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, allowing 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 DoS/griefing style attack that leverages virtual channels to destroy the PCN itself and is inherent to the design adopted by the existing Bitcoin-compatible 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.

I. INTRODUCTION

Payment channels (PCs) have emerged as one of the most promising solutions to the limited transaction throughput of permissionless blockchains, with the Lightning Network [34] 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 [11], [12], [18]. Recent work [8] 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 PCs between all pairs of users (i.e., a clique) is economically infeasible, as users must lock coins for each PC and funding occurs on-chain. On-demand creation of PCs with any potential partner is also infeasible due to the need for on-chain transactions for opening and closing each channel, which results in on-chain fees, long confirmation times (around 1h in Bitcoin) and again impacts the blockchain throughput. As a result, single PCs are instead linked together to form PCNs, using paths of PCs to connect two users instead of opening a PC between them. The interactions of PCN users can be classified into synchronization protocols and virtual channels.

Synchronization protocols. Synchronization protocols [9], [22], [30]–[32], [34] 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 increases along with 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) [4]. 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 [17].

Virtual channels. Virtual channels (VC) [7], [19]–[21], [25], [28], [33] allow two users connected by a path of PCs to establish a direct connection, bypassing intermediaries. Intuitively, a
VC is akin to a PC, but instead of being opened by an on-chain transaction, it is opened off-chain using funds from the path of PCs. Therefore, the opening phase involves all intermediaries, besides the endpoints. Once established, however, updates can proceed without the involvement of any intermediaries. In this manner, VCs overcome the aforementioned drawbacks of synchronization protocols: (i) intermediaries are no longer required to be online; (ii) the reliability of the channel does not depend on intermediaries; (iii) intermediaries do 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 intermediaries; (v) intermediaries do not learn each single VC update; (vi) a PC can host several VCs, each of which can be used to dispense up to 483 payments or potentially more VCs, bypassing the limitation on the number of payments in PCNs.

Since VCs can be used just as PCs, they constitute the most promising solution to perform repeated transactions as well as applications different from payments (e.g., [11], [12], [18]) between any pair of users connected by a path of PCs. In fact, applications built on top of PCs can be smoothly lifted to VCs, which constitutes a crucial improvement over synchronization protocols.1 For instance, VCs support Discreet Log Contracts [18], 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. For a more detailed discussion see Appendix A.

VC constructions are difficult to design, since the balance of honest parties needs to be ensured even in the presence of malicious, and possibly colluding intermediaries/endpoints. The first constructions have been proposed for blockchains supporting Turing-complete scripting languages based on the account model, like Ethereum [19–21]. In such blockchains, VC constructions are somewhat easier to design: For instance, stateful smart contracts can resolve conflicts on the current state of VCs by associating a different version number to each state update and, in case of conflict, by selecting the highest number as the valid state. Indeed, Ethereum-based constructions are based on this idea and do not suffer from the Domino attack presented in this paper. Unfortunately, this reliance on Turing-complete scripting languages makes these constructions incompatible with many of the cryptocurrencies available today, including Bitcoin itself.

It is not only of practically relevant, but also theoretically interesting to investigate what is the minimum scripting functionalities necessary to design secure VCs. Therefore, a bit later VC constructions have been proposed also for blockchains with a less expressive scripting language and based on the Unspent Transaction Output (UTXO) model (i.e., Bitcoin-compatible) [7], [25], [28]. Throughout the rest of this paper, we investigate VCs built on these blockchains if not specified otherwise. All of these 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 off-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)).

**Conceptual advancements in this work.** We show that rooted VCs are by design prone to severe drawbacks including the Domino attack (see Section III), a new DoS/griefing style 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 Bitcoin-compatible VC constructions affected by this attack, in fact the ideal functionalities against which they are proven secure do permit this attack, but also this attack is so severe that it can potentially shut down the underlying PCN, as we show in Section III-C. As a result, we argue that none of the existing Bitcoin-compatible 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 III-D). 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 PCNs, by one of the VC endpoints. The underlying PCNs are used to lock up some funds (or collateral) that are paid to the honest VC endpoint if the 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 PCNs, 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 VC to a PC 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

---

1VCs expose all the functionalities of a PC and can be used interchangeably as a building block for off-chain applications, see Section V.
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 [28]. The improvements over existing Bitcoin-compatible multi-hop VC constructions are summarized in Table I. We compare with single-hop constructions and with those relying on Turing-complete smart contracts in Table V in Appendix B.

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 Bitcoin-compatible 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. Even though VC protocols are not yet used in practice, we find it crucial to show this attack before any construction gets implemented, offering instead a secure alternative.

- 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 I).

- We introduce the notion of synchronized modification, a novel subroutine allowing parties to atomically change the value or timeout of a synchronization protocol, a contribution of independent interest. Synchronized modification, non-rooted funding, and the pay-or-revoke paradigm [9] 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 fewer 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 [25] (depending on how the VC is constructed) or cubic in Elmo [28] to constant.

**II. BACKGROUND AND NOTATION**

**A. UTXO based blockchains**

We adopt the notation for UTXO-based blockchains from [8], 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 $\theta := (\text{cash}, \phi)$: $\text{cash}$ contains the amount of coins stored in this output and $\phi$ 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 $\text{inputs}$ and the latter ones as transaction $\text{outputs}$. Formally, we define a transaction body as a tuple $\text{tx} := (\text{id}, \text{ inputs}, \text{ output})$. The identifier $\text{tx}\_\text{id} \in \{0, 1\}^*$ is assigned as the hash of the other attributes, $\text{tx}\_\text{id} := H(\text{tx}\_\text{id}, \text{tx}\_\text{output})$. We model $H$ as a random oracle. The attribute $\text{tx}\_\text{input}$ is a non-empty list of the identifiers of the transaction’s inputs and $\text{tx}\_\text{output} := (\theta_1, \ldots, \theta_n)$ 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 $\text{tx} := (\text{id}, \text{ inputs}, \text{ output}, \text{ witness})$ or for convenience also $\text{tx} := (\text{tx}, \text{ witness})$. Valid transactions can be recorded on the public ledger $\mathcal{L}$ called blockchain, with a delay of $\Delta$. 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 $\phi$ 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 OneSig(pk) or MultiSig(pk_1, 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 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 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 $\phi = \phi_1 \lor \ldots \lor \phi_n$. A conjunction of subconditions is simply written as $\phi = \phi_1 \land \ldots \land \phi_n$.

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).

B. 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 [8], [30], [31] for more details. We abstract away from the implementation details and instead model the state of the reader to [8], [30], [31] for more details. We abstract away the final state to the ledger.

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 $\alpha$ 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 $\gamma_j$, 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 $\alpha$ coins moved from left to right. We give an example in Figure 13 in Appendix D.2.

D. Blitz

There exist many different MHP protocols that synchronize the updates of channels. In particular, the Blitz [9] protocol is useful for this work. In Blitz, the PC updates are dependent on a transaction called $tx^{er}$, which acts as a global event. The PCs are synchronized in the following way: If $tx^{er}$ is posted on-chain, the updates are reverted, otherwise, they are successful. In other words, the sender sets up a MHP conditioned on a “refund enabling” transaction $tx^{er}$ in a way that the refund can be triggered, if anything goes wrong. If all channels participated honestly, the sender does not post $tx^{er}$ and the MHP goes through (see Figure 3). In a bit more detail, Blitz consists of four operations:

1) Setup. The sender $U_0$ creates a synchronization transaction $tx^{er}$ as depicted in Figure 3b, which has an output $\theta_{e_0}$ holding $\epsilon$ coins for each user except the receiver $U_n$. The value $\epsilon$ is set to the smallest possible value that the underlying blockchain allows (ideally zero); these outputs are merely to enable other transactions.

2) Open. Each channel sequentially, from sender to receiver, sets up a payment whose success or refund is conditioned on a time $T$ or transaction $tx^{er}$, as conceptualized in Figure 3a and shown in detail in Figure 3c. In a nutshell, two users $U_i, U_{i+1}$ update their channel $\gamma_i$ to a state where the amount to be paid $\alpha$ (more precisely $\alpha_i$, which encodes a per-hop fee) coming from $U_i$ can be spent as follows: Either by $U_{i+1}$ using $tx^s_i$ after time $T$ or by $U_i$ using $tx^r_i$, if $tx^{er}$ is posted on-chain. Since each $tx^s_i$ uses the corresponding output $\theta_{e_i}$ of $tx^{er}$, the UTXO model ensures that $tx^r_i$ can only be posted if $tx^{er}$ has been posted before.

3) Finalize. After the receiver has successfully set up the payment, she sends back a confirmation to the sender containing $tx^{er}$. If the sender receives a confirmation containing the $tx^{er}$ she created in the setup phase within some time, she goes idle. Otherwise, she posts $tx^{er}$, initiating the refunds (see respond).

4) Respond. Every user $U_i$ monitors the blockchain if $tx^{er}$ appears. In case it appears before $T_i$, the user will publish the refund transaction $tx^r_i$ for her channel $\gamma_i$. If the two users in $\gamma_i$ collaborate, both updates and refunds can always be performed off-chain.

In this work, we utilize a slightly modified version of Blitz as a building block. We mark the modification in green in Figure 3b and describe it in Section V-C.

E. 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 \( \text{tx}^\alpha \) 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 \( \text{tx}^\alpha \) on-chain, transforming the VC into a PC in case of a dispute.

To understand how VCs work, let us look at an example following a state-of-the-art VC construction [25]. This example is depicted in Figure 4. Assume \( U_0 \) and \( U_2 \) want to construct a VC via \( U_1 \), i.e., there exist PCs \((U_0, U_1)\) and \((U_1, U_2)\), and they wish to build a VC \((U_0, U_2)\). To open a VC, the main idea is to take the desired VC capacity \( \alpha \) and lock it in both channels, such that \( \alpha \) coins come from \( U_0 \) and \( \alpha \) coins from the intermediary \( U_1 \). These two \( \alpha \) coins are used both for funding the VC and as collateral; these coins can be spent in the following, mutually exclusive ways:

(i) by putting the funding transaction \( \text{tx}^\alpha \) on-chain, which simultaneously funds the VC and refunds the intermediary its collateral \( \alpha \), or

(ii) if both \( \alpha \) coins are not spent by a chosen punishment time \( t_{\text{pun}} \), \( U_0 \) and \( U_2 \) can each claim \( \alpha \) coins, which is the maximal amount they could hold in the VC.

Clearly, \( U_1 \), who is part of both channels, is incentivized to put \( \text{tx}^\alpha \) on-chain, as this is the only way to get her collateral back. Simultaneously, the two end-users \( U_0 \) and \( U_2 \), who are only part of one of the channels, are ensured that either \( \text{tx}^\alpha \) goes on-chain, or else they receive the full \( \alpha \).

Putting \( \text{tx}^\alpha \) on-chain is called offloading and is a safety mechanism to ensure that users can claim their rightful balance in case of a dispute. Offloading can be initiated by either \( U_0 \) or \( U_2 \) (by closing their respective channel and threatening to take the collateral if \( U_1 \) does not react), or by \( U_1 \) by simply closing both channels. We emphasize that the money of \( \text{tx}^\alpha \) comes from both underlying channels, i.e., it can only exist on-chain, if both underlying channels have been put on-chain (closed). We call this design a rooted VC.

If there is no dispute, the transactions depicted in Figure 4 remain off-chain and the underlying channels \((U_0, U_1)\) and \((U_1, U_2)\) remain open. The update of the VC requires no interaction of the intermediaries, the end-users simply update the channel \((U_0, U_2)\) as they would a PC. 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. This single-intermediary idea 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 Figure 5.
III. THE DOMINO ATTACK

A. Reasons that lead to 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 II-E, 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, which can be initiated by the VC endpoints and in some existing VC protocols [28] even by the intermediaries.

Observation 2: VC funding transaction is rooted in all underlying base channels. We recall that 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 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.

B. Attack description

The Domino attack is essentially a DoS or grieving style attack. It follows directly from the two observations mentioned above and can proceed in the following 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 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 5. 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 [25]. After the attacker is done with this step, the transaction structure among different users is as in Figure 5. The attacker can now unilaterally force the closure of all underlying channels, i.e., 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 5 (e.g., this could be LVPC), the output in the state of $(U_3, U_4)$ which is used to fund the VC $(U_0, U_3)$ 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 the punishment transaction on the bottom right in Figure 5 (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 $(U_0, U_3)$. 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 [28]. 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 functionalities explicitly allow for the Domino attack.

C. Quantitative analysis of the Domino attack

To quantify the severity of the Domino attack, we perform the following simulation. We take a current (March 2022) snapshot of the Lightning Network (LN) [5]. In this snapshot, there are 83k channels, 20k nodes and 3284 BTC (around 150M USD) locked in channels (of the largest connected component). The nodes’ connectivity varies in the LN. There are leaf nodes having only one open channel, and there are nodes with almost 3000 channels. Additionally, entities can control multiple nodes. The entities can be linked by their alias, as pointed out in [35], something we follow in this simulation as well.

Clearly, differently connected nodes can launch the Domino attack with more or less devastating effect. The better connected a node is, the more channels can be closed down. Note that for this attack, it does not matter how many coins are locked in the channels under control of the attacker and not even the number of nodes the attacker control, but instead the number of open channels and the kind of paths which exist to another node under the attacker’s control; the source and destination may be the same node.

---

2By 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.
approach for each alias. Each node can close channels are closed and they can do no further damage. The attacker repeats these steps, until all of the attacker’s Domino attack. Now, on the new network with fewer channels, one and proceeds to close the channels by performing the attack, yielding paths starting and ending at one node.

This means, using the open channels the attacker tries to be caused by existing nodes in the LN with two or more channels, as possible. Computing the optimal set of VCs the attacker would need to open to maximize the channels is computationally expensive and out of the scope of this simulation. Instead, we settle for a simpler heuristic. The attacker computes the cycle basis for a root node controlled by the attacker, yielding paths starting and ending at one node under the attacker’s control. The attacker chooses the longest one and proceeds to close the channels by performing the Domino attack. Now, on the new network with fewer channels, the attacker repeats these steps, until all of the attacker’s channels are closed and they can do no further damage.

We count the channels an attacker can close with this approach for each alias. Each node can close 1284 channels on average, which amounts to around 1.5% of all channels in the LN. However, note that around 8% of all nodes can close no channels at all, while the most well-connected entity can close around 53k channels, which more than 60% of the LN. We visualize our results in Figure 6a, where for a given interval of how many channels an entity can close, we show the percentage of nodes in this category. The source code and raw results of this simulation can be found at [6].

To make matters worse, an attacker can target specific channels with this. This allows the attacker to perform attacks similar to Route Hijacking [38], a DoS attack where an attacker strategically places a channel in a topologically important location and announces low fees. Subsequently, users will route their payments through the attacker’s channel who can then drop the requests. In the worst case this can (temporarily) disconnect parts of the network from one another. In the Domino attack, an attacker can disconnect parts of the network directly, by closing all edges that connect the two subgraphs.

Analyzing newly placed nodes. In this second analysis, we let the attacker create new channels instead of assuming an existing node is corrupted. Clearly, without any restrictions, an attacker can do more damage than in the previous simulation, i.e., by opening the same (and more) channels as the best performing node which had a bit less than 3000 channels. Taking a current average fee of 0.000031 BTC (1.27 USD) per transaction [2], this would cost an adversary around 0.186 BTC. In more detail, 0.093 BTC are needed for opening these channels and again 0.093 BTC for closing them after establishing the according VC, triggering the Domino attack. Note that the latter amount is also needed if the channels are already there (in the previous simulation).

We therefore put some restrictions on the attacker. We assume that an adversary has a certain budget to spend on fees for establishing channels over the network. Further, 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 $n$ to 11, the diameter of the LN snapshot, i.e., at this length every nodes can reach every other node.

The adversary needs to post 3 on-chain transactions per VC with the associated fees, two for establishing the two PCs encasing the victim path 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. 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 6b. Using only 1 BTC for fees, the adversary can close up to 97k honest channels, which is more than all channels in our LN snapshot (83k), and cause a cost of at least 6 BTC to the involved nodes. 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 [3].

We remark that since VCs are not used in practice, we cannot evaluate this in the real world. However, previous work has already shown the feasibility of similar DoS or griefing attacks and how they transfer to the real world [24]. For a discussion on why it is infeasible to deter this attack with fees, we refer to Appendix B. From our simulation it follows that 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.

D. More drawbacks of current VC constructions

Unlimited lifetime. Existing VC constructions such as Elmo [28] 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 IV.

**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 underlying 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. In our construction, offloading involves only a constant number of on-chain transaction 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.

### IV. Donner: Key ideas

We describe the core ideas of Donner by assuming that a slight variant of the previously described Blitz construction is used as underlying MHP protocol. As detailed below, our construction is parameterized over it, so that other functionality-equivalent MHP protocols could be deployed instead.

**High level architecture.** 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 $\alpha$ and time $T$ fully funded by $U_0$. First, $U_0$ starts with a slightly modified version of the Setup phase of a Blitz payment of $\alpha$ coins, as explained in Section II-D, let us call it Setup*. In this modified phase, $U_0$ proceeds to create a transaction $tx^{vc}$ as depicted in Figure 3b (this time, including the green part) instead of $tx^\alpha$. $tx^{vc}$ takes an input from $U_0$ and creates an output holding $\alpha$ coins and like in the Setup phase, an output holding $\epsilon$ coins for each user except the receiver $U_n$. This transaction will serve two purposes: (i) it will be the funding of the VC and (ii) it will be used to synchronize a Blitz payment.

