. Add $\mathtt{pid}$ to pidSet
   - Let $x := \mathtt{checkTxVc}(U_{i+1}, U_{i+1}.a, U_{i+1}.b, \mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_{i+1})$.
     Check that $x \neq \bot$, but instead $x = (\mathsf{sk}_{\widetilde{U_{i+1}}}, \theta_{\epsilon_{i+1}}, R_{i+1}, U_{i+2}, \mathsf{onion}_{i+2})$.
   - 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 $\vec{\theta}_i = \mathtt{genStateOutputs}(\overline{\gamma_i}, \alpha_i, T)$.
   - Check that $\mathsf{tx}^{\mathsf{r}}_i := \mathtt{genRefTx}(\vec{\theta}_i, \theta_{\epsilon_x}, U_i)$, where θ_{ϵ_x} is an output of $\mathsf{tx}^{\mathsf{vc}}$, s.t. $\theta_{\epsilon_x} \neq \theta_{\epsilon_{i+1}}$.
2) If one or more of the previous checks fail, abort. Otherwise, send $(\mathtt{sid}, \mathtt{pid}, \mathtt{OPEN}, \mathsf{tx}^{\mathsf{vc}}, \theta_{\epsilon_{i+1}}, R_i, U_i, U_{i+2}, \alpha_i, T) \xhookrightarrow{\tau_x} \mathcal{E}$.
3) If $(\mathtt{sid}, \mathtt{pid}, \mathtt{OPEN\text{-}ACCEPT}, \overline{\gamma_{i+1}}) \xhookleftarrow{\tau_x} \mathcal{E}$, generate $\sigma_{U_{i+1}}(\mathsf{tx}^{\mathsf{r}}_i)$. Otherwise stop.
4) Set $\mathsf{local}(\mathtt{pid}) := (\mathsf{tx}^{\mathsf{er}}_i, T)$, $\mathsf{left}(\mathtt{pid}) := (\overline{\gamma_i}, \vec{\theta}_i, \mathsf{tx}^{\mathsf{r}}_i)$ and $(\mathtt{sid}, \mathtt{pid}, \mathtt{open\text{-}ok}, \sigma_{U_{i+1}}(\mathsf{tx}^{\mathsf{r}}_i)) \xhookrightarrow{\tau_x} U_i$.

U_i upon $(\mathtt{sid}, \mathtt{pid}, \mathtt{open\text{-}ok}, \sigma_{U_{i+1}}(\mathsf{tx}^{\mathsf{r}}_i)) \xhookleftarrow{\tau_i+2} U_{i+1}$

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

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

6) Define $\tau_{(i+1)} := \tau_x + 1 + t_{\mathsf{u}}$.
7) If U_{i+1} is not the receiver, using the values of step 1:
   - Send $(\mathtt{sid}, \mathtt{pid}, \mathtt{OPENED}) \xhookrightarrow{\tau_{i+1}} \mathcal{E}$.
   - $(\mathsf{sk}_{\widetilde{U_{i+1}}}, \theta_{\epsilon_{i+1}}, R_{i+1}, U_{i+2}, \mathsf{onion}_{i+2}) := \mathtt{checkTxVc}(U_{i+1}, U_{i+1}.a, U_{i+1}.b, \mathsf{tx}^{\mathsf{er}}_i, \mathsf{rList}, \mathsf{onion}_{i+1})$
   - $\vec{\theta}_{i+1} := \mathtt{genStateOutputs}(\overline{\gamma_{i+1}}, \alpha_i - \mathsf{fee}, T)$
   - $\mathsf{tx}^{\mathsf{r}}_{i+1} := \mathtt{genRefTx}(\vec{\theta}_{i+1}, \theta_{\epsilon_{i+1}}, U_{i+1})$
   - Set $\mathsf{right}(\mathtt{pid}) := (\overline{\gamma_{i+1}}, \vec{\theta}_{i+1}, \mathsf{tx}^{\mathsf{r}}_{i+1}, \mathsf{sk}_{\widetilde{U_{i+1}}})$
   - Send the message $(\mathtt{sid}, \mathtt{pid}, \mathtt{open\text{-}req}, \mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_{i+2}, \vec{\theta}_{i+1}, \mathsf{tx}^{\mathsf{r}}_{i+1}) \xhookrightarrow{\tau_{i+1}} U_{i+2}$.
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 $(\mathtt{sid}, \mathtt{pid}, \mathtt{finalize}, \mathsf{tx}^{\mathsf{vc}}, \sigma_{U_n}(\mathsf{tx}^{\mathsf{vc}})) \xhookleftarrow{\tau} U_n$, 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 $\mathtt{PRE\text{-}UPDATE}$ 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 $(\mathtt{sid}, \mathtt{pid}, \mathtt{SHUTDOWN}, \alpha'_{n-1}) \xhookleftarrow{\tau_n} \mathcal{E}$

