Multi-Party Virtual State Channels

Stefan Dziembowski[1], Lisa Eckey[2], Sebastian Faust[2],
Julia Hesse[2], and Kristina Hostáková[2]

[1] stefan.dziembowski@crypto.edu.pl; University of Warsaw
[2] first.last@crisp-da.de; Technische Universität Darmstadt

Abstract. Smart contracts are self-executing agreements written in program code and are envisioned to be one of the main applications of blockchain technology. While they are supported by prominent cryptocurrencies such as Ethereum, their further adoption is hindered by fundamental scalability challenges. For instance, in Ethereum contract execution suffers from a latency of more than 15 seconds, and the total number of contracts that can be executed per second is very limited. *State channel networks* are one of the core primitives aiming to address these challenges. They form a second layer over the slow and expensive blockchain, thereby enabling instantaneous contract processing at negligible costs.
In this work we present the first complete description of a state channel network that exhibits the following key features. First, it supports virtual multi-party state channels, i.e. state channels that can be created and closed *without* blockchain interaction and that allow contracts with any number of parties. Second, the worst case time complexity of our protocol is *constant* for arbitrary complex channels. This is in contrast to the existing virtual state channel construction that has worst case time complexity linear in the number of involved parties. In addition to our new construction, we provide a comprehensive model for the modular design.

1 Introduction

Blockchain technology emerged recently as a promising technique for distributing trust in security protocols. It was introduced by Satoshi Nakamoto in [27] who used it to design *Bitcoin*, a new cryptographic currency which is maintained jointly by its users, and remains secure as long as the majority of computing power in the system is controlled by honest parties. In a nutshell, a blockchain is a system for maintaining a joint database (also called the "ledger") between several users in such a way that there is a *consensus* about its state.

In recent years the original ideas of Nakamoto have been extended in several directions. Particularly relevant to this paper are systems that support so-called *smart contracts* [32], also called *contracts* for short (see Sect. 2.1 for a more detailed introduction to this topic). Smart contracts are self-executing agreements written in a programming language that distribute money according to the results of their execution. The blockchain provides a platform where such contracts can be written down, and more importantly, be executed according to the rules of the language in which they are encoded. The most prominent blockchain system that offers support for rich smart contracts is *Ethereum*, but many other systems are currently emerging.

Unfortunately, the current approach of using blockchain platforms for executing smart contracts faces inherent scalability limitations. In particular, since all participants of such systems need to reach consensus about the blockchain contents, state changes are costly and time consuming. This is especially true for blockchains working in the so-called *permissionless* setting (like Bitcoin or Ethereum), where the set of users changes dynamically, and the number of participants is typically large. In Ethereum, for example, it can take minutes for a transaction to be confirmed, and the number of state changes per second (the so-called transaction throughput) is currently around 15-20 transactions per second. This is unacceptable for many applications, and in particular, prohibits use-cases such as "microtransactions" or many games that require instantaneous state changes.

Arguably one of the most promising approaches to tackle these problems are *off-chain techniques* (often also called "layer-2 solutions"), with one important example being *payment channels* [3]. We describe this concept in more detail in Sect. 2.1. For a moment, let us just say that the basic idea of a payment channel is to let two parties, say Alice and Bob, "lock" some coins in a smart contract on the

blockchain in such a way that the amount of coins that each party owns in the contract can be changed dynamically *without* interacting with the blockchain. As long as the coins are locked in the contract the parties can then update the distribution of these coins "off-chain" by exchanging signatures of the new balance that each party owns in the channel. At some point the parties can decide to close the channel, in which case the latest signed off-chain distribution of coins is realized on the blockchain. Besides creation and closing, the blockchain is used only in one other case, namely, when there is a *dispute* between the parties about the current off-chain balance of the channel. In this case the parties can send their latest signed balance to the contract, which will then resolve the dispute in a fair way.

This concept can be extended in several directions. *Channel networks* (e.g., the *Lightning network* over Bitcoin [30]) are an important extension which allows to securely "route" transactions over a longer path of channels. This is done in a secure way, which means that intermediaries on the path over which coins are routed cannot steal funds. Another extension is known under the name *state channels* [1]. In a state channel the parties can not only send payments but also execute smart contracts off-chain. This is achieved by letting the channel maintain in addition to the balance of the users a "state" variable that stores the current state of an off-chain contract. Both extensions can be combined resulting into so-called *state channel networks* [12, 9, 7], where simple state channels can be combined to create longer state channels. We write more about this in Sect. 2.1.

Before we describe our contribution in more detail let us first recall the terminology used in [12] on which our work relies. Dziembowski et al. [12] distinguish between two variants of two-party state channels – so-called *ledger* and *virtual* state channels[3]. Ledger state channels are created directly over the ledger, while virtual state channels are built over multiple existing (ledger/virtual) state channels to construct state channels that span over multiple parties. Technically, this is done in a recursive way by building a virtual state channel on top of two other state channels. For instance, given two ledger state channels between Alice and Ingrid, and Ingrid and Bob respectively, we may create a virtual state channel between Alice and Bob where Ingrid takes the role of an *intermediary*. Compared to ledger state channels, the main advantage of virtual state channels is that they can be opened and closed without interaction with the blockchain.

1.1 Our Contribution

Our main contribution is to propose a new construction for generalized state channel networks that exhibit several novel key features. In addition, we present a comprehensive modeling and a security analysis of our construction. We discuss further details below. The comparison to related work is presented in Sec. 1.2.

Multi-party state channels. Our main contribution is the *first full specification* of multi-party virtual state channels. A multi-party state channel allows parties to off-chain execute contracts that involve > 2 parties. This greatly broadens the applicability of state channel networks since many use cases such as online games or exchanges for digital assets require support for multi-party contracts. Our multi-party state channels are built "on top" of a network of ledger channels. Any subset of the parties can form multi-party state channels, where the only restriction is that the parties involved in the multi-party state channel are connected via a path in the network of ledger channels. This is an important distinctive feature of our construction because once a party is connected to the network it can "on-the-fly" form multi-party state channels with changing subsets of parties. An additional benefit of our construction is that our multi-party state channels are *virtual*, which allows opening and closing of the channel without interaction with the blockchain. As a consequence in the optimistic case (i.e., when there is no dispute between the parties) channels can be opened and closed instantaneously at nearly zero-costs.

At a more technical level, virtual multi-party state channel are built in a recursive way using 2-party state channels as a building-block. More concretely, if individual parties on the connecting path do not wish to participate in the multi-party state channel, they can be "cut out" via building virtual 2-party state channels over them.

[3] The startup L4 and their project *Counterfactual* [9] use a different terminology: virtual channels are called "meta channels", but the concepts are the same.

Virtual state channels with direct dispute. The second contribution of our work is to introduce the concept of "direct disputes" to virtual state channels. To better understand the concept of direct disputes let us recall the basic idea of the dispute process from [12]. While in ledger state channels disputes are always directly taken to the ledger, in the 2-party virtual state channels from [12] disputes are first attempted to be resolved by the intermediary Ingrid before moving to the blockchain. There are two advantages of such an "indirect" dispute process. First, it provides "layers of defense" meaning that Alice is forced to go to the blockchain *only* if both Bob and Ingrid are malicious. Second, "indirect" virtual state channels allow for cross-blockchain state channels because the contracts representing the underlying ledger state channels always have to deal with a single blockchain system only.

These features, however, come at the price of an increased worst case time complexity. Assuming a blockchain finality of Δ,[4] the virtual channel construction of [12] has worst case dispute timings of order $O(n\Delta)$ for virtual state channels that span over n parties. We emphasize that these worst case timing may already occur when only a *single* intermediary is corrupt, and hence may frequently happen in state channel networks with long paths.

In this work we build virtual state channels with *direct disputes*. Similar to ledger state channels, virtual state channels with direct dispute allow the members of the channel to resolve conflicts in time $O(\Delta)$, and thus, independent of the number of intermediaries involved. We call our new construction *virtual state channels with direct dispute* to distinguish them from their "indirect" counterpart [12]. To emphasize the importance of this improvement, notice that already for relatively short channels spanning over 13 ledger channels the worst case timings reduce from more than 1 day for the dispute process in [12] to less than 25 minutes in our construction. A comparison of the two types of two party state channels is presented in the following table.

	Ledger	**Direct Virtual**	**Indirect Virtual**
Creation	on chain	via subchannels	via subchannels
Dispute	on chain	on chain	via subchannels
Closure	on chain	via subchannels	via subchannels

Our final construction generalizes the one of [12] by allowing an arbitrary composition of: (a) 2-party virtual state channels with direct and indirect disputes, and (b) multi-party virtual state channels with direct disputes. We leave the design of multi-party virtual state channels with indirect dispute as an important open problem for future work.

Modeling state channel networks. Our final contribution is a comprehensive security model for designing and analysing complex state channel networks in a modular way. To this end, we use the Universal Composability framework of Canetti [4] (more precisely, its global variant [5]), and a recursive composition approach similar to [12]. One particular nice feature of our modeling approach is that we are able to re-use the ideal state channel functionality presented in [12]. This further underlines the future applicability of our approach to design complex blockchain-based applications in a modular way. Or put differently: our functionalities can be used as subroutines for any protocol that aims at fast and cheap smart contract executions.

1.2 Related Work

One of the first constructions of off-chain channels in the scientific literature was the work of Wattenhofer and Decker [10]. Since then, there has been a vast number of different works constructing protocols for off-chain transactions and channel networks with different properties [31, 21, 13, 22, 20, 23]. These papers differ from our work as they do not consider off-chain execution of arbitrary contract code, but instead focus on payments. Besides academic projects, there are also many industry projects that aim at building state channel networks. Particular relevant to our work is the Counterfactual project of L4 [9], Celer network [7] and Magmo [8]. The white-papers of these projects typically do not offer full specification of full state channel networks and instead follow a more "engineering-oriented" approach that provides descriptions for developers. Moreover, none of these works includes a formal modeling of state channels nor a security analysis.

[4] In Ethereum typically Δ equal to 6 minutes is assumed to be safe.

To the best of our knowledge, most related to our work is [12], which we significantly extend (as described above), and the recent work of Sprites [26] and its extensions [25, 24] on building multi-party *ledger* state channels. At a high-level in [26, 25, 24] a set of parties can open a multi-party ledger state channel by locking a certain amount of coins into a contract. Then, the execution of this contract can be taken "off-chain" by letting the parties involved in the channel sign the new states of the contract. In case a dispute occurs among the parties, the dispute is taken on-chain. The main differences to our work are twofold: first [26, 25, 24] do not support virtual channels, and hence opening and closing state channels requires interaction with the blockchain. Second, while we support full concurrent execution of multiple contracts in a single channel, [26, 25, 24] focuses on the off-chain execution of a single contract. Moreover, our focus is different: while an important goal of our work is formal modeling, [26] aims at improving the worst case timings in payment channel networks, and [25, 24] focus on evaluating practical aspects of state channels.

2 Overview of Our Constructions

Before we proceed with the more technical part of this work, we provide some background on the ledger and virtual state channels in Sec. 2.1 (we follow the formalism of [12]). In Sec. 2.2 we give an overview of our construction for handling "direct disputes", while in Sec. 2.3, we describe how we build and maintain multi-party virtual state channels. Below we assume that the parties that interact with the contracts own some coins in their *accounts* on the ledger. We emphasize that the description in this section is very simplified and excludes many technicalities.

2.1 Background on Contracts and State Channels [12]

Contracts. As already mentioned in Sect. 1, contracts are self-executing agreements written in a programming language. More formally, a contract can be viewed as a piece of code that is deployed by one of the parties, can receive coins from the parties, store coins on its account, and send coins to them. A contract never "acts by itself" – in other words: by default it is in an idle state and activates only when it is "woken up" by one of the parties. Technically, this is done by calling a *function* from its code. Every function call can have some coins attached to it (meaning that these coins are deduced from the account of the calling party and added to the contract account).

To be a bit more formal, we use two different terms while referring to a "contract": (i) "contract *code*" $\mathtt{C}$ – a static object written in some programming language (and consisting of a number of functions); and (ii) "contract *instance*" ν, which results from deploying the contract code $\mathtt{C}$. Each contract instance ν maintains during its lifetime a (dynamically changing) *storage*, where the current state of the contract is stored. One of the functions in contract code, called a *constructor*, is used to create an instance and its initial storage. These notions are defined formally in Sect. 3. Here, let us just illustrate them by a simple example of a contract $\mathtt{C}_{\mathsf{sell}}$ for selling a pre-image of some fixed function H. More concretely, suppose that we have two parties: Alice and Bob, and Bob is able to invert H, while Alice is willing to pay 1 coin for a pre-image of H, i.e., for any x such that $H(x) = y$ (where y is chosen by her). Moreover, if Bob fails to deliver x, then he has to pay a "fine" of 2 coins. First, the parties deploy the contract by depositing their coins into it (Alice deposits 1 coins, and Bob deposits 2 coins).[5] Denote the initial storage of the contract instance as G_0. Alice can now challenge Bob by requesting him to provide a pre-image of y. Let G_1 be the storage of the contract after this request has been recorded. If now Bob sends x such that $H(x) = y$ to the contract, $1 + 2 = 3$ coins are paid to Bob, and the contract enters a terminal state of storage G_2. If Bob fails to deliver x in time, i.e. within some time $t > \Delta$, and the contract has still storage G_1, then Alice can request the contract to pay the 3 coins to her, and the contract enters into a terminal state of storage G_3.

The contract code $\mathtt{C}_{\mathsf{sell}}$ consists of functions used to deploy the contract (see footnote 5), a function that Alice uses to send y to the contract instance, a function used by Bob to send x, and a function that Alice calls to get her coins back if Bob did not send x in time.

[5] Technically, this is done by one of the parties, Alice, say, calling a constructor function, and then Bob calling another function to confirm that he agrees to deploy this contract instance. To keep our description simple, we omit these details here.

Functionality of state channels. State channels allow two parties Alice and Bob to execute instances of some contract code C off-chain, i.e., without interacting with the ledger. These channels offer four sub-protocols that manage their life cycles: (i) *channel create* for opening a new channel; (ii) *channel update* for updating the state of a channel; (iii) *channel execute* for executing contracts off-chain; and finally (iv) *channel close* for closing a channel when it is not needed anymore. In [12] the authors consider two types of state channels: *ledger* state channels and *virtual* state channels. The functionality offered by these two variants is slightly different, which we discuss next.

Ledger state channels. Ledger state channels are constructed directly on the ledger. To this end, Alice and Bob *create* the ledger state channel γ by deploying an instance of a *state channel contract* (denoted SCC) on the ledger. The contract SCC will take the role of a judge, and resolve disputes when Alice and Bob disagree (we will discuss disputes in more detail below). During channel creation, Alice and Bob also lock a certain amount of coins into the contract. These coins can then be used for off-chain contracts. For instance, Alice and Bob may each transfer 10 coins to SCC, and hence in total 20 coins are available in the channel γ. Once the channel γ is established, the parties can *update* the state of γ (without interacting with the state channel contract). These updates serve to create new contract instances "within the channel", e.g., Alice can buy from Bob a pre-image of H and pay for it using her channel funds by deploying an instance of the $\mathsf{C}_{\mathsf{sell}}$ contract in the channel. At the end the channel is closed, and the coins are transfered back to the accounts of the parties on the ledger. The state channel contract guarantees that even if one of the parties is dishonest she cannot steal the coins of the honest party (i.e.: get more coins than she would get from an honest execution of the protocol). The mechanism behind this is described a bit later (see "*Handling disputes in channels*" on page 6).

Virtual state channels. The main novelty of [12] is the design of *virtual state channels*. A virtual state channel offers the same interface as ledger state channels (i.e.: channel creation, update, execute, and close), but instead of being constructed directly over the ledger, they are built "on top of" other state channels. Consider a setting where Alice and Bob are not directly connected via a ledger state channel, but they both have a ledger channel with an intermediary Ingrid. Call these two ledger state channels α and β, respectively (see Fig. 1, page 6). Suppose now that Alice and Bob want to execute the pre-image selling procedure using the contract $\mathsf{C}_{\mathsf{sell}}$ according to the same scenario as the one described above. To this end, they can *create* a virtual state channel γ with the help of Ingrid, but without interacting with the ledger. In this process the parties "lock" their coins in channels α and β (so that they cannot be used for any other purpose until γ is closed). The amounts of "locked" coins are as follows: in α Alice locks 1 coin and Ingrid locks 2 coins, and in β Bob locks 2 coins, and Ingrid locks 1 coin. The requirement that Ingrid locks 2 coins in α and 1 coin in β corresponds to the fact that she is "representing" Bob in channel α and "representing" Alice in channel β. Here, by "representing" we mean that she is ready to cover their commitments that result from executing the contract in γ.

Once γ is created, it can be used exactly as a ledger state channel, i.e., Alice and Bob can open a contract instance ν of $\mathsf{C}_{\mathsf{sell}}$ in γ via the virtual state channel *update* protocol and *execute* it. As in the ledger state channels, when both Alice and Bob are honest, the update and execution of ν can be done without interacting with the ledger or Ingrid. Finally, when γ is not needed anymore, it is *closed*, where closing is first tried *peacefully* via the intermediary Ingrid (in other words: Alice and Bob "register" the latest state of γ at Ingrid).

For example: suppose the execution of C_{sell} ends in the way that Alice receives 0 coins, and Bob receives 3 coins. The effect on the ledger channels is as follows: in channel α Alice receives 0 coins, and Ingrid receives 3 coins, and in channel β Bob receives 3 coins, and Ingrid receives 0 coins. Note that this is financially neutral for Ingrid who always gets backs the coins that she locked (although the distribution of these coins between α and β can be different from the original one). This situation is illustrated on Fig. 1. If the peaceful closing fails, the parties enter into a dispute which we describe next.

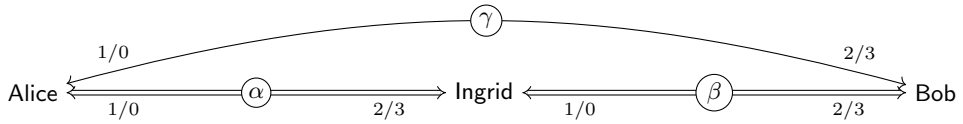

Fig. 1. Virtual channel γ built over ledger channels α and β. The labels "x/y" on the channels denote the fact that a given party locked x coins for the creation of γ, and got y coins as a result of closing γ.

Handling disputes in channels. The description above considered the case when both Alice and Bob are honest. Of course, we also need to take into account conflicts between the parties, e.g., when Alice and Bob disagree on a state update, or refuse to further execute a contract instance. Resolving such conflicts in a fair way is the purpose of the dispute resolution mechanism. The details of this mechanism appear in [12], but for completeness we included its short description in Appx. A.

In order to better understand the dispute handling, we start by providing some more technical details on the state channel off-chain execution mechanism. Let ν be a contract instance of the pre-image selling contract $\mathsf{C}_{\mathsf{sell}}$, say, and denote by G_0 its initial state. To deploy ν in the state channel both parties exchange signatures on $(G_0, 0)$, where the second parameter in the tuple will be called the *version number*. The rest of the execution is done by exchanging signatures on further states with increasing version number. For instance, suppose that in the pre-image selling contract C_{sell} (described earlier in this section) the last state on which both parties agreed on was $(G_1, 1)$ (i.e., both parties have signatures on this state tuple), and Bob wants to provide x such that $H(x) = y$. To this end, he locally evaluates the contract instance to obtain the new state $(G_2, 2)$, and sends it together with his signature to Alice. Alice verifies the correctness of both the computation and the signature, and if both checks pass, she replies with her signature on $(G_2, 2)$.

Let us now move to the dispute resolution for ledger channels and consider a setting where a malicious Alice does not reply with her signature on $(G_2, 2)$ (for example because she wants to avoid "acknowledging" that she received x). In this case, Bob can force the execution of the contract instance ν *on*-chain by *registering* in the state channel contract SCC the latest state on which both parties agreed on. To this end, Bob will send the tuple $(G_2, 2)$ together with the signature of Alice to SCC. Of course, SCC cannot accept this state immediately because it may be the case that Bob cheated by registering an outdated state.[6] Hence, the ledger contract SCC gives Alice time Δ to reply with a more recent signed state (recall that in Sec. 1.1 we defined Δ to be a constant that is sufficiently large so that every party can be sure her transaction is processed by the ledger within this time). When Δ time has passed, SCC finalizes the *state registration* process by storing the version with the highest version number in its storage. Once registration is completed, the parties can continue the execution of the contract instance on-chain.[7]

The dispute process for *virtual* state channels is much more complex than the one for the ledger channels. In particular, in a virtual state channel Alice and Bob first try to resolve their conflicts peacefully via the intermediary Ingrid. That is, both Alice and Bob first send their latest version to Ingrid who takes the role of the judge, and attempts to resolve the conflict. If this does not succeed because a dishonest Ingrid is not cooperating, then the parties resolve their dispute *on*-chain using the underlying ledger state channels α and β (and the *virtual state channel contracts* VSCC).

Longer virtual state channels via recursion. So far, we only considered virtual state channels that can be built on top of 2 ledger state channels. The authors of [12] show how virtual state channels can be used in a recursive way to build virtual state channels that span over n ledger state channels. The key feature that makes this possible is that the protocol presented in [12] is oblivious of whether the channels α or β underlying γ are ledger or virtual state channels. Hence, given a virtual state channel α between P_0 and $P_{n/2}$ and a virtual state channel β between parties $P_{n/2}$ and P_n, we can construct γ, where $P_{n/2}$ takes the role of Ingrid.

As discussed in the introduction, one main shortcoming of the recursive approach used by [12] is that even if only one intermediary is malicious[8], the worst-case time needed for dispute resolution is

[6] Notice that SCC is oblivious to what happened inside the ledger state channel γ after it was created.

[7] In the example that we considered, Bob can now force Alice bear the consequences that he revealed x to the contract instance.

[8] While it is sufficient that only one intermediary is malicious, it has to be the intermediary that was involved in the last step of the recursion, i.e., in the example from above: party $P_{n/2}$.

significantly prolonged. Concretely, even a single intermediary that works together with a malicious Alice can delay the execution of a contract instance in γ for up to $\Omega(n\Delta)$ time before it eventually is resolved on the ledger.

2.2 Virtual State Channel with Direct Dispute

The first contribution of this work is to significantly reduce the worst case timings of virtual state channels. To this end, we introduce *virtual state channels with direct dispute*, where in case of disagreement between Alice and Bob the parties do not contact the intermediaries over which the virtual state channel is constructed, but instead directly move to the blockchain. This reduces worst case timings for dispute resolution to $O(\Delta)$, and hence makes it independent of the number of parties over which the virtual channel is spanned. Let us continue with a high-level description of our construction, where we call the virtual state channels constructed in [12] *virtual state channels with indirect dispute* or *indirect virtual state channels* to distinguish them from our new construction.

Overview of virtual state channels with direct dispute. The functionality offered by virtual state channels with direct dispute can be described as a "hybrid" between ledger and indirect virtual state channels. On the one hand – similar to virtual state channels from [12] – *creation* and *closing* involves interaction with the intermediary over which the channel is built. On the other hand – similar to ledger state channels – the *update* and *execution*, in case of dispute between the end parties, is directly moved to the ledger. The latter is the main difference to indirect virtual state channels, where dispute resolution first happens peacefully via an intermediary. The advantage of our new approach is that the result of a dispute is visible to all parties and contracts that are using the same ledger. Hence, the other contracts can use the information about the result of this dispute in order to speed up the execution of their own dispute resolution procedure. This process is similar to the approach used in the Sprites paper [26], but we extend it to the case of virtual (multi-party) channels.

Before we describe in more detail the dispute process, we start by giving a high-level description of the *creation* process. To this end, consider an initial setting with two indirect virtual state channels α and β. Both α and β have length $n/2$, where α is spanned between parties P_0 and $P_{n/2}$, while β is spanned between parties $P_{n/2}$ and P_n (assume that n is a power of 2). Using the channels α and β, parties P_0 and P_n can now create a direct virtual state channel γ of length n. At a technical level this is done in a very similar way to creating an indirect virtual state channel. In a nutshell, with the help of the intermediary $P_{n/2}$ the parties update their subchannels α and β by opening instances of a special so-called *direct virtual state channel contract* `dVSCC`. The role of `dVSCC` is similar to the role of the indirect virtual state channel contract presented in [12]. It (i) guarantees balance neutrality for the intermediary (here for $P_{n/2}$), i.e., an honest $P_{n/2}$ will never loose money; and (ii) it ensures that what was agreed on in γ between the end users P_0 and P_n can be realized in the underlying subchannels α and β during closing or dispute.

Once γ is successfully created P_0 and P_n can *update* and *execute* contract instances in γ using a 2-party protocol, which is similar to the protocol used for ledger state channels (i.e., using the version number approach outlined above) as long as P_0 and P_n follow the protocol. The main difference occurs in the dispute process, which we describe next.

Direct dispute via the dispute board. Again, suppose that P_0 and P_n want to execute the pre-image selling procedure. Similarly to the example on page 6 suppose that during the execution of the contract P_0 (taking the role of Alice) refuses to acknowledge that P_n (taking the role of Bob) revealed the pre-image. Unlike in indirect virtual state channels, where P_n would first try to resolve his conflict peacefully via $P_{n/2}$, in our construction P_n registers his latest state directly on the so-called *dispute board* – denoted by $\mathcal{D}$. Since the dispute board $\mathcal{D}$ is a contract running directly on the ledger whose state can be accessed by anyone, we can reduce timings for dispute resolution from $O(n\Delta)$ to $O(\Delta)$. At a technical level, the state registration process on the dispute board is similar to the registration process for ledger channels described above. That is, when P_n registers his latest state regarding channel γ on $\mathcal{D}$, P_0 gets notified and is given time Δ to send her own version to $\mathcal{D}$. While due to the global nature of $\mathcal{D}$ all parties can see the final result of the dispute, only the end parties of γ can dispute the state of γ on $\mathcal{D}$. Our construction for direct virtual state channels uses this novel dispute mechanism also as subroutine during the update.

This enables us to reduce the worst case timings of these protocols from $O(n\Delta)$ in indirect virtual state channels to $O(\Delta)$.

The above description omits many technical challenges that we have to address in order to make the protocol design work. In particular, the closing procedure of direct virtual state channels is more complex because sometimes it needs to refer to contents on the public dispute board. Concretely, during closing of channel γ, the end parties P_0 and P_n first try to close γ peacefully via the intermediary. To this end, P_0 and P_n first attempt to update the channels α and β, respectively, in such a way that the updated channels will reflect the last state of γ. If both update requests come with the same version of γ then $P_{n/2}$ confirms the update request, and the closing of γ is completed peacefully. Otherwise $P_{n/2}$ gives the end parties some time to resolve their conflict on the dispute board $\mathcal{D}$, and takes the final result of the state registration from $\mathcal{D}$ to complete the closing of γ. Of course, also this description does not present all the details. For instance, how to handle the case when both P_0 and P_n are malicious and try to steal money from $P_{n/2}$, or a malicious $P_{n/2}$ that does not reply to a closing attempt. Our protocol addresses these issues.

Interleaving direct and indirect virtual state channels. A special feature of our new construction is that users of the system can mix direct and indirect virtual state channels in an arbitrary way. For example, they may construct an indirect virtual γ over two subchannels α and β which are direct (or where α is direct and β is indirect). This allows them to combine the benefits of both direct and indirect virtual channels. If, for instance, γ is indirect and both α and β are direct, then in case of a dispute, P_0 and P_n will first try to resolve it via the intermediary $P_{n/2}$, and only if this fails they use the dispute board. The advantage of this approach is that, as long as $P_{n/2}$ is honest, disputes between P_0 and P_n can be resolved almost instantaneously off-chain (thereby saving fees and time). On the other hand, even if $P_{n/2}$ is malicious, then disputes can be resolved fast, since the next lower level of subchannels α and β are direct, and hence a dispute with a malicious $P_{n/2}$ will be taken directly to the ledger. We believe that the optimal composition of direct and indirect virtual channels highly depends on the use-case and leave a detailed discussion on this topic for future research.

2.3 Multi-Party Virtual State Channels

The main novelty of this work is a construction of multi-party virtual state channels. As already mentioned in Sec. 1, multi-party virtual state channels are a natural generalization of 2-party channels presented in the previous sections and have two distinctive features. First, they are *multi-party*, which means that they can execute contracts involving multiple parties. Consider for instance a multi-party extension of $\mathsf{C}_{\mathsf{sell}}$ – denoted by $\mathsf{C}_{\mathsf{msell}}$ – where parties $P_1, \dots P_{t-1}$ each pay 1 coin to P_t for a pre-image of a function H, but if P_t fails to deliver a pre-image, P_t has to pay a "fine" of 2 coins to each of $P_1, \dots, P_{t-1}$ (and the contract stops). Our construction allows the parties to create an off-chain channel for executing this contract, pretty much in the same way as the standard (bilateral) channels are used for executing $\mathsf{C}_{\mathsf{sell}}$. The second main feature of our construction is that our multi-party channels are *virtual*. This means that they are built over 2-party ledger channels, and thus their creation process does not require interaction with the ledger. Our construction has an additional benefit of being highly flexible. Given ledger channels between parties P_i and P_{i+1} for $i \in \{0, \dots, n-1\}$, we can build multi-party state channels involving any subset of parties. Technically, this is achieved by cutting out individual parties P_j that do not want to participate in the multi-party state channel by building 2-party virtual state channels "over them". Moreover, we show how to generalize this for an arbitrary graph (V, E) of ledger channels, where the vertices V are the parties, and the edges E represent the ledger channels connecting the parties.

An example: a 4-party virtual state channel. To get a better understanding of our construction, we take a look at a concrete example, which is depicted in Fig. 2. We assume that five parties $P_1, \dots, P_5$, are connected by ledger state channels ($P_1 \Leftrightarrow P_2 \Leftrightarrow P_3 \Leftrightarrow P_4 \Leftrightarrow P_5$). Suppose P_1, P_3, P_4 and P_5 want to create a 4-party virtual state channel γ. Party P_2 will not be part of the channel γ but is needed to connect P_1 and P_3. In order to "cut out" P_2, parties P_1 and P_3 first construct a virtual channel denoted by $P_1 \leftrightarrow P_3$.

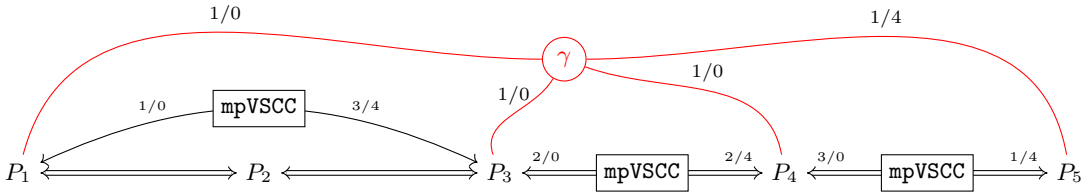

Fig. 2. Example of a multi-party virtual state channel γ between parties P_1, P_3, P_4 and P_5. In each subchannel a contract instance of `mpVSCC` is opened. Initially every party invests one coin and when the channel is closed, party P_5 owns all coins. The figure depicts the initial/final balance of parties in each of these contract instances.

Now the channel γ can be created on top of the subchannels $P_1 \leftrightarrow P_3$, $P_3 \Leftrightarrow P_4$ and $P_4 \Leftrightarrow P_5$.[9] Assume for simplicity that each party invests one coin into γ. Now in each subchannel, they open an instance of the special "multi-party virtual state channel contract" denoted as `mpVSCC`, which can be viewed as a "copy" of γ in the underlying subchannels. Note, that some parties have to lock more coins into the subchannel `mpVSCC` contract instances than others. For example in the channel $P_4 \Leftrightarrow P_5$, party P_4 has to lock three coins while P_5 only locks one coin. This is necessary, since P_4 additionally takes over the role of the parties P_1 and P_3 in this subchannel copy of γ. In other words, we require that in each `mpVSCC` contract instance, each party has to lock enough coins to match the sum of the investments of all "represented" parties.

After γ was successfully created, the parties P_1, P_3, P_4 and P_5 can open and execute multiple contracts ν in γ without talking to P_2. Let us assume that at the end of the channel lifetime party P_5 is the rightful owner of all four coins. Then after γ is successfully closed, the coins locked in the contract instances `mpVSCC` in the subchannels are unlocked in a way that reflects the final balance of γ. This means, for example, that all coins locked in subchannel $P_4 \Leftrightarrow P_5$ go to P_5. Since party P_4 now lost 3 coins in this subchannel, she needs to be compensated in the subchannel $P_3 \Leftrightarrow P_4$. Hence, the closing protocol guarantees that all four coins locked in $P_3 \Leftrightarrow P_4$ go to P_4. Since P_4 initially locked $2+3=5$ coins in the subchannels and received $4+0=4$ coins at the closing of γ, she lost 1 coin which corresponds to the final distribution in γ. As shown in Fig. 2 this process is repeated for the other subchannel $P_1 \leftrightarrow P_3$ as well.

Key ideas of the multi-party state channel update and execution. As for 2-party channels, our multi-party construction consists of 4 sub-protocol and a state registration process that is used by the parties in case of dispute. For registration our construction uses the direct dispute process outlined in Sec. 2.2, where all involved parties can register their latest state on the dispute board. One of the main differences between the 2-party and multi-party case is the way in which they handle state channel updates. Recall that in the two party case the initiating party sends an update request to the other party of the state channel, who can then confirm or reject the update request. Hence, in the two-party case it is easy for two honest parties to reach agreement whether an update was successfully completed or not.[10] In the multi-party case the protocol is significantly more complex. When the initiating party, say P_1, requests an update, she sends her update request to all other parties P_3, P_4 and P_5. The challenge is now that a malicious P_1 may for instance send a different update request to P_3 and P_4. At this point honest P_3 and P_4 have a different view on the update request. To resolve this inconsistency we may use standard techniques from the literature on authenticated broadcast protocols [11]. The problem with such an approach, however, is that it is well known [14] that broadcast has communication complexity of $O(n)$ in case most parties are dishonest. Our protocol circumvents this impossibility by a simple approach, where agreement can be reached in $O(1)$ rounds by relying on the ledger as soon as an honest party detects inconsistencies.

Let us now consider the contract execution protocol. The first attempt for constructing a protocol for multi-party state channel execution might be to use a combination of our new update protocol from

[9] To keep things simple we do not allow the recursion to build virtual channels on top on n-party channels for $n > 2$. We leave describing this extension as a possible future research direction.

[10] In case one party behaves maliciously, an agreement is reached via the state registration process.

above together with the contract execution protocol for the 2-party setting. In this case the initiating party P would locally execute the contract instance, and request an update of the multi-party state channel γ according to the new state of the contract instance. Unfortunately, this naive solution does not take into account a concurrent execution from two or more parties. For example, it may happen that P_1 and P_4 simultaneously start different contract instance executions, thereby leading to a protocol deadlock. For 2-party state channels this was resolved by giving each party a different slot when it is allowed to start a contract instance execution. In the multi-party case this approach would significantly decrease the efficiency of our protocol and in particular make its round complexity dependent on the number of involved parties. Our protocol addresses this problem by introducing a carefully designed execution scheduling, which leads to a constant time protocol.