Next, $U_0$ and $U_n$ proceed to create the initial state (see Section II-B) $tx^{\alpha}$ of the VC using $tx^{vc}$ as a funding. We emphasize that this process is exactly the same as for a PC, the only difference being that the funding transaction $tx^{vc}$ has these additional outputs holding $\epsilon$ and we do not intend to publish $tx^{vc}$ on-chain. After this step is successful, $U_0$ initiates the remaining phases of Blitz (Open, Finalize and Respond) using $tx^{vc}$. After completion, a Blitz payment of value $\alpha$ is open between $U_0$ and $U_n$ conditioned on $tx^{vc}$, i.e., it is refunded if $tx^{vc}$ is posted and otherwise successful after time $T$.

**Intuition security.** At this point, the VC is considered open and can be used exactly like a PC. The careful readers might be wondering why this VC is safe to use. After all, we detached the funding from the underlying PCs and removed the receiver $U_n$‘s ability to offload the VC. However, the sender $U_0$ did set up a Blitz payment to $U_n$ of $\alpha$ coins, which is the full capacity of the VC. By putting the VC funding inside the synchronization transaction of Blitz, we make the two actions offload the VC and refund the Blitz payment atomic. In other words, if $U_0$ does not offload, $U_n$ will automatically receive the full VC capacity via the payment after $T$.

**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 got rid of (ii): The funding $tx^{vc}$ comes solely from $U_0$, i.e., it is independent (or detached) from the PCs underlying the VC. The VC can be offloaded without closing the underlying PCs, simply by $U_0$ posting $tx^{vc}$. Once posted, all PCs can be honestly settled, updating the PC to reflect the refund or the success of the Blitz payment, as in Blitz itself or other synchronization protocols.

**Closing the VC.** One of the most essential operations of the VC operation is closing the VC honestly, i.e., off-chain. This is challenging, because closing needs to proceed in a way, such that no one is at risk of losing funds. To solve this challenge, we first observe that if the receiver $U_n$ already owns all $\alpha$ coins in the VC, the VC endpoints need merely wait until the Blitz timeout $T$ runs out. At this point, the Blitz payment will be successful automatically. But what about when $U_n$ owns $0 \leq \alpha' < \alpha$ coins in the VC? We need a protocol that atomically changes the value of the Blitz transaction from $\alpha$ to $\alpha'$. To solve this issue, we introduce a new protocol, called synchronized modification, which given a payment of value $\alpha$ tied to transaction $tx^{vc}$ and a timeout $T$, allows for updating the payment to a value $\alpha'$ such that $0 \leq \alpha' < \alpha$. This is illustrated in Figure 7.

**Synchronized modification** works as follows. We can update individual 2-party Blitz contracts to the new value $\alpha'$ from right to left. An intermediary $U_i$ is sure to not lose money, because the atomicity of Blitz ensures that in both the left $(U_{i-1}, U_i)$, having locked $\alpha$, and the right channel $(U_i, U_{i+1})$, having locked $\alpha'$, the payment is either refunded or succeeds. In the former case, $U_i$ does not lose money, as both payments are reverted. In the latter case, $U_i$ gains $\alpha$ while paying $\alpha'$, so $U_i$ gets some money. We can incentivize the participation of intermediary users with fees. Alice is incentivized to publish $tx^{vc}$ if the correct updates do not reach her (paying more money
than she owes otherwise), thereby ensuring the atomicity of the synchronized modification. If all the channels are updated, they can simply go idle waiting for the payment to be successful after $T$, or they can finalize this payment instantly by using the fast track functionality [9].

**Fair unlimited lifetime.** The timeout parameter $T$ serves an additional purpose here: It is the lifetime of the VC. VC endpoints need to close the VC before $T$ expires. Interestingly, we can use the aforementioned synchronized modification operation also for extending this 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 a synchronized 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, users are incentivized to agree as they are fine 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.

V. DONNER: PROTOCOL DESCRIPTION

A. Security and privacy goals

We informally define three security and three privacy goals for our VC construction. For a formal definition of these properties as cryptographic games (Definitions 3 to 8) and proofs (Theorems 2 to 7), we defer the reader to 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 [26]. 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 readers 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.

B. Assumptions and prerequisites

**Digital signatures.** A digital signature scheme is a tuple of algorithms $\Sigma := (\text{KeyGen}, \text{Sign}, \text{Vrfy})$. On a high level, $(pk, sk) \leftarrow \text{KeyGen}(\lambda)$ is a PPT algorithm that on input a security parameter $\lambda$ generates a keypair $(pk, sk)$. The public key $pk$ is publicly known, while the secret key $sk$ is only known to the user who generated that keypair. $\sigma \leftarrow \text{Sign}(sk, m)$ is a PPT algorithm that on input a secret key $sk$ and a message $m \in \{0,1\}^*$ generates a signature $\sigma$ of $m$. Finally, $\{0,1\} \leftarrow \text{Vrfy}(pk, \sigma, m)$ is a DPT algorithm that on input a public key $pk$, a message $m$ and a signature $\sigma$ outputs 1 iff the signature is a valid authentication tag for $m$ w.r.t. $pk$. We use a EUF-CMA secure [23] signature scheme $\Sigma$ as a black-box throughout this work.

**Payment channel notation.** We model each payment channels as a tuple: $\tau := (id, users, cash, st)$. The attribute $\tau.id \in \{0,1\}^*$ uniquely identifies a channel; $\tau.users \in \mathcal{P}^2$ identifies the two parties involved in the channel out of the set of all parties $\mathcal{P}$. Moreover, $\tau.cash \in \mathbb{R}_{\leq 0}$ denotes the total monetary capacity (i.e., the coins) of the channel and the current state is stored as a vector of outputs of $\tau.state: \tau.st := (\theta_1, \ldots, \theta_n)$. In this work, we use channels in paths from a sender to a receiver. For simplicity, we say that $\tau.left \in \tau.users$ refers to the user closer to the sender, while $\tau.right \in \tau.users$ refers to the user closer to the receiver. The balance of both users can always be inferred from the current state $\tau.st$. For convenience, we say

![Fig. 7: Synchronized modification: Safely modify the contract tied to a transaction $\tau^{\text{tx}}$ in each channel atomically. Note that $\tau^{\text{tx}}$ is the same transaction in all three cases.](image-url)
that $\gamma, \text{balance}(U)$ gives the coins owned by $U \in \gamma, \text{users}$ in this channel's latest state $\gamma, \text{st}$. Finally, we define a channel skeleton $\gamma$ for a channel $\gamma$, as $\gamma := (\gamma, \text{id}, \gamma, \text{users})$.

**Ledger and channels.** We use the ledger (or blockchain) and a PCN (both introduced in Section II) 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.

**publishTx($\mathfrak{tx}$):** The transaction $\mathfrak{tx}$ is posted on-chain after at most $\Delta$ time (the blockchain delay), if it is valid.

**updateChannel($\gamma, \mathfrak{tx}^\text{state}$):** This procedure initiates an update in the channel $\gamma, \text{state}$ to the state $\mathfrak{tx}^\text{state}$, when called by a user in $\gamma, \text{users}$. The procedure terminates after at most $t_u$ time and returns (update-ok) in case of success and (update-fail) in case of failure to both users. We call this function also to update our VC hosted on $\mathfrak{tx}^\text{vc}$.

**closeChannel($\gamma$):** This procedure closes the channel $\gamma$, called by a user in $\gamma, \text{users}$. The latest state transaction $\mathfrak{tx}^\text{state}_\text{vc}$ appears on the ledger after at most $t_c$ time.

**preCreate($\mathfrak{tx}^\text{vc}, \text{index}, U_0, U_n$):** Pre-creates the VC $\gamma, \text{vc}$, exchanging the initial state transactions with the other user in $\gamma, \text{vc}, \text{users} := (U_0, U_n)$ based on the output identified by index of the funding transaction $\mathfrak{tx}^\text{vc}$ that remains off-chain for now. It finally returns $\gamma, \text{vc}$.

**Assumptions and remarks.** 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 pathList $\leftarrow \text{GenPath}(U_0, U_n)$. 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 [36], [37] 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. We reuse the pseudo-code definitions of Setup, Open, Finalize and Respond from [9] in Figure 8.

**C. Detailed construction and pseudocode**

Recall the setting, where $U_0$ and $U_n$, connected via $U_i$ for $i \in [1, n-1]$, wish to open a bidirectional VC with capacity $\alpha$ fully funded by $U_0$. We consider the different phases of Donner: 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). For completeness, we explain the protocol including the operations of Blitz [9] below in prose, while in Figure 8(c) we show a modularized protocol based on the operations setup, open, finalize and respond. We remark that in this work, we could use any other construction providing the same functionality, e.g., this can be achieved by smart contract enabling UTXO-based chains such as the EUTXO model used in Cardano [15]. 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.

**OpenVC.** This phase makes use of a modified Blitz Setup phase (Setup' ) and Open/Finalize of Blitz. Setup': The sender $U_0$ starts by creating a transaction $\mathfrak{tx}^\text{vc}$ that contains an output $\theta_{\text{vc}}$ holding $\alpha$ coins spendable under the condition $\text{MultiSig}(U_0, U_n)$ and $n$ outputs $\theta_{\text{vc}}$ holding $\epsilon$ coins each spendable under the condition $\text{OneSign}(U_i) + \text{RelTime}(\Delta) + \Delta$ , one for every user $U_i$ for $i \in [0, n-1]$. Spending from $\theta_{\text{vc}}$, $U_0$ and $U_n$ create commitment transactions for the VC with $\gamma, \text{vc} := \text{preCreate}(\mathfrak{tx}^\text{vc}, 0, U_0, U_n)$. This function pre-creates the VC $\gamma, \text{vc}$, exchanging the initial state transactions with the other user in $\gamma, \text{vc}, \text{users} := (U_0, U_n)$ based on the output index of the funding transaction $\mathfrak{tx}^\text{vc}$ that remains off-chain for now. It finally returns $\gamma, \text{vc}$.

**Open (Blitz):** Then, each pair of users from $U_0$ to $U_n$ performs $2p\text{Setup}$ of [9], which we briefly summarize as follows. Sender $U_0$ presents its neighbor $U_1$ with $\mathfrak{tx}^\text{vc}$ and an update of their channel to a state, where $\alpha$ coins of $U_0$ are spendable under the condition $\phi = (\text{OneSign}(U_1) \land \text{AbsTime}(T)) \lor \text{MultiSig}(U_0, U_1) \land \text{RelTime}(\Delta))$. Passing along $\mathfrak{tx}^\text{vc}$ does not violate privacy, due to the usage of stealth addresses, see Appendix E.2.

Before actually updating the channel, $U_1$ gives $U_0$ its signature for $\mathfrak{tx}^\text{vc}$. $\mathfrak{tx}^\text{vc}$ takes as inputs the output holding $\alpha$ of the aforementioned proposed state update and the output $\theta_{\text{vc}}$ holding $\epsilon$ 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 $U_n$ has successfully updated its channel with its left neighbor $U_{n-1}$. Then, $U_n$ sends a confirmation to $U_0$ (Finalize).

**UpdateVC.** At this point the VC $\gamma, \text{vc}$ is considered to be open and ready to be used. An update can be performed by creating a new state $\mathfrak{tx}^\text{state}$ and calling $\text{updateChannel}(\gamma, \text{vc}, \mathfrak{tx}^\text{state})$. This function updates the VC $\gamma, \text{vc}$, changing the latest state transaction to $\mathfrak{tx}^\text{state}$ 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 (Synchronized modification).** For closing the VC, assume its final balance is $\alpha - \alpha'$ belonging to $U_0$ and $\alpha'$ 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 perform the new functionality $2p\text{Modify}$ 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 $\alpha$) only $\alpha'$ coins from $U_{n-1}$ are spendable under the condition $\phi = (\text{OneSign}(U_{n-1}) \land \text{AbsTime}(T')) \lor \text{MultiSig}(U_{n-1}, U_n) \land \text{RelTime}(\Delta))$ (closing) or the time in this condition is changed to $T'$ (prolong). For this new state, $U_n$ creates a transaction $\mathfrak{tx}^\text{vc}_{n-1}$ spending this
output and the output of tx_{\text{vc}} belonging to U_{n-1} and gives its signature for this new tx'_{n-1} to U_{n-1}. After U_{n-1} checks that the new state and new tx'_{n-1} 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, \alpha'. If it is not the same, or no such request reaches the sender, U_0 simply publishes tx_{\text{vc}} on-chain and claims tx' 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 \alpha' automatically moves from left to right to the receiver, or they perform the fast-track mechanism of [9] to immediately unlock their funds (cf. Appendix B). VC endpoints do not need to wait until T, but can close the VC well before if they wish to do so.

Respond. This phase corresponds to the phase with the same name of Blitz, which proceeds thus. Participants have to monitor the ledger if tx_{\text{vc}} is published. In case it is published and its outputs are spendable before T, each user U_i for i \in [0, n-1] needs to refund the money they staked in their right neighbor. They can either do this off-chain if their right neighbor is cooperating or in the worst case, forcefully on-chain. They can either do this off-chain if their right neighbor is cooperating or in the worst case, forcefully on-chain via tx'. Similarly, after time T has expired without tx_{\text{vc}} being published on-chain, each user U_i for i \in [1, n] can claim the money from their left channel. Again, this can happen honestly off-chain or forcefully via tx'.

Remarks. Because we detached the funding transaction from the underlying channels, we additionally get rid of the other issues presented in Section III-D. 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 VI, Donner achieves a better level of privacy. We include an illustration of the full construction and the offload operation Figures 10 and 11 in Appendix C.

VI. SECURITY ANALYSIS

A. Informal security analysis

Balance security. When the VC is opened, a Blitz [9] 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 tx_{\text{vc}} is posted. For both of these outcomes, the intermediary does not lose money. As already argued in Section IV the synchronized 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 tx_{\text{vc}} and thereby

(a) Macros: genState(\alpha_0, T, \gamma_0). Generates and returns a new channel state carrying transaction tx_{\text{state}} from the given parameters. genPay(tx_{\text{state}}0). Returns tx_{\text{state}}0 which takes tx_{\text{state}}0 output[0] as input and creates a single output := (\alpha_0, \text{OneSig}(U_{n+1})). genRef(tx_{\text{state}}1, tx_{\text{vc}}0, \theta_{\gamma_0}). Return tx_{\text{vc}}0 which takes as input tx_{\text{state}}0 output[0] and \theta_{\gamma_0} \in tx_{\text{vc}}0 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 tx_{\text{vc}}0 output := (\alpha_i + \epsilon, \text{OneSig}(U_{\text{i+1}})), where \alpha_i, U_i, U_{i+1} are taken from tx_{\text{state}}1.

(b) 2-party operation: 2pModify(\gamma_i, tx_{\text{vc}}0, \alpha_i, T')

Let T be the timeout, \alpha_i the amount and \theta_{\gamma_i} the output used for the two-party contract set up between U_{i-1} and U_i, known from 2pSetup executed in the Open [9] phase.

U_i: tx_{\text{state}}{i-1} := genState(\alpha_i, T, \gamma_{i-1}), tx_{\text{vc}}{i-1} := genRef(tx_{\text{state}}{i-1}, \theta_{\gamma_i}), then send (tx_{\text{vc}}{i-1}, \theta_{\gamma_i}, (tx_{\text{vc}}{i-1})) to U_{i-1} //\theta_{\gamma_{i-1}} known as \theta_{\gamma_{i-1}} from 2pSetup

U_{i-1} upon (tx_{\text{vc}}{i-1}, \theta_{\gamma_i}, (tx_{\text{vc}}{i-1}));

1) Extract \alpha'_i and T' from tx_{\text{state}}{i-1}''. Check that \alpha'_i \leq \alpha_i, T' > T and \alpha'_i \leq \text{genState}(\alpha_i, T', \gamma_{i-1}). If U_{i-1} = U_0, ensure that \alpha'_i \leq x + n \cdot \text{fee} where x is the final balance of U_n in the virtual channel. Check that \sigma_{\gamma_{i-1}}(tx_{\text{vc}}{i-1}') is a correct signature of U_i for tx_{\text{vc}}{i-1}'. Check that tx_{\text{vc}}{i-1}' = genRef(tx_{\text{vc}}{i-1}', \theta_{\gamma_i}) //\alpha_i, T and \theta_{\gamma_i} from 2pSetup

2) UpdateChannel(\gamma_{i-1}, tx_{\text{state}}{i-1}')

3) If, after \text{time} has expired, the message (update-ok) is returned, replace variables tx_{\text{state}}{i-1} and tx_{\text{vc}}{i-1} with tx_{\text{state}}{i-1}' and tx_{\text{vc}}{i-1}' respectively. Return (T, \alpha', T'). Else, return \bot.