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 $\theta_{\epsilon_{n-1}} \in \mathsf{tx}^{\mathsf{r}}_{n-1}.\mathsf{input}$
5) Extract T from $\vec{\theta}_{n-1}[0].\phi$
6) Let $\alpha_i := \vec{\theta}_{n-1}[0].\mathsf{cash}$. If not $0 \leq \alpha'_i \leq \alpha_i$, abort. Compute:
   - $\vec{\theta}'_{n-1} := \mathtt{genNewState}(\overline{\gamma_{n-1}}, \alpha'_{n-1}, T)$
   - $\mathsf{tx}^{\mathsf{r}'}_{n-1} := \mathtt{genRefTx}(\vec{\theta}'_{n-1}, \theta_{\epsilon_{n-1}}, U_{n-1})$
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 $(\mathtt{sid}, \mathtt{pid}, \mathtt{close\text{-}req}, \vec{\theta}'_i, \mathsf{tx}^{\mathsf{r}'}_i, \sigma_{U_{i+1}}(\mathsf{tx}^{\mathsf{r}'}_i)) \xhookleftarrow{\tau_x} U_{i+1}$

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 $\overline{\gamma_i}.\mathsf{right} \neq U_{i+1}$, abort. If $\vec{\theta}_i[0] \notin \mathsf{tx}^{\mathsf{r}}_i.\mathsf{input}$, abort.
4) Extract $\theta_{\epsilon_i} \in \mathsf{tx}^{\mathsf{r}}_i.\mathsf{input}$
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 $\mathsf{tx}^{\mathsf{r}'}_i \neq \mathtt{genRefTx}(\vec{\theta}'_i, \theta_{\epsilon_i}, U_i)$, 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 $(\mathtt{sid}, \mathtt{pid}, \mathtt{CLOSE\text{-}ACCEPT}) \xhookleftarrow{\tau_x} \mathcal{E}$, 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 $(\mathtt{ssid}_C, \mathtt{UPDATED}, \overline{\gamma_i}.\mathsf{id}, \vec{\theta}'_i) \xhookleftarrow{\tau_i+1+t_{\mathsf{u}}} \mathcal{F}_{Channel}$

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

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

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 $\theta_{\epsilon_{i-1}} \in \mathsf{tx}^{\mathsf{r}}_{i-1}.\mathsf{input}$
- $\vec{\theta}'_{i-1} := \mathtt{genNewState}(\overline{\gamma_{i-1}}, \alpha'_i + \mathsf{fee}, T)$
- $\mathsf{tx}^{\mathsf{r}'}_{i-1} := \mathtt{genRefTx}(\vec{\theta}'_{i-1}, \theta_{\epsilon_{i-1}}, U_{i-1})$
- 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:
   - $(\mathtt{sid}, \mathtt{pid}, \mathtt{VC\text{-}CLOSED}) \xhookrightarrow{\tau_i} \mathcal{E}$

<u>Emergency-Offload</u>

<u>U_0 in every round τ</u>

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

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 $(\mathtt{ssid}_L, \mathtt{POST}, \overline{\mathsf{tx}^{\mathsf{vc}}}) \xhookrightarrow{\tau} \mathcal{G}_{Ledger}$.