Combining different state channel types. Finally, we emphasize that due to our modular modeling approach, all different state channel constructions that we consider in this paper can smoothly work together in a *fully concurrent* manner. That is, given a network of ledger state channels, parties may at the same time be involved in 2-party virtual state channels with direct or indirect dispute, while also being active in various multi-party state channels. Moreover, our construction guarantees strong fairness and efficiency properties in a fully concurrent setting where all parties except for one are malicious and collude. Further details on our formal model are given in Sec. 5 and in the appendix.

3 Definitions and Notation

We assume that the set $\mathcal{P} = \{P_1, \ldots, P_m\}$ of parties that use the system is fixed. In addition, we fix a bijection $\mathsf{Order}_{\mathcal{P}} \colon \mathcal{P} \to [m]$ which on input a party $P_i \in \mathcal{P}$ returns its "order" i in the set $\mathcal{P}$. Following [12, 13] we present tuples of values using the following convention. The individual values in a tuple T are identified using keywords called *attributes*, where formally an *attribute tuple* is a function from its set of attributes to $\{0,1\}^*$. The *value of an attribute* identified by the keyword attr in a tuple T (i.e. $T(\mathsf{attr})$) will be referred to as $T.\mathsf{attr}$. This convention will allow us to easily handle tuples that have dynamically changing sets of attributes. We assume the existence of a signature scheme $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ that is existentially unforgeable against a chosen message attack (see, e.g., [18]). The ECDSA scheme used in Ethereum is believed to satisfy this definition.

3.1 On the Usage of the UC-Framework

To formally model the security of our construction, we use a UC-style model that consider protocols that operate with *coins*.[11] In particular, our model uses a synchronous version of the global UC framework (GUC) [6] which extends the standard UC framework [4] by allowing for a global setup. Since our security model is essentially the same as in [12], parts of this section are taken verbatim from there.

Protocols and adversarial model. We consider an *n-party protocol* π that runs between parties from the set $\mathcal{P} = \{P_1, \ldots, P_n\}$. A protocol is executed in the presence of an *adversary* $\mathcal{A}$ that takes as input a security parameter 1^λ (with $\lambda \in \mathbb{N}$) and an auxiliary input $z \in \{0,1\}^*$, and who can *corrupt* any party P_i at the beginning of the protocol execution (so-called static corruption). By corruption we mean that $\mathcal{A}$ takes full control over P_i including learning its internal state. Parties and the adversary $\mathcal{A}$ receive their inputs from a special entity – called the *environment* $\mathcal{Z}$ – which represents anything "external" to the current protocol execution. The environment also observes all outputs returned by the parties of the protocol. In addition to the above entities, the parties can have access to ideal functionalities $\mathcal{F}_1, \ldots, \mathcal{F}_m$. In this case we say that the protocol π *works in the* $(\mathcal{F}_1, \ldots, \mathcal{F}_m)$*-hybrid model* and write $\pi^{\mathcal{F}_1, \ldots, \mathcal{F}_m}$.

Modeling time and communication. We assume a synchronous communication network, which means that the execution of the protocol happens in rounds. Let us emphasize that the notion of rounds is just an abstraction which simplifies our model and allows us to argue about the time complexity of our protocols in a natural way. We follow [19] and formalize the notion of rounds via an ideal functionality $\widehat{\mathcal{G}}_{clock}$ representing "the clock". On a high level, the ideal functionality requires all honest parties to indicate

[11] Throughout this work, the word *coin* refers to a monetary unit.

that they are prepared to proceed to the next round before the clock is "ticked". For completeness, we include the formal description of the functionality in Appx. B. In contrast to [19], we treat the clock functionality as a *global* ideal functionality using the *global UC (GUC)* model [6]. This means that all entities (possibly including hybrid ideal functionalities) are always aware of the given round. For a formalization of the synchronous communication model and its relation to the model with real time, see, e.g, [16, 17, 28, 19].

We assume that parties of a protocol are connected via authenticated communication channels with guaranteed delivery of exactly one round. This means that if a party S sends a message m to party R in round t, party R receives this message in beginning of round $t+1$. In addition, R is sure that the message was sent by party S. The adversary can see the content of the message and can reorder messages that were sent in the same round. However, it can not modify, delay or drop messages sent between parties, or insert new messages. We formalize our assumptions on the communication channels as an ideal functionality $\mathcal{F}_{GDC}$ which is described in the figure below.[12]

Functionality $\mathcal{F}_{GDC}$

Functionality $\mathcal{F}_{GDC}$ is parameterized by a set of parties $\mathcal{P}$. It has access to the global ideal functionality $\widehat{\mathcal{G}}_{clock}$. It accepts queries of following types:

Sending messages

Upon receiving a message $(\text{send}, sid, ssid, \{(R_i, m_i)\}_{i\in[k]})$, where $R_i \in \mathcal{P}$ and $m_i \in \{0,1\}^*$, from a party $S \in \mathcal{P}$, forward this message to the adversary and for every $i \in [k]$, store (t, S, R_i, m_i) in your memory. Here t denotes the current round.

Receiving messages

Upon receiving a message $(\text{fetch}, sid, ssid)$ from a party $R \in \mathcal{P}$, consider the following two cases:

- If no message was sent to R in the previous round, then reply with a message $(\text{noMessage}, sid, ssid)$.
- Otherwise, let $\{(S_i, m_i)\}_{i\in[k]}$ be the sequence of messages sent to R in the previous round, i.e., the sequence of all pairs (S_i, m_i) such that $(t-1, S_i, R, m_i)$ is in the memory, where t is the current round. Reply to R with the message $(\text{deliver}, sid, ssid, \{(S_{\rho(i)}, m_{\rho(i)})\}_{i\in[k]})$, where $\rho\colon [k] \to [k]$ is a permutation chosen by the adversary (if the adversary does not make any choice, ρ is set to be the identity function).

Fig. 3. The guaranteed delivery authenticated communication channel functionality $\mathcal{F}_{GDC}$.

While the communication between two parties of a protocol takes exactly one round, all other communication – for example, between the adversary $\mathcal{A}$ and the environment $\mathcal{Z}$ or between a party and a hybrid ideal functionality – takes zero rounds. For simplicity, we assume that any computation made by any entity takes zero rounds as well.

Handling coins. We model the money mechanics offered by crypotcurrencies such as Bitcoin or Ethereum via a *global* ideal functionality $\widehat{\mathcal{L}}$ using the *global UC (GUC)* model [6]. The state of the ideal functionality $\widehat{\mathcal{L}}$ is public and can be accessed by all parties of the protocol π, the adversary $\mathcal{A}$ and the environment $\mathcal{Z}$. It keeps track on how much money the parties have in their accounts by maintaining a vector of non-negative (finite precision) real numbers $(x_1, \ldots, x_n)$, where each x_i is the amount of coins that P_i owns.[13]

The ledger functionality $\widehat{\mathcal{L}}$ is initiated by the environment $\mathcal{Z}$ that can also freely add and remove money in user's accounts, via the operations "add" and "remove". While parties $P_1, \ldots, P_n$ *cannot* directly perform operations on $\widehat{\mathcal{L}}$, the ideal functionalities can carry out "add" and "remove" operations on the $\widehat{\mathcal{L}}$ (and hence, indirectly, P_i's can also modify $\widehat{\mathcal{L}}$, in a way that is controlled by the functionalities). Every time an ideal functionality issues an "add" or "remove" command, this command is sent to $\widehat{\mathcal{L}}$ within Δ rounds, for some parameter $\Delta \in \mathbb{N}$. The exact round when the command is sent is determined by the adversary $\mathcal{A}$. The parameter Δ models the fact that in cryptocurrencies updates on the ledger are not

[12] The functionality $\mathcal{F}_{GDC}$ can be seen as a modification of the bounded-delay channel functionality of [19]. The main difference is that the delay is fix to 1, i.e., both lower and upper bounded by the constant 1.

[13] This is similar to the concept of a *safe* of [2].

immediate. We denote a ledger functionality $\widehat{\mathcal{L}}$ with maximal delay Δ by $\widehat{\mathcal{L}}(\Delta)$ and an ideal functionality $\mathcal{G}$ with access to $\widehat{\mathcal{L}}(\Delta)$ by $\mathcal{G}^{\widehat{\mathcal{L}}(\Delta)}$. The ledger functionality $\widehat{\mathcal{L}}$ is formally defined in Fig. 4.

Functionality $\widehat{\mathcal{L}}$

Functionality $\widehat{\mathcal{L}}$ is parameterized by a set of parties $\mathcal{P} = \{P_1, \ldots, P_n\}$, a set of ideal functionalities $\{\mathcal{F}_1, \ldots, \mathcal{F}_m\}$ and a vector $(x_1, \ldots, x_n) \in \mathbb{R}^n_{\geq 0}$ (where $\mathbb{R}_{\geq 0}$ are finite-precision non-negative reals) representing the balances of parties. The functionality accepts queries of following types:

Adding money

Upon receiving a message $(\text{add}, sid, ssid, P_i, y)$ from $\mathcal{F}_j$ for some $j \in [m]$, where $y \in \mathbb{R}_{\geq 0}$ and $i \in [n]$, set $x_i := x_i + y$. We say that *y coins are added to P_i's account in $\widehat{\mathcal{L}}$.*

Removing money

Upon receiving a message $(\text{remove}, sid, ssid, P_i, y)$ from $\mathcal{F}_j$ for some $j \in [m]$, where $y \in \mathbb{R}_{\geq 0}$ and $i \in [n]$:

- If $x_i < y$, then reply with a message $(\text{noFunds}, sid, ssid)$.
- Otherwise let $x_i := x_i - y$. We say that *y coins were removed from the account of P_i in $\widehat{\mathcal{L}}$.*

Getting balance

Upon receiving a message $(\text{getBalance}, sid, ssid, P_i)$ from the adversary or $\mathcal{F}_j$ for some $j \in [m]$ or P_j for some $j \in [n]$, where $i \in [n]$, reply with $(\text{balance}, sid, ssid, P_i, x_i)$.

Fig. 4. The ledger functionality $\widehat{\mathcal{L}}$.

The GUC-security definition. Let π be a protocol working in the $\mathcal{H}$-hybrid model with access to the global ledger $\widehat{\mathcal{L}}(\Delta)$ and the global clock $\widehat{\mathcal{G}}_{clock}$. The output of an environment $\mathcal{Z}$ interacting with a protocol π and an adversary $\mathcal{A}$ on input 1^λ and auxiliary input z is denoted as $\text{EXEC}^{\widehat{\mathcal{L}}(\Delta),\widehat{\mathcal{G}}_{clock},\mathcal{H}}_{\pi,\mathcal{A},\mathcal{Z}}(\lambda, z)$. Let $\phi_{\mathcal{F}}$ be the ideal protocol for an ideal functionality $\mathcal{F}$ with access to the global ledger $\widehat{\mathcal{L}}(\Delta)$ and the global clock $\widehat{\mathcal{G}}_{clock}$. This means that $\phi_{\mathcal{F}}$ is a trivial protocol in which the parties simply forward their inputs to the ideal functionality $\mathcal{F}$. We call parties of the ideal protocol *dummy parties*. The output of an environment $\mathcal{Z}$ interacting with a protocol $\phi_{\mathcal{F}}$ and a adversary Sim (sometimes also call *simulator*) on input 1^λ and auxiliary input z is denoted as $\text{EXEC}^{\widehat{\mathcal{L}}(\Delta),\widehat{\mathcal{G}}_{clock}}_{\phi_{\mathcal{F}},\mathsf{Sim},\mathcal{Z}}(\lambda, z)$.

We are now ready to state our main security definition which, informally, says that if a protocol π UC-realizes an ideal functionality $\mathcal{F}$, then any attack that can be carried out against the real-world protocol π can also be carried out against the ideal protocol $\phi_{\mathcal{F}}$.

Definition 1. *We say that a protocol π working in a $\mathcal{H}$-hybrid model* UC-realizes an ideal functionality $\mathcal{F}$ with respect to a global ledger $\widehat{\mathcal{L}}(\Delta)$ *and a global clock $\widehat{\mathcal{G}}_{clock}$ if for every adversary $\mathcal{A}$ there exists a adversary* Sim *such that we have*

$$\left\{\text{EXEC}^{\widehat{\mathcal{L}}(\Delta),\widehat{\mathcal{G}}_{clock},\mathcal{H}}_{\pi,\mathcal{A},\mathcal{Z}}(\lambda, z)\right\}_{\substack{\lambda\in\mathbb{N},\\ z\in\{0,1\}^*}} \overset{c}{\approx} \left\{\text{EXEC}^{\widehat{\mathcal{L}}(\Delta),\widehat{\mathcal{G}}_{clock}}_{\phi_{\mathcal{F}},\mathsf{Sim},\mathcal{Z}}(\lambda, z)\right\}_{\substack{\lambda\in\mathbb{N},\\ z\in\{0,1\}^*}}$$

(where "$\overset{c}{\approx}$" denotes computational indistinguishability of distribution ensembles, see, e.g., [15]).

Simplifying assumptions To simplify exposition, we omit the session identifiers *sid* and the sub-session identifiers *ssid*. Instead, we will use expressions like "message m is a reply to message m'". We believe that this approach improves readability. Another simplifying assumption we make is that before the protocol starts, the following public-key infrastructure is set up by some trusted party: (1) For every $i = 1, \ldots, n$ let $(pk_{P_i}, sk_{P_i}) \leftarrow \mathsf{Gen}(1^\lambda)$, (2) For every $i = 1, \ldots, n$ send the message $(sk_{P_i}, (pk_{P_1}, \ldots, pk_{P_n}))$ to P_i. We emphasize that the use of a PKI is only an abstraction, and can easily be realized using the blockchain.

Moreover, in the protocol descriptions, we do not explicitly mention that communication between parties takes place via the hybrid ideal functionality $\mathcal{F}_{GDC}$. For brevity, we say "S sends a m message to R in round t" instead of "S sends a message $(\text{send}, sid, (R, m))$ to $\mathcal{F}_{GDC}$ in round t". Similarly, when we

say "R receives a message m from S in round t" we formally mean that R sends a message (fetch, sid) to $\mathcal{F}_{GDC}$ in round t and $\mathcal{F}_{GDC}$ replies with the message (deliver, $sid, \mathcal{M}$), where $\mathcal{M}$ contains the pair (S, m).

We further adopt the notation of [4] and write $\rho^{\phi \to \pi}$ to denote a protocol that is equal to ρ except that calls to the sub-protocol ϕ (e.g., an ideal functionality) are replaced by calls to the sub-protocol π, and outputs from π are treated as if they came from ϕ.

Finally, since in our work all entities have access to the global clock, we drop the superscript $\widehat{\mathcal{G}}_{clock}$, and when it is clear from the context that a functionality has access to the ledger (e.g., it accesses $\widehat{\mathcal{L}}(\Delta)$ in its code), we will also drop the $\widehat{\mathcal{L}}(\Delta)$ superscript.

3.2 Definitions of Multi-Party Contracts and Channels

We now present our syntax for describing multi-party contracts and state channels (it has already been introduced informally in Sect. 2.1). We closely follow the notation from [12, 13].

Contracts. Let n be the number of parties involved in the contract. A *contract storage* is an attribute tuple σ that contains at least the following attributes: (1) $\sigma.\mathsf{users}\colon [n] \to \mathcal{P}$ that denotes the users involved in the contract (sometimes we slightly abuse the notation and understand $\sigma.\mathsf{users}$ as the set $\{\sigma.\mathsf{users}(1), \ldots, \sigma.\mathsf{users}(n)\}$), (2) $\sigma.\mathsf{locked} \in \mathbb{R}_{\geq 0}$ that denotes the total amount of coins that is locked in the contract, and (3) $\sigma.\mathsf{cash}\colon \sigma.\mathsf{users} \to \mathbb{R}$ that denotes the amount of coins assigned to each user. It must hold that $\sigma.\mathsf{locked} \geq \sum_{P \in \sigma.\mathsf{users}} \sigma.\mathsf{cash}(P)$. Let us explain the above inequality on the following concrete example. Assume that three parties are playing a game where each party initially invests 5 coins. During the game, parties make a bet, where each party puts 1 coin into the "pot". The amount of coins *locked* in the game did not change, it is still equal to 15 coins. However, the amount of coins assigned to each party decreased (each party has only 4 coins now) since it is not clear yet who wins the bet.

We say that a contract storage σ is *terminated* if $\sigma.\mathsf{locked} = 0$. Let us emphasize that a terminated σ does not imply that $\sigma.\mathsf{cash}$ maps to zero for every user. In fact, the concept of a terminated contract storage with non-zero cash values is important for our work since it represents "payments" performed between the users. Consider, for example, a terminated three party contract storage σ with $\sigma.\mathsf{cash}(P_1) = 1$, $\sigma.\mathsf{cash}(P_2) = 1$ and $\sigma.\mathsf{cash}(P_3) = -2$. This means that both P_1 and P_2 paid one coin to P_3.

A *contract code* consists of constructors and functions. They take as input: a contract storage σ, a party $P \in \sigma.\mathsf{users}$, round number $\tau \in \mathbb{N}$ and input parameter $z \in \{0,1\}^*$, and output: a new contract storage $\tilde{\sigma}$, information about the amount of unlocked coins $\mathsf{add}\colon \sigma.\mathsf{users} \to \mathbb{R}_{\geq 0}$ and some additional output message $m \in \{0,1\}^*$. Importantly, no contract function can ever change the set of users or create new coins. More precisely, it must hold that $\sigma.\mathsf{users} = \tilde{\sigma}.\mathsf{users}$ and $\sigma.\mathsf{locked} - \tilde{\sigma}.\mathsf{locked} \geq \sum_{P \in \sigma.\mathsf{users}} \mathsf{add}(P)$.

As described already in Sec. 2.1, a *contract instance* represents an instantiation of a contract code. Formally, a contract instance is an attribute tuple ν consisting of the contract storage $\nu.\mathsf{storage}$ and the contract code $\nu.\mathsf{code}$. To allow parties in the protocol to update contract instances off-chain, we also define a *signed contract instance version* of a contract instance which in addition to $\nu.\mathsf{storage}$ and $\nu.\mathsf{code}$ contains two additional attributes $\nu.\mathsf{version}$ and $\nu.\mathsf{sign}$. The purpose of $\nu.\mathsf{version} \in \mathbb{N}$ is to indicate the version of the contract instance. The attribute $\nu.\mathsf{sign}$ is a function that on input $P \in \nu.\mathsf{storage.users}$ outputs the signature of P on the tuple $(\nu.\mathsf{storage}, \nu.\mathsf{code}, \nu.\mathsf{version})$.

Two-party ledger and virtual state channels. Formally, a two-party state channel is an attribute tuple $\gamma = (\gamma.\mathsf{id}, \gamma.\mathsf{Alice}, \gamma.\mathsf{Bob}, \gamma.\mathsf{cash}, \gamma.\mathsf{cspace}, \gamma.\mathsf{length}, \gamma.\mathsf{Ingrid}, \gamma.\mathsf{subchan}, \gamma.\mathsf{validity}, \gamma.\mathsf{dispute})$. The attribute $\gamma.\mathsf{id} \in \{0,1\}^*$ is the identifier of the two-party state channel. The attributes $\gamma.\mathsf{Alice} \in \mathcal{P}$ and $\gamma.\mathsf{Bob} \in \mathcal{P}$ identify the two end-parties using γ. For convenience, we also define the set $\gamma.\mathsf{end\text{–}users} := \{\gamma.\mathsf{Alice}, \gamma.\mathsf{Bob}\}$ and the function $\gamma.\mathsf{other\text{–}party}$ as $\gamma.\mathsf{other\text{–}party}(\gamma.\mathsf{Alice}) := \gamma.\mathsf{Bob}$ and $\gamma.\mathsf{other\text{–}party}(\gamma.\mathsf{Bob}) := \gamma.\mathsf{Alice}$. The attribute $\gamma.\mathsf{cash}$ is a function mapping the set $\gamma.\mathsf{end\text{–}users}$ to $\mathbb{R}_{\geq 0}$ such that $\gamma.\mathsf{cash}(T)$ is the amount of coins the party $T \in \gamma.\mathsf{end\text{–}users}$ has locked in γ. The attribute $\gamma.\mathsf{cspace}$ is a partial function that is used to describe the set of all contract instances that are currently open in this channel. It takes as input a *contract instance identifier* $cid \in \{0,1\}^*$ and outputs a contract instance ν such that $\nu.\mathsf{storage.users} = \gamma.\mathsf{end\text{–}users}$. We refer to $\gamma.\mathsf{cspace}(cid)$ as the *contract instance with identifier cid in* γ. The attribute $\gamma.\mathsf{length} \in \mathbb{N}$ denotes the length of the two-party state channel.

If $\gamma.\mathsf{length} = 1$, then we call γ a two-party *ledger state channel*. The attributes $\gamma.\mathsf{Ingrid}$ and $\gamma.\mathsf{subchan}$ do not have any meaning in this case and it must hold that $\gamma.\mathsf{validity} = \infty$ and $\gamma.\mathsf{dispute} = \mathrm{direct}$.

Intuitively, this means that a ledger state channel has no intermediary and no subchannel, there is no a priory fixed round in which the channel must be closed, and potential disputes between the users are resolved directly on the blockchain.

If $\gamma.\mathsf{length} > 1$, then we call γ a two-party *virtual state channel* and the remaining attributes have the following meaning. The attribute $\gamma.\mathsf{Ingrid} \in \mathcal{P}$ denotes the identity of the intermediary of the virtual channel γ. For convenience, we also define the set $\gamma.\mathsf{users} := \{\gamma.\mathsf{Alice}, \gamma.\mathsf{Bob}, \gamma.\mathsf{Ingrid}\}$. The attribute $\gamma.\mathsf{subchan}$ is a function mapping the set $\gamma.\mathsf{end\text{–}users}$ to channel identifiers $\{0,1\}^*$. The value of $\gamma.\mathsf{subchan}(\gamma.\mathsf{Alice})$ refers to the identifier of the two-party state channel between $\gamma.\mathsf{Alice}$ and $\gamma.\mathsf{Ingrid}$. Analogously, for the value of $\gamma.\mathsf{subchan}(\gamma.\mathsf{Bob})$. The attribute $\gamma.\mathsf{validity} \in \mathbb{N}$ denotes the round in which the virtual state channel γ will be closed. Intuitively, the a priory fixed closure round upper bounds the time until when party $\gamma.\mathsf{Ingrid}$ has to play the role of an intermediary of γ.[14] At the same time, the $\gamma.\mathsf{validity}$ lower bounds the time for which the end-users can freely use the channel. Finally, the attribute $\gamma.\mathsf{dispute} \in \{\text{direct}, \text{indirect}\}$ distinguishes between virtual state channel with *direct dispute*, whose end-users contact the blockchain immediately in case they disagree with each other, and virtual state channel with *indirect dispute*, whose end-users first try to resolve disagreement via the subchannels of γ.[15]

Multi-party virtual state channel. Formally, an n-party virtual state channel γ is a tuple $\gamma := (\gamma.\mathsf{id}, \gamma.\mathsf{users}, \gamma.\mathsf{E}, \gamma.\mathsf{subchan}, \gamma.\mathsf{cash}, \gamma.\mathsf{cspace}, \gamma.\mathsf{length}, \gamma.\mathsf{validity}, \gamma.\mathsf{dispute})$. The pair of attributes $(\gamma.\mathsf{users}, \gamma.\mathsf{E})$ defines an acyclic connected undirected graph, where the set of vertices $\gamma.\mathsf{users} \subseteq \mathcal{P}$ contains the identities of the n parties of γ, and the set of edges $\gamma.\mathsf{E}$ denotes which of the users from $\gamma.\mathsf{users}$ are connected with a two-party state channel. Since $(\gamma.\mathsf{users}, \gamma.\mathsf{E})$ is an undirected graph, elements of $\gamma.\mathsf{E}$ are unordered pairs $\{P, Q\} \in \gamma.\mathsf{E}$. The attribute $\gamma.\mathsf{subchan}$ is a function mapping the set $\gamma.\mathsf{E}$ to channel identifiers $\{0,1\}^*$ such that $\gamma.\mathsf{subchan}(\{P, Q\})$ is the identifier of the two-party state channel between P and Q. For convenience, we define the function $\gamma.\mathsf{other\text{–}party}$ which on input $P \in \gamma.\mathsf{users}$ outputs the set $\gamma.\mathsf{users} \setminus \{P\}$, i.e., all users of γ except for P. In addition, we define a function $\gamma.\mathsf{neighbors}$ which on input $P \in \gamma.\mathsf{users}$ outputs the set consisting of all $Q \in \gamma.\mathsf{users}$ for which $\{P, Q\} \in \gamma.\mathsf{E}$. Finally, we define a function $\gamma.\mathsf{split}$ which, intuitively works as follows. On input the ordered pair (P, Q), where $\{P, Q\} \in \gamma.\mathsf{E}$, it divides the set of users $\gamma.\mathsf{users}$ into two subsets V_P, V_Q. The set V_P contains P and all nodes that are "closer" to P than to Q and the set V_Q contains Q and all nodes that are "closer" to Q than to P. We define the function formally in Appx. B. The attribute $\gamma.\mathsf{cash}$ is a function mapping $\gamma.\mathsf{users}$ to $\mathbb{R}_{\geq 0}$ such that $\gamma.\mathsf{cash}(P)$ is the amount of coins the party $P \in \gamma.\mathsf{users}$ possesses in the channel γ. The attributes $\gamma.\mathsf{length}$, $\gamma.\mathsf{cspace}$ and $\gamma.\mathsf{validity}$ are defined as for two-party virtual state channels. The value $\gamma.\mathsf{dispute}$ for multi-party channels will always be equal to direct, since we do not allow indirect multi-party channels. We leave adding this feature to future work. In the following we will for brevity only write multi-party channels instead of virtual multi-party state channels with direct dispute. Additionally, we note that since multi-party channels cannot have intermediaries, the sets $\gamma.\mathsf{users}$ and $\gamma.\mathsf{end\text{–}users}$ are equal.

We demonstrate the introduced definitions on two concrete examples depicted in Fig. 5. In the 6-party channel on the left, the neighbors of party P_4 are $\gamma.\mathsf{neighbors}(P_4) = \{P_3, P_5, P_6\}$ and $\gamma.\mathsf{split}(\{P_3, P_4\}) = (\{P_1, P_2, P_3\}, \{P_4, P_5, P_6\})$. In the 4-party channel on the right, the neighbors of P_4 are $\gamma.\mathsf{neighbors}(P_4) = \{P_1, P_5, P_6\}$ and $\gamma.\mathsf{split}(\{P_1, P_4\}) = (\{P_1\}, \{P_4, P_5, P_6\})$.

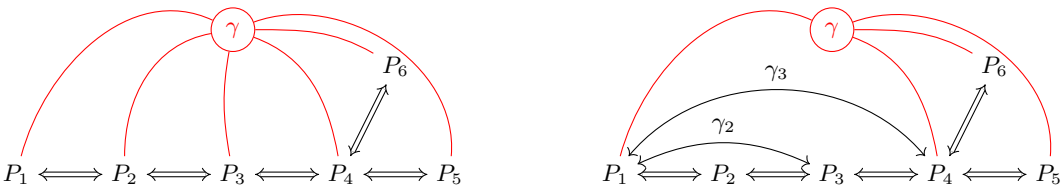

Fig. 5. Examples of multi-party channel setups: A 6-party channel on top of 5 ledger channels (left) and a 4-party channel on 2 ledger and a virtual channel γ_3 (right).

[14] In practice, this information would be used to derive fees charged by the intermediary for its service.

[15] Recall from Sec. 2 that disagreements in channels with indirect dispute might require interaction with the blockchain as well. However this happen only in the worst case when all parties are corrupt.

3.3 Security and Efficiency Goals

In the previous section, we formally defined what state channels are. Let us now give several security and efficiency goals that we aim for when designing state channels. The list below can be seen as an extension of the one from [12].

Security goals. We define security goals that guarantee that an adversary cannot steal coins from honest parties, even if he corrupts all parties except for one.

(S1) **Consensus on creation:** A state channel γ can be successfully created only if all users of γ agree with its creation.
(S2) **Consensus on updates:** A contract instance in a state channel γ can be successfully updated (this includes also creation of the contract instances) only if all end-users of γ agree with the update.
(S3) **Guarantee of execution:** An honest end-user of a state channel γ can execute a contract function f of an opened contract instance in any round $\tau_0 < \gamma.\mathsf{validity}$ on an input value z even if all other users of γ are corrupt.
(S4) **Balance security:** If the channel γ has an intermediary, then this intermediary never loses coins even if all end-users of γ are corrupt and collude.

Let us stress that while creation of a state channel has to be confirmed by *all users* of the channel, this includes the intermediary in case of a two-party virtual state channel, the update of a contract instance needs confirmation only from the *end-users* of the state channel. In other words, the intermediary of a two-party virtual state channel has the right to refuse being an intermediary but once he agrees, he can not influence how this channel is being used by the end-users. Let us also emphasize that the last property, (S4), talks only about two-party virtual state channels since, by definition, ledger and multi-party channels do not have any intermediary.

Efficiency goals. We identify four efficiency requirements. Table 1 defines which property is required from what type of channel.

(E1) **Creation in $O(1)$ rounds:** Successful creation of a state channel γ takes a constant number of rounds.
(E2) **Optimistic update/execute in $O(1)$ rounds:** In the optimistic case when all end-users of a state channel γ are honest, they can update/execute a contract instance in γ within a constant number of rounds.
(E3) **Pessimistic update/execute in $O(\Delta)$ rounds:** In the pessimistic case when some end-users of a state channel γ are dishonest, the time complexity of update/execution of a contract instance in γ depends only on the ledger delay Δ but is independent of the channel length.
(E4) **Optimistic closure in $O(1)$ rounds:** In the optimistic case when all users of $\gamma.\mathsf{users}$ are honest, the channel γ is closed in round $\gamma.\mathsf{validity} + O(1)$.

	Ledger	Virtual		
		Direct	Indirect	MP
(E1) Creation in $O(1)$		✓	✓	✓
(E2) Opt. update/execute in $O(1)$	✓	✓	✓	✓
(E3) Pess. update/execute in $O(\Delta)$	✓	✓		✓
(E4) Opt. closing in $O(1)$		✓	✓	✓

Table 1. Summary of the efficiency goals for state channels. Above, "Ledger" stands for ledger state channels, "Direct/Indirect" stand for a two party virtual state channels with direct/indirect dispute and "MP" stands for multi-party channels.

It is important to note that in the optimistic case when all users of any *virtual state channel* (i.e. multi-party, two-party with direct/indirect dispute) are honest, the time complexity of channel creation, update, execution and closure must be independent of the blockchain delay; hence in this case there cannot be any interaction with the blockchain during the lifetime of the channel.

4 State Channels ideal functionalities

Recall that the main goal of this paper is to broaden the class of virtual state channels that can be constructed. Firstly, we want virtual state channels to support direct dispute meaning that end-users of the channel can resolve disputes directly on the blockchain, and secondly, we want to design virtual multi-party state channels that can be built on top of any network of two-party state channels. In order to formalize these goals, we define an ideal functionality $\mathcal{F}_{mpch}^{\widehat{\mathcal{L}}(\Delta)}(i, \mathcal{C})$ which describes what it means to create, maintain and close multi-party as well as two-party state channels of length up to i in which contract instances from the set $\mathcal{C}$ can be opened. The functionality has access to a global ledger functionality $\widehat{\mathcal{L}}(\Delta)$ keeping track of account balances of parties in the system.

The first step towards defining $\mathcal{F}_{mpch}^{\widehat{\mathcal{L}}(\Delta)}(i, \mathcal{C})$ has already been done in [12], where the authors describe an ideal functionality, $\mathcal{F}_{ch}^{\widehat{\mathcal{L}}(\Delta)}(i, \mathcal{C})$, for ledger state channels and two-party virtual state channels with indirect dispute. For completeness, we recall this ideal functionality in Appx. C.1. The second step is to extend the ideal functionality $\mathcal{F}_{ch}^{\widehat{\mathcal{L}}(\Delta)}(i, \mathcal{C})$ such that it additionally describes how virtual state channels with direct dispute are created, maintained and closed. We denote this extended functionality $\mathcal{F}_{dch}^{\widehat{\mathcal{L}}(\Delta)}(i, \mathcal{C})$ and describe it in more detail in Sec. 4.1. As a final step, we define how multi-party channels are created, maintained and closed. This is discussed in Sec. 4.2.

Before we proceed with the description of the novel ideal functionalities, let us establish the following simplified notation. In the rest of this paper, we write $\mathcal{F}$ instead of $\mathcal{F}^{\widehat{\mathcal{L}}(\Delta)}$, for $\mathcal{F} \in \{\mathcal{F}_{ch}, \mathcal{F}_{dch}, \mathcal{F}_{mpch}\}$.

4.1 Virtual State Channels with Direct Dispute

In this section we introduce our ideal functionality $\mathcal{F}_{dch}(i, \mathcal{C})$ that allows to build any type of two party state channel (ledger state channel, virtual state channel with direct dispute and virtual state channel with indirect dispute) of length up to i in which contract instances with code from the set $\mathcal{C}$ can be opened. The ideal functionality $\mathcal{F}_{dch}(i, \mathcal{C})$ extends the ideal functionality $\mathcal{F}_{ch}(i, \mathcal{C})$ in the following way:

- Messages about ledger state channels and virtual state channels with indirect dispute are handled as in $\mathcal{F}_{ch}(i, \mathcal{C})$.
- Virtual state channels **with direct dispute** are created (resp. closed) using the procedure of $\mathcal{F}_{ch}(i, \mathcal{C})$ for creating (resp. closing) virtual channels with indirect dispute.
- Update (resp. execute) requests of contract instances in channels **with direct dispute** are handled as $\mathcal{F}_{ch}(i, \mathcal{C})$ handles such queries for ledger state channels.

Hence, intuitively, a virtual state channel γ with direct dispute is a "hybrid" between a ledger state channel and a virtual state channel with indirect dispute, meaning that it is created and closed as a virtual state channel with indirect dispute and its contract instances are updated and executed as if γ would be a ledger state channel. In the remainder of this section, we explain how $\mathcal{F}_{dch}(i, \mathcal{C})$ works in more detail and argue that it satisfies all the security and efficiency goals listed in Sec. 3.3. The formal description of the ideal functionality $\mathcal{F}_{dch}(i, \mathcal{C})$ can be found in Appx. C.1, where we also recall the functionality $\mathcal{F}_{ch}(i, \mathcal{C})$ from [12] for completeness.

If $\mathcal{F}_{dch}(i, \mathcal{C})$ receives a message about a ledger state channel or a virtual state channel with indirect dispute, then $\mathcal{F}_{dch}(i, \mathcal{C})$ behaves exactly as $\mathcal{F}_{ch}(i, \mathcal{C})$. Since $\mathcal{F}_{ch}(i, \mathcal{C})$ satisfies all the security goals and the efficiency goals (E1) – (E2) (see [12]), $\mathcal{F}_{dch}(i, \mathcal{C})$ satisfies them as well in this case. It is thus left to analyze the properties in the novel case, i.e., for virtual state channels with direct dispute.