U_i: Upon (update-ok), replace variables tx_{\text{state}}{i-1}' and tx_{\text{vc}}{i-1}' with tx_{\text{state}}{i-1} and tx_{\text{vc}}{i-1}', with tx_{\text{vc}}{i-1}' := \text{genPay}(tx_{\text{state}}{i-1}') respectively.

(c) Protocol:

(i) Setup* (see also Appendix E, Figure 16), as in [9], except:

- Create tx_{\text{vc}} instead of tx_{\text{x}} as shown in Figure 3b
- \gamma_{\text{vc}} := \text{preCreate}(tx_{\text{vc}}, 0, U_0, U_n) together with U_n after creating tx_{\text{vc}}, to create the VC commitment transactions.

(ii) Open and Finalize (see also Appendix E, Figure 16) as in [9]

UpdateVC

Either user U_i \in \gamma_{\text{vc}} users can update the VC \gamma_{\text{vc}} by creating a new state tx_{\text{state}}{i-1} and calling updateChannel(\gamma_{\text{vc}}, tx_{\text{state}}{i-1}).

CloseVC/ProlongVC (synchronized modification)

(i) InitClose/InitProlong

U_n: Let \alpha'_i be the final balance of U_n in the virtual channel and T' = T (Close) or T' > T be the new lifetime of the VC and leave \alpha'_i := \alpha_i (Prolong). Execute 2pModify(\gamma_i, tx_{\text{vc}}0, \alpha'_i, T') U_{i-1} upon (T, \alpha'_i, T'); If U_{i-1} \neq U_0, let \alpha'_i := \alpha'_i + \text{fee}.

Execute 2pModify(\gamma_{i-1}, tx_{\text{vc}}0, \alpha'_i, T')

(ii) Emergency-Offload

U_0: If U_0 has not successfully performed 2pModify with the correct value \alpha' (plus fee for each hop) until T - t_\epsilon > 3\Delta, publishTx(tx_{\text{vc}}0, \sigma_{U_0}(tx_{\text{vc}}0)). Else, update T := T'.

Respond (see also Appendix E, Figure 16) as in [9]

Fig. 8: (a) macros, (b) 2-party operation, (c) protocol
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.** $tx_{vc}$ 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 $tx_{vc}$ 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 $tx_{vc}$ 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.

### B. Security model

We rely on the synchronous, global universal composability (GUC) framework [14] to model the Donner protocol. We make use of some preliminary functionalities commonly used in the literature [7]–[9], [19], [20]. The global ledger $\mathcal{L}$ is maintained by the functionality $G_{\text{ledger}}$, which is parameterized by a signature scheme $\Sigma$ and a blockchain delay $\Delta$, 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 $G_{\text{clock}}$ and the communication by $F_{\text{GDC}}$. Finally, a functionality $F_{\text{channel}}$ handles the creation, update and closure of PCs as well as the preparation and update of the VCs.

We define an ideal functionality $F_{\text{VC}}$ 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, $F_{\text{VC}}$ is a trusted third party. Additionally, we formally define the real world hybrid protocol $\Pi$ and show that $\Pi$ emulates (or realizes) $F_{\text{VC}}$. For this, we describe a simulator $S$ that translates any attack of any adversary on $\Pi$ into an attack on $F_{\text{VC}}$.

To show that the protocol $\Pi$ realizes $F_{\text{VC}}$, we need to show that no PPT environment $E$ can distinguish between interacting with the real world and interacting with the ideal world with non-negligible probability. 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 (Definitions 3 to 8) as well as the proof that $F_{\text{VC}}$ has these properties (Theorems 2 to 7).

**Theorem 1.** For functionalities $G_{\text{ledger}}$, $G_{\text{clock}}$, $F_{\text{GDC}}$, $F_{\text{Channel}}$ and for any ledger delay $\Delta \in \mathbb{N}$, the protocol $\Pi$ UC-realizes the ideal functionality $F_{\text{VC}}$.

### VII. Evaluation and Comparison

**Communication overhead.** We implemented a small proof-of-concept that creates the raw Bitcoin transactions necessary for Donner [1]. We use the library python-bitcoin-utils and Bitcoin Script to build the transactions and tested their compatibility with Bitcoin by deploying them on the testnet. We show the results for the operations Open, Update, Close, Offload in Table II. 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 [8] as the underlying payment channel (PC) protocol, 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: $tx_{vc}$, $tx_{i}$ and two transactions for updating the state. Since $tx_{vc}$ 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, $tx_{i}$ has a size of 306 bytes, and a channel update to a state holding this contract is 742 bytes. $tx_{vc}$ does not need to be exchanged, since the left user of a channel can generate it independently. This totals to $1240+34 \cdot n$ bytes of off-chain 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 · $n + 2$ transactions or 34 · $n^2 + 1240 \cdot n + 695$ bytes for the path.

For closing a VC, the payment needs to be updated from right to left. However, $tx_{vc}$ 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 $tx_{vc}$ needs to be posted on-chain and nothing per channel. This means 192+34 · $n$ bytes and costs 0.25 + 0.04 · $n$ 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 · $k + 192 + 34 \cdot n$ bytes, which costs 1.36 · $k + 0.25 + 0.04 \cdot n$ USD for the whole path. We mark this as the pessimistic case in Table II.

**Efficiency comparison.** We compare our construction to LVPC [25] and Elmo [28] (cf. Table III), the only current Bitcoin-compatible 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
TABLE II: 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.

<table>
<thead>
<tr>
<th></th>
<th># txs</th>
<th>size (bytes)</th>
<th>on-chain cost (USD)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open</td>
<td>( 4 \cdot n + 2 )</td>
<td>( 34 \cdot n^2 + 1240 \cdot n + 695 )</td>
<td>0</td>
</tr>
<tr>
<td>Update</td>
<td>2</td>
<td>650</td>
<td>0</td>
</tr>
<tr>
<td>Close</td>
<td>( 3 \cdot n )</td>
<td>1048 ( k + 192 + 34 \cdot n )</td>
<td>0.25 + 0.04 ( n )</td>
</tr>
<tr>
<td>Offload (Optimistic)</td>
<td>1</td>
<td>1048 ( k + 192 + 34 \cdot n )</td>
<td>1.36 ( k + 0.25 + 0.04 \cdot n )</td>
</tr>
<tr>
<td>Offload (Pessimistic)</td>
<td>( 3 \cdot k + 1 )</td>
<td>1048 ( k + 192 + 34 \cdot n )</td>
<td>1.36 ( k + 0.25 + 0.04 \cdot n )</td>
</tr>
</tbody>
</table>

TABLE III: Comparison of LVPC, Elmo and Donner for a VC over from \( U_0 \) to \( U_n \). In the pessimistic offload in Donner, \( k \in [0, n] \) is the number of channels where there is a dispute.

<table>
<thead>
<tr>
<th></th>
<th># txs</th>
<th>size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open LVPC [25]</td>
<td>7 ( \cdot (n - 1) )</td>
<td>( \Theta(n^3) )</td>
</tr>
<tr>
<td>Elmo [28]</td>
<td>( \Theta(n^3) )</td>
<td>( \Theta(n^3) )</td>
</tr>
<tr>
<td>Donner</td>
<td>( 4 \cdot n + 2 )</td>
<td>( \Theta(n^3) )</td>
</tr>
<tr>
<td>Update LVPC [25]</td>
<td>2</td>
<td>3 ( n + 3 )</td>
</tr>
<tr>
<td>Elmo [28]</td>
<td>2</td>
<td>( \Theta(n^3) )</td>
</tr>
<tr>
<td>Donner</td>
<td>2</td>
<td>( \Theta(n^3) )</td>
</tr>
<tr>
<td>Close LVPC [25]</td>
<td>( 4 \cdot (n - 1) )</td>
<td>( \Theta(n^3) )</td>
</tr>
<tr>
<td>Elmo [28]</td>
<td>3 ( n + 3 )</td>
<td>( \Theta(n^3) )</td>
</tr>
<tr>
<td>Donner</td>
<td>( 3 \cdot n )</td>
<td>( \Theta(n^3) )</td>
</tr>
<tr>
<td>Offload LVPC [25]</td>
<td>( 5 \cdot (n - 1) )</td>
<td>( \Theta(n^3) )</td>
</tr>
<tr>
<td>Elmo [28]</td>
<td>3 ( n + 1 )</td>
<td>( \Theta(n^3) )</td>
</tr>
<tr>
<td>Donner</td>
<td>1</td>
<td>( \Theta(n^3) )</td>
</tr>
<tr>
<td>Offload Elmo [28]</td>
<td>( 5 \cdot (n - 1) )</td>
<td>( \Theta(n^3) )</td>
</tr>
<tr>
<td>Donner</td>
<td>( 3 \cdot k + 1 )</td>
<td>( \Theta(n^3) )</td>
</tr>
</tbody>
</table>

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 LN channels.** To simplify the formalization of this work, we built our VC construction on top of generalized payment channels (GC) [8], 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 [7] constructions rely on GCs as well, while the recursive LVPC [25] relies on simple channels that have only one state, but each update reduces the limited lifetime of the channel. (Elmo [28] needs the opcode \texttt{ANYPREVOUT} that is not supported in Bitcoin or 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 \texttt{txi} 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 [25]) 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 [8]. 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.

**VIII. 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 Discrete Log Contracts or fast and direct micropayments, without the need to create a direct PC. Unlike the underlying PCNs, the VCs are not susceptible to liveness and privacy attacks by the intermediaries and do not require fees per payment.

Acknowledgements. This work has been supported by the European Research Council (ERC) under the Horizon 2020 research (grant 771527-BROWSEC); by the Austrian Science Fund (FWF) through the projects PROFET (grant P31621) and the project W1255-N23; by the Austrian Research Promotion Agency (FFG) through the Bridge-1 project PR4DLT (grant 13808694) and the COMET K1 SBA and COMET K1 ABC; by the Vienna Business Agency through the project Vienna Cybersecurity and Privacy Research Center (VISP); by CoBloX Labs; by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development and the Christian Doppler Research Association through the Christian Doppler Laboratory Blockchain Technologies for the Internet of Things (CDL-BOT); by the National Science Foundation (NSF) under grant CNS-1846316; by Madrid regional government as part of the program S2018/TCS-4339 (BLOQUES-CM) co-funded by EIE Funds of the European Union; by the project HACRYPT (N00014-19-1-2292); by grant IJC2020-043391-I/MCIN/AEI/10.13039/501100011033 and European Union NextGenerationEU/PRTR; by PRODIGY Project (TED2021-132464-B-100) funded by MCIN/AEI/10.13039/501100011033 and the European Union NextGenerationEU/PRTR; by SCUM Project (RTI2018-102043-B-I00) MCIN/AEI/10.13039/501100011033/ERDF A way of making Europe.

References


HEN 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 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. Next, we highlight 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 \( l \cdot (BF + RF \cdot a) \cdot x < 2 \cdot TF \) 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 \( l \cdot (BF + RF \cdot a) \cdot x < l \cdot (a \cdot BF + RF \cdot a) \) 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.

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 IV. 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 III 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**

1. **Extended comparison to the state of the art in VCs**

Dziembowski et al. [20] proposed the first construction of VCs over a single intermediary. Recursive constructions [21] 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. [19], [33] 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. [7] 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.

Jourenko et al. [25] 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 [28], a VC construction that does not rely on creating intermediate VCs, by instead relying on scripting functionality not present in Bitcoin, i.e., the opcode ANYPREVOUT. In Table V we compare Donner to existing VC protocols, including those that rely on a Turing-complete scripting language or are limited to a single intermediary.

2. **Extended discussion**

**Deterring the Domino attack with fees.** One might think that the Domino attack could deterred by fees. I.e., intermediaries charge fees high enough to be compensated for having to close and reopen their channel, as well as having to claim the collateral put into the VC, in total at least three transaction per intermediary, in addition to the fees charged for the VC usage. It becomes clear, that this is an infeasible deterrence strategy and is in opposition to the aim of VCs to provide scalable and cheap payments: No user would pay three times an on-chain fee per intermediary for a VC. They would simply post an on-chain transaction or open a new direct PC.

**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, $T \geq 3\Delta + t_c$, where $\Delta$ 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 II) and $t_c$ is the time it takes to close a channel. Intermediaries can prolong the lifetime if they agree and can charge a fee based on 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 $\epsilon$ of txvc 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 [9]. Finally, reusing the stealth address and onion routing mechanism as in [9] we achieve our desired privacy properties.

**APPENDIX C**

**OPERATION EXAMPLES**

To illustrate the different operations for different VC protocols as examples, we provide the following figures. For rooted VCs, we show the construction in Figure 5 in Section II. We further show the offload operation in Figure 9, which coincides with the outcome of the Domino attack example in Section III. For Donner, we show the full construction in Figure 10 and the offload operation in Figure 11.

**APPENDIX D**

**EXTENDED BACKGROUND**

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
TABLE V: Comparison to other virtual channel protocols. 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.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Multi-hop</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Domino attack</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Path privacy</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Storage Overhead per party</td>
<td>(\Theta(n))</td>
<td>(\Theta(n)) &amp; *</td>
<td>(\Theta(n)) / (\Theta(1)) †</td>
<td>(\Theta(n))</td>
<td>(\Theta(n)) &amp; *</td>
<td>(\Theta(n))^3</td>
<td>(\Theta(1))</td>
</tr>
<tr>
<td>Unlimited lifetime</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Off-chain closing</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Dispute: txs on-chain</td>
<td>1</td>
<td>1</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Dispute: time delay</td>
<td>(\Theta(1))</td>
<td>(\Theta(1)) &amp;</td>
<td>(\Theta(1)) / (\Theta(1)) †</td>
<td>(\Theta(1))</td>
<td>(\Theta(1)) &amp;</td>
<td>(\Theta(1)) &amp;</td>
<td>(\Theta(1)) &amp;</td>
</tr>
</tbody>
</table>

Fig. 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.

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 OneSig(pk) or MultiSig(pk_1, 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 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 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 \(\phi = \phi_1 \lor \ldots \lor \phi_n\), 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 \(\phi = \phi_1 \land \ldots \land \phi_n\).

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 \(\alpha\) 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, we 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 V and in the formal model in Appendix F.3, Appendix F.4 and Appendix F.5. To be transparent
Go on-chain:

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

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

About the similarities to [9] and highlight the novelties of this work, we mark the latter in green. Further, we spell out the full protocol pseudocode, including the 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 [9] and highlight the novelties of this work, we mark the latter in green color.

Subprocedures

$\text{checkTxIn}(tx^a, n, U_0, \alpha)$:

1) Check that $tx^a$ is a transaction on the ledger $L$.

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

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

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 $U_n$.

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

Let $\tau$ be the current round. If $T \geq \tau + n(3 + 2t_o) + 3\Delta + t_e + 2 + t_o$, return $\top$. Otherwise, return $\bot$.

$\text{genTxVc(U_0, channelList, tx^a)}$:

1) Let outputList := $\emptyset$ and rList := $\emptyset$
Fig. 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 output, 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 $\phi_1 \lor \phi_2 \lor (\phi_3 \land \phi_4)$. The input of $tx$ is not shown.

\[
U_0 = \begin{bmatrix}
7.12 & 3.16 & 8.2 & 4.6 & 11.7 & 9.0 & 5.4 & 4.6
\end{bmatrix}
\]

U_1 = [12, 8, 5, 9, 10, 4, 6, 7]

\[
U_2 = \begin{bmatrix}
11.7 & 9.0 & 5.4 & 4.6 & 11.7 & 9.0 & 5.4 & 4.6
\end{bmatrix}
\]

U_3 = [12, 8, 5, 9, 10, 4, 6, 7]

Fig. 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).

2) For every channel $\gamma_i$ in channelList:
   - $(pk_{U_i}, R_i) := GenPk(\gamma_i, left.A, \gamma_i, left.B)$
   - outputList := outputList $\cup (x_j, OneSig(pk_{U_i}))$ and RelTime($t_\leftarrow + \Delta$)
   - rList := rList $\cup R_i$
3) Let $\mathcal{P} := \{\gamma_i \mid \text{left}, \gamma_i \text{right} \in \text{channelList}\}$ and nodeList be a list, where $\mathcal{P}$ is sorted from sender to receiver. Let $n := |\mathcal{P}|$.
   - 4) Shufle outputList and rList.
   - 5) Let $tx\leftarrow := (tx\leftarrow, output[0], \text{outputList})$
   - 6) Create a list $msgList \leftarrow \forall i \in [0..n]$, where $msg_i := \mathcal{H}(tx\leftarrow)$
   - 7) onion $\leftarrow CreateRoutingInfo(nodeList[0], msgList[0])$
   - 8) Return $(tx\leftarrow, onion)$.