<u>Respond</u>

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

Let t be the current round. Do the following:

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

F.5. 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 F.4) UC-realizes the ideal functionality $\mathcal{F}_{VC}$ shown in Appendix F.3.

Simulator

Local variables:

left	A map, storing the channel $\overline{\gamma_{i-1}}$ and output $\theta_{\epsilon_{i-1}}$ for a given keypair consisting of a payment id $\mathtt{pid}$ and a user U_i, or $(\bot, \bot)$ if U_i is the sending endpoint.
right	A map, storing the transaction $\mathsf{tx}^{\mathsf{r}}_i$ for a given keypair consisting of a payment id $\mathtt{pid}$ and a user U_i.
$\mathsf{rightSig}$	A map, storing the signature of the right neighbor for the transaction stored in right for a given keypair consisting of a payment id $\mathtt{pid}$ and a user U_i.

Simulator for init phase

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

Simulator for Open-VC phase

<u>Pre-create VC</u>

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

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

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

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

1) Upon $(\mathtt{sid}, \mathtt{pid}, \mathtt{open\text{-}req}, \mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_i, \vec{\theta}_{i-1}, \mathsf{tx}^{\mathsf{r}}_{i-1})$ $\overset{\tau}{\hookleftarrow} U_{i-1}$.
   Let $\alpha_{i-1} := \vec{\theta}_{i-1}[0].\mathsf{cash}$ and extract T from $\vec{\theta}_{i-1}[0].\phi$ (the

parameter of $\mathsf{AbsTime}()$). 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 $x = (\mathsf{sk}_{\widetilde{U_i}}, \theta_{\epsilon_i}, R_i, U_{i+1}, \mathsf{onion}_{i+1})$. 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 $\mathsf{tx}^{\mathsf{r}}_i := \texttt{genRefTx}(\vec{\theta}_{i-1}, \theta_{\epsilon_{i-1}}, U_i)$, where $\theta_{\epsilon_{i-1}} \in \mathsf{tx}^{\mathsf{vc}}$ and $\theta_{\epsilon_{i-1}} \neq \theta_{\epsilon_i}$.
4) $(\texttt{sid}, \texttt{pid}, \texttt{check-id}, \mathsf{tx}^{\mathsf{vc}}, \theta_{\epsilon_i}, R_i, U_{i-1}, U_i, U_{i+1}, \alpha_i, T) \overset{\tau}{\hookrightarrow} \mathcal{F}_{VC}$
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 $\mathsf{tx}^{\mathsf{r}}_{i-1}$ on behalf of U_i yielding $\sigma_{U_i}(\mathsf{tx}^{\mathsf{r}}_{i-1})$ and $(\texttt{sid}, \texttt{pid}, \texttt{open-ok}, \sigma_{U_i}(\mathsf{tx}^{\mathsf{r}}_{i-1})) \overset{\tau}{\hookrightarrow} U_{i-1}$.
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 $\mathsf{left}(\texttt{pid}, U_i) := (\overline{\gamma_{i-1}}, \theta_{\epsilon_{i-1}})$.
9) If $U_i = U_n$ (if $(\mathsf{sk}_{\widetilde{U_i}}, \theta_{\epsilon_i}, R_i, U_{i+1}, \mathsf{onion}_{i+1}) = (\top, \top, \top, \top, \top)$ holds), and U_0 is honest,[a] send $(\texttt{sid}, \texttt{pid}, \texttt{vc-open}, \mathsf{tx}^{\mathsf{vc}}) \xhookrightarrow{\tau+1+t_{\mathsf{u}}} \mathcal{F}_{VC}$. If U_0 is dishonest, create signature $\sigma_{U_n}(\mathsf{tx}^{\mathsf{vc}})$ on behalf of U_n and send $(\texttt{sid}, \texttt{pid}, \texttt{finalize}, \mathsf{tx}^{\mathsf{vc}}, \sigma_{U_n}(\mathsf{tx}^{\mathsf{vc}})) \xhookrightarrow{\tau+1+t_{\mathsf{u}}} U_0$. In both cases, send via $\mathcal{F}_{VC}$ to the dummy user U_n the message $(\texttt{sid}, \texttt{pid}, \texttt{VC-OPENED}, \mathsf{tx}^{\mathsf{vc}}, T, \alpha_{i-1}) \xhookrightarrow{\tau+1+t_{\mathsf{u}}} U_n$. Go Idle.
10) Send via $\mathcal{F}_{VC}$ to the dummy user U_i the message $(\texttt{sid}, \texttt{pid}, \texttt{OPENED}) \xhookrightarrow{\tau+1+t_{\mathsf{u}}} U_i$.
11) If U_{i+1} honest, call $\mathsf{process}(\texttt{sid}, \texttt{pid}, \mathsf{tx}^{\mathsf{vc}}, \overline{\gamma_{i-1}}, \overline{\gamma_i}, R_i, \mathsf{onion}_i, \alpha_i, T)$.
12) If U_{i+1} dishonest, go to step Simulator U_i honest, U_{i+1} dishonest step 1 with parameters $(\texttt{pid}, \mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_{i+1}, \alpha_{i-1} - \mathsf{fee}, T, \overline{\gamma_{i-1}}, \overline{\gamma_i}, \theta_{\epsilon_{i-1}}, \theta_{\epsilon_i})$.