Create and close a virtual state channel with direct dispute. The users of the virtual state channel γ, which are the end-users of the channel $\gamma.\mathsf{Alice}$ and $\gamma.\mathsf{Bob}$ and the intermediary $\gamma.\mathsf{Ingrid}$, express that they want to create γ by sending the message (create, γ) to $\mathcal{F}_{dch}(i, \mathcal{C})$. Once $\mathcal{F}_{dch}(i, \mathcal{C})$ receives such a message, it records it into the memory and locks coins in the corresponding subchannel. For example, if the sender of the message is $\gamma.\mathsf{Alice}$, $\mathcal{F}_{dch}(i, \mathcal{C})$ locks $\gamma.\mathsf{cash}(\gamma.\mathsf{Alice})$ coins of $\gamma.\mathsf{Alice}$ and $\gamma.\mathsf{cash}(\gamma.\mathsf{Bob})$ coins of $\gamma.\mathsf{Ingrid}$ in the subchannel $\gamma.\mathsf{subchan}(\gamma.\mathsf{Alice})$. If $\mathcal{F}_{dch}(i, \mathcal{C})$ records the message (create, γ) from all three parties within three rounds, then the channel γ is created. The ideal functionality informs both end-users of the channel about the successful creation by sending the message $(\text{created}, \gamma)$ to them. Since all three parties have to agree with the creation of γ, the security goal (S1) is clearly met. The successful creation takes 3 rounds, hence (E1) holds as well.

Once the virtual state channel is successfully created, $\gamma.\mathsf{Alice}$ and $\gamma.\mathsf{Bob}$ can use it (open and execute contract instance) until round $\gamma.\mathsf{validity}$ when the closing of the channel γ begins. In round $\gamma.\mathsf{validity}$, $\mathcal{F}_{dch}(i, \mathcal{C})$ first waits for τ rounds, where $\tau = 3$ if all users of γ are honest and is set by the adversary otherwise,[16] and then distributes the coins locked in the subchannels according to the final state of the channel γ. It might happen that the final state of γ contains unterminated contract instances, i.e. contract instances that still have locked coins, in which case it is unclear who owns these coins. In order to guarantee the balance security for the intermediary, the property (S4), $\mathcal{F}_{dch}(i, \mathcal{C})$ gives all of these locked coins to $\gamma.\mathsf{Ingrid}$ in *both* subchannels. The goal (E4) is met because γ is closed in round $\gamma.\mathsf{validity} + 3$ in the optimistic case.

Update a contract instance. A party P that wants to update a contract instance with identifier *cid* in a virtual state channel γ sends the message $(\text{update}, \gamma.\mathsf{id}, cid, \sigma, \mathtt{C})$ to $\mathcal{F}_{dch}(i, \mathcal{C})$. The parameter σ is the proposed new contract instance storage and the parameter $\mathtt{C}$ is the code of the contract instance. $\mathcal{F}_{dch}(i, \mathcal{C})$ informs the party $Q := \gamma.\mathsf{other\text{–}party}(P)$ about the update request and completes the update only if Q confirms it. If the party Q is honest, then it has to reply immediately. In case Q is malicious, $\mathcal{F}_{dch}(i, \mathcal{C})$ expects the reply within 3Δ rounds. Let us emphasize that the confirmation time is independent of the channel length. This models the fact that disputes are happening directly on the blockchain and not via the subchannels. In the optimistic case the update procedure takes 2 rounds and in the pessimistic case $2 + 3\Delta$ rounds; hence both update efficiency goals (E2) and (E3) are satisfied. The security property (S2) holds as well since without Q's confirmation the update fails.

Execute a contract instance. When a party P wants to execute a contract instance with identifier *cid* in a virtual state channel γ on function f and input parameters z, it sends the message $(\text{execute}, \gamma.\mathsf{id}, cid, f, z)$ to $\mathcal{F}_{dch}(i, \mathcal{C})$. The ideal functionality waits for τ rounds, where $\tau \leq 5$ in case both parties are honest and $\tau \leq 4\Delta + 5$ in case one of the parties is corrupt. The exact value of τ is determined by the adversary. Again, let us stress that the pessimistic time complexity is independent of channel length which models the fact that registration and force execution takes place directly on the blockchain. After the waiting time is over, $\mathcal{F}_{dch}(i, \mathcal{C})$ performs the function execution and informs both end-users of the channel about the result by outputting the message $(\text{execute}, \gamma.\mathsf{id}, cid, \tilde{\sigma}, \mathsf{add}, m)$. Here $\tilde{\sigma}$ is the new contract storage after the execution, add contains information about the amount of coins unlocked from the contract instance and m is some additional output message. Since the adversary can not stop the execution, and only delay it, the guarantee of execution, security property (S3), is satisfied by $\mathcal{F}_{dch}(i, \mathcal{C})$. From the description above it is clear that the two execute efficiency goals (E2) and (E3) are fulfilled as well.

Two-party state channels of length one. Before we proceed to the description of the ideal functionality $\mathcal{F}_{mpch}(i, \mathcal{C})$, let us state one simple but important observation which follows from the fact that the minimal length of a virtual state channel is 2 and the ideal functionality $\mathcal{F}_{dch}(1, \mathcal{C})$ accepts only messages about a state channel of length 1.

Observation 1 *For any set of contract codes $\mathcal{C}$ it holds that $\mathcal{F}_{dch}(1, \mathcal{C})$ is equivalent to $\mathcal{F}_{ch}(1, \mathcal{C})$.*

4.2 Virtual Multi-Party State Channels

We now introduce the functionality $\mathcal{F}_{mpch}(i, \mathcal{C})$ which allows to create, maintain and close multi-party as well as two-party state channels of length up to i in which contract instances from the set $\mathcal{C}$ can be opened. Before we give formal definition of the ideal functionality, le us provide its high level description and argue that all security and efficiency goals identified in Sec. 3.3 are met.

High level description of the ideal functionality The ideal functionality $\mathcal{F}_{mpch}(i, \mathcal{C})$ extends the functionality $\mathcal{F}_{dch}(i, \mathcal{C})$, which we described in Sec. 4.1, in the following way. In case $\mathcal{F}_{mpch}(i, \mathcal{C})$ receives a message about a two-party state channel, then it behaves exactly as the functionality $\mathcal{F}_{dch}(i, \mathcal{C})$. Since the functionality $\mathcal{F}_{dch}(i, \mathcal{C})$ satisfies all the security and efficiency goals for two-party state channels,

[16] The value of τ can be set by the adversary as long as it is smaller than some upper bound T which is of order $O(\gamma.\mathsf{length} \cdot \Delta)$.

these goals are met by $\mathcal{F}_{mpch}(i, \mathcal{C})$ as well. For the rest of this informal description, we focus on the more interesting case, when $\mathcal{F}_{mpch}(i, \mathcal{C})$ receives a message about a multi-party channel.

Parties express that they want to create the channel γ by sending the message (create, γ) to the ideal functionality $\mathcal{F}_{mpch}(i, \mathcal{C})$. Once the functionality receives such message from a party $P \in \gamma.\mathsf{users}$, it locks coins needed for the channel γ in all subchannels of γ party P is participating in. Let us elaborate on this step in more detail. For every $Q \in \gamma.\mathsf{neighbors}(P)$ the ideal functionality proceeds as follows. Let $(V_P, V_Q) := \gamma.\mathsf{split}(\{P, Q\})$ which intuitively means that V_P contains all the user of γ that are "closer" to P than to Q. Analogously for V_Q. Then $\sum_{T \in V_P} \gamma.\mathsf{cash}(T)$ coins of party P and $\sum_{T \in V_Q} \gamma.\mathsf{cash}(T)$ coins of party Q are locked in the subchannel between P and Q by the ideal functionality. If the functionality receives the message (create, γ) from all parties in $\gamma.\mathsf{users}$ within 4 rounds, then the channel γ is created. The ideal functionality informs all parties about the successful creation by outputting the message $(\text{created}, \gamma)$. Clearly, the security goal (S1) and the efficiency goal (E1) are both met.

Once the multi-party channel is successfully created, parties can use it (open and execute contract instances in it) until the round $\gamma.\mathsf{validity}$ comes. In round $\gamma.\mathsf{validity}$, the ideal functionality first waits for τ rounds, where $\tau = 3$ if all parties are honest and is set by the adversary otherwise,[17] and then unlocks the coins locked in the subchannels of γ. The coin distribution happens according to the following rules (let $\hat{\gamma}$ denote the final version of γ): If there are no unterminated contract instances in $\hat{\gamma}.\mathsf{cspace}$, then the ideal functionality simply distributes the coins back to the subchannels according to the function $\hat{\gamma}.\mathsf{cash}$. The situation is more subtle when there are unterminated contract instances in $\hat{\gamma}.\mathsf{cspace}$. Intuitively, this means that some coin of the channel are not attributed to any of the users. Our ideal functionality distributes the unattributed coins equally among the users[18] and the attributed coins according to $\hat{\gamma}.\mathsf{cash}$. Once the coins are distributed back to the subchannels, the channel γ is closed which is communicated to the parties via the message $(\text{closed}, \gamma.\mathsf{id})$. Since in the optimistic case, γ is closed in round $\gamma.\mathsf{validity} + 3$, the goal (E4) is clearly met.

The update and execute parts of the ideal functionality $\mathcal{F}_{mpch}(i, \mathcal{C})$ in case of multi-party channels are straightforward generalizations of the update and execute parts of the ideal functionality $\mathcal{F}_{dch}(i, \mathcal{C})$ in case of two-party virtual state *with direct dispute* (see Sec. 4.1).

Formal description of the ideal functionality For completeness, we give the pseudo-code description of the multi-party state channel functionality $\mathcal{F}_{mpch}(i, \mathcal{C})$ in Fig. 6. We assume that the functionality maintains a channel space Γ, were it stores all channels that were created via this functionality. Since messages that the parties send to the ideal functionality do not contain any private information, we implicitly assume that the ideal functionality forwards all messages it receives to the adversary. In addition, the adversary influences the timings; for example, the adversary decides when the parties receive messages from the functionality. Adversary's influence of this kind is implicit in the notation. By saying: "In round $\tau \leq T$ do instruction X", we mean that the adversary can decide when exactly instruction X is performed as long as it is before round T. In case the adversary does not make any choice, the instruction X is performed in round T.

Let us emphasize that since our ideal functionality is fairly complex, the description presented in Fig. 6 is simplified. In particular, it excludes some natural checks that one would expect from an ideal state channel functionality upon receiving a message from a party P. For instance, when party requests multi-party channel γ creation, the ideal functionality should check that all subchannels of γ exist. We discuss all necessary check in Appx. C.2. Furthermore, the pseudo-code of the functionality makes use of some abbreviated notation. For instance, it uses a symbolic notation for sending and receiving messages and makes use of a sub-procedure `UpdateChanSpace`. In order to keep the main body of the paper compact, we provide the detailed explanations and formal descriptions of all the abbreviated notation in Appx. B.4.

Towards realizing the ideal functionality For the rest of the paper, we focus on realization of our novel ideal functionality $\mathcal{F}_{mpch}(i, \mathcal{C})$. Our approach of realizing the ideal functionality $\mathcal{F}_{mpch}(i, \mathcal{C})$ closely follows

[17] In case at least one user is corrupt, the value of τ can be set by the adversary as long as it is smaller that some upper bound T which is of order $O(\gamma.\mathsf{length} \cdot \Delta)$.

[18] Let us emphasize that this design choice does not necessarily lead to a *fair* coin distribution. For example, when users of the multi-party channel play a game and one of the users is "about to win" all the coins when round $\gamma.\mathsf{validity}$ comes. Hence, honest parties should always agree on new contract instances only if they can enforce contract termination before time $\gamma.\mathsf{validity}$ or if they are willing to take this risk.

Functionality $\mathcal{F}_{mpch}^{\hat{\mathcal{L}}(\Delta)}(i, \mathcal{C})$

This functionality accepts messages from parties in $\mathcal{P}$. Let TimeExe(i, Δ) be the timing function for executing a contract instance in a two party state channel of length at most i, see Appx. B.5.

Multi-party virtual state channel creation

Upon $(\text{create}, \gamma) \hookleftarrow P$, where γ is a multi-party virtual state channel and $P \in \gamma.\mathsf{users}$, record the message and for every $Q \in \gamma.\mathsf{neighbors}(P)$ proceed as follows:

1. If you did not already receive (create, γ) from Q let $(V_P, V_Q) := \gamma.\mathsf{split}(P, Q)$ and remove $\sum_{T \in V_P} \gamma.\mathsf{cash}(T)$ coins from P's balance in $\gamma.\mathsf{subchan}(\{P, Q\})$ and $\sum_{T \in V_Q} \gamma.\mathsf{cash}(T)$ coins from Q's balance in $\gamma.\mathsf{subchan}(\{P, Q\})$.
2. Distinguish the following cases:
   - If within 4 rounds you record (create, γ) from all users in $\gamma.\mathsf{users}$, define $\Gamma(\gamma.\mathsf{id}) := \gamma$, send $(\text{created}, \gamma) \hookrightarrow \gamma.\mathsf{users}$ and wait for channel closing in Step 3 (while accepting update and execute messages for γ).
   - Otherwise wait until $\gamma.\mathsf{validity}$ to refund the coins that you removed from the subchannels in Step 1 within $3\Delta + \text{TimeExe}(i, \Delta) + 2$ rounds. Then stop.

Automatic closure of a multi-party virtual state channel γ in round $\gamma.\mathsf{validity}$:

Let $\hat{\gamma}$ be the current and γ be the initial version of the multi-party state channel. Let $C := \sum_{P \in \gamma.\mathsf{users}} \gamma.\mathsf{cash}(P)$ and $\hat{C} := \sum_{P \in \gamma.\mathsf{users}} \hat{\gamma}.\mathsf{cash}(P)$. Let $X := \frac{C - \hat{C}}{|\gamma.\mathsf{users}|}$.

3. If $C > \hat{C}$, then for every $P \in \gamma.\mathsf{users}$ add X to $\hat{\gamma}.\mathsf{cash}(P)$.
4. If all parties are honest, set $\tau := 3$. Otherwise set $\tau := 3\Delta + \text{TimeExe}(i, \Delta) + 2$. Wait until round $\gamma.\mathsf{validity} + \tau$.
5. For every $\{P, Q\} \in \hat{\gamma}.\mathsf{E}$:
   (a) Let $(V_P, V_Q) := \hat{\gamma}.\mathsf{split}(P, Q)$
   (b) Add $\sum_{T \in V_P} \hat{\gamma}.\mathsf{cash}(T)$ coins to P's balance in $\hat{\gamma}.\mathsf{subchan}(\{P, Q\})$ and $\sum_{T \in V_Q} \hat{\gamma}.\mathsf{cash}(T)$ coins to Q's balance in $\hat{\gamma}.\mathsf{subchan}(\{P, Q\})$.
6. Set $\Gamma(\gamma.\mathsf{id}) = \bot$ and send $(\text{closed}, \gamma.\mathsf{id}) \overset{\tau}{\hookrightarrow} \gamma.\mathsf{users}$.

Contract instance update in a multi-party virtual state channel

Upon $(\text{update}, id, cid, \tilde{\sigma}, \mathtt{C}) \overset{\tau_0}{\hookleftarrow} P$, where $\gamma := \Gamma(id)$ is a multi-party virtual state channel and $P \in \gamma.\mathsf{users}$, proceed as follows:

1. Send $(\text{update–requested}, id, cid, \tilde{\sigma}, \mathtt{C}) \overset{\tau_0 + 1}{\hookrightarrow} \gamma.\mathsf{other–party}(P)$.
2. Set $\tau := \tau_0 + 1$ in optimistic case when all parties in $\gamma.\mathsf{users}$ are honest. Else set $\tau := \tau_0 + 3\Delta + 3$.
3. If for every party $Q \in \gamma.\mathsf{other–party}(P)$ you receive $(\text{update–reply}, ok, id, cid) \overset{\tau_1 \leq \tau}{\hookleftarrow} Q$, then set $\Gamma := \mathtt{UpdateChanSpace}(\Gamma, id, cid, \tilde{\sigma}, \mathtt{C}, \mathsf{add})$, where for every $Q \in \gamma.\mathsf{users}$ the value $\mathsf{add}(Q)$ is defined as $-\tilde{\sigma}.\mathsf{cash}(Q)$ if $\gamma.\mathsf{cspace}(cid) = \bot$ and as $\sigma.\mathsf{cash}(Q) - \tilde{\sigma}.\mathsf{cash}(Q)$, where $\sigma := \gamma.\mathsf{cspace}(cid).\mathsf{storage}$, otherwise. Set $\tau_2 := \max\{\tau_0 + 3, \tau_1\}$ and send $(\text{updated}, id, cid) \overset{\tau_2}{\hookrightarrow} \gamma.\mathsf{users}$.

Contract instance execution in a multi-party virtual state channel

Upon $(\text{execute}, id, cid, f, z) \overset{\tau_0}{\hookleftarrow} P$, where $\gamma := \Gamma(id)$ is a virtual multi-party channel and $P \in \gamma.\mathsf{users}$ proceed as follows:

1. Set $\tau := \tau_0 + 6$ in the optimistic case when all parties in $\gamma.\mathsf{users}$ are honest and $\tau := \tau_0 + 4\Delta + 5$ otherwise.
2. In round $\tau_1 \leq \tau$, let $\gamma := \Gamma(id)$, $\nu := \gamma.\mathsf{cspace}(cid)$, $\sigma := \nu.\mathsf{storage}$, and $\tau' := \tau_0$ if P is honest and else τ' is set by the simulator. Compute $(\tilde{\sigma}, \mathsf{add}, m) := f(\sigma, P, \tau', z)$. If $m = \bot$, then stop. Else set $\Gamma := \mathtt{UpdateChanSpace}(\Gamma, id, cid, \tilde{\sigma}, \nu.\mathsf{code}, \mathsf{add})$ and send $(\text{executed}, id, cid, \tilde{\sigma}, \mathsf{add}, m) \overset{\tau_1}{\hookrightarrow} \gamma.\mathsf{users}$.

Two party state channel

Upon $m \overset{\tau_0}{\hookleftarrow} P$ where m is not one of the messages from above, proceed exactly as the functionality $\mathcal{F}_{dch}(i, \mathcal{C})$.

Fig. 6. The ideal functionality for multi-party virtual state channels.

the modular way we use for *defining* it. On a very high level, we first show how to construct any two party state channel, in other words, how to realize the ideal functionality $\mathcal{F}_{dch}$. This is done in Sec. 5 and Sec. 6. Thereafter, in Sec. 7, we design a protocol for multi-party channels using two party state channels in a black box way.

5 Modular Approach

In this section, we introduce our approach of realizing $\mathcal{F}_{dch}(i, \mathcal{C})$. We do not want to realize $\mathcal{F}_{dch}(i, \mathcal{C})$ from scratch, but find a modular approach which lets us reuse existing results.

Reusing standard virtual channels. We give a protocol $\Pi_{dch}(i, \mathcal{C}, \pi)$ for building two-party state channels supporting direct dispute. Our construction uses three ingredients: (1) a protocol π for virtual state channels with indirect dispute up to length i, which was shown in [12] how to build recursively from subchannels, (2) the ideal functionality $\mathcal{F}_{dch}$ for virtual channels with direct dispute up to length $i-1$ and (3) an ideal dispute board. Our protocol can roughly be described by distinguishing three cases:

Case 1: If a party receives a message about a two-party state channel of length $j < i$, then it forwards the request to $\mathcal{F}_{dch}$.

Case 2: If a party receives a message about a virtual state channel with indirect dispute and of length exactly i, then it behaves as in the protocol π.

Case 3: For the case when a party receives a message about a virtual state channel γ with direct dispute of length exactly i, we describe a new protocol using $\mathcal{F}_{dch}$ and an ideal dispute board $\mathcal{F}_{DB}$ which we will detail shortly. Central element of the new protocol will be a special contract `dVSCC` used for creating and closing γ.

In the rest of this section, we discuss each of the above mentioned building blocks and describe how the building blocks can be combined and used by our protocol $\Pi_{dch}(i, \mathcal{C}, \pi)$. We provide the description of the protocol in the next section, Sec. 6, where we also describe the special contract `dVSCC` whose instances are opened in the subchannels of γ during the creation process and guarantee that the final state of γ will be correctly reflected to the subchannels.

Ideal dispute board. Let us now informally describe our ideal functionality $\mathcal{F}_{DB}(\mathcal{C})$ for directly disputing about contracts whose code is in some set $\mathcal{C}$. On a high level, the functionality models an ideal blockchain which allows the users to achieve consensus on contract instance. For this, $\mathcal{F}_{DB}(\mathcal{C})$ maintains a public "dispute board", which is a list of contract states available to all parties. $\mathcal{F}_{DB}(\mathcal{C})$ admits two different procedures to manipulate these states: *registration* of a contract instance and *execution* of a contract instance. The registration procedure works as follows: whenever a party determines a dispute regarding a specific instance whose code is in the set $\mathcal{C}$, it can register this contract instance by sending its latest valid state to $\mathcal{F}_{DB}(\mathcal{C})$. The dispute board will now give the other party[19] of the contract instance some time to react and to send her latest state. Then, $\mathcal{F}_{DB}(\mathcal{C})$ compares both states, judges which is the latest valid one and adds it to the dispute board. To execute a function of a contract instance without the other party cooperating, the execute procedure of $\mathcal{F}_{DB}(\mathcal{C})$ can be called. We stress that the other party cannot interfere and merely gets informed about the execution. Upon receiving an execution request for a contract instance that is contained in the dispute board, $\mathcal{F}_{DB}(\mathcal{C})$ will execute the called function and update the contract instance on the dispute board according to the outcome. The formal description of the dispute board functionality $\mathcal{F}_{DB}(\mathcal{C})$ can be found in Appx. D.

Unfortunately, we cannot simply add an ideal dispute board as another hybrid functionality next to one for constructing smaller channels. In a nutshell, the reason is that the balances of virtual channels that are created via subchannels might be influenced by contracts that are in dispute. Upon closing these virtual channel, the dispute board needs to be taken into account. However, in the standard UC model it is not possible that ideal functionalities communicate their state. Thus, we will artificially allow sharing state by merging both ideal functionalities. Technically, this is done by putting a *wrapper* $\mathcal{W}_{dch}$ around both functionalities, which can be seen just as a piece of code distributing queries to the wrapped functionalities. The formal description of the wrapper can be found in Appx. E.

[19] For simplicity, we describe here how the functionality $\mathcal{F}_{DB}$ handles a dispute about a 2-party contract. $\mathcal{F}_{DB}$ handles disputes about multi-party contracts in a similar fashion.

Now that we described all ingredients, we formally state what our protocol Π_{dch} achieves and what it assumes. On a high level, our protocol gives a method to augment a 2-party state channel protocol π with indirect dispute, to also support direct dispute. Our transformation is case-tailored for channel protocols π that are build recursively out of shorter channels. That is, we do not allow an arbitrary protocol π for channels up to length i, but only one that is itself recursively build out of shorter channels. See Figure 7 for a graphical version of the Theorem 1.[20]

Theorem 1. *Let $\mathcal{C}_0$ be a set of contract codes, let $i > 1$ and $\Delta \in \mathbb{N}$. Suppose the underlying signature scheme is existentially unforgeable against chosen message attacks. Let π be a protocol that realizes the ideal functionality $\mathcal{F}_{ch}(i, \mathcal{C}_0)$ in the $\mathcal{F}_{ch}(i-1, \mathcal{C}'_0)$-hybrid world. Then protocol $\Pi_{dch}(i, \mathcal{C}_0, \pi)$ (cf. Sec. 6) working in the $\mathcal{W}_{dch}(i-1, \mathcal{C}_1, \mathcal{C}_0)$-hybrid model, for $\mathcal{C}_1 := \mathcal{C}_0 \cup \mathcal{C}'_0 \cup \mathtt{dVSCC}_i$, emulates the ideal functionality $\mathcal{F}_{dch}(i, \mathcal{C}_1)$.*

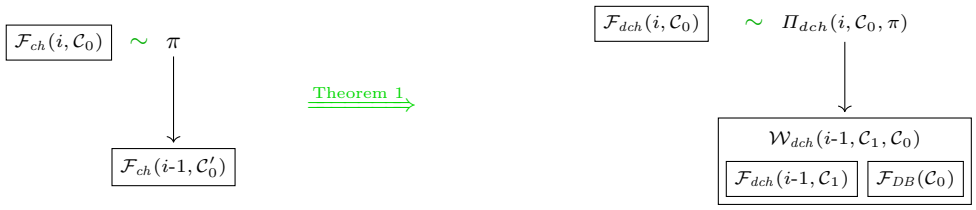

Fig. 7. We build virtual channels with direct dispute (**right**) out of standard virtual channels of the same length (**left**). The $\sim$ denotes "is UC-realized by" and $\rightarrow$ denotes usage of a hybrid functionality. The functionality $\mathcal{W}_{dch}(i\text{-}1, \mathcal{C}_1, \mathcal{C}_0)$ is a wrapper combining two functionalities, which we will discuss in detail later.

Remaining technicalities. Remember that our goal is to add direct dispute to a 2-party state channel protocol that is itself recursively build from smaller subchannels. We still need to solve two technicalities. Firstly, note that Thm. 1 yields a protocol realizing $\mathcal{F}_{dch}$ for length up to i, while it requires a *wrapped* $\mathcal{F}_{dch}$ of length up to $i-1$. Thus, to be able to apply it recursively, we introduce a technical Lemma 2 which shows how to modify $\Pi_{dch}(i, \mathcal{C}_0, \pi)$ so that it realizes the wrapped $\mathcal{F}_{dch}$.

Secondly, we can apply Thm. 1 on any level *except* for ledger channels. In a nutshell, the reason is that Thm. 1 heavily relies on using subchannels, which simply do not exist in case of ledger channels. Fortunately, this can quite easily be resolved by showing how to add our dispute board to a protocol for ledger channels. We will show how to do this with a protocol π_1 from [12] in Lemma 1. Adding the dispute board to any functionality again works by wrapping functionality $\mathcal{F}_x$ and $\mathcal{F}_{DB}$ within a wrapper $\mathcal{W}_x$. The protocol π_1 assumes an ideal functionality $\mathcal{F}_{scc}$ which models contracts on the blockchain.

The proofs of both lemmas can be found in Appx. E.

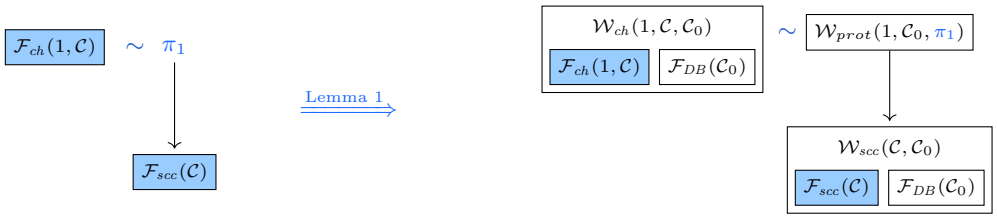

Fig. 8. Pictorial representation of Lemma 1.

Lemma 1 (The Blue Lemma). *Let $\mathcal{C}$ and $\mathcal{C}_0$ be two arbitrary sets of contract codes and let π_1 be a protocol that UC-realizes the ideal functionality $\mathcal{F}_{ch}(1, \mathcal{C})$ in the $\mathcal{F}_{scc}(\mathcal{C})$-hybrid world. Then the protocol $\mathcal{W}_{prot}(1, \mathcal{C}_0, \Pi_1)$ UC-realizes the ideal functionality $\mathcal{W}_{ch}(1, \mathcal{C}, \mathcal{C}_0)$ in the $\mathcal{W}_{scc}(\mathcal{C}, \mathcal{C}_0)$-hybrid world.*

[20] For the sake of correctness, in this section we include details about contract sets which each channel is supposed to handle. In order to understand our modular approach, their relations can be ignored. The reader can just assume that each subchannel is build in such a way that it can handle all contracts required for building all the longer channels.

Lemma 2 (The Red Lemma). *Let $i \geq 2$ and let $\mathcal{C}$ be a set of contract codes. Let Π_i be a protocol that UC-realizes the ideal functionality $\mathcal{F}_{dch}(i, \mathcal{C})$ in the $\mathcal{W}_{dch}(i-1, \mathcal{C}', \mathcal{C})$-hybrid world for some set of contract codes $\mathcal{C}'$. Then for every $\mathcal{C}_0 \subseteq \mathcal{C}$ the protocol $\mathcal{W}_{prot}(i, \mathcal{C}_0, \Pi_i)$ UC-realizes the ideal functionality $\mathcal{W}_{dch}(i, \mathcal{C}, \mathcal{C}_0)$ in the $\mathcal{W}_{dch}(i-1, \mathcal{C}', \mathcal{C})$-hybrid world.*

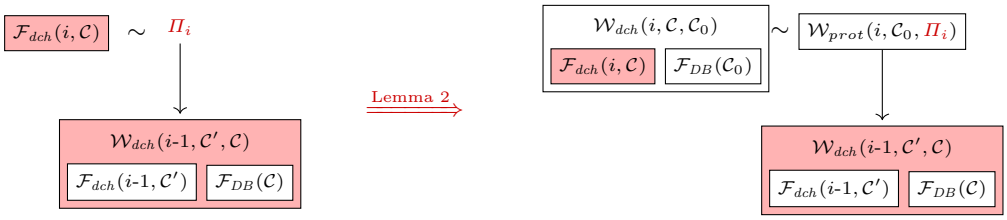

Fig. 9. Pictorial representation of Lemma 2.

We finish this section with the complete picture of our approach of building any two-party state channel of length up to 3 (Fig. 10). The picture demonstrates how we recursively realize $\mathcal{F}_{dch}$ functionalities of increasing length, as well as their wrapped versions $\mathcal{W}_{dch}$ which additionally comprise the ideal dispute board functionality. While already being required for recursively constructing $\mathcal{F}_{dch}$, $\mathcal{W}_{dch}$ will also serve us as a main building block for our protocol for multi-party channels presented in Sec. 7.

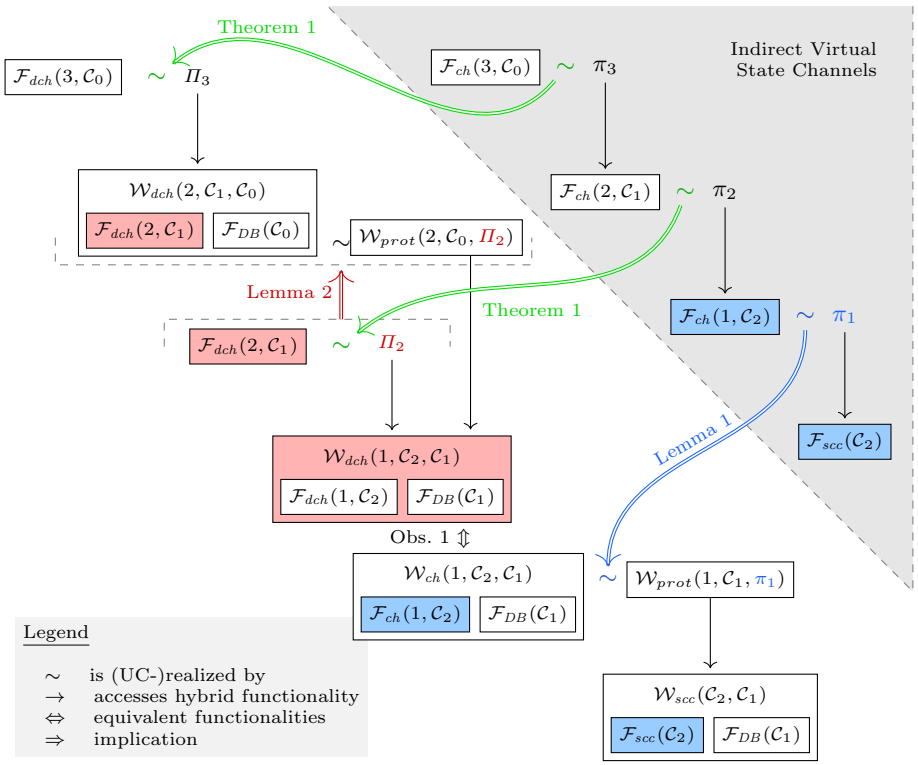

Fig. 10. The complete approach of building virtual state channels with direct dispute of length up to 3 (**top left**), from channels with indirect dispute (**gray background**). Thm. 1 and Lemma 1 allow to add direct dispute to channels. Note that the resulting recursion chain for building longer channels is disconnected due to Thm. 1 requiring $\mathcal{F}_{DB}$. Lemma 2 then reconnects the recursion chain. $\mathcal{C}_0$ is an arbitrary contract set. To build longer channels recursively, we have to allow the necessary channel contracts in each level. Thus, $\mathcal{C}_1 := \mathcal{C}_0 \cup \mathtt{C}'$, where $\mathtt{C}'$ is a special contract used for opening our target channel (i.e., longer channel supporting direct dispute, or multi-party channel). Similarly, $\mathcal{C}_2 := \mathcal{C}_1 \cup \mathcal{C}''$, where again $\mathtt{C}''$ is a special contract that is needed for the target channel. Note that it holds that $\mathcal{C}_0 \subset \mathcal{C}_1 \subset \mathcal{C}_2$, and also that the length of the channels as well as the target contract set have to be known in advance.

6 Protocol for Two-Party Channels

We now prepared to describe a concrete protocol that realizes the ideal functionality $\mathcal{F}_{dch}(i, \mathcal{C}_0)$ for $i > 1$ and any set of contract codes $\mathcal{C}_0$ in the hybrid world of the ideal functionality $\mathcal{W}_{dch}(i-1, \mathcal{C}_1, \mathcal{C}_0)$ that was recursively constructed in the previous section. Recall that the ideal functionality $\mathcal{W}_{dch}^{\hat{\mathcal{C}}(\Delta)}(i-1, \mathcal{C}_1, \mathcal{C}_0)$ combines the functionality of $\mathcal{F}_{dch}(i-1, \mathcal{C}_1)$, which is an ideal functionality that allows to build any state channel of length up to $i-1$ in which contract instance with code from $\mathcal{C}_1$ can be opened, and the functionality of the dispute board $\mathcal{F}_{DB}(\mathcal{C}_0)$ that allows parties to dispute about contract instances from the set $\mathcal{C}_0$.

Our strategy of constructing the new protocol, which we denote $\Pi_{dch}(i, \mathcal{C}_0, \pi)$, is to distinguish three cases which we already informally described in Sec. 5. We recall these three cases here and additionally explain the minimal requirements on the content of the set $\mathcal{C}_1$. On high level, this contract set contains the contract codes that must be supported by channel of length up to $i-1$:

Case 1: If a party receives a message about a two-party state channel of length $j < i$, then it forwards the request to the hybrid ideal functionality. Thus we require that $\mathcal{C}_0 \subseteq \mathcal{C}_1$.