---

Subprocedures used exclusively in UC model

**createMaps($U_0$, nodeList, $tx\leftarrow, \alpha$):**
1) For every $U_i \in \text{nodeList} \setminus U_n$ do:
   - $(pk_{U_i}, R_i) := GenPk(U_i.A, U_i.B)$
   - outputMap($U_i$) := $(x_i, OneSig(pk_{U_i}) \land \text{RelTime}(t_\leftarrow + \Delta))$
   - rMap($U_i$) := $R_i$
2) $rList = \text{rMap.values().shuffle}()$
3) Let $\theta_{\epsilon} := (\alpha, \text{MultiSig}(U_0, U_0))$
4) $tx\leftarrow := (tx\leftarrow, output[0], [\theta_{\epsilon}, outputMap.values()])$
5) Create a map stealthMap that stores for every user $U_i$ a key in outputMap the corresponding output of $tx\leftarrow$ corresponding to outputMap($U_i$)
6) Create two empty lists $\emptyset$ named msgList, userList
7) For every $U_i \in \text{nodeList}$ from $U_0$ to $U_n$ in descending order:
   - Append $[\mathcal{H}(tx\leftarrow)]$ to msgList
   - Prepend $[U_i]$ to userList
   - onion := $CreateRoutingInfo(userList, msgList)$
   - $\text{outputMap}(U_i) := \text{onion}$
8) Return $(tx\leftarrow, onions, rMap, rList, stealthMap)$

**genStateOutputs($\pi, \alpha, T$:)
1) Let $\theta_{\leftarrow} := \pi_{st}$ be the current state of the channel $\pi_{st}$.
2) Let $U_1 := \pi_{st.left} = U_{i+1} := \pi_{st.right}$.
3) $\theta_{\epsilon}$ consists of the outputs $\theta_{\epsilon} := (x_{U_1}, \text{OneSig}(U_1))$ and $\theta_{\epsilon + 1} := (x_{U_{i+1}}, \text{OneSig}(U_{i+1}))$ holding the balances of the two users.
4) Create the output vector $\theta_{\epsilon} := (\theta_0, \theta_1, \theta_2)$, where:
   - $\theta_0 := (\alpha, \text{MultiSig}(U_0, U_{i+1}) \land \text{RelTime}(T))$
   - $\theta_1 := (x_{U_1} - \alpha, \text{OneSig}(U_1))$
   - $\theta_2 := (x_{U_{i+1}} - \alpha, \text{OneSig}(U_{i+1}))$
5) Return $\theta_{\epsilon}$.

**genNewState($\pi, \alpha, T$:)
1) Let $\theta_{\epsilon} := \pi_{st}$.
2) Let $\alpha := \theta_{\epsilon}[0].cash$
3) Set $\theta_0 := (\alpha, \theta_{\epsilon}[0].\phi)$
4) Set $\theta_1 := (\theta[1].cash + \alpha - \alpha', \theta_{\epsilon}[1].\phi)$
5) Set $\theta_2 := \theta[2]$
6) Return vector $\theta_{\epsilon} := (\theta_0, \theta_1, \theta_2)$

**genRefTx($\theta, \alpha, U_1$):
1) Create a transaction $tx'_{\epsilon}$ with $tx'_{\epsilon}.input := [\theta, \theta_{\epsilon}]$ and $tx'_{\epsilon}.output := [\theta, \text{OneSig}(U_1)]$
2) Return $tx'_{\epsilon}$

**genPayTx($\theta, U_{i+1}$):
1) Create a transaction $tx'_{\leftarrow}$ with $tx'_{\leftarrow}.input := [\theta]$ and $tx'_{\leftarrow}.output := [\theta, \text{OneSig}(U_{i+1})]$
2) Return $tx'_{\leftarrow}$

\footnote{Possibly other outputs $\{\theta_{\epsilon}'\}_{\geq 1}$ 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.}
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.

2. Prerequisites

Stealth addresses. In order to hide the underlying path, we use stealth addresses [39] for the outputs in the transaction tx\textsuperscript{ex}. 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 \(P\) [39].

Onion routing. Like in the Lightning Network, we rely on onion routing [29] techniques like Sphinx [16] 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 encryption of routing information and a payload. Each user \(U\) controls two private keys \(p\) and \(b\). The user \(U\) then can receive as \(\sigma\), \(\nu\) and only the \(\nu\) knows its internal state. The parties and the adversary observes the messages that are output by the protocol.

Further, we assume that parties communicate via authenticated channels with guaranteed delivery after precisely one round. This is modeled by the ideal functionality \(\mathcal{F}_{\text{GDC}}\): 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 \(A\) 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 \(\mathcal{F}_{\text{GDC}}\) we refer to [19].

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

2. Ledger and channels

We use the global ideal functionality \(\mathcal{G}_{\text{Ledger}}\) to model a UTXO based blockchain, parameterized by \(\Delta\), an upper bound on the number of rounds it takes for a valid transaction to be accepted (the blockchain delay) and a signature scheme \(\Sigma\). \(\mathcal{G}_{\text{Ledger}}\) communicates with a fixed set of parties \(\mathcal{P}\). The environment \(\mathcal{E}\) first initializes \(\mathcal{G}_{\text{Ledger}}\) by setting up a key pair \((sk_{P}, pk_{P})\) for every party \(P\in\mathcal{P}\) and registers it to the ledger by sending \((\text{sid.REGISTER}, pk_{P})\) to \(\mathcal{G}_{\text{Ledger}}\). 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 \((\text{sid.POST}, \mathbf{tx})\). If a transaction is valid, it will be appear on \(\mathcal{L}\) after at most \(\Delta\) 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 \(A\). For a more accurate formalization, we refer to works such as [10]. To increase readability, we opted for these simplifications.

Channels are handles by the functionality \(\mathcal{F}_{\text{Channel}}\) [7], which is an extension of [8] and builds on top of \(\mathcal{G}_{\text{Ledger}}\). \(\mathcal{F}_{\text{Channel}}\) allows 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 \(t_{o}\) 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 \(t_{o}\) rounds and closing a channel, regardless if the parties are cooperating or not, takes at most \(t_{c}\) rounds. Finally, \(t_{c}\) 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

APPENDIX F

UC MODELING

For our formal security analysis, we utilize the global UC framework (GUC) [14]. 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 [7]–[9], [19]–[21].

1. Preliminaries, communication model and threat model

A protocol \(\Pi\) is executed between a set of parties \(\mathcal{P}\) and runs in the presence of an adversary \(A\), who receives as input a security parameter \(\lambda \in \mathbb{N}\) along with an auxiliary input \(z \in \{0,1\}^{*}\). \(A\) 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 \(A\) takes control over \(P_{i}\) and learns its internal state. The parties and the adversary \(A\) 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 \(\mathcal{G}_{\text{clock}}\) [27], 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.
between them are already open. We present an API along with an explanation of $F_{Channel}$ in Figure 18 and of $G_{Ledger}$ below. For increased readability, we hide the calls to $G_{clock}$ and $F_{GDC}$ in our notation. Instead of explicitly calling these functionalities, we write $(msg) \rightarrow X$ to denote sending message $(msg)$ to $X$ in round $t$ and $(msg) \leftarrow X$ to denote receiving message $(msg)$ from $X$ at time $t$. The sending/receiving entity as well as $X$ are either a party $P \in P$, the environment $E$, the simulator $S$ or another ideal functionality.

**Interface of $G_{Ledger}(\Delta, \Sigma)$ [8]**

<table>
<thead>
<tr>
<th>Parameters:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Delta$: upper bound on the number of rounds it takes a valid transaction to be published on $L$</td>
<td></td>
</tr>
<tr>
<td>$\Sigma$: a digital signature scheme</td>
<td></td>
</tr>
</tbody>
</table>

**API:**

Messages from $E$ via a dummy user $P \in P$:

- $(\text{sid}, \text{REGISTER}, \text{pk}_P) \xrightarrow{\tau} P$:
  
  This function adds an entry $(\text{pk}_P, P)$ to PKI consisting of the public key $\text{pk}_P$ and the user $P$, if it does not already exist.

- $(\text{sid}, \text{POST}, \mathbb{E}) \xrightarrow{\tau} P$:
  
  This function checks if $\mathbb{E}$ is a valid transaction and if yes, accepts it on $L$ after at most $\Delta$ rounds.

1) The UC-security definition: Closely following [9], we define $\Pi$ as a hybrid protocol that accesses the ideal functionalities $F_{\text{prelim}}$ consisting of $F_{Channel}$, $G_{Ledger}$, $F_{GDC}$ and $G_{clock}$. An environment $\mathcal{E}$ that interacts with $\Pi$ and an adversary $\mathcal{A}$ will on input a security parameter $\lambda$ and an auxiliary input $z$ output $\text{EXEC}_{\Pi, \mathcal{A}, \mathcal{E}}^{F_{\text{prelim}}}(\lambda, z)$. Moreover, $\phi_{F_{VC}}$ denotes the ideal protocol of ideal functionality $F_{VC}$, where the dummy users simply forward their input to $F_{VC}$. It has access to the same functionalities $F_{\text{prelim}}$. The output of $\phi_{F_{VC}}$ on input $\lambda$ and $z$ when interacting with $\mathcal{E}$ and a simulator $S$ is denoted as $\text{EXEC}_{\phi_{F_{VC}}, S, \mathcal{E}}^{F_{\text{prelim}}}(\lambda, z)$.

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

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

$$
\{ \text{EXEC}_{\Pi, \mathcal{A}, \mathcal{E}}^{F_{\text{prelim}}} (\lambda, z) \}_{\lambda \in \mathbb{N}, z \in \{0,1\}^*} \approx^{\Sigma} \{ \text{EXEC}_{\phi_{F_{VC}}, S, \mathcal{E}}^{F_{\text{prelim}}} (\lambda, z) \}_{\lambda \in \mathbb{N}, z \in \{0,1\}^*}
$$

where $\approx$ denotes computational indistinguishability.

3. Ideal functionality

In this section we explain our ideal functionality (IF) $F_{VC}$ in prose. Note that the IF is capable of outputting an error message, e.g., when a transaction does not appear on the ledger after instructing the simulator. We remark that the only protocols that realize this IF that are of interest to us are the ones that never output error. The cases where error is output are not meaningful to us and any guarantees are lost. We use the extended macros defined in Appendix 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 [9]. To be transparent about the similarities to [9] 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, $F_{VC}$ takes care of creating all necessary object, such as tx\textsuperscript{vc}, the onions, the stealth addresses, etc. and calls PRE-CREATE of $F_{Channel}$ to set up the VC with $U_n$. Afterwards, $F_{VC}$ 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, $F_{VC}$ instructs the simulator $S$ to simulate the view of the attacker. Additionally, $F_{VC}$ 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 $S$ 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, $F_{VC}$ expects a confirmation from $U_n$ via $S$. If an incorrect or no confirmation was received.
Upon the VC is open, the two endpoints can use the following phases:

1. **Pre-update**: This phase is similar to the Open-VC phase, but it involves the creation of a new state for the VC. If no such confirmation was received until $t - t_\text{c} = 3\Delta$, publish $\text{tx}^{v_c}(\sigma_i, \text{tx}_i^{v_c})$.

2. **Update-VC**: If $\text{tx}_i^{\text{state}}$ is not a correct signature of $U_{i+1}$ for the $\text{tx}_i^{v_c}$ created in step 2, return $\bot$.

3. **Close/InitProlong**: Execute $2\text{pModify}(\gamma, \text{tx}^{v_c}, \alpha_i, T')$.

4. **Emergency-Offload**: If $U_0$ has not successfully performed $2\text{pModify}$ with the correct value $\alpha_i$ (plus fee for each hop) until $T - t_\text{c} = 3\Delta$, publish $\text{tx}^{v_c}(\sigma_i^{v_c}, \text{tx}_i^{v_c})$. Else, update $T' = T$.

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

1. If $\tau_i < T - t_\text{c} = 3\Delta$ and $\text{tx}^{v_c}$ on the blockchain, $\text{closeChannel}(\gamma_{i-1})$ and, after $\text{tx}^{\text{state}}_{i-1}$ is accepted on the blockchain within at most $t_\text{c}$ rounds, wait $\Delta$ rounds.

2. If $\tau_i > T$, $\gamma_{i-1}$ is closed and $\text{tx}^{\text{state}}_{i-1}$ is on the blockchain, but not $\text{tx}^{\text{state}}_{i-1}$, publish $\text{tx}^{\text{state}}_{i-1}(\sigma_i, \text{tx}^{\text{state}}_{i-1}))$.

**Fig. 16**: Pseudocode of the protocol.

in the correct round, the IF instructs the simulator to publish $\text{tx}^{v_c}$, offloading the VC.

**Update-VC**: While the VC is open, the two endpoints can use $\text{PRE-UPDATE}$ of $\mathcal{F}_{\text{Channel}}$ 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 $\mathcal{F}_{\text{VC}}$ 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 $S$ 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

**Fig. 17**: Protocol for 2-party channel update.
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 \( E \) via a dummy user \( P \):

- \((\text{id}, \text{CREATE}, \pi, \text{id}_p) \xrightarrow{P} \pi\):
  Let \( \pi \) be the attribute tuple \((\pi, \text{id}, \pi, \text{users}, \pi, \text{cash}, \pi, \text{st})\), where \( \pi, \text{id} \in \{0, 1\}^* \) is the identifier of the channel, \( \pi, \text{users} \subseteq P \) are the users of the channel (and \( \pi, \text{users} \subseteq P \) are the users of the channel and \( P \subseteq \pi, \text{users} \)), \( \pi, \text{cash} \in \mathbb{R}^{>0} \) is the total money in the channel and \( \pi, \text{st} = \pi, \text{st} \) is the initial state of the channel. When invoked, this function asks \( \pi, \text{otherParty} \) to create a new channel.

- \((\text{id}, \text{UPDATE}, \text{id}, \theta) \xrightarrow{P} \pi\):
  Let \( \pi \) be the channel where \( \pi, \text{id} = \text{id} \). When invoked by \( P \subseteq \pi, \text{users} \) and both parties agree, the channel \( \pi \) (if it exists) is updated to the new state \( \theta \). 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 \( t_o \) is the upper bound that an update takes. In the successful case, \((\text{id}, \text{UPDATE}, \text{id}, \theta) \xrightarrow{t_o} \pi, \text{users} = \text{output} \).

Fig. 18: Interface of \( \mathcal{F}_{\text{Channel}}(T, k) [8] \).

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, \( \mathcal{F}_{\text{VC}} \) instructs \( S \) to offload the VC.

Respond. This phase is executed in every round and in it, \( \mathcal{F}_{\text{VC}} \) observes if a transaction \( tx^c \) 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, \( \mathcal{F}_{\text{VC}} \) 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, \( \mathcal{F}_{\text{VC}} \) instructs the simulator to publish the payment transaction.
Continue: //Continue after a dishonest user
1) Upon (sid, pid, continue, nodeList, tx, onions, rMap, 
    rList, stealthMap, α1, T, γn−1) ↝ S
2) Open(pid, nodeList, tx, onions, rMap, rList, 
    stealthMap, α1, T, γn−1).

Check: //Sim. can check that id was not yet used
1) Upon (sid, pid, check-id, tx, θ, R, U1−1, U1, U1+1, 
    α, T) ↝ S
2) If (pid, U) \notin idSet, let idSet := idSet ∪ \{(pid, U}\} and send the message 
   (sid, pid, OPEN, tx, θ, R, U1−1, U1+1, α1, T) ↝ U
3) If (sid, pid, OPEN-ACCEPT, τ) ↝ U, (sid, pid, ok, τ) ↝ S.

VC-Open: //Mark VC as opened
1) Upon (sid, pid, vc-open, tx) ↝ S, let Ψ := Ψ ∪ 
   \{(pid, tx)\}.

Register: //Sim. can register a channel
1) Upon (sid, pid, register, τ, θ, tx, θ, R, T) ↝ S
2) G := Ψ ∪ \{(pid, τ, θ, tx, T, θ, R)\}.

Open(pid, nodeList, tx, onions, rMap, rList, stealthMap, α1, T, 
    γn−1): Let τ be the current round and U1 := γn−1 ethos.
1) If (pid, U) ∈ idSet, go idle.
2) idSet := idSet ∪ \{(pid, U)\}.
3) If an entry after U in nodeList exists and is \( \downarrow \), go idle.
4) If \( U_1 = U \) ethos (i.e., last entry in nodeList), set \( U_1+1 := \top \). Else, wait \( U_1+1 \) from nodeList (the entry after \( U \)).
5) \( \bar{R}_i := rMap(U) \) and \( \bar{θ}_i := stealthMap(U) \)
6) \( \bar{θ}_{i−1} := genStateOuts(γn−1, α1, T) \) if \( \bar{θ}_{i−1} = \top \), go idle. Else, wait \( I \) round.
7) (sid, pid, OPEN-ACCEPT, τ) \( \neq \) U i, go idle. Else, wait \( I \) round.
8) (sid, pid, OPEN-ACCEPT, τ) \( \neq \rightarrow U, \) go idle. Else, wait \( I \) round.
9) (ssidc, UPDATE, γn−1, id, \bar{θ}_i−1) \( \neq \rightarrow F_{Channel} \)
10) (ssidc, UPDATED, τ, id) \( \rightarrow \rightarrow F_{Channel} \), else go idle.