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

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

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 $(\texttt{sid}, \texttt{pid}, \texttt{offload}, \mathsf{tx}^{\mathsf{vc}}, U_0) \overset{\tau}{\hookleftarrow} \mathcal{F}_{VC}$ 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 $(\texttt{ssid}_L, \texttt{POST}, \overline{\mathsf{tx}^{\mathsf{vc}}}) \overset{\tau}{\hookrightarrow} \mathcal{G}_{Ledger}$.

b) Case U_n honest, U_0 dishonest

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

c) Case U_n dishonest, U_0 honest

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

Simulator for Close-VC phase

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

1) Upon $(\texttt{sid}, \texttt{pid}, \texttt{close}, \alpha'_{i-1}, \overline{\gamma_{i-1}}) \overset{\tau}{\hookleftarrow} \mathcal{F}_{VC}$ or upon being called by the simulator $\mathcal{S}$ itself in round τ with parameters $(\texttt{pid}, \alpha'_{i-1}, \overline{\gamma_{i-1}})$.
2) Retrieve $(\overline{\gamma_{i-1}}, \theta_{\epsilon_{i-1}}) := \mathsf{left}(\texttt{pid}, U_i)$.
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) $\vec{\theta}'_{i-1} := \texttt{genNewState}(\overline{\gamma_{i-1}}, \alpha'_i, T)$
6) $\mathsf{tx}^{\mathsf{r}}_i := \texttt{genRefTx}(\vec{\theta}'_{i-1}, \theta_{\epsilon_{i-1}}, U_{i-1})$
7) Create the signature $\sigma_{U_i}(\mathsf{tx}^{\mathsf{r}'}_{i-1})$ on U_i's behalf.
8) Send $(\texttt{sid}, \texttt{pid}, \texttt{close-req}, \vec{\theta}'_{i-1}, \mathsf{tx}^{\mathsf{r}'}_{i-1}, \sigma_{U_i}(\mathsf{tx}^{\mathsf{r}'}_{i-1})) \overset{\tau}{\hookrightarrow} U_{i-1}$.
9) If $(\texttt{ssid}_C, \texttt{UPDATED}, \overline{\gamma_{i-1}}.\mathsf{id}, \vec{\theta}'_{i-1}) \xhookleftarrow{\tau+1+t_{\mathsf{u}}} \mathcal{F}_{Channel}$, send $(\texttt{sid}, \texttt{pid}, \texttt{replace}, \overline{\gamma_{i-1}}, \vec{\theta}'_{i-1}) \xhookrightarrow{\tau+1+t_{\mathsf{u}}} \mathcal{F}_{VC}$.