Case 2: If a party receives a message about a virtual state channel with indirect dispute and of length exactly i, then it behaves as in the protocol π.[21]

Case 3: For the case when a party receives a message about a virtual state channel γ with direct dispute of length exactly i, we describe a new protocol using (a) shorter state channels as building blocks that provide monetary guarantees and (b) the dispute board for fair resolution of disagreements. For (a) we need the subchannels of γ to support contract instances of a special contract code which we denote $\texttt{dVSCC}_i$ and formally define later in this section. Thus we require that $\texttt{dVSCC}_i \in \mathcal{C}_1$.

The first two cases are rather straightforward and we refer the reader to Appx. G for more details. Here we focus on the most interesting case – Case 3. We begin with the explanation of the contract code $\texttt{dVSCC}_i$ and, thereafter, we describe all parts of the protocol $\Pi_{dch}(i, \mathcal{C}_0, \pi)$ for Case 3.

6.1 Contract Code $\texttt{dVSCC}_i$

In order to create a virtual state channel with direct dispute, let us denote it γ, parties of the channel need to open a special contract instance in both subchannels of γ. We call these special contract instances *Direct Virtual State Channel Contract* instances and denote their code $\texttt{dVSCC}_i$. The parameter i denotes the length of the virtual channel γ.

Let us demonstrate the role of $\texttt{dVSCC}_i$ on a concrete example for $i = 3$ that is depicted in Fig. 11. Let us consider four parties P_1, P_2, P_3 and P_4 and assume that the following state channels already exist: ledger channels $P_1 \Leftrightarrow P_2$, $P_2 \Leftrightarrow P_3$, $P_3 \Leftrightarrow P_4$ and a virtual state channel (with or without) direct dispute of length 2 between $P_1 \leftrightarrow P_3$. In order to build a virtual state channel with direct dispute γ between P_1 and P_4, an instance of $\texttt{dVSCC}_3$ has to be built in both $P_1 \leftrightarrow P_3$ and $P_3 \Leftrightarrow P_4$. The instances are generated using the *constructor* of $\texttt{dVSCC}_3$ which we denote $\texttt{Init}_3^{\mathsf{d}}$. The contract instance in $P_1 \leftrightarrow P_3$ is identified by $cid_1 := P_1||\gamma.\mathsf{id}$ and one can view it as a "copy" of the virtual state channel γ, where party P_3 plays the role of party P_4. Analogously, the contract instance in $P_3 \Leftrightarrow P_4$, with identifier $cid_4 := P_4||\gamma.\mathsf{id}$, can be viewed as a "copy" of the virtual state channel γ, where party P_3 plays the role of party P_1. So for example, if γ is a virtual state channel in which each party initially invests one coin, then (i) P_1 has to lock one coin in cid_1, P_4 has to lock one coin in cid_4 and (ii) P_3 has to lock one coin in the contract instance cid_1 and one coin in the contract instance cid_4. Once both cid_1 and cid_4 are opened, the virtual state channel γ is considered to be created and can be used by parties P_1 and P_4 until round $\gamma.\mathsf{validity}$, when it must be closed. During the closing procedure, parties execute the contract instance cid_1 and cid_4 on the contract function $\texttt{Close}_3^{\mathsf{d}}$, whose purpose is to redistribute the locked coins back to the subchannels according to the final state of the virtual state channel γ. In the example in Fig. 11, we assume that party P_4 has two coins in γ when round $\gamma.\mathsf{validity}$ comes and party P_1 has none. Hence P_1 does not get any coin back in the channel $P_1 \leftrightarrow P_3$, P_3 gets two coins back in $P_1 \leftrightarrow P_3$ and none in $P_3 \Leftrightarrow P_4$ and P_4 gets two coins back in $P_3 \Leftrightarrow P_4$.

[21] Recall that π is a protocol for virtual state channels with indirect dispute up to length i, which was shown in [12] how to build recursively from subchannels.

To summarize, the role of the $\texttt{dVSCC}_i$ contract instances is to (i) guarantee the balance neutrality for the intermediary of the channel γ and (ii) guarantee the end-users of the channel γ that whatever they agreed on in the virtual state channel during its lifetime will be reflected to the subchannels when round $\gamma.\mathsf{validity}$ comes. The interface of the contract code $\texttt{dVSCC}_i$ can be found in Fig. 12. The description of the contract functions is presented together with the protocol $\Pi_{dch}(i, \mathcal{C}_0, \pi)$ later in this section. The formal definition can be found in Appx. G.

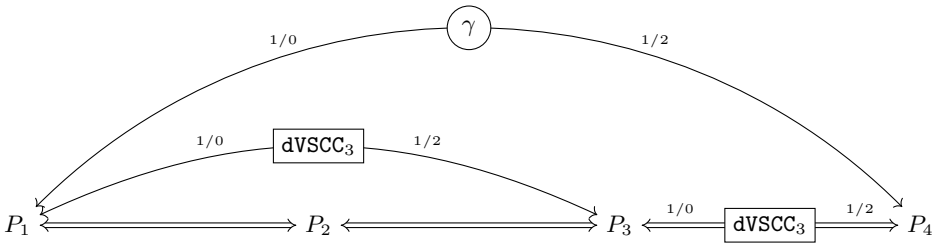

Fig. 11. Example of a virtual state channel with direct dispute γ of length 3 in which both end users P_1 and P_4 initially invest one coin. When the channel is being closed, party P_4 possesses both coins of the channel γ. In both subchannels of γ, a contract instance of the code $\texttt{dVSCC}_3$ has to be opened. The figure depicts the initial/final balances of parties in each of these contract instances.

Interface of the contract $\texttt{dVSCC}_i$

Attributes:
- Mandatory attributes: $\mathsf{users}, \mathsf{locked}, \mathsf{cash}_\mathsf{L}, \mathsf{cash}_\mathsf{R}$ (see Sec. 3.2)
- $\mathsf{virtual\text{–}channel}$: stores the initial version of the virtual state channel γ;

Functions:
- $\texttt{Init}_i^{\mathrm{d}}$: the constructor of the contract (see page 45);
- $\texttt{Close}_i^{\mathrm{d}}$: a contract function called when the virtual state channel is being closed (see page 49).

Fig. 12. Interface of the contract $\texttt{dVSCC}_i$

6.2 Protocol Description

Below we provide an intuitive explanation of each part of the protocol $\Pi_{dch}(i, \mathcal{C}_0, \pi)$ for the Case 3. The formal description can be found in Appx. G. Let us emphasize that some protocol parts are very similar to the protocol for ledger state channel or the protocol for virtual state channel with indirect dispute which were described in [12].

Create a virtual state channel with direct dispute As already explained in Sec 6.1, in order to create a virtual state channel γ with direct dispute and length i, parties have to open a $\texttt{dVSCC}_i$ contract instance in both subchannels of γ. This is done using the "update" mechanism of the subchannels. Since subchannels of γ are state channels of length $< i$, update of their contract instance can be done in a black-box way via the hybrid ideal functionality $\mathcal{W}_{dch}^{\hat{\mathcal{C}}(\Delta)}(i-1, \mathcal{C}_1, \mathcal{C}_0)$. Let us explain how this is done in more detail.

In order to create the virtual state channel γ, party $\gamma.\mathsf{Alice}$ runs (locally) the $\texttt{dVSCC}_i$ constructor $\texttt{Init}_i^{\mathrm{d}}(\gamma.\mathsf{Alice}, \tau_0, \gamma)$, where τ_0 is the current round, and obtains the initial contract instance storage σ_A. Thereafter, she requests an update by sending the message $(\text{update}, id_A, cid_A, \sigma_A, \texttt{dVSCC}_i)$ to the hybrid ideal functionality, where id_A is the identifier of the subchannel between her and $\gamma.\mathsf{Ingrid}$ and $cid_A := \gamma.\mathsf{Alice}||\gamma.\mathsf{id}$. The party $\gamma.\mathsf{Bob}$ proceeds analogously. If in round τ_0+1, party $\gamma.\mathsf{Ingrid}$ receives two symmetric update requests from the ideal functionality, meaning that $\gamma.\mathsf{Alice}$'s proposal corresponds to the result of $\texttt{Init}_i^{\mathrm{d}}(\gamma.\mathsf{Alice}, \tau_0, \gamma)$ and $\gamma.\mathsf{Bob}$'s proposal corresponds to the result of $\texttt{Init}_i^{\mathrm{d}}(\gamma.\mathsf{Alice}, \tau_0, \gamma)$, she confirms both update requests. Parties $\gamma.\mathsf{Alice}$ and $\gamma.\mathsf{Bob}$ signal to each other in round $\tau_0 + 2$ that the updates were successfully completed and output "created" to the environment which completes the channel creation. To conclude, successful creation of the channel γ takes 3 rounds.

Register a contract instance in a virtual state channel with direct dispute As long as end-users of the virtual state channel γ with direct dispute and length i behave honestly, they can update/execute contract instance in the channel γ without communicating with the hybrid ideal functionality $\mathcal{W}_{dch}^{\hat{\mathcal{C}}(\Delta)}(i-1, \mathcal{C}_1, \mathcal{C}_0)$ which models the dispute board and the subchannels of γ. However, once the end-users disagree with each other on the latest state of a contract instance in the channel γ, they need some third party to fairly resolve their disagreement. In case of virtual state channel with direct dispute, the dispute board plays the role of such a judge which intuitively means that, as soon as parties disagree about a contract instance, they go directly to the blockchain to resolve their disagreement. Let us emphasize that this is different than in the case of a virtual state channel without direct dispute, where parties tried to resolve their dispute via the subchannels.

Parties might run into dispute about a contract instance when they update/execute the contract instance or when the are closing the virtual state channel γ. In order to avoid code repetition, we define the dispute process as a separate registration procedure, `RegisterDirect`(P, id, cid). The input parameter P denotes the initiating party of the dispute process, the parameter id identifies the virtual state channel γ and cid is the identifier of the contract instance parties disagree on.

On a high level, the initiating party P first sends her version of the contract instance, let us denote it ν^P, to the dispute board.[22] The dispute board verifies the validity of ν^P and informs the other party of the virtual state channel, let us denote this party Q, that registration was requested. The party Q has the opportunity to react on the registration request by submitting her own version of the contract instance, let us denote it ν^Q. After a certain amount of rounds, which is sufficient for Q to react, the initiating party P can request finalization of the registration. The dispute board compares the two submitted contract instance versions ν^P and ν^Q and registers the one with higher version number. Parties interact with the dispute board three times during the registration process; hence, the time complexity of the registration if bounded by 3Δ, where Δ is the "blockchain delay" parameter (see Sec. 3.2).

Before we proceed to the next part of the protocol $\Pi_{dch}(i, \mathcal{C}_0, \pi)$, let us briefly discuss one technicality. Looking ahead, when a party P wants to open a new contract instance ν in the channel γ, she sends her signature on the first version of the contract instance to the other party Q. If Q does not reply to P's request by sending a valid signature on the proposed contract instance, P is in a difficult situation. Note that P does not have any valid version of the contract instance that she could submit to the dispute board and initiate the registration process. On the other had, Q can decide at any later point to register ν since it has P's signature. Because of this special case, the dispute board allows P to initiate the registration process by submitting the set $\{P, Q\}$ instead of a valid contract instance version. The dispute board then knows that it should inform the party Q about P's registration request and expect an immediate reply. If Q does not react, then the dispute board marks the contract instance cid in the channel γ as "unregistrable", which prevents Q from registering ν in some later round.

Update a contract instance in a virtual state channel with direct dispute The main idea of the update subprotocol is that parties of the virtual state channel γ exchange signatures on the new contract instance. More precisely, the party initiating the update, let us denote this party P, signs the new contract instance version and contacts the other party of the channel, let us denote it Q, to sign the new contract instance version as well. The party Q can either (i) agree with the update and send her signature on the new contract instance version back to P, or (ii) disagree with the update in which case Q signs the original version of the contract instance but with higher version number, or (iii) in case Q is malicious, it can simply abort and not reply, reply with an invalid message etc. Note that in the third case, party P does not know which contract instance version is the latest valid one. In order to force Q to either accept or reject the update, P initiates the registration procedure by executing the procedure `RegisterDirect` in which parties contact the dispute board to resolve the disagreement about the latest valid state of the contract instance. In case Q registers the new contract instance version on the dispute board, update is successfully completed. Hence, the update protocol takes in the optimistic case 2 rounds and in the pessimistic case, when registration on the dispute board is needed, it takes up to $3\Delta + 2$ rounds.

Execute a contract instance in a virtual state channel with direct dispute Execution of a contract instance in a virtual state channel γ with direct dispute and length i works on high level as follows. If both end-users of the channel γ are honest, then they first try to execute the contract instance in a peaceful way.

[22] In fact, party sends the message to the hybrid ideal functionality $\mathcal{W}_{dch}^{\hat{\mathcal{C}}(\Delta)}(i-1, \mathcal{C}_1, \mathcal{C}_0)$ which internally runs the code of the dispute board ideal functionality $\mathcal{F}_{DB}$.

This means that the initiating party P locally executes the function to obtain the new contract instance version. Then P signs it and sends details of the execution and the signature to the other party of the channel, let us denote this party Q. Party Q verifies the correctness of P's local execution and validity of P's signature and if no mistakes are found, Q sends her signature on the new contract instance version to P. If the peaceful execution fails, party P initiates the registration procedure to reach consensus with Q on the latest valid version. In case Q registers the contract instance version after the execution, then the execution process is completed. Otherwise, P has to execute the contract instance forcefully via the dispute board.

Before we proceed to the channel closing procedure, let us discuss one technicality. Since we allow parties to execute contract instances fully concurrently, we need to tackle the problematic situation when both end-users want to peacefully execute the same contract instance in the same round. To this end, we assign a "time slot" to each party, in which it is allowed to propose a peaceful execution request. More precisely, the party $\gamma.\mathsf{Alice}$ can send a peaceful execution request to party $\gamma.\mathsf{Bob}$ only if the round number $\tau = 1 \mod 4$. Party $\gamma.\mathsf{Bob}$ can send a peaceful update request to party $\gamma.\mathsf{Alice}$ only if the round number $\tau = 3 \mod 4$. Hence, if an honest party P receives the execution message from the environment, the party might need to wait for 3 rounds until it is allowed to propose the execution to the other party.

To summarize, the execution protocol takes in the optimistic case up to 5 rounds and up to $4\Delta + 5$ rounds in the pessimistic case when the registration and force execution via the dispute board are needed.

Close a virtual state channel with direct dispute The closing procedure of a virtual state channel with direct dispute γ starts automatically in round $\gamma.\mathsf{validity}$. The goals of the closing procedure is to unlock the coins that were locked in the subchannels of γ when the channel γ was created. Importantly, the coins have to be unlocked back to the subchannels in a way that corresponds to the final distribution in the virtual state channel γ. Let us now give an high level explanation on how this is done.

The end-users of the channel, $\gamma.\mathsf{Alice}$ and $\gamma.\mathsf{Bob}$, first try to unlock the coins back to the subchannels in a peaceful way by updating the $\mathtt{dVSCC}_i$ contract instances in the subchannels. More precisely, party $\gamma.\mathsf{Alice}$ locally executes the $\mathtt{dVSCC}_i$ contract instance with identifier $cid_A := \gamma.\mathsf{Alice}||\gamma.\mathsf{id}$ on the function $f = \mathtt{Close}_i^{\mathsf{d}}$ and input parameter z containing all contract instances ever opened in γ, i.e. z contains $\nu := \gamma.\mathsf{cspace}(cid)$ if $\nu \neq \bot$. The function $\mathtt{Close}_i^{\mathsf{d}}$ is designed in such a way that it processes every contract instance $\nu \in z$ and adjusts the cash attributes of cid_A according to the coin distribution in ν. Once all contract instances from z are processed, the balances in cid_A correspond to the final balances of γ. Hence $\mathtt{Close}_i^{\mathsf{d}}$ can unlock coins from cid_A and output a terminated contract instance storage. The party $\gamma.\mathsf{Alice}$ uses the result of the local execution as an update proposal for the contract instance cid_A. The update request is made via the hybrid ideal functionality $\mathcal{W}_{dch}^{\widehat{\mathcal{L}}(\Delta)}(i-1, \mathcal{C}_1, \mathcal{C}_0)$. The party $\gamma.\mathsf{Bob}$ proceeds analogously with the $\mathtt{dVSCC}_i$ contract instance with identifier $cid_B := \gamma.\mathsf{Alice}||\gamma.\mathsf{id}$.

If the intermediary of the channel, party $\gamma.\mathsf{Ingrid}$, receives two symmetric update requests from the hybrid ideal functionality $\mathcal{W}_{dch}^{\widehat{\mathcal{L}}(\Delta)}(i-1, \mathcal{C}_1, \mathcal{C}_0)$, it confirms both of them. However, if the update requests are not symmetric, which implies that at least one of the end-users is trying to close γ with a false view on the set $\gamma.\mathsf{cspace}$, $\gamma.\mathsf{Ingrid}$ does not confirm any of the updates. This forces the end-users to first publicly reach consensus on the content of the set $\gamma.\mathsf{cspace}$ and only then complete the closing procedure.

Hence, parties $\gamma.\mathsf{Alice}$ and $\gamma.\mathsf{Bob}$ register all the contract instances in $\gamma.\mathsf{cspace}$ on the dispute board which guarantees a unique global view on the set $\gamma.\mathsf{cspace}$. This is done by running `RegisterDirect`. After the registration procedure, $\gamma.\mathsf{Alice}$ requests execution of cid_A on the function $\mathtt{Close}_i^{\mathsf{d}}$ and $\gamma.\mathsf{Bob}$ requests execution of cid_B on the function $\mathtt{Close}_i^{\mathsf{d}}$. If one of the parties did not request the execution, the intermediary can, after some time, request it herself. Let us emphasize that since the execution of cid_A and cid_B is done via the hybrid ideal functionality $\mathcal{W}_{dch}^{\widehat{\mathcal{L}}(\Delta)}(i-1, \mathcal{C}_1, \mathcal{C}_0)$, the input parameter z is equal to the set $\gamma.\mathsf{cspace}$ as it appears on the dispute board in both executions. This guarantees symmetric termination of both cid_A and cid_B.

The closing procedure is completed in round $\gamma.\mathsf{validity} + 3$ in the optimistic case when all users of γ are honest. Otherwise it is completed before round $\gamma.\mathsf{validity} + 3\Delta + 2 + \mathrm{TimeExe}(\lceil i/2 \rceil)$, where $\mathrm{TimeExe}(\lceil i/2 \rceil)$ represents the upper bound on the time complexity of force execution in a virtual state channel of length at most $\lceil i/2 \rceil$ (see Appx. B.5 for the precise definitions). Let us emphasize that the above high level description excludes some technical details which can be found in the formal description in Appx. G.

7 Protocol for Multi-Party Channels

In this section we describe a concrete protocol that realizes the ideal functionality $\mathcal{F}_{mpch}(i, \mathcal{C}_0)$ for $i \in \mathbb{N}$ and any set of contract codes $\mathcal{C}_0$ in the $\mathcal{W}_{dch}(i, \mathcal{C}_1, \mathcal{C}_0)$-hybrid world. Recall that $\mathcal{W}_{dch}(i, \mathcal{C}_1, \mathcal{C}_0)$ is a functionality wrapper (cf. Sec. 5) combining the dispute board $\mathcal{F}_{DB}(\mathcal{C}_0)$ and the ideal functionality $\mathcal{F}_{dch}(i, \mathcal{C}_1)$ for building two-party state channels of length up to i supporting contract instances whose codes are in $\mathcal{C}_1$. Our strategy of constructing a protocol $\Pi_{mpch}(i, \mathcal{C}_0)$ for multi-party channels is to distinguish two cases. These cases also outline the minimal requirements on the set of supported contracts $\mathcal{C}_1$:

Case 1: If a party receives a message about a two-party state channel, it forwards the request to the hybrid ideal functionality. Thus, we require $\mathcal{C}_0 \subset \mathcal{C}_1$.

Case 2: For the case when a party receives a message about a multi-party channel γ, we design a new protocol that uses (a) the dispute board for fair resolution of disagreements between the users of γ and (b) two-party state channels as a building block that provides monetary guarantees. For (b) we need the subchannels of γ to support contract instances of a special code $\texttt{mpVSCC}_i$; hence, $\texttt{mpVSCC}_i \in \mathcal{C}_1$.

Since case 1 is rather straightforward, we refer the reader to Appx. H for technical details. Here we discuss case 2 in more detail, by first describing the special contract code $\texttt{mpVSCC}_i$ and then the protocol for multi-party channels.

7.1 Multi-Party Channel Contract

In order to create a multi-party channel γ, parties of the channel need to open a special two-party contract instance in each subchannel of γ (recall the example depicted in Fig. 2 in Sec. 2.3). We denote the code of these instances $\texttt{mpVSCC}_i$, where $i \in \mathbb{N}$ is the maximal length of the channel in which an instance of $\texttt{mpVSCC}_i$ can be opened. A contract instance of $\texttt{mpVSCC}_i$ in a subchannel of γ between two parties P and Q can be understood as a "copy" of γ, where P plays the role of all parties from the set V_P and Q plays the role of parties from the set V_Q, where $(V_P, V_Q) := \gamma.\mathsf{split}(\{P, Q\})$. The purpose of the $\texttt{mpVSCC}_i$ contract instances is to guarantee to every user of γ that he gets the right amount of coins back to his subchannels when γ is being closed in round $\gamma.\mathsf{validity}$. And this must be true even if all other parties collude.[23]

The contract has in addition to the mandatory attributes $\mathsf{users}, \mathsf{locked}, \mathsf{cash}$ (see Sec. 3.2) one additional attribute $\mathsf{virtual\text{–}channel}$ storing the initial version of the multi-party channel γ. The interface of the contract code $\texttt{dVSCC}_i$ can be found in Fig. 13. The description of the contract functions is presented together with the protocol $\Pi_{mpch}(i, \mathcal{C}_0)$ later in this section. The formal definition can be found in Appx. G.

Interface of the contract code $\texttt{mpVSCC}_i$

Attributes: $\mathsf{users}, \mathsf{locked}, \mathsf{cash}$ (mandatory attributes, see Sec. 3.2),
$\mathsf{virtual\text{–}channel}$ (stores the initial version of the multi-party channel γ);
Functions:
$\texttt{Init}_i^{\texttt{mp}}$: the constructor of the contract (see page 50)
$\texttt{Close}_i^{\texttt{mp}}$: a contract function called when the multi-party channel is being closed (see page 54)

Fig. 13. Interface of the contract code $\texttt{mpVSCC}_i$.

7.2 Protocol Description

We now informally explain the main ideas of the protocol $\Pi_{mpch}(i, \mathcal{C}_0)$ for the Case 2, i.e. when a message about a multi-party channel is received by the parties. We discuss each part of the protocol separately. The formal description of the protocol can be found in Appx. H.

[23] This statement assumes that the only contract instances that can be opened in the multi-party channel are the ones whose code allows any user to enforce termination before time $\gamma.\mathsf{validity}$.

Create a multi-party channel. Parties are instructed by the environment to create a multi-party channel γ via the message (create, γ). As already explained before, parties have to add an instance of $\texttt{mpVSCC}_i$ to every subchannel of γ. This is, on high level, done as follows. Let P and Q be the two parties of a two party channel α which is a subchannel of γ. Let us assume for now that $\mathsf{Order}_{\mathcal{P}}(P) < \mathsf{Order}_{\mathcal{P}}(Q)$ (see Sec. 3.2 for the definition of $\mathsf{Order}_{\mathcal{P}}$). If P receives the message (create, γ) in round τ_0, it requests an update of a contract instance in the state channel α via the hybrid ideal functionality. As parameters of this request, P chooses the channel identifier $cid := P||Q||\gamma.\mathsf{id}$, the contract storage $\texttt{Init}_i^{\texttt{mp}}(P, Q, \tau_0, \gamma)$ and contract code $\texttt{mpVSCC}_i$. Recall that $\texttt{Init}_i^{\texttt{mp}}$ is the constructor of the special contract $\texttt{mpVSCC}_i$. If the party Q also received the message (create, γ) in round τ_0, it knows that it should receive an update request from the hybrid ideal functionality in round $\tau_0 + 1$. If this is indeed the case, Q inspects P's proposal and confirms the update.

Assume that the environment sends (create, γ) to all users of γ in the same round τ_0. If all parties follow the protocol, in round $\tau_0 + 2$ all subchannels of γ should contain a new contract instance with the contract code $\texttt{mpVSCC}_i$. However, note that a party $P \in \gamma.\mathsf{users}$ only has information about subchannels it is part of, i.e. about subchannels $S_P := \{\alpha \in \gamma.\mathsf{subchan} \mid P \in \alpha.\mathsf{end\text{–}users}\}$. To this end, every honest party P sends a message "create–ok" to every other party if all subchannels in S_P contain a new $\texttt{mpVSCC}_i$ instance in round $\tau_0 + 2$. Hence, if all parties are honest, latest in round $\tau_0 + 3$ every party knows that the creation process of γ is completed successfully. However, if there is a malicious party P that sends the "create–ok" to all parties except for one, let us call it Q, then in round $\tau_0 + 3$ only Q thinks that creation failed. In order to reach total consensus on creation among honest parties, Q signals the failure by sending a message "create–not–ok" to all other parties.

To conclude, an honest party outputs $(\text{created}, \gamma)$ to the environment if (1) it received "create–ok" from all parties in round $\tau_0 + 3$ and (2) did not receive any message "create–not–ok" in round $\tau_0 + 4$.

Register a contract instance in a multi-party channel. As long as users of the multi-party channel γ behave honestly, they can update/execute contract instances in the channel γ by communicating with each other. However, once the users disagree, they need some third party to fairly resolve their disagreement. The dispute board, modeled by the hybrid ideal functionality $\mathcal{W}_{dch}(i, \mathcal{C}_1, \mathcal{C}_0)$, plays the role of such a judge.

Parties might run into dispute when they update/execute the contract instance or when they are closing the channel γ. In order to avoid code repetition, we define the dispute process as a separate procedure $\texttt{mpRegister}(P, id, cid)$. The input parameter P denotes the initiating party of the dispute process, the parameter id identifies the channel γ and cid is the identifier of the contract instance parties disagree on. The initiating party submits its version of the contract instance, ν^P, to the dispute board which then informs all other parties about P's registration request. If a party Q has a contract instance version with higher version number, i.e. $\nu^Q.\mathsf{version} > \nu^P.\mathsf{version}$, then Q submits this to the dispute board. After a certain time, which is sufficient for other parties to react to P's registration request, any party can complete the process by sending "finalize" to the dispute board which then informs all parties about the result.

Update a contract instance in a multi-party channel. In order to update the storage of a contract instance in a multi-party channel from σ to $\tilde{\sigma}$, the environment sends the message $(\text{update}, id, cid, \tilde{\sigma}, \texttt{C})$ to one of the parties P, which becomes the *initiating party*. Let τ_0 denote the round in which P receives this message. On a high level the update protocol works as follows. P sends the signed new contract storage $\tilde{\sigma}$ to all other parties of γ. Each of these parties $Q \in \gamma.\mathsf{other\text{–}party}(P)$ verifies if the update request is valid (i.e., if P's signature is correct) and outputs the update request to the environment. If the environment confirms the request, Q also signs the new contract storage $\tilde{\sigma}$ and sends it as part of the "update–ok" message to the other channel parties. In case the environment does not confirm, Q sends a rejection message "update–not–ok" which contains Q's signature on the original storage σ but with a version number that is increased by two, i.e., if the original version number was w, then Q signs σ with $w + 2$.

If in round $\tau_0 + 2$ a party $P \in \gamma.\mathsf{users}$ is missing a correctly signed reply from at least one party, it is clear that someone misbehaved. Thus, P initiates the registration procedure to resolve the disagreement via the dispute board.

If P received at least one rejection message, it is unclear to P if there is a malicious party or not. Note that from P's point of view it is impossible to distinguish whether (a) one party sends the "update–not–ok" message to P and the message "update–ok" to all other parties, or (b) one honest party simply does not agree with the update and sends the "update–not–ok" message to everyone. To resolve this

uncertainty, P communicates to all other parties that the update failed by sending the signed message (update–not–ok, $\sigma, w+2$) to all other parties. If all honest parties behave as described above, in round $\tau_0 + 3$ party P must have signatures of all parties on the original storage with version number $w+2$; hence, consensus on rejection is reached. If P does not have all the signatures at this point, it is clear that at least one party is malicious. Thus, P initiates the registration which enforces the consensus via the dispute board.

If P receives a valid "update–ok" from all parties in round $\tau_0 + 2$, she knows that consensus on the updated storage $\tilde{\sigma}$ will eventually be reached. This is because in worst case, P can register $\tilde{\sigma}$ on the dispute board. Still, P has to wait if no other party detects misbehavior and starts the dispute process or sends a reject message in which case P initiates the dispute. If none of this happens, all honest parties output the message "updated" in round $\tau_0 + 3$. Otherwise they output the message after the registration is completed.

Execute a contract instance in a multi-party channel. The environment triggers the execution process by sending the instruction (execute, id, cid, f, z) to a party P in round τ_0. P first tries to perform the execution of the contract instance with identifier cid in a channel γ with identifier id peacefully, i.e. without touching the blockchain. An intuitive design of this process would be to let P compute $f(z)$ locally and send her signature on the new contract storage (together with the environment's instruction) to all other users of γ. Every other user Q would verify this message by recomputing $f(z)$ and confirm the new contract storage by sending her signature on it to the other users of γ.

It is easy to see that this intuitive approach fails when two (or more) parties want to peacefully execute the same contract instance cid in the same round. While in two party channels this can be solved by assigning "time slots" for each party, this idea cannot be generalized to the n-party case, without blowing up the number of rounds needed for peaceful execution from $O(1)$ to $O(n)$. To keep the peaceful execution time constant, we let each contract instance have its own *execution period* which consists of four rounds:

Round 1: If P received (execute, id, cid, f, z) in this or the previous 3 rounds, it sends (peaceful–request, id, cid, f, z, τ_0) to all other parties.

Round 2: P locally sorts[24] all requests it received in this round (potentially including its own from the previous round), locally performs all the executions and sends the signed resulting contract storage to all other parties.

Round 3: If P did not receive valid signatures on the new contract storage from all other parties, it starts the registration process.

Round 4: Unless some party started the registration process, P outputs an execution success message.

If the peaceful execution fails, i.e. one party initiates registration, all execution requests of this period must be performed forcefully via the dispute board.

Close a multi-party channel. The closing procedure of a multi-party channel begins automatically in round $\gamma.\mathsf{validity}$. Every pair of parties $\{P, Q\} \in \gamma.\mathsf{E}$ tries to peacefully update the $\mathtt{mpVSCC}_i$ contract instance, let us denote its identifier cid, in their subchannel $\alpha := \gamma.\mathsf{subchan}(\{P, Q\})$. More precisely, both parties locally execute the function $\mathtt{Close}_i^{\mathtt{mp}}$ of contract instance cid with input parameter $z := \gamma.\mathsf{cspace}$ – the tuple of all contract instances that were ever opened in γ. The function $\mathtt{Close}_i^{\mathtt{mp}}$ adjusts the balances of users in cid according to the provided contract instances in z and unlocks all coins from cid back to α.

If the peaceful update fails, then at least one party is malicious and either does not communicate or tries to close the channel γ with a false view on the set $\gamma.\mathsf{cspace}$. In this case, users have to register all contract instances of γ on the dispute board. This guarantees a fixed global view on $\gamma.\mathsf{cspace}$. Once the registration process is over, the $\mathtt{mpVSCC}_i$ contract instances in the subchannels can be terminated using the execute functionality of $\mathcal{W}_{dch}(i, \mathcal{C}_1, \mathcal{C}_0)$ on function $\mathtt{Close}_i^{\mathtt{mp}}$. Since the set $\gamma.\mathsf{cspace}$ is now publicly available on the dispute board, the parameter z will be the same in all the $\mathtt{mpVSCC}_i$ contract instance executions in the subchannels. Technically, this is taken care of by the wrapper $\mathcal{W}_{ch}(i, \mathcal{C}_1, \mathcal{C}_0)$ which overwrites the parameter z of every execution request with function $\mathtt{Close}_i^{\mathtt{mp}}$ to the relevant content of the dispute board. See Sec. 5 and Appx. E for more details.

[24] We assume a fixed ordering on peaceful execution requests. See Appx. H for more details.

Theorem 2. *Suppose the underlying signature scheme is existentially unforgeable against chosen message attacks. For every set of contract codes $\mathcal{C}_0$, every $i \geq 1$ and every $\Delta \in \mathbb{N}$, the protocol $\Pi_{mpch}(i, \mathcal{C}_0)$ in the $\mathcal{W}_{dch}(i, \mathcal{C}_1, \mathcal{C}_0)$-hybrid model emulates the ideal functionality $\mathcal{F}_{mpch}(i, \mathcal{C}_0)$.*

8 Conclusion

We presented the first full specification and construction of a state channel network that supports multiparty channels. The pessimistic running time of our protocol can be made constant for arbitrary complex channels. While we believe that this is an important contribution by it self, we also think that it is very likely that the techniques developed by us will have applications beyond the area of off-chain channels. In particular, the modeling of multiparty state channels that we have in this paper can be potentially useful in other types of off-chain protocols, e.g., in Plasma [29]. We leave extending our approach to such protocols as an interesting research direction for the future.

Acknowledgments. This work was partly supported by the German Research Foundation (DFG) Emmy Noether Program FA 1320/1-1, the DFG CRC 1119 CROSSING (project S7), the Ethereum Foundation grant *Off-chain labs: formal models, constructions and proofs*, the Foundation for Polish Science (FNP) grant TEAM/2016-1/4, the German Federal Ministry of Education and Research (BMBF) *iBlockchain* project, by the Hessen State Ministry for Higher Education, Research and the Arts (HMWK) and the BMBF within CRISP, and by the Polish National Science Centre (NCN) grant 2014/13/B/ST6/03540.

We thank Jan Camenisch for useful discussions on the UC model.

References

[1] I. Allison. *Ethereum's Vitalik Buterin explains how state channels address privacy and scalability.* 2016.

[2] I. Bentov and R. Kumaresan. "How to Use Bitcoin to Design Fair Protocols". In: *CRYPTO 2014, Part II*. Ed. by J. A. Garay and R. Gennaro. Vol. 8617. LNCS. Springer, Heidelberg, Aug. 2014, pp. 421–439. DOI: 10.1007/978-3-662-44381-1_24.

[3] *Bitcoin Wiki: Payment Channels*. https://en.bitcoin.it/wiki/Payment_channels. 2018.

[4] R. Canetti. "Universally Composable Security: A New Paradigm for Cryptographic Protocols". In: *42nd FOCS*. IEEE Computer Society Press, Oct. 2001, pp. 136–145.