12) If \( U_1 = U \) ethos:
   • Ψ := Ψ ∪ \{(pid, tx)\}
   • (sid, pid, VC-OPENED, tx, T, α1−1) \( \rightarrow \rightarrow U \)
   • If \( U_0 \) is dishonest, send (sid, pid, finalize, tx) \( \rightarrow \rightarrow S \)

13) Else:
   • (sid, pid, OPENED) \( \rightarrow \rightarrow U \)
   • If \( U_1+1 \) ethos, execute Open(pid, nodeList, tx, onions, 
     rMap, rList, stealthMap, α1−1 = fee, \( \bar{θ}_{i−1} \) = stealthMap(U1+1)
   • Else, send (sid, pid, open, tx, R, list, onions+1, α1−1 = fee, T, \( \bar{θ}_{i−1}−1, \bar{θ}_{i−1} \) \( \rightarrow \rightarrow S \), where onions+1 := onions(U1+1) and \( \bar{θ}_{i−1}−1 := stealthMap(U1+1) \)

Finalize-Open (executed at every round)

For every (pid, U0) ∈ Φ, keyList() do the following:
1) Let (τ, tx, U0) = Φ(pid, U0). If for the current round \( τ \) it holds that \( \tau = τ_r \), go idle.
2) If \( U_0 \) ethos, check if (pid, tx) ∈ Ψ. If yes, let Ψ := Ψ \ \{(pid, tx)\} and go idle.
3) If \( U_0 \) dishonest and (sid, pid, confirmed, tx, \sigma_{U_0}(tx)) \( \tau \rightarrow S \) such that tx = tx and \sigma_{U_0}(tx) is \( U_0 \)’s valid signature of tx, go idle.
4) Send (sid, pid, offload, tx, U0) \( \rightarrow \rightarrow S \) and remove key and value for key (pid, U0) from Φ. tx must be on \( L \) in round \( τ' \leq \tau + \Delta \). Otherwise, output (sid, ERROR) \( \rightarrow \rightarrow U_0 \).

Update-VC

While VC is open, the sending and the receiving endpoint can update the VC using PRE-UPDATE of FChannel just as they would a ledger channel.

Close-VC

Let \( τ \) be the current round.

Start:
1) Upon (sid, pid, SHUTDOWN, α', \( \gamma_{n−1} \)) \( \rightarrow \rightarrow U \), for parameter \( pid \), fetch entry (pid, \( \gamma_{n−1}, \bar{θ}_i, tx, T, \theta', R_i \)) from \( Γ \), s.t. \( \gamma_{n−1} ethos \) = \( \gamma_{n−1} ethos \). If there is no such entry, go idle.
2) Let \( U_{n−1} := \gamma_{n−1} ethos \).
3) If \( U_{n−1} ethos \) is not the endpoint in VC, go idle.
4) If \( U_{n−1} ethos \) honest, execute Close(pid, \( \gamma_{n−1}, α' \)).
5) Else, send (sid, pid, close, α', \( \gamma_{n−1}, \gamma_{n−1} ethos \)) \( \rightarrow \rightarrow S \).

Continue-Close: //Continue after a dishonest user
1) Upon (sid, pid, continue-close, \( \gamma_{n−1}, α' \)) \( \rightarrow \rightarrow S \)
2) Close(pid, \( \gamma_{n−1}, α' \)).

Close(pid, \( \gamma_{n−1}, α' \)): Let \( τ \) be the current round and \( U_i := \gamma_{n−1} ethos \).
1) For the parameters \( pid \) and \( \gamma_{n−1} ethos \), fetch entry (pid, \( \gamma_{n−1}, \bar{θ}_i, tx, T, \theta', R_i \)) from \( Γ \). If there is no entry where the parameters \( pid \) and \( \gamma_{n−1} ethos \) match, go idle.
2) If \( \gamma_{n−1} ethos \) ethos \( \bar{θ}_i ethos \), go idle.
3) Let \( α_i := \bar{θ}_i ethos \). If not \( 0 < α_i ethos < α_i ethos \), go idle.
4) \( \bar{θ}_i := genNewState(\gamma_{n−1} ethos, T) \). If \( \bar{θ}_i ethos = \top ethos \), \( \gamma_{n−1} ethos \) ethos. Else, wait \( I \) round.
5) (sid, pid, CLOSE, \( \gamma_{n−1} ethos, α' ethos \)) \( \rightarrow \rightarrow U_i \).
6) If not (sid, pid, CLOSE-ACCEPT) \( \rightarrow \rightarrow U_i \), go idle.
7) (ssidc, UPDATE, \( \gamma_{n−1} ethos, \bar{θ}_i ethos \)) \( \rightarrow \rightarrow F_{Channel} \).
8) If not (ssidc, UPDATED, \( \gamma_{n−1} ethos, \bar{θ}_i ethos \)) \( \rightarrow \rightarrow F_{Channel} \), go idle.
9) G := \( Γ \) \( \Gamma \) \( \gamma_{n−1} ethos, \bar{θ}_i ethos, tx, T, \theta', R_i \).
10) G := G \( \Gamma \) \( \gamma_{n−1} ethos, \bar{θ}_i ethos, tx, T, \theta', R_i \).
11) If \( U_i = U_0 \),
   • (sid, pid, VC-CLOSED) \( \rightarrow \rightarrow U_1 \).
   • Remove key and value for key (pid, U0) from Φ.
12) Else:
   • Retrieve \( \gamma_{n−1} ethos \) from \( Γ \) ethos \( \gamma_{n−1} ethos \), s.t. \( \gamma_{n−1} ethos \) ethos = \( \gamma_{n−1} ethos \).
   • (sid, pid, CLOSED) \( \rightarrow \rightarrow U_0 \).
   • If \( U_i ethos \ ethos \) honest, execute Close(pid, \( \gamma_{n−1} ethos, α' ethos \) ethos, \( \gamma_{n−1} ethos \) ethos, \( \gamma_{n−1} ethos \) ethos).
   • Else, send (sid, pid, close, \( α' ethos \) ethos, \( \gamma_{n−1} ethos \) ethos, \( \gamma_{n−1} ethos \) ethos).

Replace: //Update the state currently saved by the IF
1) Upon (sid, pid, replace, \( \gamma_{n−1}, \bar{θ}_i ethos \)) \( \rightarrow \rightarrow S \), let \( U_i := \gamma_{n−1} ethos \).
2) For parameters \( pid \) and \( \gamma_{n−1} ethos \), fetch entry (pid, \( \gamma_{n−1}, \bar{θ}_i ethos, T, \theta_i ethos, R_i \) ∈ \( Γ \).
3) G := G \( \Gamma \) \( \gamma_{n−1}, \bar{θ}_i ethos, Tx, T, \theta_i ethos, R_i \).
4) G := G \( \Gamma \) \( \gamma_{n−1}, \bar{θ}_i ethos, Tx, T, \theta_i ethos, R_i \).
5) If \( U_i = U_0 \), remove key and value for key (pid, U0) from Φ.

Emergency-Offload (executed at every round)
Let $\tau$ be the current round. For every $(\text{pid}, U_0) \in \Phi$.keyList() do the following:

1) For $\text{pid}$ and a channel $\gamma_0$ where $\gamma_0.left = U_0$, fetch entry $(\text{pid}, \gamma_0, \theta_0, \text{tx}^\infty, T, \theta_0, R_0) \in \Gamma$.

2) If $\tau < T - t_c - 2\Delta$, continue with next loop iteration.

3) Else, let $(\gamma_i, \text{tx}^\infty, U_i) = \Phi(\text{pid}, U_0)$. Send $(\text{sid}, \text{pid}, \text{offload}, \text{tx}^\infty, U_i) \trans S$. $\text{tx}^\infty$ must be on $L$ in round $\tau^i \leq \tau + \Delta$. Otherwise, output $(\text{sid}, \text{ERROR}) \trans U_i \text{.}

4) Remove key and value for key $(\text{pid}, U_0)$ from $\Phi$.

Respond (executed at the end of every round)

Let $t$ be the current round. For every element $(\text{pid}, \gamma_i, \theta_i, \text{tx}^\infty, T, \theta_i, R_i) \in \Gamma$, check if $\gamma_i.st = \theta_i$ and $\text{tx}^\infty$ is on $L$. If yes, do the following:

Revoke: If $\gamma_i.left$ honest and $t < T - t_c - 2\Delta$ do the following.

- Set $\Gamma := \Gamma \setminus \{(\text{pid}, \gamma_i, \theta_i, \text{tx}^\infty, T, \theta_i, R_i)\}$.
- $(\text{ssid}, \text{CLOSE}, \gamma_i, \text{id}) \trans \text{FChannel}$.
- At time $t + t_c$, a transaction $tx$ on $\text{tx}.output = \gamma_i.st$ has to be on $L$. If not, do the following. If $(\text{ssid}, \text{PUNISHED}, g_i, \text{id}) \trans \gamma_i$, go idle. Else, send $(\text{sid}, \text{ERROR}) \trans \gamma_i$.

Wait: If $\Delta$ rounds do the following. If

- $(\text{sid}, \text{pid}, \text{post-refund}, \gamma_i, \theta_i, R_i) \trans \gamma_i$.
- At time $t_c^\forall < T$, check whether a transaction $tx$ on $\text{tx}.output = \gamma_i.st$ is on $L$ with $\text{tx}.input = \theta_i, \text{tx}.output[0]$ on $\text{tx}.output[0]$ and $\text{tx}.output = \theta_i, \text{tx}.output[0]$. $\text{tx}.output[0]$ containing the channel with its left neighbor and the state and the transaction $\text{tx}_i$ for $U_i$’s left channel in the communication.

Force-Pay: Else, if a transaction $tx$ on $\text{tx}.output = \gamma_i.st$ is on $L$.

- Set $\Gamma := \Gamma \setminus \{(\text{pid}, \gamma_i, \theta_i, \text{tx}^\infty, T, \theta_i, R_i)\}$.
- $(\text{sid}, \text{pid}, \text{post-pay}, \gamma_i, \text{id}) \trans \text{FChannel}$.
- In round $t + \Delta$ transaction $\text{tx}^\infty_{\gamma_i} \text{tx}.input = \theta_i, \text{tx}.output[0]$ and $\text{tx}.output = \theta_i, \text{tx}.output[0]$. $\text{tx}.output[0]$ containing the channel with its right neighbor, the state, the transaction $\text{tx}_i$ and the key necessary for signing the refund transaction in the payment $\text{pid}$.

right: A map, sorting for a given $\text{pid}$ $U_i$’s local copy of $\text{tx}^\infty$ and $T$ in a tuple $(\text{tx}^\infty, T)$.

rightSig: A map, storing for a given $\text{pid}$ $U_i$’s local copy of $\text{tx}^\infty$ and $T$ in a tuple $(\text{tx}^\infty, T)$.

4. Protocol

In this section we give the formal protocol $\Pi$ 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 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 V. The main differences lie in having VC ids that allow handling multiple different VCs, the notion of time and the environment $E$. Briefly, the protocol starts with $E$ invoking $U_0$ to set up the initial objects and pre-create the VC with $U_0$. 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 $\text{tx}^\infty$ 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 [9].

Protocol $\Pi$

Let $fee \in \mathbb{N}$ be a system parameter known to every user.

Local variables of $U_i$ (all initially empty):

- $\text{pidSet}$: A set storing every payment id for the user that has participated in to prevent duplicates.
- $\text{paySet}$: A map storing tuples $(\text{pid}, \gamma, U_n)$ where $\text{pid}$ is an id, $\gamma$ is the round in which a confirmation is expected from the receiver $U_n$ for the payments that have been opened by this user.
- $\text{local}$: A map, storing for a given $\text{pid}$ $U_i$’s local copy of $\text{tx}^\infty$ and $T$ in a tuple $(\text{tx}^\infty, T)$.
- $\text{left}$: A map, storing for a given $\text{pid}$ $U_i$’s local copy of $\text{tx}^\infty$ and $T$ in a tuple $(\text{tx}^\infty, T)$.

Setup: In every round, every node $U_0 \in \mathcal{P}$ does the following. We denote $\gamma_0$ as the current round.

$U_0$ upon $(\text{sid}, \text{pid}, \text{SETUP}, \text{channelList}, \text{tx}^\infty, \alpha, T, \gamma_0) \trans E$

1) If $\text{pid} \notin \text{pidSet}$, abort. Add $\text{pid}$ to $\text{pidSet}$.
2) Let $x := \text{checkChannels}(\text{channelList}, U_0)$. If $x = \bot$, abort.
   Else, let $U_n := x$. If $\gamma_0$ is not the full channel between $U_0$ and his right neighbor $U_1 := \gamma_0.right$ (corresponding to the channel skeleton $\gamma_0$ 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 $n := \text{channelList}$. If $\text{check}(n, T) = \bot$, abort.
4) If $\text{checkTxIn}(\text{tx}^\infty, n, U_n, \alpha) = \bot$, abort.
5) $(\text{tx}^\infty, \text{onions}, \text{rMap}, \text{rList}, \text{stealthMap}) := \text{createMaps}(U_0, \text{nodeList}, \text{tx}^\infty, \alpha)$.
6) $(\text{tx}^\infty, \text{rList}, \text{onionO}) := \text{genTxVc}(U_0, \text{channelList}, \text{tx}^\infty)$
7) $\text{paySet} := \text{paySet} \cup \{(\text{pid}, \gamma_i, \tau := t + n \cdot (2 + t_c) + 2 + t_0, U_n)\}$
8) $(\text{sk}_0, \theta_0, R_0, U_0, \text{onion}) := \text{checkTxVc}(U_0, U_0, \text{tx}^\infty, \text{rMap}, \text{rList}, \text{onionO})$
9) Set load $(\text{pid}) := (\text{tx}^\infty, T)$.
10) Send $(\text{sid}, \text{pid}, \text{pre-create-vc}, \gamma_0, \text{tx}^\infty, T) \trans U_n$, wait 1 round.
11) Send $(\text{ssid}, \text{PRE-CREATE}, \gamma_0, \text{tx}^\infty, 0, T = \gamma_0) \trans \text{FChannel}$.
12) If not $(\text{ssid}, \text{PRE-CREATED}, \gamma_0, \text{id}) \trans \text{FChannel}$, go idle.
13) Set $\alpha_0 := \alpha + \text{fee} \cdot (n - 1)$ and compute:
   - $\theta_0 := \text{genStateOutputs}(\gamma_0, \alpha_0, T)$
   - $\text{tx}_0 := \text{genRefTx}(\theta_0, \theta_0, U_0)$

\footnote{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 $\text{tx}_0$, but we omit this for readability.}
14) Set right(pid) := ($\sigma_0$, $\vec{\theta}_0$, tx$_0$, sk$_{U_0}$).
15) Send $(sid,pid,open-req,tx^e,rList,\sigma_0,tx'_0)$ to $U_1$.

$U_n$ upon $(sid,pid,pre-create-vc,tx^e,T) \xrightarrow{\tau_n} U_0$

1) $(ssid_C,PRE-CREATE,tx^e,0,T-\tau_n) \xrightarrow{\tau_n} FC_{\text{Channel}}$

2) If not $(ssid_C,PRE-CREATED,tx^e,\text{id}) \xrightarrow{\tau_n} FC_{\text{Channel}}$ mark VC as unusable.

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

$U_{i+1}$ upon $(sid,pid,open-req,tx^e,rList,\sigma_0,tx'_0) \xrightarrow{\tau_x} U_i$

1) Perform the following checks:
   - Verify that $\tau_x \neq \tau$ and instead $x = (sk_{U_i-1},\vec{\theta}_{i+1},R_{i+1},U_{i+2},\sigma_{i+2})$.
   - Set $\alpha_i = \vec{\theta}_{i}[0]\text{cash}$ and extract $T$ from $\vec{\theta}_{i-1}[0]\phi$.
   - Check that there exists a channel between $U_i$ and $U_{i+1}$ and call this channel $\tau_i$. Verify that $\vec{\theta}_i = \sigma_{i+1}(tx')$. Otherwise stop.

2) If one or more of the previous checks fail, abort. Otherwise, send $(sid,pid,OPEN,tx^e,\vec{\theta}_{i+1},R_i,U_{i+1},\sigma_{i+1},T) \xrightarrow{\tau} \mathcal{E}$.

3) If $(sid,pid,OPEN-ACCEPT,\tau_{i+1}) \xrightarrow{\tau} \mathcal{E}$, generate $\sigma_{i+1}(tx')$. Otherwise stop.

4) Set local(pid) := $(tx^e_i,T)$, left(pid) := $(\tau_i,\vec{\theta}_i,tx_i)$ and $(sid,pid,open-ok,\sigma_{i+1}(tx'_i)) \xrightarrow{\tau} U_i$.

$U_i$ upon $(sid,pid,open-ok,\sigma_{i+1}(tx'_i)) \xrightarrow{\tau_{i+2}} U_{i+1}$

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

5) Check that $\sigma_{i+1}(tx'_i)$ is a valid signature for $tx'_i$. If yes, set rightSig(pid) := $\sigma_{i+1}(tx'_i)$ and $(ssid_C,UPDATE,\tau_i,\vec{\theta}_i) \xrightarrow{\tau_{i+2}} FC_{\text{Channel}}$.