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

1) Upon $(\texttt{sid}, \texttt{pid}, \texttt{close-req}, \vec{\theta}'_i, \mathsf{tx}^{\mathsf{r}'}_i, \sigma_{U_{i+1}}(\mathsf{tx}^{\mathsf{r}'}_i)) \overset{\tau}{\hookleftarrow} U_{i+1}$, let $\overline{\gamma_i}$ the channel between U_i and U_{i+1}.
2) Let $\mathsf{tx}^{\mathsf{r}}_i := \mathsf{right}(\texttt{pid}, U_i)$. 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 $\theta_{\epsilon_i} \in \mathsf{tx}^{\mathsf{r}}_i.\mathsf{input}$
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 $\vec{\theta}'_i \neq \texttt{genNewState}(\overline{\gamma_i}, \alpha'_i, T)$, abort.
9) If $\mathsf{tx}^{\mathsf{r}'}_i \neq \texttt{genRefTx}(\vec{\theta}'_i, \theta_{\epsilon_i}, U_i)$, 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 $(\texttt{sid}, \texttt{pid}, \texttt{CLOSE}, \alpha'_i) \overset{\tau}{\hookrightarrow} U_i$ and expect the answer $(\texttt{sid}, \texttt{pid}, \texttt{CLOSE-ACCEPT}) \overset{\tau}{\hookleftarrow} U_i$, otherwise go idle.
12) Send $(\texttt{ssid}_C, \texttt{UPDATE}, \overline{\gamma_i}.\mathsf{id}, \vec{\theta}'_i) \overset{\tau}{\hookrightarrow} \mathcal{F}_{Channel}$.
13) Expect $(\texttt{ssid}_C, \texttt{UPDATED}, \overline{\gamma_i}.\mathsf{id}, \vec{\theta}'_i) \xhookleftarrow{\tau+t_{\mathsf{u}}} \mathcal{F}_{Channel}$, else go idle.

14) Send $(\mathtt{sid}, \mathtt{pid}, \mathtt{replace}, \overline{\gamma_i}, \vec{\theta}_i') \overset{\tau}{\hookrightarrow} \mathcal{F}_{VC}$.
15) Set $\mathsf{right}(\mathtt{pid}, U_i) := \mathsf{tx}_i^{\mathsf{r}'}$
16) Retrieve $(\overline{\gamma_{i-1}}, \theta_{\epsilon_{i-1}}) := \mathsf{left}(\mathtt{pid}, U_i)$.
17) If $U_i = U_0$, send via $\mathcal{F}_{VC}$ to the dummy user U_i the message $(\mathtt{sid}, \mathtt{pid}, \mathtt{VC\text{-}CLOSED}) \xhookrightarrow{\tau+t_{\mathsf{u}}} U_i$. Go idle.
18) Send via $\mathcal{F}_{VC}$ to the dummy user U_i the message $(\mathtt{sid}, \mathtt{pid}, \mathtt{CLOSED}) \xhookrightarrow{\tau+t_{\mathsf{u}}} U_i$.
19) If U_{i-1} honest, send $(\mathtt{sid}, \mathtt{pid}, \mathtt{continue\text{-}close}, \overline{\gamma_{i-1}}, \alpha_i' + \mathsf{fee}) \xhookrightarrow{\tau+t_{\mathsf{u}}} \mathcal{F}_{VC}$
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 $(\mathtt{sid}, \mathtt{pid}, \mathtt{post\text{-}refund}, \overline{\gamma_i}, \mathsf{tx}^{\mathsf{vc}}, \theta_{\epsilon_i}, R_i) \overset{\tau}{\hookleftarrow} \mathcal{F}_{VC}$.
   - Extract α_i and T from $\overline{\gamma_i}.\mathsf{st.output}[0]$.
   - If U_{i+1} is honest, create the transaction $\mathsf{tx}_i^{\mathsf{r}} := \mathtt{genRefTx}(\overline{\gamma_i}.\mathsf{st}[0], \theta_{\epsilon_i}, U_i)$. Else, let $\mathsf{tx}_i^{\mathsf{r}} := \mathsf{right}(\mathtt{pid}, U_i)$
   - Extract $\mathsf{pk}_{\widetilde{U_i}}$ from 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 $(\mathtt{ssid}_L, \mathtt{POST}, \overline{\mathsf{tx}_i^{\mathsf{r}}}) \overset{\tau}{\hookrightarrow} \mathcal{G}_{Ledger}$.
2) Upon $(\mathtt{sid}, \mathtt{pid}, \mathtt{post\text{-}pay}, \overline{\gamma_i}) \overset{\tau}{\hookleftarrow} \mathcal{F}_{VC}$
   - Extract α_i and T from $\overline{\gamma_i}.\mathsf{st.output}[0]$. 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 $(\mathtt{ssid}_L, \mathtt{POST}, \overline{\mathsf{tx}_i^{\mathsf{p}}}) \overset{\tau}{\hookrightarrow} \mathcal{G}_{Ledger}$.