[5] R. Canetti and T. Rabin. "Universal Composition with Joint State". In: *CRYPTO 2003*. Ed. by D. Boneh. Vol. 2729. LNCS. Springer, Heidelberg, Aug. 2003, pp. 265–281.

[6] R. Canetti et al. "Universally Composable Security with Global Setup". In: *TCC 2007*. Ed. by S. P. Vadhan. Vol. 4392. LNCS. Springer, Heidelberg, Feb. 2007, pp. 61–85.

[7] *Celer Network*. https://www.celer.network. 2018.

[8] T. Close. *Nitro Protocol*. Cryptology ePrint Archive, Report 2019/219. https://eprint.iacr.org/2019/219. 2019.

[9] *Counterfactual*. https://counterfactual.com. 2018.

[10] C. Decker and R. Wattenhofer. "A Fast and Scalable Payment Network with Bitcoin Duplex Micropayment Channels". In: *Stabilization, Safety, and Security of Distributed Systems*. Ed. by A. Pelc and A. A. Schwarzmann. Springer International Publishing, 2015, pp. 3–18. ISBN: 978-3-319-21741-3.

[11] D. Dolev and H. R. Strong. "Authenticated Algorithms for Byzantine Agreement". In: *SIAM J. Comput.* 12.4 (1983), pp. 656–666. DOI: 10.1137/0212045. URL: https://doi.org/10.1137/0212045.

[12] S. Dziembowski et al. "General State Channel Networks". In: *ACM CCS 18*. 2018, pp. 949–966.

[13] S. Dziembowski et al. *Perun: Virtual Payment Hubs over Cryptographic Currencies*. conference version accepted to the 40th IEEE Symposium on Security and Privacy (IEEE S&P) 2019. 2017. URL: http://eprint.iacr.org/2017/635.

[14] J. A. Garay et al. "Round Complexity of Authenticated Broadcast with a Dishonest Majority". In: *48th FOCS*. IEEE Computer Society Press, Oct. 2007, pp. 658–668.

[15] O. Goldreich. *Foundations of Cryptography: Volume 1*. New York, NY, USA: Cambridge University Press, 2006. ISBN: 0521035368.

[16] D. Hofheinz and J. Mueller-Quade. *A Synchronous Model for Multi-Party Computation and the Incompleteness of Oblivious Transfer*. Cryptology ePrint Archive, Report 2004/016. http://eprint.iacr.org/2004/016. 2004.

[17] Y. T. Kalai et al. "Concurrent Composition of Secure Protocols in the Timing Model". In: *Journal of Cryptology* 20.4 (Oct. 2007), pp. 431–492.

[18] J. Katz and Y. Lindell. *Introduction to Modern Cryptography (Chapman & Hall/Crc Cryptography and Network Security Series)*. Chapman & Hall/CRC, 2007. ISBN: 1584885513.

[19] J. Katz et al. "Universally Composable Synchronous Computation". In: *TCC 2013*. Ed. by A. Sahai. Vol. 7785. LNCS. Springer, Heidelberg, Mar. 2013, pp. 477–498. DOI: 10.1007/978-3-642-36594-2_27.

[20] R. Khalil and A. Gervais. *NOCUST - A Non-Custodial 2nd-Layer Financial Intermediary*. Cryptology ePrint Archive, Report 2018/642. https://eprint.iacr.org/2018/642. 2018.

[21] R. Khalil and A. Gervais. "Revive: Rebalancing Off-Blockchain Payment Networks". In: *ACM CCS 17*. Ed. by B. M. Thuraisingham et al. ACM Press, 2017, pp. 439–453.

[22] J. Lind et al. "Teechain: Reducing Storage Costs on the Blockchain With Offline Payment Channels". In: *Proceedings of the 11th ACM International Systems and Storage Conference, SYSTOR 2018*. ACM, 2018, p. 125. DOI: 10.1145/3211890. URL: http://doi.acm.org/10.1145/3211890.3211904.

[23] G. Malavolta et al. "Concurrency and Privacy with Payment-Channel Networks". In: *ACM CCS 17*. Ed. by B. M. Thuraisingham et al. ACM Press, 2017, pp. 455–471.

[24] P. McCorry et al. *Pisa: Arbitration Outsourcing for State Channels*. Cryptology ePrint Archive, Report 2018/582. https://eprint.iacr.org/2018/582. 2018.

[25] P. McCorry et al. "You sank my battleship ! A case study to evaluate state channels as a scaling solution for cryptocurrencies". In: 2018.

[26] A. Miller et al. "Sprites: Payment Channels that Go Faster than Lightning". In: *CoRR* abs/1702.05812 (2017). URL: http://arxiv.org/abs/1702.05812.

[27] S. Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. http://bitcoin.org/bitcoin.pdf. 2009.

[28] J. B. Nielsen. "On Protocol Security in the Cryptographic Model". PhD thesis. Aarhus University, 2003.

[29] J. Poon and V. Buterin. *Plasma: Scalable Autonomous Smart Contracts*. 2017. URL: http://plasma.io/plasma.pdf.

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

[31] S. Roos et al. "Settling Payments Fast and Private: Efficient Decentralized Routing for Path-Based Transactions". In: *NDSS*. 2018.

[32] N. Szabo. *Smart Contracts: Building Blocks for Digital Markets*. Extropy Magazine. 1996.

Appendix (Supplementary Material)

A Further Background on Contracts and State Channels [12]

Some prior work on channels has already been provided in Sec. 2.1. In this section we provide some further details on the dispute handling procedure that was mentioned there. In order to better understand it, we start by providing some more technical details on the state channel off-chain execution mechanism. Let ν be a contract instance of the pre-image selling contract $\mathsf{C}_{\mathsf{sell}}$, say, and denote by G_0 its initial state. To deploy ν in the state channel both parties exchange signatures on $(G_0, 0)$, where the second parameter in the tuple will be called the *version number* (and y is the value whose pre-image Alice wants to learn). The rest of the execution is done by exchanging signatures on further states with increasing version number. For instance, suppose in the pre-image selling contract C_{sell} (described earlier in this section) the last state on which both parties agreed on was $(G_1, 1)$ (i.e., both parties have signatures on this state tuple), and Bob wants to provide x' such that $H(x') = y$. To this end, he locally evaluates the contract instance to obtain the new state $(G_2, 2)$, and sends it together with his signature to Alice. Alice verifies the correctness of both the computation and the signature, and if both checks pass, he replies with his signature on $(G_3, 3)$.

Let us start by taking a look at the dispute resolution for ledger channels and consider a setting where a malicious Alice does not reply with her signature on $(G_3, 3)$ (for example because she wants to avoid "acknowledging" that she received x'). In this case, Bob can force the execution of the contract instance ν *on*-chain by registering in the state channel contract SCC the latest state on which both parties agreed on. To this end, Bob will send the tuple $(G_2, 2)$ together with the signature of Alice to SCC. Of course, SCC cannot accept this state immediately because it may be the case that Bob cheated by registering an outdated state.[25] Hence, the ledger contract SCC gives Alice time Δ to reply with a more recent signed state (recall that in Sec. 1.1 we defined Δ to be a constant that is sufficiently large so that every party can be sure her transaction is processed by the ledger within this time). When Δ time has passed, SCC finalizes the registration process by storing the version with the highest version number in its storage. Once registration is completed, the parties can continue the execution of the contract instance on-chain.[26] As described in Sec. 2.1 the dispute process for virtual state channels is significantly more complex.

B Additional Notation and Definitions

B.1 The Clock Functionality

As discussed in Sec. 3.1, we formalize the notion of rounds via the ideal functionality $\widehat{\mathcal{G}}_{clock}$ representing the clock. On high level, the ideal functionality requires all honest parties to indicate that they are prepared to proceed to the next round before the clock is "ticked". We consider the clock functionality presented by Katz et. al [19] with two minor modifications. Firstly, since we use $\widehat{\mathcal{G}}_{clock}$ as a global ideal functionality, it additionally maintains a internal "round counter" t. And secondly, because we consider only static corruption, the set of honest parties is fixed and hence the functionality does not need to include a mechanism for corrupting parties "on the fly".

Ideal functionality $\widehat{\mathcal{G}}_{clock}$

The functionality is parameterized by a set of parties $\mathcal{P} = \{P_1, \ldots, P_n\}$. The functionality maintains an internal counter t which is initially set to 0. In addition, for each party P_i it maintains a flag d_i initially set to 0. The functionality accepts the following messages:

Round Ok

Upon receiving the message (RoundOK) from a party $P_i \in \mathcal{P}$, set $d_i := 1$. If for all honest $P_j \in \mathcal{P}$, $d_j = 1$, then set $t := t + 1$ and reset $d_j := 0$ for all $j \in [n]$. In any case, leak (RoundOK, P_i) to the adversary.

[25] Notice that SCC is oblivious to what happened inside the ledger state channel γ after it was created.

[26] In the example that we considered, Bob can now force Alice bear the consequences of the fact that he revealed x' to the contract instance.

Round Request

Upon receiving the message (RequestRound) from $P_i \in \mathcal{P}$, reply to P_i by sending (t, d_i).

B.2 Formal Definition of a Contract Code

We will now define formally the notion of *contract code* that was already described informally in Sec. 3.2. Formally a contract code consists of some functions (in Ethereum they are written in Solidity) that operate on contract storage. The set of possible contract storages is usually restricted (e.g. the functions expect that it has certain attributes defined). We call the set of restricted storages the *admissible contract storages* and typically denote it by Λ.

Formally, we define a *contract code* as a tuple $\mathtt{C} = (\Lambda, g_1, \ldots, g_r, f_1, \ldots, f_s)$, where Λ are the admissible contract storages and $g_1, \ldots, g_r$ are functions called *contract constructors*, and $f_1, \ldots, f_s$ are called *contract functions*. Each contract constructor g_i is a function that takes as input a tuple (P, τ, z), with $P \in \mathcal{P}, \tau \in \mathbb{N}$, and $z \in \{0,1\}^*$, and produces as output an admissible contract storage σ or a special symbol $\perp$ (in which case we say that the contract construction *failed*). The meaning of these parameters is as follows: P is the identity of the party that called the function, τ is the current round, and z is used to pass additional parameters to g_i. The constructors are used to create a new instance of the contract. If the contract construction did not fail, then $g_i(P, \tau, z)$ is the initial storage of a new contract instance.

Each contract function f_i takes as input a tuple (σ, P, τ, z), with $\sigma \in \Lambda$ being an admissible contract storage, $P \in \sigma.\mathsf{users}$, $\tau \in \mathbb{N}$ and $z \in \{0,1\}^*$ (the meaning of these parameters is as before). It outputs a tuple $(\tilde{\sigma}, \mathsf{add}, m)$, where $\tilde{\sigma}$ is the new contract storage (that replaces contract storage σ in the contract instance), values $\mathsf{add}\colon [n] \to \mathbb{R}_{\geq 0}$ correspond to the amount of coins that were *unlocked* from the contract storage to each user (as a result of the execution of f_i), and $m \in \{0,1\}^* \cup \{\perp\}$ is an *output message*. If the output message is $\perp$, we say that the execution *failed* (we assume that the execution always fails if a function is executed on input that does not satisfy the constraints described above, e.g., it is applied to σ that is not admissible). If the output message $m \neq \perp$, then we require that $\tilde{\sigma}$ is an admissible contract storage and the attribute users in $\tilde{\sigma}$ is identical to the one in σ. In addition, it must hold that $\sum_{i \in [n]} \mathsf{add}(i) = \sigma.\mathsf{locked} - \tilde{\sigma}.\mathsf{locked}$. Intuitively, this condition guarantees that executions of a contract functions can never result in unlocking more coins than what was originally locked in the contract storage.

B.3 Formal Definition of the Function γ.split

In order to formally define the function $\gamma.\mathsf{split}$, we first define an auxiliary function $\mathsf{Decompose}$ which, intuitively, decomposes a graph into connected components. More precisely, the function $\mathsf{Decompose}$ on input a graph $G = (V, E)$ outputs $G_1, \ldots, G_\ell$, where (i) $\bigcup_{i \in [\ell]} G_i = \left(\bigcup_{i \in [\ell]} V_i, \bigcup_{i \in [\ell]} E_i\right) = (V, E) = G$, (ii) every G_i is a connected graph and (iii) for every $i \neq j$ it holds that $V_i \cap V_j = \emptyset$. Using the function $\mathsf{Decompose}$, we now define the function $\gamma.\mathsf{split}$. On input (P, Q), where $\{P, Q\} \in \gamma.\mathsf{E}$, proceeds as follows

1. Set $V' := \gamma.\mathsf{users}$ and $E' := \gamma.\mathsf{E} \setminus \{P, Q\}$
2. Set $(V_1, E_1) \cup (V_2, E_2) := \mathsf{Decompose}(V', E')$
3. If $P \in V_1$, then set $(V_P, V_Q) := (V_1, V_2)$, otherwise set $(V_P, V_Q) := (V_2, V_1)$.
4. Output (V_P, V_Q).

B.4 Notation Simplifying the Formal Descriptions

In order to simplify the notation in the description of the ideal functionalities and protocols, we introduce symbolic notation for sending and receiving messages. Instead of the instruction "Send the message msg to party P in round τ", we write $msg \overset{\tau}{\hookrightarrow} P$. Instead of the instruction "Send the message msg to all parties in the set $\mathcal{S}$ in round τ", we write $msg \overset{\tau}{\hookrightarrow} \mathcal{S}$. By $msg \overset{\tau}{\hookleftarrow} P$ we mean that an entity (i.e. the ideal functionality) receives a message msg from party P in round τ, and we use $msg \overset{\tau \leq \tau_1}{\hookleftarrow} P$ when an entity receives msg from party P latest in round τ_1.

Recall from Sec. 3.2 that each entity (ideal functionality or party in a protocol), stores and maintains a set of all state channels it is aware of. This set is called *channel space* and denoted Γ. When we want

to emphasize that we are referring to a local version of a state channel stored by some entity T, we add T to the superscript. So for instance, $\gamma^T := \Gamma^T(id)$ denotes T's local version of the state channel γ with identifier id as stored in T's channel space Γ^T.

We define an auxiliary procedure `UpdateChanSpace`, which takes as input a channel space Γ, a channel identifier id, a contract instance identifier cid, a new contract instance storage $\tilde{\sigma}$ together with the contract code $\mathtt{C}$ and a vector of values add representing the required change in the cash values of the state channel with identifier id. The procedure sets $\Gamma(id).\mathsf{cspace}(cid) := (\tilde{\sigma}, \mathtt{C})$, adds $\mathsf{add}(T)$ coins to $\Gamma(id).\mathsf{cash}(T)$ for every $T \in \gamma.\mathsf{users}$. Finally, it outputs the updated channel space Γ. In Fig. 14 we define the procedure formally.

$\mathtt{UpdateChanSpace}(\Gamma, id, cid, \tilde{\sigma}, \mathtt{C}, \mathsf{add})$
Let $\gamma := \Gamma(id)$ and make the following updates: 1. For every $T \in \gamma.\mathsf{users}$ add $\mathsf{add}(T)$ coins to $\gamma.\mathsf{cash}(T)$ 2. Set $\gamma.\mathsf{cspace}(cid)$ equal to the tuple $(\tilde{\sigma}, \mathtt{C})$. Output Γ with the updated contract instance cid in the state channel γ.

Fig. 14. Auxiliary procedure for updating the channel space.

It will be often the case that the values of the input parameter add will correspond to the difference between the amount of coins locked in the contract instance before the update and the amount of coins in the new contract instance. To simplify the descriptions even further and avoid code repetition, we define another auxiliary procedure $\mathtt{UpdateChanSpace}^*$ which will derive the parameter add automatically from the new contract storage $\tilde{\sigma}$. See the formal description in Fig. 15.

$\mathtt{UpdateChanSpace}^*(\Gamma, id, cid, \tilde{\sigma}, \mathtt{C})$
Let $\gamma := \Gamma(id)$ and $\sigma := \gamma.\mathsf{cspace}(cid).\mathsf{storage}$. For every $P \in \sigma.\mathsf{users}$ define the value x_P as follows: if $\sigma = \bot$, the set $x_P := 0$. Else set $x_P := \sigma.\mathsf{cash}(P)$. Make the following updates: 1. For every $P \in \sigma.\mathsf{users}$ add $x_P - \tilde{\sigma}.\mathsf{cash}(P)$ coins to $\gamma.\mathsf{cash}(P)$. 2. Set $\gamma.\mathsf{cspace}(cid)$ equal to the tuple $(\tilde{\sigma}, \mathtt{C})$. Output Γ with the updated contract instance cid in the state channel γ.

Fig. 15. Modification of the auxiliary procedure for updating the channel space.

We define both $\mathtt{UpdateChanSpace}^*$ and `UpdateChanSpace` in case a party wants to update the private extended version of the contract instance. Notice that in this case procedures will take additional two parameters: the new version number and the signatures created by the parties.

We also define the auxiliary procedure that takes as input a signed contract instance version, verifies if the instance is correctly signed by all users of the contract instance and outputs a decision bit.

$\mathtt{VerifyInstance}(id, cid, \nu)$
1. Let $b := 1$, $\sigma := \nu.\mathsf{storage}$, $\mathtt{C} := \nu.\mathsf{code}$ and $w := \nu.\mathsf{version}$. 2. If $\sigma \notin \mathsf{code}.\Lambda$, then set $b := 0$. 3. For every $P \in \sigma.\mathsf{users}$: if $\mathsf{Vrfy}_{pk_P}(id, cid, \sigma, \mathtt{C}, w; \nu.\mathsf{sign}(P)) \neq 1$, set $b := 0$. 4. Return b.

Fig. 16. An auxiliary procedure that verifies validity of a signed contract instance version.

In addition to the channel space, each party P maintains a set Γ^P_{aux} containing additional information about the contract instances in the open state channels of the party. The tuple $aux := \Gamma^P_{aux}(id, cid)$ has the following attributes: $aux.\mathsf{next\text{-}version} \in \mathbb{N}$ denoting the version number to be used during the next update of the contract instance (id, cid);[27]; $aux.\mathsf{corrupt} \in \{0, 1\}$ which is set to 1 the first time parties run into disagreement about the contract instance (id, cid); $aux.\mathsf{registered} \in \{0, 1\}$ which is set to 1 the

[27] For technical reasons (see [12]) it is not always the case that $\Gamma(id).\mathsf{cspace}(cid).\mathsf{version} + 1 = \Gamma_{aux}(id, cid).\mathsf{next\text{-}version}$.

if the contract instance (id, cid) is registered (on the blockchain in case of ledger state channel and in the subchannels in case of virtual state channel); and if $\Gamma^P(id)$ is a virtual state channel, then aux has an addition attribute $aux.\mathsf{toExecute}$ which is a set containing all functions that party P requested to "forcefully" execute via the subchannels in case of a virtual state channel.

For better readability of the protocol descriptions, we write "Mark (id, cid) as corrupt" instead of the instruction "Set $\Gamma_{aux}(id, cid).\mathsf{corrupt} := 1$". Similarly, we write "Mark (id, cid) as registered" instead of the instruction "Set $\Gamma_{aux}(id, cid).\mathsf{registered} := 1$".

B.5 Timing Functions

In the description of the ideal functionality we use two "timing functions": $\mathrm{TimeExeReq}(i, \Delta)$, that represents the maximal number of rounds it takes to inform a party that execution of a contract instance in a two-party ledger channel or virtual state channel with indirect dispute of length $i > 0$ was requested by the other party, and $\mathrm{TimeExe}(i, \Delta)$ that represents the maximal number of rounds it takes to execute of a contract instance in a two-party ledger channel or virtual state channel with indirect dispute of length $i > 0$. Both of these functions were formally defined in [12] as

$$\mathrm{TimeExeReq}(i, \Delta) := 20i + 8i\Delta - 5$$
$$\mathrm{TimeExe}(i, \Delta) := 40i + 16\Delta i + 32\Delta + 67.$$

C State Channel Ideal Functionalities

In this section we formally define the ideal functionality for two-party state channels, $\mathcal{F}_{dch}^{\hat{\mathcal{C}}(\Delta)}(i, \mathcal{C})$, that was informally described in Sec. 4.1. Since this functionality is an extension of the functionality $\mathcal{F}_{ch}^{\hat{\mathcal{C}}(\Delta)}(i, \mathcal{C})$ defined in [12], we recall this functionality here as well. Before we do so, let us make some assumptions that apply to both of these functionalities.

We assume that every state channel functionality formally defined in this section maintains a channel space Γ, were it stores all channels that were created via this functionality. Since messages that the parties send to the ideal functionality do not contain any private information, we implicitly assume that the ideal functionality forwards all messages it receives to the adversary. In addition, the adversary influences the timings; for example, the adversary decides when the parties receive messages from the functionality. Adversary's influence of this kind is implicit in the notation. By saying: "In round $\tau \leq T$ do instruction X", we mean that the adversary can decide when exactly instruction X is performed as long as it is before round T. In case the adversary does not make any choice, the instruction X is performed in round T.

In the UC model, an entity can send only one input to another entity per activation. For example, if a functionality wants to send a message m to party P and party Q, it needs to be activated twice. In the descriptions of our functionalities, we frequently write that a functionality sends m to two or more parties in the same round. Formally, this means that the ideal functionality does not "tick the clock" on behalf of honest parties before all pending messages are sent, hence sufficient amount of activation is enforced.

C.1 Ideal Functionality for Two-party Channels

We now formally define the ideal functionality $\mathcal{F}_{dch}^{\hat{\mathcal{C}}(\Delta)}(i, \mathcal{C})$ that was informally described already in Sec. 4.1. The functionality allows two parties to create maintain and close a two-party state channel up to length i in which contract instances with code from the set $\mathcal{C}$ can be opened. Recall that the functionality is defined exactly as the functionality $\mathcal{F}_{ch}^{\hat{\mathcal{C}}(\Delta)}(i, \mathcal{C})$, which was introduced in [12], in case it receives a message about a ledger state channel or a virtual state channel with indirect dispute. For completeness, we present the functionality $\mathcal{F}_{ch}^{\hat{\mathcal{C}}(\Delta)}(i, \mathcal{C})$ (with slightly modified notation) below as well.

Ideal Functionality $\mathcal{F}_{dch}^{\hat{\mathcal{C}}(\Delta)}(i, \mathcal{C})$

This functionality accepts messages from parties in $\mathcal{P}$. Upon $m \overset{\tau_0}{\longleftarrow} P$, depending on m distinguish the following cases:

– If $m = (\text{create}, \gamma)$, where γ is a virtual state channel with direct dispute of length $\leq i$, execute the "Virtual state channel creation" procedure of $\mathcal{F}_{ch}^{\widehat{\mathcal{L}}(\Delta)}(i, \mathcal{C})$.
– If $m = (\text{update}, id, cid, \tilde{\sigma}, \mathtt{C})$ and $\gamma := \Gamma(id)$ is a virtual state channel with direct dispute of length $\leq i$, then execute the "Contract instance update" procedure of $\mathcal{F}_{ch}^{\widehat{\mathcal{L}}(\Delta)}(i, \mathcal{C})$ as if γ would be a ledger state channel.
– If $m = (\text{execute}, id, cid, f, z)$ and $\gamma := \Gamma(id)$ is a virtual state channel with direct dispute of length $\leq i$, then execute the "Contract instance execute" procedure of $\mathcal{F}_{ch}^{\widehat{\mathcal{L}}(\Delta)}(i, \mathcal{C})$ as if γ would be a ledger state channel.
– If none of the former cases apply, proceed as the functionality $\mathcal{F}_{ch}^{\widehat{\mathcal{L}}(\Delta)}(i, \mathcal{C})$.

Ideal Functionality $\mathcal{F}_{ch}^{\widehat{\mathcal{L}}(\Delta)}(i, \mathcal{C})$

This functionality accepts messages from parties in $\mathcal{P}$. We abbreviate $A := \gamma.\mathsf{Alice}$, $B := \gamma.\mathsf{Bob}$ and $I := \gamma.\mathsf{Ingrid}$.

Ledger state channel creation

Upon $(\text{create}, \gamma) \xleftarrow{\tau_0} A$ where $\gamma.\mathsf{length} = 1$ proceed as follows:

1. Within Δ rounds remove $\gamma.\mathsf{cash}(A)$ coins from A's account on $\widehat{\mathcal{L}}$.
2. If $(\text{create}, \gamma) \xleftarrow{\tau_1 \leq \tau_0 + \Delta} B$, remove within 2Δ rounds $\gamma.\mathsf{cash}(B)$ coins from B's account on $\widehat{\mathcal{L}}$ and then set $\Gamma(\gamma.\mathsf{id}) := \gamma$, send $(\text{created}, \gamma) \hookrightarrow \gamma.\mathsf{end\text{–}users}$ and stop.
3. Otherwise upon $(\text{refund}, \gamma) \xleftarrow{> \tau_0 + 2\Delta} A$, within Δ rounds add $\gamma.\mathsf{cash}(A)$ coins to A's account on $\widehat{\mathcal{L}}$.

Virtual state channel creation

1. Upon $(\text{create}, \gamma) \hookleftarrow P$, where $\gamma.\mathsf{length} \in \{2, \ldots, i\}$ and $P \in \gamma.\mathsf{end\text{–}users} \cup \{I\}$, record the message and distinguish:
   – If $P \in \gamma.\mathsf{end\text{–}users}$ proceed as follows: If you have not yet received (create, γ) from I, then remove $\gamma.\mathsf{cash}(P)$ coins from P's balance in the subchannel $\gamma.\mathsf{subchan}(P)$ and $\gamma.\mathsf{cash}(\gamma.\mathsf{other\text{-}party}(P))$ coins from I's balance in the subchannel $\gamma.\mathsf{subchan}(P)$.
   – If $P = I$, then for both $P \in \gamma.\mathsf{end\text{–}users}$ proceed as follows: If you have not yet received (create, γ) from P then remove $\gamma.\mathsf{cash}(P)$ coins from P's balance in $\gamma.\mathsf{subchan}(P)$, and $\gamma.\mathsf{cash}(\gamma.\mathsf{other\text{-}party}(P))$ coins from I's balance in $\gamma.\mathsf{subchan}(P)$.
2. Distinguish the following two cases:
   – If within 3 rounds you record (create, γ) from all users in $\gamma.\mathsf{end\text{–}users} \cup \{\gamma.\mathsf{Ingrid}\}$, then define $\Gamma(\gamma.\mathsf{id}) := \gamma$, send $(\text{created}, \gamma) \hookrightarrow \gamma.\mathsf{end\text{–}users}$ and wait for channel closing in Step 3 (in the meanwhile accepting the update and execute messages concerning γ).
   – Else wait until round $\gamma.\mathsf{validity}$. Then within $2 \cdot (\text{TimeExeReq}(\lceil j/2 \rceil) + \text{TimeExe}(\lceil j/2 \rceil))$ rounds, where $j := \gamma.\mathsf{length}$, refund the coins that you removed from the subchannels in Step 1 and stop.

Automatic closure of virtual state channel γ when round $\gamma.\mathsf{validity}$ comes:

3. Let $j := \gamma.\mathsf{length}$. If all parties from $\gamma.\mathsf{users}$ are honest, set $T := 15$. Else let $T := 2 \cdot (\text{TimeExe Req}(\lceil j/2 \rceil) + \text{TimeExe}(\lceil j/2 \rceil))$.
4. Within T rounds proceed as follows: Let $\hat{\gamma}$ be the current version of the virtual state channel, i.e. $\hat{\gamma} := \Gamma(\gamma.\mathsf{id})$, and let $\hat{c}_A := \hat{\gamma}.\mathsf{cash}(A)$ and $\hat{c}_B := \hat{\gamma}.\mathsf{cash}(B)$.
5. Add $\hat{c}_A$ coins to A's balance and $\hat{c}_B$ coins to I's balance in $\gamma.\mathsf{subchan}(A)$. Add $\hat{c}_A$ coins to I's balance and $\hat{c}_B$ coins to B's balance in $\gamma.\mathsf{subchan}(B)$. If there exists $cid \in \{0,1\}^*$ such that $\sigma_{cid} := \hat{\gamma}.\mathsf{cspace}(cid).\mathsf{storage} \neq \bot$ and $\hat{c} := \sigma_{cid}.\mathsf{locked} > 0$, then add $\hat{c}$ coins to I's balance in both $\gamma.\mathsf{subchan}(A)$ and $\gamma.\mathsf{subchan}(B)$. Erase $\hat{\gamma}$ from Γ and $(\text{closed}, \gamma.\mathsf{id}) \hookrightarrow \gamma.\mathsf{end\text{–}users}$.

Contract instance update

Upon $(\text{update}, id, cid, \tilde{\sigma}, \mathtt{C}) \xleftarrow{\tau_0} P$, let $\gamma := \Gamma(id)$, $j = \gamma.\mathsf{length}$. If $\mathtt{C} \notin \mathcal{C}$ or $j > i$, then stop. Else proceed as follows:

1. Send $(\text{update–requested}, id, cid, \tilde{\sigma}, \mathtt{C}) \xhookrightarrow{\tau_0+1} \gamma.\mathsf{other–party}(P)$ and set $T := \tau_0 + 1$ in optimistic case when both parties in $\gamma.\mathsf{end–users}$ are honest. Else if $j = 1$, set $T := \tau_0 + 3\Delta + 1$ and if $j > 1$, set $T := \tau_0 + 4 \cdot \text{TimeExeReq}(\lceil j/2 \rceil) + 1$.
2. If $(\text{update–reply}, ok, id, cid) \xhookleftarrow{\tau_1 \leq T} \gamma.\mathsf{other–party}(P)$, then set $\Gamma := \mathtt{UpdateChanSpace}^*(\Gamma, id, cid, \tilde{\sigma}, \mathtt{C})$. Then send $(\text{updated}, id, cid) \xhookrightarrow{\tau_1+1} \gamma.\mathsf{end–users}$ and stop.

Contract instance execution

Upon $(\text{execute}, id, cid, f, z) \xhookleftarrow{\tau_0} P$, let $\gamma := \Gamma(id)$ and $j := \gamma.\mathsf{length}$. If $j > i$, then stop. Else set T_1 and T_2 as:

– In the optimistic case when both parties in $\gamma.\mathsf{end–users}$ are honest, set $T_1 := \tau_0 + 4$ and $T_2 := \tau_0 + 5$.
– In the pessimistic case when at least one party in $\gamma.\mathsf{end–users}$ is corrupt, set $T_1, T_2 := \tau_0 + 4\Delta + 5$ if $j = 1$ and set $T_1 := \tau_0 + 2 \cdot \text{TimeExeReq}(\lceil j/2 \rceil) + 5$, $T_2 := \tau_0 + 4 \cdot \text{TimeExeReq}(\lceil j/2 \rceil) + 5$ if $j > 1$.

1. In round $\tau_1 \leq T_1$, send $(\text{execute–requested}, id, cid, f, z) \xhookrightarrow{\tau_1} \gamma.\mathsf{other–party}(P)$.
2. In round $\tau_2 \leq T_2$, let $\gamma := \Gamma(id)$, $\nu := \gamma.\mathsf{cspace}(cid)$, $\sigma := \nu.\mathsf{storage}$, and $\tau := \tau_0$ if P is honest and else τ is set by the simulator. Compute $(\tilde{\sigma}, \mathsf{add}, m) := f(\sigma, P, \tau, z)$. If $m = \bot$, then stop. Else set $\Gamma := \mathtt{UpdateChanSpace}(\Gamma, id, cid, \tilde{\sigma}, \nu.\mathsf{code}, \mathsf{add})$ and send $(\text{executed}, id, cid, \tilde{\sigma}, \mathsf{add}, m) \xhookrightarrow{\tau_2} \gamma.\mathsf{end–users}$.

Ledger state channel closure

Upon $(\text{close}, id) \xhookleftarrow{\tau_0} P$, let $\gamma = \Gamma(id)$. If $\gamma.\mathsf{length} \neq 1$, then stop. Else wait for at most 7Δ rounds and then distinguish:

– If there exists $cid \in \{0,1\}^*$ such that $\sigma_{cid} := \gamma.\mathsf{cspace}(cid).\mathsf{storage} \neq \bot$ and $\sigma_{cid}.\mathsf{locked} \neq 0$, then stop.
– Otherwise wait up to Δ rounds to add $\gamma.\mathsf{cash}(A)$ coins to A's account and $\gamma.\mathsf{cash}(B)$ coins to B's account on the ledger $\widehat{\mathcal{L}}$. Then set $\Gamma(id) := \bot$, send $(\text{closed}, id) \xhookrightarrow{\tau_2 \leq \tau_0 + 8\Delta} \gamma.\mathsf{end–users}$ and stop.

Let us note that the time complexity of executing a contract instance in a virtual state channel with direct dispute of length i has the same upper bound as the time complexity of executing contract instance in a ledger state channel. Consequently, the timing functions $\text{TimeExeReq}(i, \Delta)$ and $\text{TimeExe}(i, \Delta)$, see Appx. B.5, upper bound the execution time complexity for *any two-party state channel*.

C.2 Simplifying Descriptions of State Channel Functionalities

Let us emphasize that the formal descriptions of all our state channel ideal functionalities $\mathcal{F}_{ch}^{\widehat{\mathcal{L}}(\Delta)}(i, \mathcal{C})$, $\mathcal{F}_{dch}^{\widehat{\mathcal{L}}(\Delta)}(i, \mathcal{C})$ and $\mathcal{F}_{mpch}^{\widehat{\mathcal{L}}(\Delta)}(i, \mathcal{C})$, which were presented in Sec. 4.2, are simplified. In particular, the descriptions exclude many natural checks that one would expect from an ideal state channel functionality $\mathcal{F} \in \{\mathcal{F}_{ch}^{\widehat{\mathcal{L}}(\Delta)}(i, \mathcal{C}), \mathcal{F}_{dch}^{\widehat{\mathcal{L}}(\Delta)}(i, \mathcal{C}), \mathcal{F}_{mpch}^{\widehat{\mathcal{L}}(\Delta)}(i, \mathcal{C})\}$ upon receiving requests from a party P. The intuition behind this is that we do not want a virtual state channel protocol to "work" for malformed or invalid requests. Let us give a few examples of such invalid requests of P that we want $\mathcal{F}$ to refuse.

– P wants to create a ledger channel, but does not have enough coins.
– P wants to create a virtual channel using an intermediary I but there does not exist any state channel between P and I yet.
– P wants to update or execute a contract instance in a channel that was never created, or that P is not participating in, or that was already closed.
– P wants to execute a non-existing function of a contract instance.
– P provides malformed inputs (e.g., missing or unknown parameters).

Since the descriptions of our functionalities are already very complex, we formally define a *wrapper* $\mathcal{W}_{checks}^{\widehat{\mathcal{L}}(\Delta)}(i, \mathcal{C}, \mathcal{F})$ that takes care of all these and other necessary checks. We then let $\mathcal{F}(i, \mathcal{C}) := \mathcal{W}_{checks}^{\widehat{\mathcal{L}}(\Delta)}(i, \mathcal{C}, \mathcal{F}^{\widehat{\mathcal{L}}(\Delta)}(i, \mathcal{C}))$, for $\mathcal{F} \in \{\mathcal{F}_{ch}^{\widehat{\mathcal{L}}(\Delta)} \mathcal{F}_{dch}^{\widehat{\mathcal{L}}(\Delta)}, \mathcal{F}_{mpch}^{\widehat{\mathcal{L}}(\Delta)}\}$ with $\mathcal{W}_{checks}^{\widehat{\mathcal{L}}(\Delta)}$ as defined below.