$U_{i+1}$ upon $(ssid_C,UPDATE,\tau_i,\vec{\theta}_i) \xrightarrow{\tau_{i+1+tu}} FC_{\text{Channel}}$

6) Define $\tau_{i+1} := \tau_x + 1 + tu$.
7) If $U_{i+1}$ is not the receiver, using the values of step 1:
   - Send $(sid,pid,OPENED) \xrightarrow{\tau_{i+1}} \mathcal{E}$.
   - $(sk_{U_{i+1}},\vec{\theta}_{i+1},R_{i+1},U_{i+2},\sigma_{i+2},\text{id}) := \text{checkTxVC}(U_{i+1},U_{i+1},b,tx^e_{i+1},rList,\text{id})$
   - $\vec{\theta}_{i+1} := \text{genStateOut}(\tau_{i+1},\vec{\theta}_i-\text{fee},T)$.
   - $tx^e_{i+1} := \text{genRefTx}(\vec{\theta}_{i+1},tx_{i+1},U_{i+1})$.
   - Set right(pid) := $(\tau_{i+1},\vec{\theta}_{i+1},tx^e_{i+1})$.
   - Send the message $(sid,pid,open-req,tx^e_{i+1},rList,\sigma_{i+1}(tx'_i)) \xrightarrow{\tau_{i+1}} U_{i+2}$.

8) If $U_{i+1}$ is the receiver:
   - msg := GetRoutingInfo(\text{onion}_{i+1},U_{i+1})
   - Create the signature $\sigma_{i+1}(tx'_i)$ as confirmation and send $(sid,pid,finalize,tx^e_{i+1},\sigma_{i+1}(tx'_i)) \xrightarrow{\tau_{i+1}} U_0$. Send the message $(sid,pid,VC-OPENED,tx^e_{i+1},T,\alpha_i) \xrightarrow{\tau_{i+1}} \mathcal{E}$.

Finalize

For every entry $(pid,\sigma,tx^e) \in \text{paySet}$ do the following if $\tau = \pi$:

1) Upon receiving $(sid,pid,finalize,tx^e,\sigma_{i+1}(tx'_i)) \xrightarrow{\tau} U_n$, continue if $\sigma_{i+1}(tx'_i)$ is a valid signature for $tx^e$. Otherwise, go to step (3).

2) Let $(x,T) = \text{local(pid)}$. If $x = tx^e$, go idle. Otherwise, continue with the next step.

3) Sign $tx^e_{i+1}$ yielding $\sigma_{i+1}(tx^e_{i+1})$ and set $tx^e_{i+1} := (tx^e_{i+1},(\sigma_{i+1}(tx^e_{i+1})))$.

   Send $(sid_{\text{L}},POST,tx^e_{i+1}) \xrightarrow{\tau} \mathcal{G}_{\text{Ledger}}$ and remove $(pid,\sigma,tx^e) \in \text{paySet}$.

Update VC

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

Close VC

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

$U_n$ upon $(sid,pid,SHUTDOWN,\alpha'_{i+1}) \xrightarrow{\tau_n} \mathcal{E}$

1) If $pid \notin \text{pidSet}$, abort.
2) If $U_n$ is not the receiving endpoint set in the VC, abort.
3) Retrieve $(\tau_{i-1},\vec{\theta}_{i-1},tx^e_{i-1}) := \text{left(pid)}$.
4) Extract $\alpha_{i-1} \in tx^e_{i-1}$.input
5) Extract $T$ from $\vec{\theta}_{i-1}[0]\phi$.
6) Let $\alpha_i := \vec{\theta}_{i-1}[0]\text{cash}$. If not $\alpha_i \leq \alpha_i$, abort.

7) Create the signature $\sigma_{i+1}(tx^e_{i+1})$

   8) Send $(sid,pid\text{,close-req,}tx^e_{i+1},\sigma_{i+1}(tx^e_{i+1})) \xrightarrow{\tau} U_{i+1}$.

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

$U_i$ upon $(sid,pid,close-req,\vec{\theta}_i,tx^e_{i+1},\sigma_{i+1}(tx^e_{i+1})) \xrightarrow{\tau} U_{i+1}$

1) If $\vec{\theta}_i \notin \text{pidSet}$, abort.
2) If $\vec{\theta}_i \neq \text{tx}_{i+1}$, input, abort.
3) If $\tau_x \neq \tau_i$, abort.
4) Extract $\alpha_i \in tx^e_{i+1}$.input
5) Extract $T$ from $\vec{\theta}_{i}[0]\phi$ and $\alpha_i := \vec{\theta}_{i}[0]\text{cash}$.
6) Extract $T'$ from $\vec{\theta}_i[0]\phi$ and $\alpha_i := \vec{\theta}_i[0]\phi$.
7) If $T \neq T'$, abort. If not $\alpha_i \leq \alpha_i$, abort.
8) If $\vec{\theta}_i \neq \text{genNewState}(\tau_i,\alpha_i,\vec{\theta}_i)$, abort.
9) If $tx^e_{i} \neq \text{genRefTx}(\vec{\theta}_i,tx^e_{i},U_i)$, abort.
10) If $\sigma_{i+1}(tx^e_{i+1})$ is not a valid signature for $tx^e_{i+1}$, abort.
11) Send $(sid,pid,\text{CLOSE,}\alpha_i) \xrightarrow{\tau} \mathcal{E}$.
12) If not $(sid,pid,\text{CLOSE-ACCEPT}) \xrightarrow{\tau} \mathcal{E}$, abort.
13) $(ssid_C,UPDATE,\tau_i,\vec{\theta}_i) \xrightarrow{\tau_i} \mathcal{F}_{\text{Channel}}$. 

26
14) Set left(pid) := (\( \pi_i, \theta'_i, t x'_i \) )
\[ U_i \text{ upon (ssid}_i, \text{UPDATED}, \pi_i, \theta'_i, t x'_i) \xrightarrow{t_x + t_u} F_{\text{Channel}} \]

15) Let \( t_1 := t_x + t_u \)
16) Set rightSig(pid) := \( \sigma_{U_i} (t x'_i) \) and set right(pid) := (\( \pi_i, \theta'_i, t x'_i, sk_{U_i} \) )
17) If \( U_i \) is not the sending endpoint:
   - Retrieve \( (\pi_i, \theta'_i, t x'_i) := \text{left}(\text{pid}) \)
   - Extract \( \theta_{i-1} \in t x'_i \text{ input} \)
   - \( \theta'_{i-1} := \text{genNewState}(\pi_{i-1}, \alpha'_i + \text{fee}, T) \)
   - \( t x'_i := \text{genRefTx}(\theta'_{i-1}, \pi_i, \theta_i, U_{i-1}) \)
   - Create the signature \( \sigma_{U_i} (t x'_i) \)
   - Send \((\text{sid, pid, close-req, } \theta'_{i-1}, t x'_i, \sigma_{U_i} (t x'_i)) \)
     \( \xrightarrow{t_x + t_u} U_{i-1} \).
   - \((\text{sid, pid, CLOSED}) \xrightarrow{t} E \)
18) If \( U_i \) is the sending endpoint:
   - \((\text{sid, pid, VC-CLOSED}) \xrightarrow{t_x + t_u} E \)

\section*{Emergency-Offload}

\( U_0 \) in every round \( \tau \)

For every entry \((\text{pid, } \pi, U_n) \in \text{paySet}\) do the following:

1) Let \((t x^c, T) := \text{local}(\text{pid}) \).
2) If \( T - t_x < 3 \Delta \), continue with next loop iteration.
3) Remove \((\text{pid, } \pi, U_n) \) from \text{paySet}.
4) Sign \( t x^c \) yielding \( \sigma_{U_n} (t x^c) \) and set \( t x^c := (t x^c, (\sigma_{U_n} (t x^c))) \).
   - Send \((\text{sid}_i, \text{POST, } t x^c) \xrightarrow{t} G_{\text{Ledger}} \).

\section*{5. Simulation}

In this section we provide the code for the simulator \( S \), 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 \( F_{VC} \) shown in Appendix F.3.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
\textbf{Simulator} & \\
\hline
\textbf{Local variables:} & \\
\hline
\textbf{left} & A map, storing the channel \( \pi_{i-1} \) and output \( \theta_{i-1} \) for a given keypair consisting of a payment id \( \text{pid} \) and a user \( U_i \), or \((\perp, \perp)\) if \( U_i \) is the sending endpoint. \\
\hline
\textbf{right} & A map, storing the transaction \( t x'_i \) for a given keypair consisting of a payment id \( \text{pid} \) and a user \( U_i \). \\
\hline
\textbf{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 \( \text{pid} \) and a user \( U_i \). \\
\hline
\end{tabular}
\end{table}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
\textbf{Simulator for init phase} & \\
\hline
\textbf{Upon (sid, init, } t_x^\text{init} & \xrightarrow{t_x^\text{init}} F_{VC} \text{ and send} \\
(\text{sid, init-ok, } t_x^\text{init}, t_u) & \xrightarrow{t_x^\text{init}} F_{VC} \).
\hline
\end{tabular}
\end{table}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
\textbf{Simulator for Open-VC phase} & \\
\hline
\textbf{Pre-create VC} & \\
\hline
1) Upon (sid, pid, pre-create-vc, \( t x^c_i, T) \xrightarrow{t} U_0 \) if \( U_0 \) dishonest, go to step (3). \\
2) Upon (sid, pid, pre-create-vc, \( t x^c_i, T) \xrightarrow{t} F_{VC} \) if \( U_0 \) honest, do the following. If \( U_0 \) honest, go to step (3). If \( U_0 \) dishonest, send (sid, pid, pre-create-vc, \( t x^c_i, T) \xrightarrow{t} U_n \) and go idle. \\
3) (ssid, PRE-CREATE, \( t x^c_i, 0, T - \tau \) \xrightarrow{t} F_{Channel} \).
\hline
\textbf{a) Case U_1 is honest, U_{i+1} dishonest} & \\
\hline
1) Upon receiving (sid, pid, open, \( t x^c_i, rList, \text{onion}_{i+1}, \alpha_i, T, \pi_{i+1}, \theta_{i+1}, \theta_i, \alpha_i) \xrightarrow{t} F_{VC} \) or upon being called by the simulator \( S \) itself in round \( \tau \) with parameters (pid, \( t x^c_i, rList, \text{onion}_{i+1}, \alpha_i, T, \pi_{i+1}, \theta_{i+1}, \theta_i, \alpha_i \)). \\
2) Let \( U_i := \pi_{i+1}.left \text{ and } U_{i+1} := \pi_{i+1}.right \). \\
3) \( \theta_i := \text{genStateOuts}(\pi_i, \alpha_i, T) \).
\hline
\textbf{tx'_i := genRefTx(\theta_i, U_i, U_{i+1})} & \\
\hline
\textbf{5) (sid, pid, open-req, } t x^c_i, rList, \text{onion}_{i+1}, \theta'_i, t x'_i) \xrightarrow{t} U_{i+1} \text{ and check that} \\
\sigma_{U_{i+1}} (t x'_i) \text{ is a valid signature for } t x'_i. \text{ If not, go idle.} \\
\hline
\textbf{Set rightSig(pid, U_{i+1}) := } & \sigma_{U_{i+1}} (t x'_i). \text{ right(pid, U_{i+1}) := } t x'_i \text{ and go idle.} \\
\hline
\textbf{8) Send (ssid, UPDATE, } \pi_{i+1}, \theta'_i, t x'_i) \xrightarrow{t} F_{Channel} \).
\hline
\end{tabular}
\end{table}
9) If not \((\text{ssid}_{\text{c}}, \text{UPDATED}, \gamma_{\text{id}}, \tilde{\theta}_i) \xrightarrow{\tau+2+u_{\gamma}} \mathcal{F}_{\text{Channel}}, \text{go } \) idle.
10) Set \(\text{left}(\text{pid}, U_i) := (\gamma_{\text{id}}-1, \theta_{i-1})\).
11) Send \((\text{sid}, \text{pid}, \text{register}, \gamma_{\text{id}}, \tilde{\theta}_i, \text{tx}_{\text{c},i}, T, \theta_{i-1}, R) \xrightarrow{\tau} \mathcal{F}_{\text{VC}}.

b) Case \(U_i\) is honest, \(U_{i-1}\) dishonest

1) Upon \((\text{sid}, \text{pid}, \text{open}, \text{req}, \text{tx}_{\text{c},i}, \text{rList}, \text{onion}, \tilde{\theta}_{i-1}, \text{tx}_{\text{c},i-1}) \xrightarrow{\tau} U_{i-1}\).
   Let \(\alpha_{i-1} := \tilde{\theta}_{i-1}[0].\text{cash}\) and extract \(T\) from \(\tilde{\theta}_{i-1}[0].\phi\) (the parameter of AbsTime()). Let \(\gamma_{\text{id}} \neq 1\) be the channel between \(U_{i-1}\) and \(U_i\).
2) Let \(x := \text{checkTxVc}(U_i, \text{U}, a, b, \text{tx}_{\text{c},i}, \text{rList}, \text{onion}).\)
   Check that \(x \neq \bot\), but instead \(x = (\text{sk}_{\text{U}_i}, \tilde{\theta}_i, R, U_{i-1}, \text{onion}_{i+1})\). Otherwise, go idle.
3) Check that there exists a channel between \(U_i\) and \(U_{i+1}\) and call this channel \(\gamma_{i-1}\). Verify that \(\tilde{\theta}_i := \text{genStateOutputs}(\gamma_{i-1}, \alpha_{i-1}, T)\) and \(\text{tx}_{i-1} := \text{genRefTx}(\tilde{\theta}_i, \gamma_{i-1}, U_i)\), where \(\theta_{i-1} \in \text{tx}_{\text{c},i}\) and \(\theta_{i-1} \neq \tilde{\theta}_i\).
4) \((\text{sid}, \text{pid}, \text{check-id}, \text{tx}_{\text{c},i}, \theta_{i-1}, R, U_{i-1}, U_{i+1}, \alpha_i, T) \xrightarrow{\tau} \mathcal{F}_{\text{VC}}.
5) If not \((\text{sid}, \text{pid}, \text{ok}, \gamma_{i-1}) \xrightarrow{\tau} \mathcal{F}_{\text{VC}}, \text{go idle. Let } U_{i+1} := \gamma_{i-1}.right.\)
6) Sign \(tx_{i-1}^{\prime}\) on behalf of \(U_i\) yielding \(\sigma_{U_i}(tx_{i-1})\).
7) Upon \((\text{ssid}_{\text{c}}, \text{UPDATED}, \gamma_{i-1}, \text{id}, \tilde{\theta}_{i-1}) \xrightarrow{\tau+1+u_{\gamma}} \mathcal{F}_{\text{Channel}}, \text{send }\)
   \((\text{sid}, \text{pid}, \text{register}, \gamma_{i-1}, \tilde{\theta}_{i-1}, \text{tx}_{\text{c},i}, T, \bot, \bot) \xrightarrow{\tau} \mathcal{F}_{\text{VC}}.
   \text{Otherwise, go idle.}
8) Set \(\text{left}(\text{pid}, U_i) := (\gamma_{i-1}, \theta_{i-1}).\)
9) If \(U_i = U_n\) (if \((\text{sk}_{\text{U}}, \theta_{i}, R, U_{i+1}, \text{onion}_{i+1}) = (T, T, T, T, T)\) holds), and \(U_0\) is honest, send \((\text{sid}, \text{pid}, \text{open}, \text{tx}_{\text{c},i}) \xrightarrow{\tau+1+u_{\gamma}} \mathcal{F}_{\text{VC}}.\)
   If \(U_0\) is dishonest, create signature \(\sigma_{U_0}(\text{tx}_{\text{c},i})\) on behalf of \(U_0\) and send \((\text{sid}, \text{pid}, \text{finalize}, \text{tx}_{\text{c},i}, \sigma_{U_0}(\text{tx}_{\text{c},i})) \xrightarrow{\tau+1+u_{\gamma}} U_0.\)
   In both cases, send via \(\mathcal{F}_{\text{VC}}\) to the dummy user \(U_n\) the message \((\text{sid}, \text{pid}, \text{OPEN}, \text{tx}_{\text{c},i}, T, \alpha_{i-1}) \xrightarrow{\tau+1+u_{\gamma}} U_0).\) Go Idle.
10) Send via \(\mathcal{F}_{\text{VC}}\) to the dummy user \(U_i\) the message \((\text{sid}, \text{pid}, \text{OPEN}) \xrightarrow{\tau+1+u_{\gamma}} U_i).\)
11) If \(U_{i+1}\) honest, call process \((\text{sid}, \text{pid}, \text{tx}_{\text{c},i}, \gamma_{i-1}, \tilde{\theta}_i, R, \text{onion}, \gamma_{i-1}, T).\)
12) If \(U_{i+1}\) dishonest, go to step \(\text{Simulator for Close-VC phase}\) with parameters \((\text{pid}, \text{tx}_{\text{c},i}, \gamma_{i-1}, \alpha_{i-1}) = \text{fee}, \gamma_{i-1}, \tilde{\theta}_i, \gamma_{i-1}, \theta_{i-1}).\)

process\((\text{sid}, \text{pid}, \text{tx}_{\text{c},i}, \gamma_{i-1}, \gamma_{i-1}, \tilde{\theta}_i, \text{onion}, \gamma_{i-1}, T)\)

\(\text{Let } \tau \text{ be the current round.}\)

1) Initialize \(\text{nodeList} := \{U_i\}\) and onions, \(\text{rMap}, \text{stealthMap}\) as empty maps.
2) \((U_{i+1}, \text{msg}, \text{onion}_{i+1}) := \text{GetRoutingInfo}(\text{onion}_{i}).\)
3) \(\text{stealthMap}(U_i) := \theta_i.\)
4) \(\text{rMap}(U_i) := R_i.\)
5) While \(U_i\) and \(U_{i+1}\) honest:
   a) \(x := \text{checkTxVc}(U_{i+1}, U_{i+1}, a, b, \text{tx}_{\text{c},i}, \text{rList}, \text{onion}_{i+2}).\)
      \(\text{o If } x = \bot, \text{append } U_{i+1} \text{ and then } \bot \text{ to } \text{nodeList} \text{ and break the loop.}\)
      \(\text{o If } x = \bot, \text{append } U_{i+1} \text{ to } \text{nodeList} \text{ and break the loop.}\)
      \(\text{Else, if } x = (\text{sk}_{U_{i+1}}, \theta_{i+1}, U_{i+2}, \text{onion}_{i+2}), \text{do the following.}\)
      \(\text{o Append } U_{i+1} \text{ to } \text{nodeList}\)
      \(\text{o } \text{onions}(U_{i+2}) := \text{onion}_{i+2}\)
      \(\text{o } \text{rMap}(U_{i+2}) := R_{i+2}\)
we show that the execution ensembles \( \mathcal{F} \) are \( \equiv \mathcal{F} \) with the view of the environment \( \mathcal{E} \). These functionalities might in turn interact with the view of the environment \( \mathcal{E} \). A more formally, we show that the execution ensembles \( \mathcal{E} \) 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 \( \tau \). 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 \( \text{obsSet}(m, \mathcal{F}, \tau) \), returning a set of all of \( \mathcal{E} \) observable actions which are triggered by calling \( \mathcal{F} \) with message \( m \) in round \( \tau \).