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 [1], 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 6 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:

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

TABLE 6: 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 $\mathsf{tx}^{\mathsf{r}}_i$ is correct

- $\mathtt{EXEC}_{\mathcal{F}_{VC},\mathcal{S},\mathcal{E}} := \mathsf{obsSet}(m_1, \mathcal{F}_{Channel}, \tau_0 + 1) \cup \mathsf{obsSet}(m_1, \mathcal{F}_{Channel}, \tau_0 + 1)$ 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 7. We define the following messages.

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

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 $\mathtt{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_{10}[\tau + 1]\}$

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 $\mathtt{EXEC}_{\mathcal{F}_{VC},\mathcal{S},\mathcal{E}} := \{m_{10}[\tau + 1]\}$

TABLE 7: 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}.

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, user 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, functionality $\mathcal{F}_{VC}$ 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 8. 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 alternatively `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 alternatively `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 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 $\texttt{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_{13}[\tau], m_{17}[\tau + t_{\mathsf{u}}]\} \cup \mathsf{obsSet}(m_{15}, \tau, \mathcal{F}_{Channel})$.

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

TABLE 8: 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).

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 $\texttt{EXEC}_{\mathcal{F}_{VC},\mathcal{S},\mathcal{E}} := \{m_{13}[\tau], m_{17}[\tau + t_{\mathsf{u}}]\} \cup \mathsf{obsSet}(m_{15}, \tau, \mathcal{F}_{Channel})$.

□

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 now only for an honest U_0. We use message $m_{11} := (\texttt{ssid}_L, \texttt{POST}, \overline{\mathsf{tx}^{\mathsf{vc}}})$ 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 $\texttt{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \mathsf{obsSet}(m_{11}, \mathcal{G}_{Ledger}, \tau)$.

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 $\texttt{EXEC}_{\mathcal{F}_{VC},\mathcal{S},\mathcal{E}} := \mathsf{obsSet}(m_{11}, \mathcal{G}_{Ledger}, \tau)$.

□

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 9. We define the following messages.

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

U_i **honest, revoke**

Real world: In every round τ, U_i performs `RESPOND`, which provides a decision on whether or not to do the following. If yes, U_i performs `REVOKE`, which results in message m_{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 $\texttt{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_{20}[\tau + t_{\mathsf{c}} + 2\Delta]\} \cup \mathsf{obsSet}(m_{18}, \mathcal{F}_{Channel}, \tau) \cup \mathsf{obsSet}(m_{19}, \mathcal{G}_{Ledger}, \tau + t_{\mathsf{c}} + \Delta)$

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

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 $\texttt{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_{22}[\tau+\Delta]\} \cup \mathsf{obsSet}(m_{21}, \mathcal{G}_{Ledger}, \tau)$

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 $\texttt{EXEC}_{\mathcal{F}_{VC},\mathcal{S},\mathcal{E}} := \{m_{22}[\tau + \Delta]\} \cup \mathsf{obsSet}(m_{21}, \mathcal{G}_{Ledger}, \tau)$

□

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.