Authors of [12] define the ideal functionality $\mathcal{F}_{ch}^{\widehat{\mathcal{L}}(\Delta)}(i, \mathcal{C})$ with respect to a set of restricted environments $\mathcal{E}_{res}$ in order to reduce the complexity of the functionality description. It is straightforward to verify that restrictions defining the set $\mathcal{E}_{res}$ translate to checks performed by our wrapper as defined below, i.e, that our wrapped functionalities directly drop all queries that a restricted environment $\mathcal{E}_{res}$ would not be allowed to make. We note that handling invalid requests using a wrapper was already pointed out as an alternative approach in [12].

Wrapper: $\mathcal{W}_{checks}^{\hat{\mathcal{L}}(\Delta)}(i, \mathcal{C}, \mathcal{F})$

The wrapper is defined only if $\mathcal{F} \in \{\mathcal{F}_{ch}^{\hat{\mathcal{L}}(\Delta)}(i, \mathcal{C}), \mathcal{F}_{dch}^{\hat{\mathcal{L}}(\Delta)}(i, \mathcal{C}), \mathcal{F}_{mpch}^{\hat{\mathcal{L}}(\Delta)}(i, \mathcal{C})\}$. Below, we abbreviate $A := \gamma.\mathsf{Alice}$, $B := \gamma.\mathsf{Bob}$ and $I := \gamma.\mathsf{Ingrid}$.

Create

Upon receiving $(\text{create}, \gamma) \xleftarrow{\tau_0} P$ make the following checks: $\mathcal{F}.\Gamma(\gamma.\mathsf{id}) = \bot$ and there is no state channel γ' with $\gamma.\mathsf{id} = \gamma'.\mathsf{id}$ currently being created; γ is a valid state channel according to the definition given in Sec. 3.2; $\gamma.\mathsf{cspace}(cid) = \bot$ for every $cid \in \{0,1\}^*$. Depending on the type of the channel the wrapper additionally checks:

Ledger channel: both parties of the ledger state channel have enough funds on the ledger for the channel creation;[a]

Two-party virtual channel: $j := \gamma.\mathsf{length} \leq i$; if $\gamma.\mathsf{dispute} = \text{indirect}$, then $\gamma.\mathsf{validity} > \tau_0 + 2 + 4 \cdot \text{TimeExeReq}(\lceil j/2 \rceil)$ and if $\gamma.\mathsf{dispute} = \text{direct}$, then $\gamma.\mathsf{validity} > \tau_0 + 2 + 3\Delta$; and the following holds for the subchannels:

- If $P \in \gamma.\mathsf{end\text{–}users}$, then $\alpha := \mathcal{F}.\Gamma(id_P) \neq \bot$ for $id_P := \gamma.\mathsf{subchan}(P)$; $\alpha.\mathsf{end\text{–}users} = \{P, I\}$; $\alpha.\mathsf{length} \leq \lceil j/2 \rceil$; $\alpha.\mathsf{validity} > \gamma.\mathsf{validity} + 2\text{TimeExeReq}(\lceil j/2 \rceil) + 2\text{TimeExe}(\lceil j/2 \rceil)$; if $\alpha.\mathsf{dispute} = \text{indirect}$, then check if $\alpha.\mathsf{cspace}(cid) = \bot$ for every $cid \in \{0,1\}^*$ and if there is no other virtual state channel being created over α; both P and I have enough funds in α.[b]
- If $P = \gamma.\mathsf{Ingrid}$, then $\alpha := \mathcal{F}.\Gamma(id_A) \neq \bot$ for $id_A := \gamma.\mathsf{subchan}(A)$; $\beta := \mathcal{F}.\Gamma(id_B) \neq \bot$ for $id_B := \gamma.\mathsf{subchan}(B)$; $\alpha.\mathsf{end\text{–}users} = \{A, I\}$; $\beta.\mathsf{end\text{–}users} = \{B, I\}$; $j = \alpha.\mathsf{length} + \beta.\mathsf{length}$, $\alpha.\mathsf{length} \leq \lceil j/2 \rceil$ and $\beta.\mathsf{length} \leq \lceil j/2 \rceil$; $\min\{\alpha.\mathsf{validity}, \beta.\mathsf{validity}\} > \gamma.\mathsf{validity} + 2\text{TimeExeReq}(\lceil j/2 \rceil) + 2\text{TimeExe}(\lceil j/2 \rceil)$; if $\alpha.\mathsf{dispute} = \text{indirect}$, then check if $\alpha.\mathsf{cspace}(cid) = \bot$ for every $cid \in \{0,1\}^*$ and if there is no other virtual state channel being created over α; (analogously for β); A and I have enough funds in α and B and I have enough funds in β. [b]

Multi-party virtual channel: $\gamma.\mathsf{dispute} = \text{direct}$, $j := \gamma.\mathsf{length} \leq i$; $\gamma.\mathsf{validity} > \tau_0 + 3\Delta + 3$; for every $Q \in \gamma.\mathsf{neighbors}(P)$: $\alpha := \mathcal{F}.\Gamma(id_Q) \neq \bot$ for $id_Q := \gamma.\mathsf{subchan}(\{P, Q\})$; $\alpha.\mathsf{end\text{–}users} = \{P, Q\}$; $\alpha.\mathsf{length} \leq i$; $\alpha.\mathsf{validity} > \gamma.\mathsf{validity} + 3\Delta + \text{TimeExe}(i) + 2$; if $\alpha.\mathsf{dispute} = \text{indirect}$, then $\alpha.\mathsf{cspace}(cid) = \bot$ for every $cid \in \{0,1\}^*$ and there is no other virtual state channel being opened over α; both P and Q have enough funds in α. [b]

If one of the above checks fail, then drop the message. Otherwise proceed as the functionality $\mathcal{F}$. In addition, if $P = A$, $\gamma.\mathsf{length} = 1$, P is an honest party and $\mathcal{F}$ does not output $(\text{created}, \gamma)$ before round $\tau_0 + 2\Delta$, act as $\mathcal{F}$ upon receiving $(\text{refund}, \gamma) \xleftarrow{\tau_0 + 2\Delta + 1} A$.

Update

Upon receiving $(\text{update}, id, cid, \tilde{\sigma}, \mathtt{C}) \xleftarrow{\tau_0} P$ make the following checks: $\gamma := \mathcal{F}.\Gamma(id) \neq \bot$; $P \in \gamma.\mathsf{users}$; $\tau_0 < \gamma.\mathsf{validity}$; $\mathtt{C} \in \mathcal{C}$; $\tilde{\sigma} \in \mathtt{C}.\Lambda$; $\tilde{\sigma}.\mathsf{locked} = \sum_{Q \in \gamma.\mathsf{users}} \tilde{\sigma}.\mathsf{cash}(Q))$, all parties have enough cash in the state channel for the contract instance update. If $\nu := \gamma.\mathsf{cspace}(cid) \neq \bot$, then the following must hold: $\nu.\mathsf{code} = \mathtt{C}$; $\sigma.\mathsf{user} = \tilde{\sigma}.\mathsf{user}$, where $\sigma := \nu.\mathsf{storage}$; $\sigma.\mathsf{locked} = \sum_{Q \in \gamma.\mathsf{users}} \sigma.\mathsf{cash}(Q)$. If $\gamma.\mathsf{dispute} = \text{indirect}$, then $\gamma.\mathsf{cspace}(cid^*) = \bot$ for every $cid^* \in \{0,1\}$. If one of the above checks fails, then drop the message. Otherwise proceed as the functionality $\mathcal{F}$.

Upon receiving $(\text{update–reply}, ok, id, cid) \xleftarrow{\tau_0} P$ make the following checks: the message is a reply to the message $(\text{update–requested}, id, cid, \tilde{\sigma}, \mathtt{C})$ sent by $\mathcal{F}$ to P (see Appx. 3.1 what we formally mean by "reply"); there is no other update or execution of the contract instance cid in channel $\gamma := \mathcal{F}.\Gamma(id)$ currently going on. In addition, if $\gamma.\mathsf{dispute} = \text{indirect}$, then check is there is no virtual state channel currently being created over γ. If one of the above checks fails, then drop the message. Otherwise proceed as the functionality $\mathcal{F}$.

Execute

Upon receiving $(\text{execute}, id, cid, f, z) \xleftarrow{\tau_0} P$, make the following checks $\gamma := \mathcal{F}.\Gamma(id) \neq \bot$; $P \in \gamma.\mathsf{users}$; $\tau_0 < \gamma.\mathsf{validity}$; $\gamma.\mathsf{cspace}(cid) \neq \bot$, $f \in \gamma.\mathsf{cspace}(cid).\mathsf{code}$. If one of the above checks fails, then drop the message. Otherwise proceed as the functionality $\mathcal{F}$.

Close

Upon $(\text{close}, id) \xleftarrow{\tau_0} P$ make the following checks: $\gamma := \mathcal{F}.\Gamma(id) \neq \bot$; $P \in \gamma.\mathsf{users}$; $\gamma.\mathsf{length} = 1$. If one of the checks fails, then drop the message. Otherwise proceed as the functionality $\mathcal{F}$.

[a] In case more ledger state channels are being created at the same time, all parties have enough funds for all ledger state channels that are being created.

[b] In case more virtual state channels are being created at the same time, both parties have enough funds for all of them.

D The Dispute Board Functionality

As already informally introduced in Sec. 5, our protocol for virtual state chanenls with optional direct dispute makes use of an ideal dispute board. In this section, we formally introduce such a board as the ideal functionality $\mathcal{F}_{DB}(\mathcal{C}_0)$. In a nutshell, $\mathcal{F}_{DB}(\mathcal{C}_0)$ maintains a list of latest valid contract states that parties reached consensus about ($\mathcal{F}_{DB}(\mathcal{C}_0)$ itself serves as a judge for this consensus). For this, the ideal functionality maintains a set $\mathcal{D}$ which we call the *dispute board*, where it stores all registered instances of contracts from a set $\mathcal{C}_0$ which were opened in direct (multi-party) virtual state channels. In particular, elements of dispute board are tuples of the form (id, cid, ν), there id denotes the identifier of a direct virtual channel, cid is an identifier of a contract instance and ν is the registered version of the contract instance. Sometimes we slightly abuse the notation and treat $\mathcal{D}$ as a function which on input $(id, cid, \mathcal{Q})$ outputs ν if the set contains a tuple (id, cid, ν) such that, $\nu.\mathsf{storage.users} = \mathcal{Q}$ and $\bot$ otherwise. In addition, for every $id \in \{0,1\}^*$, we define the set $\mathcal{D}_{id} := \{\nu \mid \exists cid \in \{0,1\}^* \colon \nu = \mathcal{D}(id, cid, \nu.\mathsf{storage.users}) \neq \bot\}$, i.e., the set of contract instances that are currently in dispute in channel id.

In addition to the dispute board $\mathcal{D}$, the functionality maintains an auxiliary set $\mathcal{D}_{aux}$, where it stores all contract instances whose registration was requested but not completed yet. The elements of $\mathcal{D}_{aux}$ are tuples of the form $(T, id, cid, \nu, \tau_0)$, where $T \in \mathcal{P}$ denotes the party who submitted the contract instance version ν in time τ_0. The parameters $id, cid, \mathcal{Q}$ are as for $\mathcal{D}$. When it is convenient, we understand $\mathcal{D}_{aux}$ as a function which on input $(id, cid, \mathcal{Q})$ outputs $(T, id, cid, \nu, \tau_0)$ if such tuple is stored in the auxiliary set (s.t. $\nu.\mathsf{storage.users} = \mathcal{Q}$) and $\bot$ otherwise.

Since messages that the parties send to the dispute board functionality do not contain any private information, we implicitly assume that $\mathcal{F}_{DB}$ forwards all messages it receives to the adversary. Further, the adversary influences the timings when parties receive outputs from $\mathcal{F}_{DB}$ by imposing a delay $\leq \Delta$ on them. This is implicit in the notation $m \xrightarrow{\tau_1 \leq \tau_0 + \Delta} P$. Note that $\mathcal{F}_{DB}$ is not parameterized with $\widehat{\mathcal{L}}(\Delta)$ since it does not open or close any channels, and thus does not need to modify account balances of parties.

Functionality Dispute Board $\mathcal{F}_{DB}(\mathcal{C}_0)$

This functionality accepts messages from parties in $\mathcal{P}$.

Contract instance registration

Upon $(\text{instance–register}, id, cid, \nu) \xleftarrow{\tau_0} P$**,** proceed as follows:

1. If $\mathcal{D}(id, cid, \nu.\mathsf{storage.users}) \neq \bot$, then stop.
2. If $\nu \subseteq \mathcal{P}$, then set $\mathcal{Q} := \nu$, $\nu := \bot$ and goto step 4. Else set $\mathcal{Q} := \nu.\mathsf{storage.users}$ and goto step 3.
3. If $\mathtt{VerifyInstance}(id, cid, \nu) \neq 1$ or $\nu.\mathsf{code} \notin \mathcal{C}_0$, then stop.
4. If $P \notin \mathcal{Q}$, then stop.
5. If $\mathcal{D}_{aux}(id, cid, \mathcal{Q}) = \bot$, then set $\mathcal{D}_{aux}(id, cid, \mathcal{Q}) := (P, id, cid, \nu, \tau_0)$ and send $(\text{instance–registering}, id, cid, \nu, \mathtt{direct}) \xrightarrow{\tau_1 \leq \tau_0 + \Delta} \mathcal{Q}$ and stop. Else let $(Q, id, cid, \widehat{\nu}, \widehat{\tau}_0) := \mathcal{D}_{aux}(id, cid, \mathcal{Q})$.
6. If $\nu = \bot$ or $\mathcal{Q} \neq \widehat{\nu}.\mathsf{storage.users}$, then stop.
7. If $\widehat{\nu} = \bot$ or $\nu.\mathsf{storage.version} > \widehat{\nu}.\mathsf{storage.version}$, then set $\mathcal{D}_{aux}(id, cid, \mathcal{Q}) := (P, id, cid, \nu, \tau_0)$ and send $(\text{instance–registering}, id, cid, \nu, \mathtt{direct}) \xrightarrow{\tau_1 \leq \tau_0 + \Delta} \mathcal{Q}$.

Upon $(\text{finalize–register}, id, cid, \mathcal{Q}) \xleftarrow{\tau_2} P$**,** s.t. $P \in \mathcal{Q}$ proceed as follows:

– If $\mathcal{D}(id, cid, \mathcal{Q}) = \bot$ and $(P, id, cid, \widehat{\nu}, \widehat{\tau}_0) := \mathcal{D}_{aux}(id, cid, \mathcal{Q})$ such that $\tau_2 - \widehat{\tau}_0 \geq 2\Delta$, then set $\tilde{\nu} =$ **"unregisterable"** if $\widehat{\nu} = \bot$ and $\tilde{\nu} := (\widehat{\nu}.\mathsf{storage}, \widehat{\nu}.\mathsf{code})$ otherwise, send (instance–registered, $id, cid, \tilde{\nu}, \mathtt{direct}$) $\xhookrightarrow{\tau_3 \leq \tau_2 + \Delta}$ $\mathcal{Q}$ and set $\mathcal{D}(id, cid, \mathcal{Q}) := \tilde{\nu}$.
– Else ignore this call.

Contract instance execution

Upon (instance–execute, $id, cid, \mathcal{Q}, f, z, \tau$) $\xleftarrow{\tau_0} P$, s.t. $P \in \mathcal{Q}$ set $\nu := \mathcal{D}(id, cid, \mathcal{Q})$ and proceed as follows:
1. If $\nu = \bot$, $\tau_0 - \tau > 6$ or $f \notin \nu.\mathsf{code}$, then stop. Else let $\sigma := \nu.\mathsf{storage}$.
2. Within Δ rounds, send (execute–requested, $id, cid, f, z, \tau, \mathtt{direct}$) $\xhookrightarrow{\tau_1 \leq \tau_0 + \Delta}$ $\sigma.\mathsf{users}$.
3. If $\nu.\mathsf{code} = \mathtt{dVSCC}$ and the identifier cid can be parsed as $T||id^*$ for $T \in \sigma.\mathsf{users}$, set $z := \mathcal{D}_{id^*}$.
4. If $\nu.\mathsf{code} = \mathtt{mpVSCC}$ and the identifier cid can be parsed as $T_L||T_R||id^*$ for $T_L, T_R \in \sigma.\mathsf{users}$, set $z := \mathcal{D}_{id^*}$.
5. Compute $(\tilde{\sigma}, \mathsf{add}, m) := f(\sigma, P, \tau, z)$. If $m = \bot$, then stop. Else set $\mathcal{D}(id, cid, \mathcal{Q}).\mathsf{storage} := \tilde{\sigma}$ and send (instance–executed, $id, cid, \tilde{\sigma}, \mathsf{add}, m, \mathtt{direct}$) $\xhookrightarrow{\tau_1 \leq \tau_0 + \Delta}$ $\sigma.\mathsf{users}$ and stop.

E Additional Material on Adding the Dispute Board

This section contains technical details on our modular approach of building channels with direct dispute. In the entire section, we will explicitly denote a functionality $\mathcal{F}$ having access to the ledger with $\mathcal{F}^{\widehat{\mathcal{L}}(\Delta)}$, to emphasize that, e.g., the dispute board functionality $\mathcal{F}_{DB}$ does *not* have access to it.

E.1 Ideal Functionality Wrappers

Ideally, we would like to make the dispute board functionality $\mathcal{F}_{DB}$ available to parties as a separate hybrid functionality. However, we envision a modular approach where channels of length i are build using an ideal functionality for channels of length $i-1$ (such as $\mathcal{F}_{dch}^{\widehat{\mathcal{L}}(\Delta)}(i-1, \mathcal{C})$). In particular, we will build the channel of length i by opening special contracts in the subchannels. Any contracts that are opened off-chain in the channel of length i might influence the balance in that channel. Thus, if the channel is closed, the contents of the dispute board have to be taken into account to compute the final balances of the parties.

Unfortunately, the UC model does not allow hybrid functionalities to communicate with each other[28]. To circumvent this technicality we merge both $\mathcal{F}_{ch}^{\widehat{\mathcal{L}}(\Delta)}$ and $\mathcal{F}_{DB}$ (or, similarly, $\mathcal{F}_{dch}^{\widehat{\mathcal{L}}(\Delta)}$ and $\mathcal{F}_{DB}$) into a single functionality by putting a wrapper around them which distributes calls. We note that this is just a choice of presentation and we could also write down the resulting functionality in one piece of code.

More detailed, the wrapper allows disputing about contract instances only if they are not opened in any of the subchannels (this is because the subchannels have their own interface for direct dispute via, e.g., $\mathcal{F}_{dch}^{\widehat{\mathcal{L}}(\Delta)}(i-1, \mathcal{C})$). Also, as just already described above, in case a function (i.e., the `Close` function) of some special contract types that allow to open a channel id^* is executed, the contract instances that are in dispute in channel id^*.

Wrapper $\mathcal{W}_{ch}^{\widehat{\mathcal{L}}(\Delta)}(i, \mathcal{C}, \mathcal{C}_0)$

Let $\mathcal{F}_{DB} := \mathcal{F}_{DB}(\mathcal{C}_0)$ and $\mathcal{F} := \mathcal{F}_{ch}^{\widehat{\mathcal{L}}(\Delta)}(i, \mathcal{C})$.
Upon receiving $m \xleftarrow{\tau_0} P$ depending on m proceed as follows:

– If $m \in \{$(instance–register, id, cid, ν), (finalize–register, id, cid), (instance–execute, id, cid, f, z, τ)$\}$
  • If $\mathcal{F}.\Gamma(id) = \bot$, then proceed as the functionality $\mathcal{F}_{DB}$.
  • Else, drop the query.
– If $m =$ (execute, id, cid, f, z) and $\gamma := \mathcal{F}.\Gamma(id) \neq \bot$, set $\nu := \gamma.\mathsf{cspace}(cid)$ and proceed as follows:

[28] We note that there are certain exceptions for this rule, namely when the functionalities are modeled as global [6], or instances of the *same* functionality communicating via some shared setup [5]. Our model does not fit these cases.

1. If $\nu.\mathsf{code} = \mathtt{dVSCC}_{i+1}$ and $cid = T||id^*$ for $T \in \nu.\mathsf{storage.users}$, set $z := \mathcal{F}_{DB}.\mathcal{D}_{id^*}$.
2. If $\nu.\mathsf{code} = \mathtt{mpVSCC}_i$ and $cid = T_L||T_R||id^*$ for $T_L, T_R \in \nu.\mathsf{storage.users}$, set $z := \mathcal{F}_{DB}.\mathcal{D}_{id^*}$.
3. Proceed as the functionality $\mathcal{F}$.

– If none of the former cases applies, proceed as the functionality $\mathcal{F}$.

Analogously, we define a wrapper $\mathcal{W}_{dch}^{\widehat{\mathcal{C}}(\Delta)}(i, \mathcal{C}, \mathcal{C}_0)$ which combines $\mathcal{F}_{dch}^{\widehat{\mathcal{C}}(\Delta)}(i, \mathcal{C})$ and $\mathcal{F}_{DB}(\mathcal{C}_0)$. The following corollary follows directly from Observation 1.

Corollary 1. *For any sets of contract codes $\mathcal{C}$ and $\mathcal{C}_0$ it holds that the ideal functionality $\mathcal{W}_{dch}^{\widehat{\mathcal{C}}(\Delta)}(1, \mathcal{C}, \mathcal{C}_0)$ is equivalent to the ideal functionality $\mathcal{W}_{ch}^{\widehat{\mathcal{C}}(\Delta)}(1, \mathcal{C}, \mathcal{C}_0)$*

In our constructions, we reuse the protocol for building virtual state channels (without direct dispute in non-ledger channels) from [12] which uses a smart contract functionality $\mathcal{F}_{scc}$. This smart contract functionality is, on a high level, similar to $\mathcal{F}_{DB}$. Wrapping this functionality works exactly as wrapping virtual state channels functionalities, with only slight changes of the format of queries and indices of contracts (grey boxes show these differences).

Wrapper $\mathcal{W}_{scc}^{\widehat{\mathcal{C}}(\Delta)}(\mathcal{C}, \mathcal{C}_0)$

Let $\mathcal{F} := \mathcal{F}_{scc}^{\widehat{\mathcal{C}}(\Delta)}(\mathcal{C})$, $\mathcal{F}_{DB} := \mathcal{F}_{DB}(\mathcal{C}_0)$. Upon receiving $m \overset{\tau_0}{\hookleftarrow} P$ depending on m proceed as follows:

– If $m \in \{(\text{instance–register}, id, cid, \nu), (\text{finalize–register}, id, cid), (\text{instance–execute}, id, cid, f, z, \tau)\}$
  • If $\mathcal{F}.\Gamma(id) = \bot$, then proceed as the functionality $\mathcal{F}_{DB}$.
  • Else, drop the query.

– If $m = (\text{instance–execute}, id, cid, f, z, \tau)$ and $\gamma := \mathcal{F}.\Gamma(id) \neq \bot$, set $\nu := \gamma.\mathsf{cspace}(cid)$ and proceed as follows:
  1. If $\nu.\mathsf{code} = \mathtt{dVSCC}_2$ and $cid = T||id^*$ for $T \in \nu.\mathsf{storage.users}$, set $z := \mathcal{F}_{DB}.\mathcal{D}_{id^*}$.
  2. If $\nu.\mathsf{code} = \mathtt{mpVSCC}_1$ and $cid = T_L||T_R||id^*$ for $T_L, T_R \in \nu.\mathsf{storage.users}$, set $z := \mathcal{F}_{DB}.\mathcal{D}_{id^*}$.
  3. Proceed as the functionality $\mathcal{F}$.

– If none of the former cases applies, proceed as the functionality $\mathcal{F}$.

E.2 Protocol Wrapper

We now detail what is necessary to let a virtual state channel protocol realize a wrapped virtual state channel functionality. In an nutshell, the protocol participants need to get instructions on how to handle direct dispute queries that are now possible through the added dispute board. The parties will merely decide whether to forward such queries to $\mathcal{F}_{DB}$ or to drop them, and they will internally keep track of the dispute board to be able to add it to execute queries when necessary. Also, they have to relay all outputs of the dispute board $\mathcal{F}_{DB}$ to the environment.

Protocol Wrapper $\mathcal{W}_{prot}^{\widehat{\mathcal{C}}(\Delta)}(i, \mathcal{C}_0, \pi)$

Let $\mathcal{H}$ be the hybrid ideal functionality of the protocol π. Distinguish the following two cases:

$i = 1$: If $\mathcal{H} \neq \mathcal{F}_{scc}^{\widehat{\mathcal{C}}(\Delta)}(\mathcal{C})$ for some set of contract codes $\mathcal{C}$, then the wrapper is not defined. Otherwise set $j := 2$ and $\mathcal{H}_0 := \mathcal{W}_{scc}^{\widehat{\mathcal{C}}(\Delta)}(\mathcal{C}, \mathcal{C}_0)$.

$i > 1$: If $\mathcal{H} \neq \mathcal{W}_{dch}^{\widehat{\mathcal{C}}(\Delta)}(i-1, \mathcal{C}', \mathcal{C})$ for some sets of contract codes $\mathcal{C}, \mathcal{C}'$, then the wrapper is not defined. Otherwise set $j := i$ and $\mathcal{H}_0 := \mathcal{W}_{dch}^{\widehat{\mathcal{C}}(\Delta)}(i-1, \mathcal{C}', \mathcal{C} \cup \mathcal{C}_0)$.

We assume that every party P maintains a local copy of the dispute board $\mathcal{D}^P$ and has access to the hybrid ideal functionality $\mathcal{H}_0$.

Party P

Upon receiving $m \overset{\tau_1}{\hookleftarrow} \mathcal{Z}$ depending on m proceed as follows:

Direct dispute only for *known* contracts in *unknown* channels:

– If $m = (\text{instance–register}, id, cid, \nu)$, then distinguish the following two cases
  - If P did not already sent $(\text{created}, \gamma) \xhookrightarrow{\tau_0 \leq \tau_1} \mathcal{Z}$ for any γ with $\gamma.id = id$ and $\nu.\mathsf{code} \in \mathcal{C}_0$, then forward the message to $\mathcal{H}_0$.
  - Else, drop the query.
– If $m \in \{(\text{finalize–register}, id, cid), (\text{instance–execute}, id, cid, f, z, \tau)\}$, then distinguish
  - If P did not already sent $(\text{created}, \gamma) \xhookrightarrow{\tau_0 \leq \tau_1} \mathcal{Z}$ for any γ with $\gamma.id = id$, then forward the message to $\mathcal{H}_0$.
  - Else, drop the query.

Take into account dispute board upon closing channels:

– If $m = (\text{execute}, id, cid, f, z)$ and P already sent $(\text{created}, \gamma) \xhookrightarrow{\tau_0 \leq \tau_1} \mathcal{Z}$ for some γ with $\gamma.id = id$, P sets $\nu := \gamma.\mathsf{cspace}(cid)$, initializes $\alpha := \bot$ and proceeds as follows:
  1. If $\nu.\mathsf{code} = \mathtt{dVSCC}_j$ and $cid = T||id^*$ for $T \in \nu.\mathsf{storage.users}$, set $\alpha := id^*$.
  2. If $\nu.\mathsf{code} = \mathtt{mpVSCC}_{j-1}$ and $cid = T_L||T_R||id^*$ for $T_L, T_R \in \nu.\mathsf{storage.users}$, set $\alpha := id^*$.
  3. If $\alpha \neq \bot$, set $z := \mathcal{D}^P_\alpha$.
  4. Proceed as in the protocol $\pi^{\mathcal{H} \to \mathcal{H}_0}$.
– If none of the former cases applies, proceed as in the protocol $\pi^{\mathcal{H} \to \mathcal{H}_0}$.

Relay output of dispute board to $\mathcal{Z}$:

Upon receiving $m \xhookleftarrow{\tau_3} \mathcal{H}_0$, depending on m distinguish

– If $m \in \{(\text{instance–registering}, id, cid, \nu, \mathtt{direct}), (\text{execute–requested}, id, cid, f, z, \tau, \mathtt{direct})\}$, then output $m \xhookrightarrow{\tau_3} \mathcal{Z}$.
– If $m = (\text{instance–registered}, id, cid, \tilde{\nu}, \mathtt{direct})$, then set $\mathcal{D}^P(id, cid, \mathcal{Q}) := \tilde{\nu}$, where $\mathcal{Q} := \tilde{\nu}.\mathsf{storage.users}$, and output $m \xhookrightarrow{\tau_3} \mathcal{Z}$.
– If $m = (\text{instance–executed}, id, cid, \tilde{\sigma}, \mathsf{add}, m^*, \mathtt{direct})$, then set $\mathcal{D}^P(id, cid, \mathcal{Q}).\mathsf{storage} := \tilde{\sigma}$, where $\mathcal{Q} := \tilde{\nu}.\mathsf{storage.users}$, and output $m \xhookrightarrow{\tau_3} \mathcal{Z}$.
– If none of the former cases applies, proceed as in the protocol $\pi^{\mathcal{H} \to \mathcal{H}_0}$.

F Proofs of Lemmas

F.1 Proof of Lemma 1

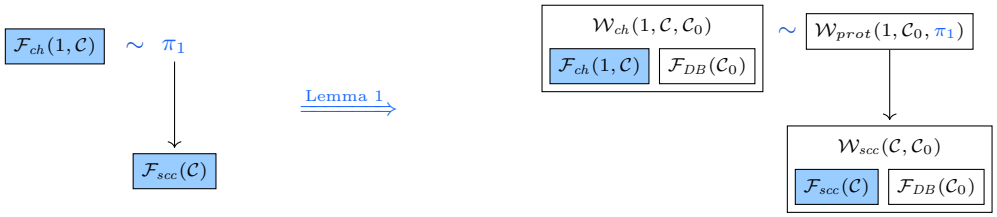

Fig. 17. Pictorial representation of Lemma 1.

Proof. Let Sim_Π be the simulator of Π which exists by the prerequisite of the lemma. Throughout the proof, we will refer to queries instance–register, finalize–register and instance–execute as *dispute queries*.

We exploit the fact that every time $\mathcal{Z}$ is active, all entities in the system are aware of the same channels. This is because $\mathcal{F}_{DB}$ does not let parties create or close any channels, and parties and $\mathcal{F}_{ch}$ having created different channels would contradict the UC security of Π. Let us elaborate on this. From the UC-security of Π, it follows that real and ideal parties send their confirmation messages of channel creation (as well as closure) at the same time. Here, by time we mean at the same point in the order of execution). This is clear since, if this would not be the case, then $\mathcal{Z}$ could easily distinguish by observing at which time confirmation messages such as $(\text{created}, \gamma)$ of the parties arrive. We stress that this is independent of the corruption status. Further, in the ideal world $\mathcal{F}_{ch}$ updates the channel spaces of the parties in the same activation as it outputs the confirmation message.

Since the wrappers all equally decide to only call $\mathcal{F}_{DB}(\mathcal{C}_0)$ if $\Gamma(id) = \bot$ or, equivalently, they never confirmed creation of a channel with this id, either all entities call $\mathcal{F}_{DB}(\mathcal{C}_0)$ or none. In the first case, the simulator's task is to play the real-world adversary communicating with $\mathcal{F}_{DB}(\mathcal{C}_0)$. We will give more details on the simulation below. In the second case, $\mathcal{F}_{DB}(\mathcal{C}_0)$ is not involved in the ideal execution apart from informing $\mathcal{F}_{ch}$ about contracts that are in dispute in either virtual or multi-party channels. We will re-use Sim_{Π} and adjust it in case of execute messages from $\mathcal{Z}$, in which case Sim_{Π} is adjusted to take into account the contents of the (correctly simulated) dispute board.

More formally, the simulator Sim will internally run a copy of the hybrid world, which comprises parties executing the wrapped protocol and a hybrid ideal functionality $\mathcal{W}_{scc}(\mathcal{C})$. Additionally, Sim talks to the ideal functionality $\mathcal{W}_{ch}(1, \mathcal{C}, \mathcal{C}_0)$. Recall that there are no private inputs and thus Sim learns all inputs that $\mathcal{W}_{ch}$ receives from honest parties (we assume that $\mathcal{W}_{ch}$ adds a prefix indicating which party the message was received from). Inputs of corrupted parties Sim gets anyway directly from $\mathcal{Z}$. Thus, Sim can internally run a "perfect" copy of $\mathcal{F}_{DB}(\mathcal{C}_0)$, of which we denote by $\mathcal{D}^{\mathsf{Sim}}$ the dispute board. Perfect here means that, throughout the simulation, it will hold that $\mathcal{D}^{\mathsf{Sim}} = \mathcal{D}$, where $\mathcal{D}$ is maintained by $\mathcal{F}_{DB}(\mathcal{C}_0)$ in the real execution. Sim behaves as $\mathsf{Sim}_{\Pi}^{\mathcal{F}_{ch}(1,\mathcal{C}) \to \mathcal{W}_{ch}(1,\mathcal{C},\mathcal{C}_0)}$ in the case of create and update queries. Regarding queries from the ideal functionality $\mathcal{W}_{ch}(1, \mathcal{C}, \mathcal{C}_0)$, Sim either acts as Sim_{Π} or, if Sim_{Π} does not know the query (which means that it came from $\mathcal{F}_{DB}$) and it is not an execute query, Sim forwards the query to the simulated $\mathcal{F}_{DB}$. We now detail how execute queries are treated.

Execute a contract instance in a ledger channel. We will re-use the simulator Sim_{Π} for executing a contract instance. Note that the only difference between $\mathcal{W}_{prot}$ and Π, respectively $\mathcal{W}_{ch}$ and $\mathcal{F}_{ch}$, regarding this query arises for closing virtual and multi-party channel contracts ($\mathtt{dVSCC}_2$ and $\mathtt{mpVSCC}_1$). In the wrapped scenario of Lemma 1, all latest contract states that are in dispute in these channels and that have to be taken into account when closing virtual or multi-party channels are available on the dispute board, which Sim is informed about and maintains in $\mathcal{D}^{\mathsf{Sim}}$. Sim now updated execute queries before handing them over to Sim_{Π}. Formally, Sim behaves as follows.

Sim for Lemma 1.