In this proof, we do a case-by-case analysis of each corruption setting. We start with the view of the environment \( \mathcal{E} \) 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, we simulate the Finalize as well as the Respond phase and the Pay, Finalize and Respond phase in [9], parts of the corresponding proofs are taken verbatim from there.

**Lemma 1.** Let \( \Sigma \) be an EUF-CMA secure signature scheme. Then, the Open-VC phase of \( \Pi \) \( \mathcal{G} \)-emulates the Open-VC phase of functionality \( \mathcal{F} \).

**Proof.**

We compare the execution ensembles for the open phase in the real and the ideal world. In Table VI 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:

- \( m_{0} := (\text{sid}, \text{pid}, \text{pre-create-vc}, \tau_{\text{vc}}, tx_{\text{vc}}, T) \)
- \( m_{1} := (\text{sid}, \text{pid}, \text{PRE-CREATE}, \tau_{\text{vc}}, tx_{\text{vc}}, 0, T - \tau) \)
- \( m_{2} := (\text{sid}, \text{pid}, \text{PRE-CREATE}, \tau_{\text{vc}}, \text{id}) \)
- \( m_{3} := (\text{sid}, \text{pid}, \text{open-req}, tx_{\text{vc}}, \ell \text{-list}, \text{onion}_{i+1}, \theta_{i}, tx_{i}^{\prime}) \)
- \( m_{4} := (\text{sid}, \text{pid}, \text{OPEN}, tx_{\text{vc}}, \theta_{i+1}, R_{i}, U_{i}, U_{i+2}, \alpha_{i}, T) \)
- \( m_{5} := (\text{sid}, \text{pid}, \text{OPEN-ACCEPT}, \tau_{\text{post-refund}}) \)
- \( m_{6} := (\text{sid}, \text{pid}, \text{open-ok}, \sigma_{\text{U}_{i+1}(tx_{i}^{\prime})}) \)
- \( m_{7} := (\text{ssid}, \text{UPDATE}, \tau_{\text{id}}, \theta_{i}) \)
- \( m_{8} := (\text{ssid}, \text{UPDATE}, \tau_{\text{id}}, \theta_{i}) \)
- \( m_{9} := (\text{sid}, \text{pid}, \text{OPENED}) \) or, if sent by the receiver, \( m_{9} := (\text{sid}, \text{pid}, \text{VC-OPENED}, tx_{\text{vc}}, T, \alpha_{i}) \)

**Setup.**

**Real world:** An honest \( U_{0} \) performs \text{setup} in \( \tau_{0} \) to set up the initial objects and to pre-create the VC with \( U_{n} \). In round \( \tau_{0}, U_{0} \) sends \( m_{0} \) to \( U_{n} \) (which \( E \) sees in round \( \tau_{0} + 1 \) only if \( U_{n} \) is corrupted) and then, after waiting 1 round, \( m_{1} \) to \( \mathcal{F}_{\text{Channel}} \). Note that an honest \( U_{n} \) receiving \( m_{0} \) in some round \( \tau \), sends also a message \( m_{1} \) to \( \mathcal{F}_{\text{Channel}} \). If \( \mathcal{F}_{\text{Channel}} \) received two valid messages \( m_{1} \) from \( U_{0} \) and \( U_{n} \), it returns \( m_{2} \). Depending on the corruption setting, the ensemble

- \( \text{EXEC}_{\Pi, A, \text{E}} := \{ m_{0}[\tau_{0} + 1] \} \cup \text{obsSet}(m_{1}, \mathcal{F}_{\text{Channel}}, \tau_{0} + 1) \) for \( U_{0} \) honest, \( U_{n} \) corrupted
- \( \text{EXEC}_{\Pi, A, \text{E}} := \text{obsSet}(m_{1}, \mathcal{F}_{\text{Channel}}, \tau_{0} + 1) \cup \text{obsSet}(m_{1}, \mathcal{F}_{\text{Channel}}, \tau_{0} + 1) \) for \( U_{0} \) honest, \( U_{n} \) honest, where \( m_{1} \) is sent by each user.
- \( \text{EXEC}_{\Pi, A, \text{E}} := \text{obsSet}(m_{1}, \mathcal{F}_{\text{Channel}}, \tau) \) for \( U_{0} \) corrupted, \( U_{n} \) honest

**Ideal world:** For an honest \( U_{0} \), \( \mathcal{F}_{\text{VC}} \) performs \text{setup} in \( \tau_{0} \) to set up the initial objects and to pre-create the VC. In round \( \tau_{0}, \mathcal{F}_{\text{VC}} \) asks \( S \) to send \( m_{0} \) to a dishonest \( U_{0} \) (who receives it in round \( \tau_{0} + 1 \)), or, if \( U_{n} \) is honest send \( m_{1} \) to \( \mathcal{F}_{\text{Channel}} \) in \( \tau_{0} + 1 \) on behalf of \( U_{n} \). In both cases, \( \mathcal{F}_{\text{VC}} \) sends \( m_{1} \) to \( \mathcal{F}_{\text{Channel}} \) in \( \tau_{0} + 1 \). If \( U_{0} \) is dishonest and \( U_{n} \) honest, \( S \) waits for a message \( m_{0} \) from \( U_{n} \) in some round \( \tau \) and sends \( m_{1} \) to \( \mathcal{F}_{\text{Channel}} \). If \( \mathcal{F}_{\text{Channel}} \) received two valid messages \( m_{1} \) from \( U_{0} \) and \( U_{n} \), it returns \( m_{2} \). Depending on the corruption setting, the ensemble

- \( \text{EXEC}_{\Pi, A, \text{E}} := \{ m_{0}[\tau_{0} + 1] \} \cup \text{obsSet}(m_{1}, \mathcal{F}_{\text{Channel}}, \tau_{0} + 1) \) for \( U_{0} \) honest, \( U_{n} \) corrupted
TABLE VI: 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).

<table>
<thead>
<tr>
<th>Real World</th>
<th>Ideal World</th>
<th>Output</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setup: Prot.OpenVC.Setup 1-15</td>
<td>IF.OpenVC.Setup 1-16, IF.OpenVC.Setup 7-10, IF.OpenVC.Setup 13-15</td>
<td></td>
<td>Pre-creates VC, performs setup and contacts next user</td>
</tr>
<tr>
<td>Create State: Prot.OpenVC.Open 6-8</td>
<td>IF.OpenVC.Open 12-13, Sim.OpenVC.a 1-5</td>
<td>m₆, 2, m₄, m₃</td>
<td>Upon m₆ sends message m₄ to $E$. Then, creates the objects to send in m₃ and sends it to next user or finalizes.</td>
</tr>
<tr>
<td>Check State: Prot.OpenVC.Open 1-4</td>
<td>IF.OpenVC.Check 1-4, IF.OpenVC.Check 5-7</td>
<td>m₄, m₅</td>
<td>Checks if objects in m₄ are correct, sends m₅ to $E$ and on m₃, sends m₃ to Uᵢ</td>
</tr>
<tr>
<td>Check Sig: Prot.OpenVC.Check 5</td>
<td>IF.OpenVC.Check 9-11</td>
<td>n/a</td>
<td>n/a</td>
</tr>
</tbody>
</table>

• EXEC$_{FVC,S,E} := \text{obsSet}(m₃,F_{Channel};τ₀ + 1) \cup \text{obsSet}(m₄,F_{Channel};τ₀ + 1)$ for Uᵢ honest, Uᵢ₊₁ honest
• EXEC$_{FVC,S,E} := \text{obsSet}(m₃,F_{Channel};τ)$ for Uᵢ corrupted, Uᵢ₊₁ honest

Open. 1. $Uᵢ$ honest, $Uᵢ₊₁$ corrupted.

Real world: After $Uᵢ$ performs either Setup or Create_State, it sends $m₃$ to $Uᵢ₊₁$ in the current round $τ$. The environment $E$ controls $A$ and therefore $Uᵢ₊₁$ and will see $m₃$ in round $τ + 1$. If $Uᵢ₊₁$ replies with a correct message $m₆$ in round $τ + 2$, $Uᵢ$ will perform Check_Sig and call $F_{Channel}$ with message $m₇$ in the same round. The ensemble is EXEC$_{A,E} := \{m₃[τ+1] \cup \text{obsSet}(m₇,F_{Channel},τ+2)$

Ideal world: After $F_{VC}$ performs either Setup or Simulator performs Create_State, the simulator sends $m₃$ to $Uᵢ₊₁$ in the current round $τ$. $E$ will see $m₃$ in round $τ + 1$. If $Uᵢ₊₁$ replies with a correct message $m₆$ in round $τ + 2$, the simulator will perform Check_Sig and call $F_{Channel}$ with message $m₇$ in the same round. The ensemble is EXEC$_{FVC,S,E} := \{m₃[τ+1] \cup \text{obsSet}(m₇,F_{Channel},τ+2)$

2. $Uᵢ$ honest, $Uᵢ₊₁$ honest.

Real world: After $Uᵢ$ performs either Setup or Create_State, it sends $m₃$ to $Uᵢ₊₁$ in the current round $τ$. $Uᵢ₊₁$ performs Check_State and sends $m₄$ to $E$ in round $τ + 1$. If $E$ replies with $m₅$, $Uᵢ₊₁$ replies with $m₆$. $Uᵢ$ receives this in round $τ + 2$, performs Check_Sig and sends $m₇$ to $F_{Channel}$. $Uᵢ₊₁$ expects the message $m₈$ in round $τ + 2 + tₙ$ and will then send $m₉$ to $E$. Afterwards it continues with either Create_State or Finalize. The ensemble is EXEC$_{A,E} := \{m₃[τ+1], m₆[τ+2+tₙ] \cup \text{obsSet}(m₇,F_{Channel},τ+2)$

Ideal world: After $F_{VC}$ performs either Setup or is invoked by itself (in step Open.13) or by the simulator (in step process.6) in the current round $τ$, $F_{VC}$ performs the procedure Open. This behaves exactly like Create_State, Check_State and Check_Sig. However, since every object is created by $F_{VC}$, the checks are omitted. The procedure Open outputs the messages $m₄$ in round $τ + 1$ and if $E$ replies with $m₅$, calls $F_{Channel}$ with $m₇$ in $τ + 2$. Finally, if $m₈$ is received in round $τ + 2 + tₙ$, outputs $m₉$ to $E$. The ensemble is EXEC$_{FVC,S,E} := \{m₄[τ+1], m₆[τ+2+tₙ] \cup \text{obsSet}(m₇,F_{Channel},τ+2)$

3. $Uᵢ$ corrupted, $Uᵢ₊₁$ honest.

Real world: After $Uᵢ₊₁$ receives the message $m₃$ from $Uᵢ$, it performs Check_State and sends $m₄$ to $E$ in the current round $τ$. If $E$ replies with $m₅$, $Uᵢ₊₁$ sends $m₆$ to $Uᵢ$. If $Uᵢ₊₁$ receives the message $m₆$ from $F_{Channel}$ in round $τ + 1 + tₙ$, it sends $m₉$ to $E$. The ensemble is EXEC$_{H,A,E} := \{m₄[τ], m₆[τ+1], m₉[τ+1+tₙ]$}

Ideal world: After the simulator receives $m₃$ from $Uᵢ$, it performs Check_State together with $F_{VC}$ and $F_{VC}$ sends $m₄$ to $E$. If $E$ replies with $m₅$, $F_{VC}$ asks the simulator to send $m₆$ to $Uᵢ$. All of this happens in the current round $τ$. If the simulator receives $m₈$ in round $τ + 1 + tₙ$, it sends $m₉$ to $E$. The ensemble is EXEC$_{FVC,S,E} := \{m₄[τ], m₆[τ+1], m₉[τ+1+tₙ]$}

Note that we do not care about the case were both $Uᵢ$ and $Uᵢ₊₁$ 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 $F_{VC}$.

Proof: Again, we consider the execution ensembles of the interaction between users $Uᵢ$ and $U₀$ for three different cases. We match the sequences and where they are used in the ideal and real world in Table VII. We define the following messages:

- $m₁₀ := \{\text{sid}, \text{pid}, \text{finalize}, tx^{vc}, \sigma_{Uᵢ}(tx^{vc})\}$
- $m₁₁ := \{\text{ssid}, \text{POST}, tx^{vc}\}$

1. $Uᵢ$ honest, $U₀$ corrupted. (ideal to real)

Real world: After performing Finalize in the current round $τ$, $U₀$ sends $m₁₀$ to $U₀$, which $E$ sees in $τ + 1$. The ensemble is EXEC$_{H,A,E} := \{m₁₀[τ+1]$}

Ideal world: After either $F_{VC}$ or the simulator performs Finalize in the current round $τ$, the simulator sends $m₁₀$ to $U₀$, which $E$ sees in $τ + 1$. The ensemble is EXEC$_{FVC,S,E} := \{m₁₀[τ+1]$}

2. $Uᵢ$ honest, $U₀$ honest. (ideal to real)

Real world: After performing Finalize in the current round $τ$, $U₀$ sends $m₁₀$ to $U₀$, which $E$ sees in $τ + 1$. The ensemble is EXEC$_{H,A,E} := \{m₁₀[τ+1]$}

Ideal world: After either $F_{VC}$ or the simulator performs Finalize in the current round $τ$, the simulator sends $m₁₀$ to $U₀$, which $E$ sees in $τ + 1$. The ensemble is EXEC$_{FVC,S,E} := \{m₁₀[τ+1]$}
TABLE VII: Explanation of the sequence names used in Lemma 2 and where they can be found.

<table>
<thead>
<tr>
<th>Real World</th>
<th>Ideal World</th>
<th>Output</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FINALIZE</td>
<td>Prot.OpenVC.Open 8</td>
<td>n/a</td>
<td>m_{10} Sends finalize message to U_0</td>
</tr>
<tr>
<td>CHECK_FINALIZE</td>
<td>Prot.Finalize 1-3</td>
<td>n/a</td>
<td>m_{11} Checks if tx^{i'} is the same, if not, publishes it to ledger with m_{11}.</td>
</tr>
</tbody>
</table>

Real world: After performing FINALIZE in the current round \( \tau \), \( U_i \) 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 \( G_{\text{Ledger}} \) in round \( \tau' \). The ensemble is \( \text{EXEC}_{H,A,E} := \text{obSet}(m_{11}, G_{\text{Ledger}}, \tau') \)

Ideal world: Either \( F_{VC} \) or the simulator performs FINALIZE in the current round \( \tau \). In the meantime, functionality \( F_{VC} \) performs CHECK_FINALIZE and will, if the checks in FINALIZE failed or it was performed in a incorrect round \( \tau' \), \( F_{VC} \) will instruct the simulator to send \( m_{11} \) to \( G_{\text{Ledger}} \) in rounds \( \tau' \). The ensemble is \( \text{EXEC}_{F_{VC},S,E} := \text{obSet}(m_{11}, G_{\text{Ledger}}, \tau') \)

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 \( G_{\text{Ledger}} \) in round \( \tau' \). The ensemble is \( \text{EXEC}_{H,A,E} := \text{obSet}(m_{11}, G_{\text{Ledger}}, \tau') \)

Ideal world: The simulator and \( F_{VC} \) perform CHECK_FINALIZE and should the simulator not receive a correct message \( m_{10} \) in the correct round, \( F_{VC} \) will instruct the simulator to send \( m_{11} \) to \( G_{\text{Ledger}} \) in round \( \tau' \). The ensemble is \( \text{EXEC}_{F_{VC},S,E} := \text{obSet}(m_{11}, G_{\text{Ledger}}, \tau') \)

Lemma 3. Let \( \Sigma \) be a EUF-CMA secure signature scheme. Then, the Update phase of protocol \( II \) GUC-emulates the Update phase of functionality \( F_{VC} \).