TABLE 9: 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.

F.6. Discussion on security and privacy goals

We state our security and privacy goals informally in Section 5.1. In this section we formally define these goals as cryptographic games on top of the ideal functionality $\mathcal{F}_{VC}$ described in Appendix F.3 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 [1].

F.6.1. 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 $1/2 + \epsilon$, 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 [10]. Sphinx [12], for instance, is a realization of this. We use it in Donner, extended with a per-hop payload (see also Section 5.2).

F.6.2. 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 $\textbf{PathBalance}(U) := \gamma_i.\mathsf{balance}(U) + \gamma_{i+1}.\mathsf{balance}(U)$. Intuitively then, we say that a virtual channel (VC) protocol achieves *balance security* if the $\textbf{PathBalance}(U)$ for each honest intermediary U does not decrease..

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 $\alpha + n \cdot \epsilon$ 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 $\mathtt{sid}$ and $\mathtt{pid}$ to two random identifiers. Then, the challenger simulates opening a VC from the OpenVC phase on input ($\mathtt{sid}$, $\mathtt{pid}$, $\mathtt{SETUP}$, $\mathsf{channelList}$, $\mathsf{tx}^{\mathsf{in}}$, α, 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 $\textbf{PathBalance}(U)$ 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 $\textbf{PathBalance}(U)$ 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 $\textbf{PathBalance}(U_i)$ 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.6.3. 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 $\alpha + n \cdot \epsilon$ 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 $\mathtt{sid}$ and $\mathtt{pid}$ 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.

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 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 4.*

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 $T - 3\Delta - t_{\mathsf{c}}$. 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.6.4. 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 $T - 3\Delta - t_{\mathsf{c}}$.

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 $\mathsf{tx}^{\mathsf{in}}.\mathsf{output}[0]$ holds at least $\alpha + n \cdot \epsilon$ 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 $\mathtt{sid}$ and $\mathtt{pid}$ 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 $T - 3\Delta - t_{\mathsf{c}}$.

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 $T - 3\Delta - t_{\mathsf{c}}$. □

F.6.5. 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, $\mathsf{channelList}_0$ 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 $\mathsf{tx}^{\mathsf{in}}.\mathsf{output}[0]$ holds at least $\alpha + n \cdot \epsilon$ coins, where n is the length of the path defined in $\mathsf{channelList}_b$. Finally, the adversary sends two queries $(\mathsf{channelList}_0, \mathsf{tx}^{\mathsf{in}}, \alpha, T)$ and $(\mathsf{channelList}_1, \mathsf{tx}^{\mathsf{in}}, \alpha + \mathsf{fee}, T)$ to the challenger. The challenger sets $\mathtt{sid}$ and $\mathtt{pid}$ to two random identifiers. Moreover, the challenger picks a bit b at random and simulates 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 $1/2 + \epsilon$, 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 $(\mathtt{sid}, \mathtt{pid}, \mathtt{open}, \mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_{i+1}, \alpha_i, T, \overline{\gamma_{i-1}}, \overline{\gamma_i}, \theta_{\epsilon_{i-1}}, \theta_{\epsilon_i})$ 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, $\theta_{\epsilon_{i-1}}$, 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, $\theta_{\epsilon_{i-1}}$ and θ_{ϵ_i} By design, the adversary knows that outputs $\theta_{\epsilon_{i-1}}$ 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 $\mathtt{checkTxIn}$. 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.6.6. 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 $\alpha + n \cdot \epsilon$ 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 $1/2 + \epsilon$, 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 $(\mathtt{sid}, \mathtt{pid}, \mathtt{open}, \mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_{i+1}, \alpha_i, T, \overline{\gamma_{i-1}}, \overline{\gamma_i}, \theta_{\epsilon_{i-1}}, \theta_{\epsilon_i})$ 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.6.7. 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 $\alpha + n \cdot \epsilon$ 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 $1/2 + \epsilon$, 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 $1/2 + \epsilon$, where ϵ denotes a negligible value. □