Sim handles queries as $\mathsf{Sim}_{\Pi}^{\mathcal{F}_{ch}(1,\mathcal{C}) \to \mathcal{W}_{ch}(1,\mathcal{C},\mathcal{C}_0)}$ except for the following:

- Upon $(P, \text{execute}, id, cid, f, z) \overset{\tau_0}{\hookleftarrow} \mathcal{W}_{ch}(1, \mathcal{C}, \mathcal{C}_0)$, if P already sent $(\text{created}, \gamma) \overset{\tau \leq \tau_0}{\hookrightarrow} \mathcal{Z}$ for some γ with $\gamma.id = id$, set $\nu := \gamma.\mathsf{cspace}(cid)$, $\alpha := \bot$ and proceed as follows:
  1. If $\nu.\mathsf{code} = \mathtt{dVSCC}_2$ and $cid = T||id^*$ for $T \in \nu.\mathsf{storage.users}$, set $z^* = \mathcal{D}^{\mathsf{Sim}}_{id^*}$.
  2. If $\nu.\mathsf{code} = \mathtt{mpVSCC}_1$ and $cid = T_L||T_R||id^*$ for $T_L, T_R \in \nu.\mathsf{storage.users}$, set $z^* = \mathcal{D}^{\mathsf{Sim}}_{id^*}$.

  Proceed as $\mathsf{Sim}_{\Pi}^{\mathcal{F}_{ch}(1,\mathcal{C}) \to \mathcal{W}_{ch}(1,\mathcal{C},\mathcal{C}_0)}$ upon receiving $(P, \text{execute}, id, cid, f, z^*) \overset{\tau_0}{\hookleftarrow} \mathcal{W}_{ch}(1, \mathcal{C}, \mathcal{C}_0)$.

Let us now elaborate on indistinguishability of direct dispute queries for contracts in non-ledger channels. Considering the code of $\mathcal{F}_{DB}(\mathcal{C}_0)$, it is clear that the only adversarial influence on the ideal functionality for direct dispute is determining the rounds in which parties receive their outputs. However, since $\mathcal{F}_{DB}(\mathcal{C}_0)$ exists also in the real world and we assume the real-world adversary to be the dummy adversary, $\mathcal{Z}$ is in fact determining the delays and sends them to Sim. This means that Sim can perfectly simulate by just relaying the delays to $\mathcal{F}_{DB}(\mathcal{C}_0)$.

F.2 Proof of Lemma 2

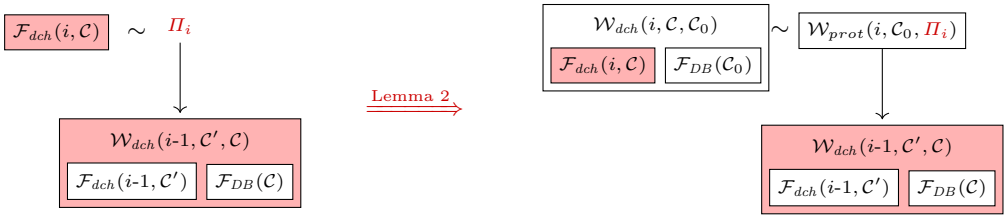

Fig. 18. Pictorial representation of Lemma 2.

Proof. Our simulation strategy will be similar to Lemma 1. The simulator Sim will internally run a copy of the hybrid world, which comprises parties executing the wrapped protocol Π_i and a hybrid ideal functionality $\mathcal{W}_{dch}^{\hat{\mathcal{L}}(\Delta)}(i-1,\mathcal{C}',\mathcal{C})$. Additionally, Sim is connected to the ideal functionality $\mathcal{W}_{dch}^{\hat{\mathcal{L}}(\Delta)}(i,\mathcal{C},\mathcal{C}_0)$.

Again, UC-security of Π_i yields that, from the point of view of $\mathcal{Z}$, all entities in the system are aware of the same set of channels. Further, the wrappers are designed to send queries to $\mathcal{F}_{DB}$ according to the openend channels, and thus either all entities forward the query to $\mathcal{F}_{DB}$ or none. In this case, we describe how the simulator handles queries that might result in $\mathcal{F}_{DB}$ acting differently in both worlds due to their mismatching contract set parameters. However, note that since the protocol wrapper requires an honest party to drop all dispute queries for contracts that are not in $\mathcal{C}_0$, we have to consider only dispute queries for contracts that are in $\mathcal{C} \setminus \mathcal{C}_0$ asked through corrupted parties.

If a query is not forwarded to $\mathcal{F}_{DB}$, the only difference to the unwrapped execution of Π_i is that $\mathcal{W}_{dch}^{\hat{\mathcal{L}}(\Delta)}(i,\mathcal{C},\mathcal{C}_0)$ will equip certain execute queries with infos from the dispute board. Sim will thus run Sim_{Π_i}'s code for simulating execute queries but first add the infos from the simulated dispute board. It follows from the code of Sim below that real (i.e., hybrid) and simulated $\mathcal{F}_{DB}(\mathcal{C})$ get the same queries and thus maintain the same dispute board, which renders execute queries in the ideal and the real world indistinguishable.

More formally, Sim behaves as Sim_{Π_i} in the case of create or update queries. For the remaining queries, i.e. execute and the three dispute queries, we now give a detailed description of Sim.

Sim for Lemma 2.

Sim handles all queries as Sim_{Π_i} except for the following:

- If a corrupted party P sends $(\text{instance–register}, id, cid, \nu) \xleftarrow{\tau_0} P$ or $(\text{finalize–register}, id, cid, \mathcal{Q}) \xleftarrow{\tau_0} P$ or $(\text{instance–execute}, id, cid, \mathcal{Q}, f, z, \tau) \xleftarrow{\tau_0} P$, if $cid \in \mathcal{C} \setminus \mathcal{C}_0$, then Sim does not input anything into $\mathcal{W}_{dch}^{\hat{\mathcal{L}}(\Delta)}(i,\mathcal{C},\mathcal{C}_0)$ on behalf of the corrupted P. Instead, Sim forwards the query on behalf of the simulated P to the simulated $\mathcal{W}_{dch}^{\hat{\mathcal{L}}(\Delta)}(i-1,\mathcal{C}',\mathcal{C})$. Whatever the simulated P outputs, Sim sends to $\mathcal{Z}$ as claimed output of the corrupted P.
- Upon $(P, \text{execute}, id, cid, f, z) \xleftarrow{\tau_0} \mathcal{W}_{dch}^{\hat{\mathcal{L}}(\Delta)}(i,\mathcal{C},\mathcal{C}_0)$, if P already sent $(\text{created}, \gamma) \xrightarrow{\tau \leq \tau_0} \mathcal{Z}$ for some γ with $\gamma.id = id$, set $\nu := \gamma.\mathsf{cspace}(cid)$, $\alpha := \bot$ and proceed as follows:
  1. If $\nu.\mathsf{code} = \mathtt{dVSCC}_i$ and $cid = T||id^*$ for $T \in \nu.\mathsf{storage.users}$, set $z^* = \mathcal{D}_{id^*}^{\mathsf{Sim}}$.
  2. If $\nu.\mathsf{code} = \mathtt{mpVSCC}_{i+1}$ and $cid = T_L||T_R||id^*$ for $T_L, T_R \in \nu.\mathsf{storage.users}$, set $z^* = \mathcal{D}_{id^*}^{\mathsf{Sim}}$.

  Proceed as $\mathsf{Sim}_{\Pi_i}{}^{\mathcal{F}_{ch} \to \mathcal{W}_{dch}^{\hat{\mathcal{L}}(\Delta)}(i,\mathcal{C},\mathcal{C}_0)}$ upon receiving $(P, \text{execute}, id, cid, f, z^*) \xleftarrow{\tau_0} \mathcal{W}_{dch}^{\hat{\mathcal{L}}(\Delta)}(i,\mathcal{C},\mathcal{C}_0)$.

G Additional material on the Protocol for Virtual State Channels

In this section, we provide the full description of the protocol $\Pi_{dch}(i,\mathcal{C}_0)$ for two-party virtual channels that was already described informally in Sec. 6. Recall that we distinguish thee cases: (1) party receives a message about a state channel of length $j < i$, (2) If a party receives a message about a virtual state channel without direct dispute and of length exactly i and (3) party receives a message about a virtual state channel γ with direct dispute and of length exactly i.

Shorter channels Case 1 is rather straightforward. If a party P receives a message about creating/closing a state channel of length $j < i$, then the party immediately forwards the message to the hybrid ideal functionality $\mathcal{W}_{dch}^{\hat{\mathcal{L}}(\Delta)}(i-1,\mathcal{C}_1,\mathcal{C}_0)$. Analogously, if the hybrid ideal functionality outputs a message that the state channel was created/closed, then the party simply forwards this message to the environment. In case a party P receives a message m which is of the form $(\text{update}, id, cid, \sigma, \mathtt{C})$, $(\text{update–reply}, ok, id, cid)$ or $(\text{execute}, id, cid, f, z)$ and id is an identifier of a state channel of length $j < i$, then the party P sightly modifies the message m before sending it to the hybrid ideal functionality. In particular, party P extends the contract instance identifier cid by adding a prefix "short". For example, if $m = (\text{execute}, id, cid, f, z)$, then the party sends the message $m' = (\text{execute}, id, \text{short}||cid, f, z)$ to $\mathcal{W}_{dch}^{\hat{\mathcal{L}}(\Delta)}(i-1,\mathcal{C}_1,\mathcal{C}_0)$. Analogously, if a party P receives a message from the hybrid ideal functionality that the contract instance with identifier $\text{short}||cid$ was updated/executed, the party removes the prefix "short" before forwarding it to the environment.

The reason for adding/removing this prefix is to prevent the environment from instructing honest parties to update/execute a contract instance with identifier $P||id$, where $P \in \mathcal{P}$ and id is an identifier of a virtual state channel of length i. Contract instance identifiers of this form are used for contract instances with the code $\texttt{dVSCC}_i$ that are created in the subchannels of a virtual state channel of length i and hence should be updated/executed only when the virtual state channel is being created or closed.

Channels with indirect dispute In Case 2, i.e. when the party receives a message about a virtual state channel with indirect dispute of length i, the the party follows the protocol π. This means, that it behaves exactly as in the protocol π except that calls to the hybrid ideal functionality $\mathcal{F}_{ch}(i-1, \mathcal{C}'_0)$ are replaced by calls to the hybrid functionality $\mathcal{W}_{dch}^{\widehat{\mathcal{L}}(\Delta)}(i-1, \mathcal{C}_1, \mathcal{C}_0)$. Analogously, messages received from $\mathcal{W}_{dch}^{\widehat{\mathcal{L}}(\Delta)}(i-1, \mathcal{C}_1, \mathcal{C}_0)$ are interpreted as messages from $\mathcal{F}_{ch}(i-1, \mathcal{C}'_0)$.

Channels with direct dispute This case was discussed in detail already in Sec. 6.2. We provide the formal description of each protocol part below.

Protocol $\Pi_{dch}(i, \mathcal{C}_0, \pi)$: Create a virtual state channel with direct dispute

Let us abbreviate $\mathcal{H} := \mathcal{W}_{dch}^{\widehat{\mathcal{L}}(\Delta)}(i-1, \mathcal{C}_1, \mathcal{C}_0)$.

Party $P \in \gamma.\mathsf{end\text{–}users}$ upon $(\text{create}, \gamma) \xleftarrow{\tau_0} \mathcal{Z}$:

1. Let $cid_P := P||\gamma.\mathsf{id}$ and $id_P := \gamma.\mathsf{subchan}(P)$.
2. Compute $\tilde{\sigma}_P := \texttt{Init}_i^{\mathtt{d}}(P, \tau_0, \gamma)$ and send $(\text{update}, id_P, cid_P, \tilde{\sigma}_P, \texttt{dVSCC}_i) \xhookrightarrow{\tau_0} \mathcal{H}$.

Party I upon $(\text{create}, \gamma) \xleftarrow{\tau_0} \mathcal{Z}$:

3. Let $id_A := \gamma.\mathsf{subchan}(\gamma.\mathsf{Alice})$, $id_B := \gamma.\mathsf{subchan}(\gamma.\mathsf{Bob})$, $cid_A := \gamma.\mathsf{Alice}||\gamma.\mathsf{id}$ and $cid_B := \gamma.\mathsf{Bob}||\gamma.\mathsf{id}$.
4. Compute $\tilde{\sigma}_A := \texttt{Init}_i^{\mathtt{d}}(\gamma.\mathsf{Alice}, \tau_0, \gamma)$ and $\tilde{\sigma}_B := \texttt{Init}_i^{\mathtt{d}}(\gamma.\mathsf{Bob}, \tau_0, \gamma)$.
5. If both messages $(\text{update–requested}, id_A, cid_A, \tilde{\sigma}_A, \mathtt{C}) \xleftarrow{\tau_0+1} \mathcal{H}$ and $(\text{update–requested}, id_B, cid_B, \tilde{\sigma}_B, \mathtt{C}) \xleftarrow{\tau_0+1} \mathcal{H}$ are received, then set $\Gamma^I(\gamma.\mathsf{id}) := \gamma$ and send $(\text{update–reply}, ok, id_A, cid_A) \xhookrightarrow{\tau_0+1} \mathcal{H}$ and $(\text{update–reply}, ok, id_B, cid_B) \xhookrightarrow{\tau_0+1} \mathcal{H}$ and wait until time $\gamma.\mathsf{validity}$. Else stop.

Back to $P \in \gamma.\mathsf{end\text{–}users}$

6. If $(\text{updated}, id_P, cid_P) \xleftarrow{\tau_0+2} \mathcal{H}$, then send $(\text{create–ok}, \gamma) \xhookrightarrow{\tau_0+2} \gamma.\mathsf{other\text{–}party}(P)$. If $(\text{create–ok}, \gamma) \xleftarrow{\tau_0+3} \gamma.\mathsf{other\text{–}party}(P)$, then set $\Gamma^P(\gamma.\mathsf{id}) := \gamma$ and output $(\text{created}, \gamma) \xhookrightarrow{\tau_0+3} \mathcal{Z}$.
7. Wait until time $\gamma.\mathsf{validity}$.

Constructor $\texttt{Init}_i^{\mathtt{d}}(P, \tau, \gamma)$

If $P \notin \gamma.\mathsf{end\text{–}users}$ or $\gamma.\mathsf{cash}(\gamma.\mathsf{Alice}) < 0$ or $\gamma.\mathsf{cash}(\gamma.\mathsf{Bob}) < 0$ or $\gamma.\mathsf{cspace}(cid) \neq \bot$ for some $cid \in \{0,1\}^*$ or $\tau \leq \gamma.\mathsf{validity} + 3\Delta + 2$, then output $\bot$. Else output the attribute tuple σ defined as follows:

$$\sigma.\mathsf{users} := \begin{cases} (\gamma.\mathsf{Alice}, \gamma.\mathsf{Ingrid}), & \text{if } P = \gamma.\mathsf{Alice}, \\ (\gamma.\mathsf{Ingrid}, \gamma.\mathsf{Bob}), & \text{if } P = \gamma.\mathsf{Bob}, \end{cases}$$

$$\sigma.\mathsf{locked} := \gamma.\mathsf{cash}(\gamma.\mathsf{Alice}) + \gamma.\mathsf{cash}(\gamma.\mathsf{Bob}),$$

$$(\sigma.\mathsf{cash}(\sigma.\mathsf{users}(1)), \sigma.\mathsf{cash}(\sigma.\mathsf{users}(2))) := (\gamma.\mathsf{cash}(\gamma.\mathsf{Alice}), \gamma.\mathsf{cash}(\gamma.\mathsf{Bob})),$$

$$\sigma.\mathsf{virtual\text{–}channel} := \gamma.$$

Protocol $\Pi_{dch}(i, \mathcal{C}_0, \pi)$: procedure $\texttt{RegisterDirect}(P, id, cid)$

Let us abbreviate $\mathcal{H} := \mathcal{W}_{dch}^{\widehat{\mathcal{L}}(\Delta)}(i-1, \mathcal{C}_1, \mathcal{C}_0)$ and let τ_0 be the staring round of the procedure.

Party P

1. Let $\gamma^P := \Gamma^P(id)$ and $Q := \gamma.\mathsf{other\text{–}party}(P)$. If $\gamma^P.\mathsf{cspace}(cid) = \bot$, then set $\nu^P := \{P, Q\}$. Else set $\nu^P := \gamma^P.\mathsf{cspace}(cid)$.
2. Send $(\text{instance–register}, id, cid, \nu^P) \overset{\tau_0}{\hookrightarrow} \mathcal{H}$ and go to Step 4.

Party Q

3. Upon $(\text{instance–registering}, id, cid, \nu^P, \texttt{direct}) \overset{\tau_1}{\hookleftarrow} \mathcal{H}$, let $\gamma^Q := \Gamma^Q(id)$ and $\nu^Q := \gamma^Q.\mathsf{cspace}(cid)$. If $\nu^Q \neq \bot$, then send $(\text{instance–register}, id, cid, \nu^Q) \overset{\tau_1}{\hookrightarrow} \mathcal{H}$ and goto Step 5.

Back to party P

4. In round $\tau_2 := \tau_0 + 2\Delta$, send $(\text{finalize–register}, id, cid, \gamma.\mathsf{end\text{–}users}) \overset{\tau_2}{\hookrightarrow} \mathcal{H}$.

End for both $T \in \gamma.\mathsf{end\text{–}users}$

5. Upon $(\text{instance–registered}, id, cid, \tilde{\nu}, \texttt{direct}) \hookleftarrow \mathcal{H}$, mark (id, cid) as registered and set $\Gamma^T := \texttt{Update ChanSpace}^*(\Gamma^T, id, cid, \tilde{\nu})$.

Protocol $\Pi_{dch}(i, \mathcal{C}_0, \pi)$: Contract instance update

Party P upon $(\text{update}, id, cid, \tilde{\sigma}, \texttt{C}) \overset{\tau_0}{\hookleftarrow} \mathcal{Z}$

1. If (id, cid) is marked as corrupt, then stop. Else let $\gamma^P := \Gamma^P(id)$ and $\nu^P := \gamma^P.\mathsf{cspace}(cid)$. If $\nu^P = \bot$, then set $w^P := 1$, else let $w^P := \Gamma^P_{aux}(id, cid).\mathsf{next\text{-}version}$.
2. Compute $s_P := \mathsf{Sign}_{sk_P}(id, cid, \tilde{\sigma}, \texttt{C}, w^P)$.
3. Send $(\text{update}, s_P, id, cid, \tilde{\sigma}, \texttt{C}) \overset{\tau_0}{\hookrightarrow} Q$ and go to Step 6.

Party Q upon $(\text{update}, s_P, id, cid, \tilde{\sigma}, \texttt{C}) \overset{\tau_1}{\hookleftarrow} P$

4. If (id, cid) is marked as corrupt, then stop. Let $\gamma^Q := \Gamma^Q(id)$ and $\nu^Q := \gamma^Q.\mathsf{cspace}(cid)$. If $\nu^Q = \bot$, then set $w^Q := 1$, else set $w^Q := \Gamma^Q_{aux}(id, cid).\mathsf{next\text{-}version}$.
5. If $\mathsf{Vrfy}_{pk_P}(id, cid, \tilde{\sigma}, \texttt{C}, w^Q; s_P) \neq 1$ or $\texttt{C} \notin \mathcal{C}_0$, then mark (id, cid) as corrupt and stop. Else output $(\text{update–requested}, id, cid) \overset{\tau_1}{\hookrightarrow} \mathcal{Z}$ and consider the following two cases
   - If $(\text{update–reply}, ok, id, cid) \overset{\tau_1}{\hookleftarrow} \mathcal{Z}$, then compute the signature $s_Q := \mathsf{Sign}_{sk_Q}(id, cid, \tilde{\sigma}, \texttt{C}, w^Q)$, send $(\text{update–ok}, s_Q) \overset{\tau_1}{\hookrightarrow} P$, set $\Gamma^Q := \texttt{UpdateChanSpace}^*(\Gamma^Q, id, cid, \tilde{\sigma}, \texttt{C}, w^Q, \{s_P, s_Q\})$, set $\Gamma^Q_{aux}(id, cid).\mathsf{next\text{-}version} := w^Q + 1$, output $(\text{updated}, id, cid) \overset{\tau_1+1}{\hookrightarrow} \mathcal{Z}$ and stop.
   - Else compute $s_Q := \mathsf{Sign}_{sk_Q}(id, cid, \nu^Q.\mathsf{storage}, \nu^Q.\mathsf{code}, w^Q + 1)$, send $(\text{update–not–ok}, s_Q) \overset{\tau_1}{\hookrightarrow} P$ and set $\Gamma^Q_{aux}.\mathsf{next\text{-}version}(id, cid) := w^Q + 2$ and stop.

Back to party P

6. Distinguish the following three cases:
   - If $(\text{update–ok}, s_Q) \overset{\tau_2=\tau_0+2}{\longleftarrow} Q$, where $\mathsf{Vrfy}_{pk_Q}(id, cid, \tilde{\sigma}, \texttt{C}, w^P; s_Q) = 1$, then set $\Gamma^P := \texttt{Update ChanSpace}^*(\Gamma^P, id, cid, \tilde{\sigma}, \texttt{C}, w^P, \{s_P, s_Q\})$, update $\Gamma^P_{aux}(id, cid).\mathsf{next\text{-}version} := w^P + 1$, output $(\text{updated}, id, cid) \overset{\tau_2}{\hookrightarrow} \mathcal{Z}$ and stop.
   - If $(\text{update–not–ok}, s_Q) \overset{\tau_2=\tau_0+2}{\longleftarrow} Q$, where $\mathsf{Vrfy}_{pk_Q}(id, cid, \nu^P.\mathsf{storage}, \nu^P.\mathsf{code}, w^P+1; s_Q) = 1$, then compute $s_P := \mathsf{Sign}_{sk_P}(id, cid, \nu^P.\mathsf{storage}, \nu^P.\mathsf{code}, w^P + 1)$, set $\Gamma^P := \texttt{UpdateChanSpace}^*(\Gamma^P, id, cid, \nu^P.\mathsf{storage}, \nu^P.\mathsf{code}, w^P + 1, \{s_P, s_Q\})$, set $\Gamma^P_{aux}(id, cid).\mathsf{next\text{-}version} := w^P + 2$ and stop.

– If none of the above cases applies, mark (id, cid) as corrupt and in round τ_0+2 call the subprocedure $\texttt{RegisterDirect}(P, id, cid)$. After the subprocedure execution (in round $\tau_3 \leq \tau_0 + 3\Delta + 2$), if $\Gamma^P(id).\mathsf{cspace}(cid) = (\tilde{\sigma}, \mathtt{C})$, then output $(\text{updated}, id, cid) \overset{\tau_3}{\hookrightarrow} \mathcal{Z}$.

Protocol $\Pi_{dch}(i, \mathcal{C}_0, \pi)$: Contract instance execution

Let us abbreviate $\mathcal{H} := \mathcal{W}_{dch}^{\widehat{\mathcal{C}}(\Delta)}(i-1, \mathcal{C}_1, \mathcal{C}_0)$.

Party P upon $(\text{execute}, id, cid, f, z) \overset{\tau_0}{\hookleftarrow} \mathcal{Z}$

1. Let $\gamma^P := \Gamma^P(id), \nu^P := \gamma^P.\mathsf{cspace}(cid)$, $\sigma^P := \nu^P.\mathsf{storage}$, $\mathtt{C}^P := \nu^P.\mathsf{code}$ and set $w^P := \Gamma^P_{aux}(id, cid).\mathsf{next\text{-}version}$.
2. Set $\tau_1 := \tau_0 + x$, where x is the smallest offset such that $\tau_1 = 1 \mod 4$ if $P = \gamma^P.\mathsf{Alice}$ and $\tau_1 = 3 \mod 4$ if $P = \gamma^P.\mathsf{Bob}$.
3. For round $\tau \in [\tau_0, \tau_1]$ proceed as follows: If (id, cid) is marked as corrupt, goto Step 5.
4. In round τ_1, compute $(\tilde{\sigma}, \mathsf{add}, m) := f(\sigma^P, P, \tau_0, z)$. If $m = \bot$, then stop. Otherwise compute $s_P := \mathsf{Sign}_{sk_P}(id, cid, \tilde{\sigma}, \mathtt{C}^P, w^P)$, send $(\text{peaceful–request}, id, cid, f, z, s_P, \tau_0) \overset{\tau_1}{\hookrightarrow} Q$ and goto Step 12.
5. If (id, cid) is marked as corrupt but not registered, then execute $\texttt{RegisterDirect}(P, id, cid)$.
6. Goto Step 13.

Party Q upon $(\text{peaceful–request}, id, cid, f, z, s_P, \tau_0) \overset{\tau_Q}{\hookleftarrow} P$

7. Let $\gamma^Q := \Gamma^Q(id), \nu^Q := \gamma^Q.\mathsf{cspace}(cid)$, $\sigma^Q := \nu^Q.\mathsf{storage}$, $\mathtt{C}^Q := \nu^Q.\mathsf{code}$, $w^Q := \Gamma^Q_{aux}(id, cid).\mathsf{next\text{-}version}$. If $\gamma^Q = \bot$, $P, Q \notin \gamma^Q.\mathsf{end\text{–}users}$, $\nu^Q = \bot$ or $f \notin \mathtt{C}^Q$, then goto step 11.
8. If $P = \gamma^Q.\mathsf{Alice}$ and $\tau_Q \mod 4 \neq 2$ or if $P = \gamma.\mathsf{Bob}$ and $\tau_Q \mod 4 \neq 0$, then goto step 11.
9. If $\tau_0 \notin [\tau_Q - 4, \tau_Q - 1]$, then goto step 11.
10. If (id, cid) is not marked as corrupt in Γ^Q_{aux}, do:
    (a) Compute $(\tilde{\sigma}, \mathsf{add}, m) := f(\sigma^Q, P, \tau_0, z)$.
    (b) If $m = \bot$ or $\mathsf{Vrfy}_{pk_P}(id, cid, \tilde{\sigma}, \mathtt{C}^Q, w^Q; s_P) \neq 1$, then goto step 11. Else proceed.
    (c) Output $(\text{execute–requested}, id, cid, f, z, \tau_0) \overset{\tau_Q}{\hookrightarrow} \mathcal{Z}$.
    (d) Sign $s_Q := \mathsf{Sign}_{sk_Q}(id, cid, \tilde{\sigma}, \mathtt{C}^Q, w^Q)$, send $(\text{peaceful–confirm}, id, cid, f, z, s_Q) \overset{\tau_Q}{\hookrightarrow} P$, set $\Gamma^Q := \texttt{UpdateChanSpace}(\Gamma^Q, id, cid, \tilde{\sigma}, \mathtt{C}^Q, \mathsf{add}, w^Q, \{s_P, s_Q\})$, $\Gamma^Q_{aux}(id, cid).\mathsf{next\text{-}version} := w^Q + 1$.
    (e) Output $(\text{executed}, id, cid, \tilde{\sigma}, \mathsf{add}(\gamma.\mathsf{Alice}), \mathsf{add}(\gamma.\mathsf{Bob}), m) \overset{\tau_Q+1}{\hookrightarrow} \mathcal{Z}$ and stop.
11. Mark (id, cid) as corrupt.

Back to party P

12. Distinguish the following two cases
    – If $(\text{peaceful–confirm}, id, cid, f, z, s_Q) \overset{\tau_2=\tau_1+2}{\hookleftarrow} Q$ such that $\mathsf{Vrfy}_{pk_Q}(id, cid, \tilde{\sigma}, \mathtt{C}^P, w^P; s_Q) = 1$, then set $\Gamma^P := \texttt{UpdateChanSpace}(\Gamma^P, id, cid, \tilde{\sigma}, \mathtt{C}^P, \mathsf{add}, w^P, \{s_P, s_Q\})$, $\Gamma^P_{aux}.\mathsf{next\text{-}version} := w^P + 1$, output $(\text{executed}, id, cid, \tilde{\sigma}, \mathsf{add}(\gamma.\mathsf{Alice}), \mathsf{add}(\gamma.\mathsf{Bob}), m) \overset{\tau_2}{\hookrightarrow} \mathcal{Z}$ and stop.
    – Else mark (id, cid) as corrupt and run $\texttt{RegisterDirect}(P, id, cid)$. Once the procedure is executed (in round $\tau_3 \leq \tau_0 + 3\Delta + 5$) and it holds that $\Gamma^P(id).\mathsf{cspace}(cid).\mathsf{storage} = \tilde{\sigma}$ (i.e. Q registered the contract instance version after execution), then output $(\text{executed}, id, cid, \tilde{\sigma}, \mathsf{add}(\gamma.\mathsf{Alice}), \mathsf{add}(\gamma.\mathsf{Bob}), m) \overset{\tau_3}{\hookrightarrow} \mathcal{Z}$ and stop. Else goto the next step.
13. Send $(\text{instance–execute}, id, cid, f, z) \overset{\tau_3}{\hookrightarrow} \mathcal{H}$.

Back to party Q

14. If $(\text{execute–requested}, id, cid, f, z, \tau, \texttt{direct}) \hookleftarrow \mathcal{H}$, output $(\text{execute–requested}, id, cid, f, z, \tau) \hookrightarrow \mathcal{Z}$.

End for both parties $T = A, B$

15. If (instance–executed, id, cid, f, $\tau, \widehat{\sigma}, \mathsf{add}, m, \mathtt{direct}$) $\xleftarrow{\tau_4}$ $\mathcal{H}$, set $\Gamma^T := \mathtt{UpdateChanSpace}(\Gamma^T, id, cid, \widehat{\sigma}, \mathtt{C}^T, \mathsf{add})$ and output (executed, id, cid, $\widehat{\sigma}$, $\mathsf{add}(\gamma.\mathsf{Alice}), \mathsf{add}(\gamma.\mathsf{Bob}), m$) $\xhookrightarrow{\tau_4}$ $\mathcal{Z}$.

Protocol $\Pi_{dch}(i, \mathcal{C}_0, \pi)$: Close a virtual state channel with direct dispute

Let γ be a virtual state channel with direct dispute and length i requested to be created in round τ_0. Let $cid_T := T||\gamma.\mathsf{id}$ and $id_T := \gamma.\mathsf{subchan}(T)$ for each party $T \in \gamma.\mathsf{end–users}$. Let $\mathrm{TE}_{\lceil i/2 \rceil} := \mathrm{TimeExe}(\lceil i/2 \rceil)$ denote the upper bound of number of rounds needed to execute a contract instance in a virtual state channel of length $\leq \lceil i/2 \rceil$ and $\mathrm{TR}_{\lceil i/2 \rceil} := \mathrm{TimeExeReq}(\lceil i \rceil)$ denote the upper bound of number of rounds needed to inform a party that an execution request was made in virtual state channel of length $\leq \lceil i/2 \rceil$.

We abbreviate $\mathcal{H} := \mathcal{W}_{dch}^{\widehat{\mathcal{C}}(\Delta)}(i-1, \mathcal{C}_1, \mathcal{C}_0)$, $A := \gamma.\mathsf{Alice}$, $B := \gamma.\mathsf{Bob}$ and $I := \gamma.\mathsf{Ingrid}$.

Party $T \in \gamma.\mathsf{end–users}$ in round $\tau_v := \gamma.\mathsf{validity}$

1. Let $\nu^T := \Gamma^T(id_T).\mathsf{cspace}(cid_T)$.
2. Distinguish the following cases
   - If $\Gamma^T(id) = \bot$ and $\nu^T = \bot$, then do nothing.
   - If $\Gamma^T(id) = \bot$ but $\nu^T \neq \bot$, then wait until round $\tau_2 := \tau_v + 3\Delta + 2$ to send (execute, id_T, cid_T, $\mathtt{Close}_i^{\mathsf{d}}, \bot$) $\hookrightarrow \mathcal{H}$ and stop.
   - If none of the above cases applies, then proceed as follows
     (a) Set $z := \emptyset$. For every $cid \in \{0,1\}^*$ such that $\gamma.\mathsf{cspace}(cid) \neq \bot$, set $z := z \cup \gamma.\mathsf{cspace}(cid)$.
     (b) Compute $(\tilde{\sigma}, \mathsf{add}, \text{contract–closed}) := \mathtt{Close}(\nu^T.\mathsf{storage}, T, \tau_v + 3\Delta + 2, z)$.
     (c) Send (update, s_T, id_T, cid_T, $\tilde{\sigma}$, $\mathtt{dVSCC}_i$) $\xhookrightarrow{\tau_v}$ $\mathcal{H}$ and goto Step 4.

Party I in round $\tau_v + 1$

3. Distinguish the following two cases:
   - If for both $T \in \{A, B\}$ you receive (update–requested, id_T, cid_T, $\tilde{\sigma}_T$, $\mathtt{dVSCC}_i$) $\xleftarrow{\tau_v+1}$ $\mathcal{H}$ and they are such that (i) $\tilde{\sigma}_A.\mathsf{cash}(A) = \tilde{\sigma}_B.\mathsf{cash}(I)$ and $\tilde{\sigma}_A.\mathsf{cash}(I) = \tilde{\sigma}_B.\mathsf{cash}(B)$, and (ii) $\tilde{\sigma}_A.\mathsf{cash}(A) + \tilde{\sigma}_A.\mathsf{cash}(I) = 0$, send (update–reply, ok, id_A, cid_A) $\xhookrightarrow{\tau_v+1}$ $\mathcal{H}$ and (update–reply, ok, id_B, cid_B) $\xhookrightarrow{\tau_v+1}$ $\mathcal{H}$ and stop.
   - Otherwise, goto Step 5.

Back to party $T \in \gamma.\mathsf{end–users}$

4. Distinguish the following cases:
   - If you receive (updated, id_T, cid_T) $\xleftarrow{\tau_v+2}$ $\mathcal{H}$, then goto Step 7.
   - Otherwise, proceed as follows:
     (a) For $cid \in \{0,1\}^*$ that is not marked as registered, execute (in parallel) the subprocedure $\mathtt{RegisterDirect}(T, id, cid)$.
     (b) Once the registration procedure is over (at round $\tau_1 \leq \tau_v + 3\Delta + 2$), send (execute, id_T, cid_T, $\mathtt{Close}_i^{\mathsf{d}}, \bot$) $\xhookrightarrow{\tau_1}$ $\mathcal{H}$ and goto Step 6.

Party I:

5. Let $\tau_2 := \tau_v + 3\Delta + 2 + \mathrm{TR}_{\lceil \frac{i}{2} \rceil}$. If for $T \in \{A, B\}$ you did not receive (execute–requested, id_T, cid_T, $\mathtt{Close}_i^{\mathsf{d}}, z$) $\xleftarrow{\leq \tau_2}$ $\mathcal{H}$, send (execute, id_T, cid_T, $\mathtt{Close}_i^{\mathsf{d}}, \bot$) $\xhookrightarrow{\tau_2}$ $\mathcal{G}$

End for both $T \in \gamma.\mathsf{end–users}$

6. Upon receiving (executed, id_T, cid_T, σ_T, add_T, contract–closed) $\hookleftarrow \mathcal{H}$ go to step 7.
7. Let $Q := \gamma.\mathsf{other–party}(T)$ and let τ_4 be the current round and let $\tau_6 := \tau_v + 3\Delta + 2 + \mathrm{TE}_{\lceil i/2 \rceil}$. Set $\Gamma^T(id) := \bot$ and send (closed, ok, id) $\xhookrightarrow{\tau_4}$ Q and distinguish:
   - If you have received (closed, ok, id) $\xleftarrow{\leq \tau_4}$ Q, wait for one round and output (closed, id) $\xhookrightarrow{\tau_4+1}$ $\mathcal{Z}$.
   - Otherwise wait if you receive (closed, ok, id) $\xleftarrow{\tau_5 < \tau_6}$ Q. In that case output (closed, id) $\xhookrightarrow{\tau_5}$ $\mathcal{Z}$ and else wait until round τ_6 to output (closed, id) $\xhookrightarrow{\tau_6}$ $\mathcal{Z}$.