Proof: Trivially, this the update phase is the same, as the pre-update messages are simply forwarded to \( F_{\text{Channel}} \) in both the real and the ideal world.

Lemma 4. Let \( \Sigma \) be a EUF-CMA secure signature scheme. Then, the Close phase of protocol \( II \) GUC-emulates the Close phase of functionality \( F_{VC} \).

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 VIII. We define the following messages.

- \( m_{15} := \text{update}(\text{sid}_C, \text{UPDATE}, \overline{\tau}, \text{id}_r, \overline{\theta}_i) \)
- \( m_{16} := \text{update}(\text{sid}_C, \overline{\tau}, \text{id}_r, \overline{\theta}_i) \)
- \( m_{17} := (\text{sid}, \text{pid}, \text{CLOSED}) \) or, if sent by the sender, \( m_{17} := (\text{sid}, \text{pid}, \text{VC-CLOSED}) \)

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 \( \tau \). The environment \( E \) controls \( A \) and therefore \( U_i \) and will see \( m_{12} \) in round \( \tau + 1 \). The ensemble is \( \text{EXEC}_{H,A,E} := \{m_{12}[\tau + 1]\} \)

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

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 \( \tau \). \( U_i \) receives this message in \( \tau + 1 \) and carries out NEW, sending \( m_{13} \) to \( E \) in \( \tau + 1 \) and, upon \( m_{14} \) in \( \tau + 1 \), sends \( m_{15} \) in \( \tau + 1 \) to \( F_{\text{Channel}} \). After a successful update \( (m_{16} \) is received), \( U_i \) sends \( m_{17} \) to \( E \) in \( \tau + t_u \) and continues with \( U_{i-1} \), if it exists. The ensemble is \( \text{EXEC}_{H,A,E} := \{m_{13}[\tau + 1], m_{17}[\tau + t_u] \} \cup \text{obSet}(m_{15}, m_{17}[\tau + 1], F_{\text{Channel}}) \)

Ideal world: After \( F_{VC} \) 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 \( \tau \), \( F_{VC} \) performs the procedure Close. This behaves exactly like \( \text{CLOSE} \) and PROCEED_CLOSE. However, since every object is created by \( F_{VC} \), the checks are omitted. The procedure Close outputs the messages \( m_{13} \) in round \( \tau + 1 \) and if \( E \) replies with \( m_{14} \), calls \( F_{\text{Channel}} \) with \( m_{15} \) in \( \tau + 1 \). Finally, if \( m_{16} \) is received in round \( \tau + 1 \) and \( t_u \), outputs \( m_{17} \) to \( E \) and continues for \( U_{i-1} \), if it exists. The ensemble is \( \text{EXEC}_{F_{VC},S,E} := \{m_{13}[\tau + 1], m_{17}[\tau + 1] \} \cup \text{obSet}(m_{15}, m_{17}[\tau + 1], F_{\text{Channel}}) \)

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

Real world: After \( U_i \) receives the message \( m_{12} \) from \( U_{i+1} \) in round \( \tau \), it performs \( \text{CLOSE} \) and sends \( m_{13} \) to \( E \) in \( \tau \). If \( E \) replies with \( m_{14} \) in the same round, \( U_i \) sends \( m_{15} \) to \( F_{\text{Channel}} \) in \( \tau \). After receiving \( m_{16} \) in \( \tau + t_u \), performs PROCEED_CLOSE, sending \( m_{17} \) to \( E \) and continues with \( U_{i-1} \), if it exists. The ensemble is \( \text{EXEC}_{H,A,E} := \{m_{13}[\tau + 1], m_{17}[\tau + t_u] \} \cup \text{obSet}(m_{15}, m_{17}[\tau + 1], F_{\text{Channel}}) \)

Ideal world: After \( S \) receives \( m_{12} \) from \( U_{i+1} \) in round \( \tau \), performs the steps \( \text{CLOSE} \), sending \( m_{13} \) to \( E \) in \( \tau \). If \( E \) replies with \( m_{14} \) in the same round, \( S \) sends \( m_{15} \)
Lemma 5. Let $\Sigma$ be a EUF-CMA secure signature scheme. Then, the Emergency-Offload phase of protocol II GUC-emulates the Emergency-Offload phase of functionality $F_{VC}$.

Proof: Again, we consider the execution ensembles, but now only for an honest $U_0$. We use message $m_{11} := (\text{ssid}_L, \text{POST}, \overline{\text{EXEC}}\tau)$ from before.

Real world: An honest $U_0$ checks every round and each of its VCs (with a certain $p_i$), if the VC has already been closed, see Prot.EmergencyOffload 1-4. If it has not within a certain round $\tau$, $U_0$ sends $m_{11}$ to $G_{\text{Ledger}}$ in $\tau$. The ensemble is $\text{EXEC}_{\text{F}_V,C,S,E} := \{m_{11}[\tau], m_{17}[\tau + t_u]\} \cup \text{obsSet}(m_{15}, \tau, F_{\text{Channel}})$.

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

Lemma 6. Let $\Sigma$ be a EUF-CMA secure signature scheme. Then, the Respond phase of protocol II GUC-emulates the Respond phase of functionality $F_{VC}$.

Proof: Again, we consider the execution ensembles. This 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 IX. We define the following messages.

- $m_{18} := (\text{ssid}_L, \text{CLOSE}, \overline{\text{EXEC}}\tau, p_i)$
- $m_{19} := (\text{ssid}_L, \text{POST}, \overline{\text{EXEC}}\tau)$
- $m_{20} := (\text{sid}, p_i, \text{REVOKED})$
- $m_{21} := (\text{sid}, \text{POST}, \overline{\text{EXEC}}_{\text{POST}})$
- $m_{22} := (\text{sid}, p_i, \text{FORCE-PAY})$

$U_i$ honest, revoke.

Real world: In every round $\tau$, $U_i$ 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 $F_{\text{Channel}}$ in round $\tau$. If the channel that is sent in $m_{18}$ is closed, $U_i$ sends $m_{19}$ to $G_{\text{Ledger}}$ in round $\tau + t_e + \Delta$. Finally, if the transaction sent in $m_{19}$ appears on $L$ in $\tau + t_e + 2\Delta$, $U_i$ sends $m_{20}$ to $E$. The ensemble is $\text{EXEC}_{\text{F}_V,C,S,E} := \{m_{20}[\tau + t_e + 2\Delta]\} \cup \text{obsSet}(m_{18}, F_{\text{Channel}}, \tau)$. $U_i$ honest, force-pay.

Real world: In every round $\tau$, $U_i$ 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 message $m_{20}$ to $G_{\text{Ledger}}$ in round $\tau$. If the channel that is sent in $m_{21}$ is closed, $U_i$ sends $m_{19}$ to $G_{\text{Ledger}}$ in round $\tau + t_e + \Delta$. Finally, if the transaction sent in $m_{19}$ appears on $L$, $F_{VC}$ sends $m_{20}$ to $E$. The ensemble is $\text{EXEC}_{\text{F}_V,C,S,E} := \{m_{20}[\tau + t_e + 2\Delta]\} \cup \text{obsSet}(m_{18}, F_{\text{Channel}}, \tau)$.

Thereom 1. (again) Let $\Sigma$ be an EUF-CMA secure signature scheme. Then, for functionalities $G_{\text{Ledger}}$, $G_{\text{Clocks}}$, $F_{\text{GDC}}$, $F_{\text{Channel}}$ and for any ledger delay $\Delta \in \mathbb{N}$, the protocol II UC-realizes the ideal functionality $F_{VC}$.

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

6. Discussion on security and privacy goals

We state our security and privacy goals informally in Section V-A. In this section we formally define these goals.
as cryptographic games on top of the ideal functionality $F_{VC}$ described in Appendix F.3 and then show that $F_{VC}$ 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 [9].

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 $P_b, R_b ← \text{GenPk}(A_0, B_0)$. 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 $A$, the adversary wins the aforementioned game with probability at most $1/2 + \epsilon$, where $\epsilon$ 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 [13]. Sphinx [16], for instance, is a realization of this. We use it in Donner, extended with a per-hop payload (see also Section V-B).

2) Balance security: Given a path $\text{channelList} := \gamma_0, \ldots, \gamma_{n-1}$ and given a user $U$ such that $\gamma_i.\text{right} = U$ and $\gamma_i.\text{left} = U$, we say that the balance of $U$ in the path is $\text{PathBalance}(U) := \gamma_n.\text{balance}(U) + \gamma_{n+1}.\text{balance}(U)$. Intuitively then, we say that a virtual channel (VC) protocol achieves balance security if the $\text{PathBalance}(U)$ for each honest intermediary $U$ does not decrease.

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

The challenger sets $\text{sid}$ and $\text{pid}$ to two random identifiers. Then, the challenger simulates opening a VC from the OpenVC phase on input $(\text{sid}, \text{pid}, \text{SETUP}, \text{channelList}, \text{tx}^n, \alpha, T, \text{PathBalance}(U))$. 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 $\text{PathBalance}(U)$ is lower after the VC execution.

Definition 3 (Balance security). We say that a VC protocol achieves balance security if for every PPT adversary $A$, 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 $\text{PathBalance}(U)$ is lower after the VC execution.

An intermediary $U_i$ potentially has coins locked up in the state stored in $F_{\text{Channel}}$ 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 $\text{PathBalance}(U_i)$ 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 $\alpha_{i-1}$ of coins locked up with $U_i$. $U_i$ has the same amount (minus a fee) $\alpha_i$ locked up with $U_{i+1}$.

4. The adversary disrupts the VC execution after $U_i$ and
have updated their channel for closing. In this case, $U_{i-1}$ has a non-negative amount $\alpha_{i-1}$ of coins locked up with $U_i$. $U_i$ has the smaller amount $\alpha_i'$ 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 $\alpha_{i-1}'$ of coins locked up with $U_i$. $U_i$ has the same amount (minus a fee) $\alpha_i'$ 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 $tx^c$ is posted before $T-3\Delta-tc$ 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 $tx^c$ is not posted before $T-3\Delta-tc$, $U_i$ always gets the collateral from $U_{i-1}$ via the Respond phase of $\mathcal{F}_{VC}$, keeping either a neutral or positive path balance.

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 channelList, a transaction $tx^m$, a virtual channel capacity $\alpha$ and a channel lifetime $T$, such that the output of $tx^m$.output[0] holds at least $\alpha + n \cdot \epsilon$ coins, where $n$ is the length of the path defined in channelList. The adversary sends the tuple (channelList, $tx^m$, $\alpha$, $T$) to the challenger.

The challenger sets $sid$ and $pid$ to two random identifiers. Then, the challenger simulates opening a VC from the OpenVC phase on input $(sid, pid, SETUP, channelList, tx^m, \alpha, T, \gamma)$. 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 $(n-1) \cdot \text{fee}$) 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 $A$, 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 $\mathcal{S}$) the channel between $U_0$ and $U_1$ to exactly the final balance $\alpha_i' (= x_{U_0} \cdot \text{fee})$ in the CloseVC phase before the round $T-3\Delta-tc$. Or, if not, $\mathcal{F}_{VC}$ has instructed the simulator to publish $tx^c\alpha$, allowing the balance to be enforceable 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 $(n-1) \cdot \text{fee}$.

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.

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 $T-3\Delta-tc$.

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

The challenger sets $sid$ and $pid$ to two random identifiers. Then, the challenger simulates opening a VC from the OpenVC phase on input $(sid, pid, SETUP, channelList, tx^m, \alpha, T, \gamma)$. 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 $T-3\Delta-tc$.

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

\textbf{Theorem 4} (Donner achieves reliability). \textit{Donner virtual channel executions achieve reliability as defined in Definition 5.}

\textbf{Proof:} This follows directly from $F_{VC}$. 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 $T - 3\Delta - \tau_c$. 

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 channelList$_0$ and channelList$_1$ that differ on the number of intermediary nodes between the sender and the adversary. In particular, channelList$_0$ is formed by $U_1, U^*, U_2, U_3$ whereas the path 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 $tx^m$, a VC capacity $\alpha$ as well as a channel life time $T$ such that the output $tx^m, output[0]$ holds at least $\alpha + n \cdot \epsilon$ coins, where $n$ is the length of the path defined in channelList$_0$. Finally, the adversary sends two queries (channelList$_0, tx^m, \alpha, T$) and (channelList$_1, tx^m, \alpha + \epsilon, T$) to the challenger. The challenger sets $\text{sid}$ and $\text{pid}$ to two random identifiers. Moreover, the challenger picks a bit $b$ at random and simulates the OpenVC phase on input $(\text{sid}, \text{pid}, \text{SETUP}, \text{channelList}_b, tx^m, \alpha, T, \theta)$. 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.

\textbf{Definition 6} (Sender anonymity). We say that a VC protocol achieves sender anonymity if for every PPT adversary $A$, the adversary wins the aforementioned game with probability at most $1/2 + \epsilon$, where $\epsilon$ denotes a negligible value.

\textbf{Theorem 5} (Donner achieves sender anonymity). \textit{Donner virtual channel executions achieve sender anonymity as defined in Definition 6.}

\textbf{Proof:} The message $(\text{sid}, \text{pid}, \text{open}, tx^m, rList, onion_{i+1}, T, \tau_{i-1}, \tau_i, \theta_{i+1}, \theta_i)$ that $F_{VC}$ sends to the simulator in the OpenVC phase, is leaked to the adversary. By looking at $\tau_{i-1}, \tau_i$ 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 $T, \tau_{i-1}$ and $\tau_i$. Since the amount to be sent was increased fee for the path channelList$_1$, the amount $\alpha_i$ for $U_i$ is identical for both cases. This leaves $tx^m, rList, \theta_{i+1}, \theta_i$ 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 $tx^m$, rList, $\theta_{i+1}$ and $\theta_i$. By design, the adversary knows that outputs $\theta_{i+1}$ belongs to its left neighbor $U_1$ and $\theta_i$ to itself. We defined that the output, that serves as input for $tx^m$, has never been used and is unblockable to the sender and check this in checkTxIn. Looking at the outputs of $tx^m$, 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 $tx^m$ belongs observing rList. This contradicts our assumption of unblockable stealth addresses.

2. The adversary finds out by looking at onion$_{i+1}$. The adversary controlling $U^*$ can open to 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 unblockability of stealth addresses or secure anonymous communication networks. Note that the both receiver anonymity and its proof are analogous to the sender anonymity. 

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 channelList$_0$ and channelList$_1$ that differ on the nodes other than the adversary neighbors. In particular, the path channelList$_0$ is formed by $U_0, U_1, U^*, U_2, U_3$ whereas the path 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. 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 $tx^m$, a VC capacity $\alpha$ as well as a life time $T$ such that the output $tx^m, output[0]$ holds at least $\alpha + n \cdot \epsilon$ coins. Finally, the adversary sends two queries (channelList$_0, tx^m, \alpha, T$) and (channelList$_1, tx^m, \alpha, T$) to the challenger.

The challenger sets $\text{sid}$ and $\text{pid}$ to two random identifiers. Moreover, the challenger picks a bit $b$ at random and simulates the setup and open phases on input $(\text{sid}, \text{pid}, \text{SETUP}, \text{channelList}_b, tx^m, \alpha, T, \theta)$. 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.

\textbf{Definition 7} (Path privacy). We say that a VC protocol achieves path privacy if for every PPT adversary $A$, the
adversary wins the aforementioned game with probability at most $1/2 + \epsilon$, where $\epsilon$ 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 $(\text{sid}, \text{pid}, \text{open}, \text{tx}^{\text{vc}}, \text{rList}, \text{onion}_{i+1}, \alpha_i, T, \gamma_1, \gamma_2, \theta_{i-1}, \theta_i)$ to the adversary. Again, the adversary can find out the correct bit $b$ by looking at (i) $\text{tx}^{\text{vc}}$ and $\text{rList}$ or (ii) at $\text{onion}_{i+1}$. 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.

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 intermediaries learns about number of transactions being exchanged and their amount. Formally, consider the following game. The adversary selects a channelList, a transaction $\text{tx}^{\text{in}}$, a virtual channel capacity $\alpha$ and a channel lifetime $T$ such that the output $\text{tx}^{\text{in}}.\text{output}[0]$ holds at least $\alpha + n \cdot \epsilon$ coins, where $n$ is the length of the path defined in channelList. The adversary sends the tuple $(\text{channelList}, \text{tx}^{\text{in}}, \alpha, T)$ to the challenger.

The challenger sets $\text{sid}$ and $\text{pid}$ 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 $\tau$ 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 $\tau$ 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 $U_{\alpha}'$'s final balance is $\alpha'$, 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 $T - t_\text{c} - 3\Delta$, 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 $A$, the adversary wins the aforementioned game with probability at most $1/2 + \epsilon$, where $\epsilon$ 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 $\mathcal{F}_{\text{VC}}$ and $\mathcal{F}_{\text{Channel}}$. The only information regarding the VC updates are sent by either VC endpoint to $\mathcal{F}_{\text{VC}}$ (in the Update phase) and forwarded to $\mathcal{F}_{\text{Channel}}$, 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 $\mathcal{F}_{\text{Channel}}$ 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 $\alpha'$ 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 $\epsilon$ denotes a negligible value.