Function $\mathtt{Close}_i^{\mathsf{d}}(\sigma, P, \tau, z)$

Let $L := \sigma.\mathsf{users}(1), R := \sigma.\mathsf{users}(2)$, $\gamma := \sigma.\mathsf{virtual\text{–}channel}$, $A := \gamma.\mathsf{Alice}, B := \gamma.\mathsf{Bob}, I := \gamma.\mathsf{Ingrid}$.

1. Make the following checks: $\gamma \neq \bot$; $P \in \{L, R\}$; if $P = \{A, B\}$, then $\tau \geq \gamma.\mathsf{validity} + 3\Delta + 2$; if $P = I$, then $\tau \geq \gamma.\mathsf{validity} + 3\Delta + 2 + \mathrm{TR}_{\lceil \frac{i}{2} \rceil}$. If one of the checks fails, output $(\sigma, 0, 0, \bot)$.
2. Let $\sigma^{(0)} := \sigma$ and parse $(\nu_1, \ldots, \nu_\ell) := z$.
3. For every $k \in [\ell]$ proceed as follows:
   (a) Let $\sigma_n := \nu_i.\mathsf{storage}$ and define $\sigma^{(k+1)} := \sigma^{(k)}$.
   (b) If $\sigma_n.\mathsf{users} = \gamma.\mathsf{end\text{–}users}$, then adjust the cash values of L and R as $\sigma^{(k+1)}.\mathsf{cash}(L) := \sigma^{(k)}.\mathsf{cash}(L) - \sigma_n.\mathsf{cash}(A)$ and $\sigma^{(k+1)}.\mathsf{cash}(R) := \sigma^{(k)}.\mathsf{cash}(R) - \sigma_n.\mathsf{cash}(B)$.
4. Set $\tilde{\sigma} := \sigma^{(\ell)}$. Let $\mathit{invest}_L := \gamma.\mathsf{cash}(A)$, $\mathit{invest}_R := \gamma.\mathsf{cash}(B)$ denote the initial balance of parties in the contract when it was created and let $\mathit{final}_L := \tilde{\sigma}.\mathsf{cash}(L)$ and $\mathit{final}_R := \tilde{\sigma}.\mathsf{cash}(R)$ denote the current balance. Distinguish the following two situations:
   – If $\mathit{final}_L \geq 0$ and $\mathit{final}_R \geq 0$ and $X := (\mathit{invest}_L - \mathit{final}_L) + (\mathit{invest}_R - \mathit{final}_R) \geq 0$, then set $\tilde{\sigma}.\mathsf{cash}(L) := (\mathit{invest}_L - \mathit{final}_L)$ and $\mathsf{add}(L) := \mathit{final}_L$. Analogously for $\tilde{\sigma}.\mathsf{cash}(R)$ and $\mathsf{add}(R)$. If $X > 0$, then set $\tilde{\sigma}.\mathsf{cash}(I) := \tilde{\sigma}.\mathsf{cash}(I) + X$ and add X coins to $\mathsf{add}(L)$ if $I = L$ and to $\mathsf{add}(R)$ if $I = R$. Let $\mathsf{add} := (\mathsf{add}(L), \mathsf{add}(R))$
   – Otherwise set $\tilde{\sigma}.\mathsf{cash}(L) := 0$, $\tilde{\sigma}.\mathsf{cash}(R) := 0$ and $\mathsf{add} := (\mathsf{add}(L), \mathsf{add}(R)) := (\mathit{invest}_L, \mathit{invest}_R)$.
5. Set $\tilde{\sigma}.\mathsf{locked} := 0$, $\tilde{\sigma}.\mathsf{virtual\text{–}channel} := \bot$ and output $(\tilde{\sigma}, \mathsf{add}, \text{contract–closed})$.

H Additional material on the Protocol for Virtual Multi-Party Channels

In this section, we provide the full description of the protocol $\Pi_{mpch}(i, \mathcal{C}_0)$ for multi-party channels which was already described informally in Sec. 7.2. Recall that we distinguish two cases: when messages about two-party state channels are received by the parties and when a multi-party channel is received by the parties.

Two-party state channels If a party P receives a message about creating/closing a two party state channel, then the party immediately forwards it to the hybrid ideal functionality $\mathcal{H} := \mathcal{W}_{dch}(i, \mathcal{C}_1, \mathcal{C}_0)$. Analogously, if the hybrid ideal functionality outputs a message that a two party state channel was created/closed, then the party simply forwards this message to the environment.

In case a party P receives a message m which is of the form $(\text{update}, \mathit{id}, \mathit{cid}, \sigma, \mathtt{C})$, $(\text{update–reply}, \mathit{ok}, \mathit{id}, \mathit{cid})$ or $(\text{execute}, \mathit{id}, \mathit{cid}, f, z)$ and id is an identifier of a two party state channel, then for technical reasons the party P sightly modifies the message m before sending it to the hybrid ideal functionality. In particular, party P extends the contract instance identifier cid by adding a prefix "TwoParty". For example, if $m = (\text{execute}, \mathit{id}, \mathit{cid}, f, z)$, then the party sends the message $m' = (\text{execute}, \mathit{id}, \text{TwoParty}||\mathit{cid}, f, z)$ to $\mathcal{H}$. Analogously, if a party P receives a message from the hybrid ideal functionality $\mathcal{H}$ that a contract instance with identifier TwoParty$||\mathit{cid}$ was updated/executed, the party removes the prefix "TwoParty"' before forwarding it to the environment.

The reason for adding/removing this prefix is to prevent the environment from instructing honest parties to update/execute a contract instance with identifier $P||Q||\mathit{id}$, where $P, Q \in \mathcal{P}$ and id is an identifier of a multi-party channel. Looking ahead, contract instance identifiers of this form are used for contract instances with the code $\mathtt{mpVSCC}_i$ that are created in the subchannels of a multi-party channel and hence should be updated/executed only when a multi-party virtual state channel is being created or closed. For the rest of this section, we focus on the protocol description of the case when a party receives a message about a multi-party channel.

Create a multi-party channel

Protocol $\Pi_{mpch}(i, \mathcal{C})$: Create a multi-party virtual state channel

We denote the hybrid functionality as $\mathcal{H} := \mathcal{W}_{dch}(i, \mathcal{C}_1, \mathcal{C}_0)$ and the virtual state contract as $\mathtt{C} := \mathtt{mpVSCC}_i$.

Party $P \in \gamma.\mathsf{end\text{–}users}$ upon $(\text{create}, \gamma) \overset{\tau_0}{\hookleftarrow} \mathcal{Z}$

1. For every $Q \in \gamma.\mathsf{neighbors}(P)$, set $id_Q := \gamma.\mathsf{subchan}(\{P, Q\})$ and distinguish the following cases:
   Case 1: If $\mathsf{Order}_{\mathcal{P}}(P) < \mathsf{Order}_{\mathcal{P}}(Q)$, then compute $\tilde{\sigma}_Q := \mathtt{Init}_i^{\mathtt{mp}}(P, Q, \tau_0; \gamma)$ and set $cid_Q := P||Q||\gamma.\mathsf{id}$. Then send $(\text{update}, id_Q, cid_Q, \tilde{\sigma}_Q, \mathtt{C}) \overset{\tau_0}{\hookrightarrow} \mathcal{H}$.
   Case 2: If $\mathsf{Order}_{\mathcal{P}}(P) > \mathsf{Order}_{\mathcal{P}}(Q)$, then compute $\tilde{\sigma}_Q := \mathtt{Init}_i^{\mathtt{mp}}(Q, P, \tau_0; \gamma)$ and set $cid_Q := Q||P||\gamma.\mathsf{id}$. If $(\text{update–requested}, id_Q, cid_Q, \tilde{\sigma}_Q, \mathtt{C}) \overset{\tau_0+1}{\hookleftarrow} \mathcal{H}$ reply by sending $(\text{update–reply}, ok, id_Q, cid_Q) \overset{\tau_0+1}{\hookrightarrow} \mathcal{H}$.
2. If for every $Q \in \gamma.\mathsf{neighbors}(P)$ you receive $(\text{updated}, id_Q, cid_Q) \overset{\tau_0+2}{\hookleftarrow} \mathcal{H}$, then send $(\text{create–ok}, \gamma) \overset{\tau_0+2}{\hookrightarrow} \gamma.\mathsf{other–party}(P)$. Else stop.
3. In round $\tau_0 + 3$ distinguish the following two cases:
   – If for every party $T \in \gamma.\mathsf{other–party}(P)$ you received $(\text{create–ok}, \gamma) \overset{\tau_0+3}{\hookleftarrow} T$, then goto step 4
   – Else send $(\text{create–not–ok}, \gamma) \overset{\tau_0+3}{\hookrightarrow} \gamma.\mathsf{other–party}(P)$ and stop.
4. In round $\tau_0 + 4$, distinguish the following two cases:
   – If for any party $T \in \gamma.\mathsf{other–party}(P)$ you received $(\text{create–not–ok}, \gamma) \overset{\tau_0+4}{\hookleftarrow} T$, then stop.
   – Else set $\Gamma^P(\gamma.\mathsf{id}) := \gamma$, output $(\text{created}, \gamma) \overset{\tau_0+4}{\hookrightarrow} \mathcal{Z}$.

Constructor $\mathtt{Init}_i^{\mathtt{mp}}(P, Q, \tau, \gamma)$

If $\{P, Q\} \notin \gamma.\mathsf{E}$ or $\gamma.\mathsf{cash}(T) < 0$ for some $T \in \gamma.\mathsf{users}$ or $\gamma.\mathsf{cspace}(cid) \neq \bot$ for some $cid \in \{0,1\}^*$ or $\gamma.\mathsf{validity} < \tau + 3\Delta + 3$, then output $\bot$. Else let $(V_P, V_Q) := \gamma.\mathsf{split}(P, Q)$ and output the attribute tuple σ defined as follows:

$$\begin{aligned}
\sigma.\mathsf{users} &:= (P, Q) \\
\sigma.\mathsf{locked} &:= \sum_{T \in \gamma.\mathsf{users}} \gamma.\mathsf{cash}(T), \\
(\sigma.\mathsf{cash}(\sigma.\mathsf{users}(1)), \sigma.\mathsf{cash}(\sigma.\mathsf{users}(2))) &:= \left(\sum_{T \in V_P} \gamma.\mathsf{cash}(T), \sum_{T \in V_Q} \gamma.\mathsf{cash}(T) \right), \\
\sigma.\mathsf{virtual–channel} &:= \gamma.
\end{aligned}$$

Registration of a contract instance The informal description of the registration procedure was provided already in Sec. 7.2. Before we present the formal definition, let us discuss one technicality here. A party P can initiate the registration process even though it does not have any valid version of the contract instance, i.e. $\nu^P = \bot$. Looking ahead, this can happen in the situation when a new contract instance is being created in the channel and some party does not confirm neither reject the contract instance creation (see the subprotocol "Update contract instance" for more details). In this case, the initiating party sends the set of all other parties in the channel γ, instead of its version of the contract instance; more precisely, it sends $\nu^P := \gamma.\mathsf{other–party}(P)$. The dispute board thus knows to which parties it should send the registration request.

Protocol $\Pi_{mpch}(i, \mathcal{C})$: Procedure $\mathtt{mpRegister}(P, id, cid)$

We denote $\mathcal{H} := \mathcal{W}_{dch}(i, \mathcal{C}_1, \mathcal{C}_0)$.

Party P:

1. Let $\gamma^P := \Gamma^P(id)$, $\nu^P := \gamma^P.\mathsf{cspace}(cid)$, and let τ_0 be the current round. If $\nu^P = \bot$, then set $\nu^P := \gamma.\mathsf{other–party}(P)$.
2. Send the message $(\text{dispute}, id, cid) \overset{\tau_0}{\hookrightarrow} \gamma^P.\mathsf{other–party}(P)$ and $(\text{instance–register}, id, cid, \nu^P) \overset{\tau_0}{\hookrightarrow} \mathcal{H}$ and set $\tau_P := \tau_0 + 2\Delta$.

Party $Q \in \gamma.\mathsf{other–party}(P)$ upon $(\text{instance–registering}, id, cid, \nu) \overset{\tau_1}{\hookleftarrow} \mathcal{H}$

3. Let $\gamma^Q := \Gamma^Q(id)$ and $\nu^Q := \gamma^Q.\mathsf{cspace}(cid)$. If $\nu^Q \neq \perp$ and $\nu^Q.\mathsf{version} > \nu.\mathsf{version}$, then send (instance–register, id, cid, ν^Q) $\overset{\tau_1}{\hookrightarrow} \mathcal{H}$.
4. Set $\tau_Q := \tau_1 + \Delta$.

For every $T \in \gamma.\mathsf{users}$

5. If not (instance–registered, $id, cid, \tilde{\nu}$) $\overset{\leq \tau_T}{\hookleftarrow} \mathcal{H}$, then send (finalize–register, $id, cid, \gamma.\mathsf{users}$) $\overset{\tau_T}{\hookrightarrow} \mathcal{H}$.
6. Upon (instance–registered, $id, cid, \tilde{\nu}$) $\overset{\leq \tau_T + \Delta}{\hookleftarrow} \mathcal{H}$, mark (id, cid) as registered and set $\Gamma^T := \mathtt{UpdateChanSpace}^*(\Gamma^T, id, cid, \tilde{\nu})$.

Update a contract instance

Protocol $\Pi_{mpch}(i, \mathcal{C})$: Contract instance update

Party P upon (update, $id, cid, \tilde{\sigma}, \mathtt{C}$) $\overset{\tau_0}{\hookleftarrow} \mathcal{Z}$

1. If (id, cid) is marked as corrupt in Γ^P_{aux}, stop. Otherwise, let $\gamma^P := \Gamma^P(id)$. If $\gamma^P.\mathsf{cspace}(cid) = \perp$ set $w^P := 1$, else let $w^P := \Gamma^P_{aux}(id, cid).\mathsf{next\text{-}version}$.
2. Compute $s_P := \mathsf{Sign}_{sk_P}(id, cid, \tilde{\sigma}, \mathtt{C}, w^P)$.
3. Send (update, $s_P, id, cid, \tilde{\sigma}, \mathtt{C}$) $\overset{\tau_0}{\hookrightarrow} \gamma.\mathsf{other\text{–}party}(P)$ and goto Step 7.

Every party $Q \in \gamma.\mathsf{other\text{–}party}(P)$ upon (update, $s_P, id, cid, \tilde{\sigma}, \mathtt{C}$) $\overset{\tau_1}{\hookleftarrow} P$

4. If (id, cid) is marked as corrupt in Γ^Q_{aux}, then stop.
5. Let $\gamma^Q := \Gamma^Q(id)$ and $\nu^Q := \gamma^Q.\mathsf{cspace}(cid)$. If $\nu^Q = \perp$, then set $w^Q := 1$, else set $w^Q := \Gamma^Q_{aux}(id, cid).\mathsf{next\text{-}version}$.
6. If $\mathsf{Vrfy}_{pk_P}(id, cid, \tilde{\sigma}, \mathtt{C}, w^Q; s_P) \neq 1$, then mark (id, cid) as corrupt and stop. Else output (update–requested, id, cid) $\overset{\tau_1}{\hookrightarrow} \mathcal{Z}$ and consider the following two cases
   - If (update–reply, ok, id, cid) $\overset{\tau_1}{\hookleftarrow} \mathcal{Z}$, then sign $s_Q := \mathsf{Sign}_{sk_Q}(id, cid, \tilde{\sigma}, \mathtt{C}, w^Q)$, send (update–ok, id, cid, s_Q) $\overset{\tau_1}{\hookrightarrow} \gamma.\mathsf{other\text{–}party}(Q)$ and goto Step 7.
   - Else compute the signature $s_Q := \mathsf{Sign}_{sk_Q}(id, cid, \nu^Q.\mathsf{storage}, \nu^Q.\mathsf{code}, w^Q{+}1)$, send message (update–not–ok, id, cid, s_Q) $\overset{\tau_1}{\hookrightarrow} \gamma.\mathsf{other\text{–}party}(Q)$, set $\Gamma^Q_{aux}.\mathsf{next\text{-}version}(id, cid) := w^Q + 2$ and stop.

Every party $T \in \gamma.\mathsf{users}$

7. For every $Q \in \gamma.\mathsf{users} \setminus \{P, T\}$, set $ok(Q) := -1$, set $ok(P) := 0$ and proceed as follows:
   - If you receive (update–ok, id, cid, s_Q) $\overset{\tau_2 = \tau_0 + 2}{\hookleftarrow} Q$, where $ok(Q) = -1$ and $\mathsf{Vrfy}_{pk_Q}(id, cid, \tilde{\sigma}, \mathtt{C}, w^T; s_Q) = 1$, then set $ok(Q) := 0$.
   - If you receive (update–not–ok, s_Q) $\overset{\tau_2 = \tau_0 + 2}{\hookleftarrow} Q$, where $ok(Q) \leq 0$ and $\mathsf{Vrfy}_{pk_Q}(id, cid, \nu^T.\mathsf{storage}, \nu^T.\mathsf{code}, w^T + 1; s_Q) = 1$, then set $ok(Q) := 1$ compute $s_T := \mathsf{Sign}_{sk_T}(id, cid, \nu^T.\mathsf{storage}, \nu^T.\mathsf{code}, w^T + 1)$ and send (update–not–ok, id, cid, s_T) $\overset{\tau_2}{\hookrightarrow} \gamma.\mathsf{other\text{–}party}(T)$
8. In round $\tau_3 := \tau_2 + 1$, distinguish the following cases
   - If there exists $Q \in \gamma.\mathsf{other\text{–}party}(T)$ such that $ok(Q) = -1$, then goto Step 9.
   - If for all $Q \in \gamma.\mathsf{other\text{–}party}(T)$ it holds that $ok(Q) = 0$ but in round τ_3 you receive (update–not–ok, s_Q) $\overset{\tau_3}{\hookleftarrow} Q$ from some $Q \in \gamma.\mathsf{other\text{–}party}(T)$, where $\mathsf{Vrfy}_{pk_Q}(id, cid, \nu^T.\mathsf{storage}, \nu^T.\mathsf{code}, w^T + 1; s_Q) = 1$, then goto Step 9.
   - If for all $Q \in \gamma.\mathsf{other\text{–}party}(T)$ it holds that $ok(Q) = 0$ and you do not receive (update–not–ok, s_Q) $\overset{\tau_3}{\hookleftarrow} Q$ from any $Q \in \gamma.\mathsf{other\text{–}party}(T)$, then set $\Gamma^T := \mathtt{UpdateChanSpace}^*(\Gamma^T, id, cid, \tilde{\sigma}, \mathtt{C}, w^T, \{s_Q\}_{Q \in \gamma.\mathsf{users}})$, set $\Gamma^T_{aux}(id, cid).\mathsf{next\text{-}version} := w^T + 1$, output (updated, id, cid) $\overset{\tau_3}{\hookrightarrow} \mathcal{Z}$ and stop.
   - Else proceed as follows:
     (a) For all $Q \in \gamma.\mathsf{other\text{–}party}(T)$: If you receive (update–not–ok, s_Q) $\overset{\tau_3}{\hookleftarrow} Q$, where $\mathsf{Vrfy}_{pk_Q}(id, cid, \nu^T.\mathsf{storage}, \nu^T.\mathsf{code}, w^T + 1; s_Q) = 1$, then set $ok(Q) := 1$.

(b) If for all $Q \in \gamma.\mathsf{other\text{–}party}(T)$ it holds that $ok(Q) = 1$, then set $\Gamma^T := \mathtt{UpdateChanSpace}^*(\Gamma^T, id, cid, \sigma, \mathtt{C}, w^T + 1, \{s_Q\}_{Q \in \gamma.\mathsf{users}})$, set $\Gamma^Q_{aux}.\mathsf{next\text{-}version}(id, cid) := w^T + 2$ and stop.
(c) Else goto Step 9.

9. Let $\tau_4 \leq \tau_0 + 3$ be the current round. Mark (id, cid) as corrupt and call the subprocedure $\mathtt{mpRegister}(T, id, cid)$. After the subprocedure execution (in round $\tau_5 \leq \tau_4 + 3\Delta$), if $\gamma^T.\mathsf{cspace}(cid) = (\tilde{\sigma}, \mathtt{C})$, then output $(\text{updated}, id, cid) \overset{\tau_5}{\hookrightarrow} \mathcal{Z}$.

Execution of a contract instance As already explained in Sec. 7.2, our protocol requires a fixed ordering of peaceful execution requests. Therefore, before we present the formal description of the execution protocol, we present the ordering considered in this work. Recall that a peaceful execution request is a tuple of the form (f, z, τ, P), where f is the function to be executed on input parameter z. The value of τ denotes the time-stamp of the request and P is the requesting party.

1. The requests are ordered by party, meaning that requests from party $P \in \mathcal{P}$ will be processed before requests from $Q \in \mathcal{P}$ if $\mathsf{Order}_{\mathcal{P}}(P) < \mathsf{Order}_{\mathcal{P}}(Q)$.
2. Requests with lower time-stamp are processed first.
3. Requests are executed along the function order inside the contract code.
4. Lastly, the lexicographic ordering of binary input representation is applied.

Protocol $\Pi_{mpch}(i, \mathcal{C})$: Contract instance execution

We denote $\mathcal{H} := \mathcal{W}_{dch}(i, \mathcal{C}_1, \mathcal{C}_0)$. In addition, we assume that every party $P \in \mathcal{P}$ maintains the following three sets $\Gamma^P_{aux}(id, cid).\mathsf{toPeacefullyExecute}$, $\Gamma^P_{aux}(id, cid).\mathsf{toForceExecute}$ and $\Gamma^P_{aux}(id, cid).\mathsf{PeacefullyExecuted}$ in its auxiliary channel space.

Party P

Upon $(\text{execute}, id, cid, f, z) \overset{\tau_c}{\hookleftarrow} \mathcal{Z}$**:**

1. Add the tuple (f, z, τ_c, P) to the set $\Gamma^P_{aux}(id, cid).\mathsf{toPeacefullyExecute}$.
2. Wait for at most three rounds until the round number, let us denote it τ_0, is such that $\tau_0 = 0 \mod 4$.
3. Let $E \subseteq \Gamma^P_{aux}(id, cid).\mathsf{toPeacefullyExecute}$ consist of all $(f', z', \tau'_c, P) \in \Gamma^P_{aux}(id, cid).\mathsf{toPeacefullyExecute}$, where $\tau'_c \in [\tau_0 - 3, \tau_0]$. Distinguish the following two situations:
   - If (id, cid) is marked as corrupt in Γ^P_{aux}, then
     (a) Add elements from E to the set $\Gamma^P_{aux}(id, cid).\mathsf{toForceExecute}$
     (b) If (id, cid) is not marked as registered yet, then run the procedure $\mathtt{mpRegister}(P, id, cid)$.
     (c) Goto step 12.
   - Else for every $(f', z', \tau'_c, P) \in E$ send $(\text{peaceful–request}, id, cid, f', z', \tau'_c) \overset{\tau_c}{\hookrightarrow} \mathcal{Q}$ and then goto step 5.

Upon $(\text{peaceful–request}, id, cid, f, z, \tau_c) \overset{\tau}{\hookleftarrow} Q$**:**

4. Let $\gamma := \Gamma^P(id), \nu := \gamma.\mathsf{cspace}(cid), \sigma := \nu.\mathsf{storage}$.
   - If at least one of the following is true: $\tau \neq 1 \mod 4$; $\tau_c \notin [\tau - 4, \tau - 1]$; $\gamma = \bot$; $P, Q \notin \gamma.\mathsf{users}$; $\nu = \bot$; $f \neq \nu.\mathsf{code}$; $\tau_c \geq \gamma.\mathsf{validity}$; (id, cid) is marked as corrupt in Γ^P_{aux}, then mark (id, cid) as corrupt in Γ^P_{aux}.
   - Else add tuple (f, z, τ_c, Q) to the set $\Gamma^P_{aux}(id, cid).\mathsf{toPeacefullyExecute}$ and then goto step 5.

Local execution

5. Wait for at most one round until the round number, let us denote it τ_1, is such that $\tau_1 = 1 \mod 4$.
6. Let $\gamma := \Gamma^P(id)$, $\sigma^{(0)} := \gamma.\mathsf{cspace}(cid).\mathsf{storage}$, $\mathtt{C} := \nu.\mathsf{code}$ and $w := \Gamma_{aux}(id, cid).\mathsf{next\text{-}version}$. Let $E \subseteq \Gamma^P_{aux}(id, cid).\mathsf{toPeacefullyExecute}$ consist of all tuples $e^{(k)} \in \Gamma_{aux}(id, cid).\mathsf{toPeacefullyExecute}$, where $e^{(k)} = (f^{(k)}, z^{(k)}, \tau_c^{(k)}, T^{(k)})$ for $\tau_c^{(k)} \in [\tau_1 - 4, \tau_1 - 1]$.
7. Let $\ell := |E|$ and $E = \{e^{(1)}, \ldots, e^{(\ell)}\}$ be such that for $i < j$ the following holds:
   - If $T^{(i)} \neq T^{(j)}$, then $\mathsf{Order}_{\mathcal{P}}(T^{(i)}) < \mathsf{Order}_{\mathcal{P}}(T^{(j)})$.
   - If $T^{(i)} = T^{(j)}$, then either (i) $\tau_c^{(i)} < \tau_c^{(j)}$ or (ii) $\tau_c^{(i)} = \tau_c^{(j)}$ and $f^{(i)} <_{\mathtt{C}} f^{(j)}$, where $<_{\mathtt{C}}$ is total ordering of the contract functions defined by the contract code $\mathtt{C}$, or (iii) $\tau_c^{(i)} = \tau_c^{(j)}$, $f^{(i)} = f^{(j)}$ and $z^{(i)} \leq_{\text{lex}} z^{(j)}$, where $\leq_{\text{lex}}$ is the lexicographic ordering of binary strings.
8. For $k = 1$ to ℓ do
   (a) Compute $(\sigma^{(k)}, \mathsf{add}^{(k)}, m^{(k)}) := f(\sigma^{(k-1)}, T^{(k)}, \tau_0^{(k)}, z^{(k)})$.

(b) If $m^{(k)} = \perp$ set $\sigma^{(k)} := \sigma^{(k-1)}$. Else add $(\text{executed}, id, cid, \sigma^{(k)}, \mathsf{add}^{(k)}, m^{(k)})$ to the set $\Gamma^P_{aux}(id, cid).\mathsf{PeacefullyExecuted}$.

9. For every $T \in \gamma.\mathsf{users}$, set $\mathsf{add}(T) := \sum_{k\in[\ell]} \mathsf{add}^{(k)}(T)$.
10. Set $\tilde{\sigma} := \sigma^{(\ell)}$ and compute $s_P := \mathsf{Sign}_{sk_P}(id, cid, \tilde{\sigma}, \mathtt{C}, w)$ and send $(\text{peaceful–confirm}, id, cid, s_P) \overset{\tau_1}{\hookrightarrow} \gamma.\mathsf{other–party}(P)$.
11. In round $\tau_2 := \tau_1 + 1$ distinguish:
    - If $(\text{peaceful–confirm}, id, cid, s_Q) \overset{\tau_2}{\hookleftarrow} Q$, where $\mathsf{Vrfy}_{pk_Q}(id, cid, \sigma^{(\ell)}, \mathtt{C}, w; s_Q) = 1$, from every party $Q \in \gamma.\mathsf{other–party}(P)$, then proceed as follows
      (a) Set $\Gamma^P := \mathtt{UpdateChanSpace}(\Gamma^P, id, cid, \tilde{\sigma}, \mathtt{C}, \mathsf{add}, w, \{s_Q\}_{Q\in\gamma.\mathsf{users}})$
      (b) If in round $\tau_3 := \tau_2 + 1$ you receive $(\text{dispute}, id, cid) \overset{\tau_3}{\hookleftarrow} Q$ from some party $Q \in \gamma.\mathsf{other–party}(P)$, then participate in the registration procedure
      (c) Let $\tau_4 \le \tau_3 + 3\Delta$ be the current round. Then for every $e \in \Gamma_{aux}.\mathsf{PeacefullyExecuted}(id, cid)$, output $e \overset{\tau_3}{\hookrightarrow} \mathcal{Z}$ and stop.
    - Else mark (id, cid) as corrupt and execute the registration subprocedure $\mathtt{mpRegister}(P, id, cid)$. If after the registration procedure is executed, in round distinguish:
      - If $\Gamma^P(id).\mathsf{cspace}(cid).\mathsf{storage} = \tilde{\sigma}$, then output $e \hookrightarrow \mathcal{Z}$ for every $e \in \Gamma_{aux}.\mathsf{PeacefullyExecuted}(id, cid)$ and stop.
      - Otherwise add every tuple $(f^{(k)}, z^{(k)}, \tau_c^{(k)}, P) \in E$ to the set $\Gamma^P_{aux}(id, cid).\mathsf{toForceExecute}$ and goto step 12.

Force execution

12. Let τ_4 be the current round. Let $E = \Gamma^P_{aux}(id, cid).\mathsf{toForceExecute}$.
13. For every $(f, z, \tau_c, P) \in E$ send $(\text{instance–execute}, id, cid, f, z) \overset{\tau_4}{\hookrightarrow} \mathcal{H}$.

Upon $(\text{instance–executed}, id, cid, \widehat{\sigma}, \mathsf{add}, m) \hookleftarrow \mathcal{H}$**:**
Set $\Gamma^P := \mathtt{UpdateChanSpace}(\Gamma^P, id, cid, \widehat{\sigma}, \mathtt{C}^T, \mathsf{add})$, output $(\text{executed}, id, cid, \widehat{\sigma}, \mathsf{add}, m) \hookrightarrow \mathcal{Z}$ and stop.

Closing a multi-party channel

Protocol $\Pi_{mpch}(i, \mathcal{C})$: Close a multi-party virtual state channel γ

We denote the hybrid functionality as $\mathcal{H} := \mathcal{W}_{dch}(i, \mathcal{C}_1, \mathcal{C}_0)$, the virtual state contract as $\mathtt{C} := \mathtt{mpVSCC}_i$. Let $\mathrm{TE}_i := \mathrm{TimeExe}(i)$ be the upper bound on the rounds needed to execute a two party virtual state channel of length $\le i$. We assume that every party maintains a function $\mathsf{terminated}\colon \gamma.\mathsf{users} \to \{0, 1\}$, to keep track on updated contract instances in the subchannels of γ.

Party $P \in \gamma.\mathsf{end–users}$ when round $\tau_0 := \gamma.\mathsf{validity}$ comes

For every $Q \in \gamma.\mathsf{neighbors}(P)$ set $cid_Q := P||Q||\gamma.\mathsf{id}$ if $\mathsf{Order}_{\mathcal{P}}(P) < \mathsf{Order}_{\mathcal{P}}(Q)$ and $cid_Q := Q||P||\gamma.\mathsf{id}$ otherwise. Let $id_Q := \gamma.\mathsf{subchan}(\{P, Q\})$ and $\sigma_Q := \Gamma^P(id_Q).\mathsf{storage}.\mathsf{cspace}(cid_Q)$

Case 1: If $\Gamma^P(\gamma.\mathsf{id}) = \perp$, then check for every $Q \in \gamma.\mathsf{neighbors}(P)$ if $\sigma_Q \neq \perp$. If this is the case, wait $3\Delta+2$ rounds to send $(\text{execute}, id_Q, cid_Q, \mathtt{Close}_i^{\mathtt{mp}}, \perp) \overset{\tau_0+3\Delta+2}{\hookrightarrow} \mathcal{H}$ and stop.

Case 2: If $\Gamma^P(\gamma.\mathsf{id}) \neq \perp$, then proceed as follows:

1. Set $z := \emptyset$. For every $cid \in \{0, 1\}^*$ such that $\gamma.\mathsf{cspace}(cid) \neq \perp$, add $\gamma.\mathsf{cspace}(cid)$ to z.
2. For every $Q \in \gamma.\mathsf{neighbors}(P)$, set $\mathsf{terminated}(Q) := 0$ and compute $(\tilde{\sigma}_Q, \mathsf{add}, m) := \mathtt{Close}_i^{\mathtt{mp}}(\sigma_Q, Q, \tau_0 + 3\Delta + 2, z)$.
3. For each neighbor of higher order $(\mathsf{Order}_{\mathcal{P}}(P) < \mathsf{Order}_{\mathcal{P}}(Q))$ send $(\text{update}, id_Q, cid_Q, \tilde{\sigma}_Q, \mathtt{C}) \overset{\tau_0}{\hookrightarrow} \mathcal{H}$.
4. For each neighbor of lower order $(\mathsf{Order}_{\mathcal{P}}(P) > \mathsf{Order}_{\mathcal{P}}(Q))$ upon receiving $(\text{update–requested}, id_Q, cid_Q, \tilde{\sigma}_Q, \mathtt{C}) \overset{\tau_0+1}{\hookleftarrow} \mathcal{H}$ reply by sending $(\text{update–reply}, ok, id_Q, cid_Q) \overset{\tau_0+1}{\hookrightarrow} \mathcal{H}$.
5. For every $Q \in \gamma.\mathsf{neighbors}(P)$ check the following:
   - If you receive $(\text{updated}, id_Q, cid_Q) \overset{\tau_0+2}{\hookleftarrow} \mathcal{H}$, set $\mathsf{terminated}(Q) := 1$.
   - If you did not receive $(\text{updated}, id_Q, cid_Q) \overset{\tau_0+2}{\hookleftarrow} \mathcal{H}$
     (a) For every $(cid, \nu) \in z$ execute $\mathtt{mpRegister}(P, \gamma.\mathsf{id}, cid)$ if $(\gamma.\mathsf{id}, cid)$ is not marked as registered in Γ^P_{aux} yet.
     (b) At $\tau_1 := \tau_0 + 3\Delta + 2$ send $(\text{execute}, id_Q, cid_Q, \mathtt{Close}_i^{\mathtt{mp}}, \perp) \overset{\tau_1}{\hookrightarrow} \mathcal{H}$.
     (c) Upon receiving $(\text{executed}, id_Q, cid_Q, \tilde{\sigma}_Q, \mathsf{add}, m) \overset{\tau_2 \le \tau_1 + \mathrm{TE}_i}{\hookleftarrow} \mathcal{H}$, set $\mathsf{terminated}(Q) := 1$.
6. Let τ_P be the first round in which $\mathsf{terminated}(Q) = 1$ for every $Q \in \gamma.\mathsf{neighbors}(P)$.

– If $\tau_P = \tau_0 + 2$ and you did not receive $(\text{dispute}, \gamma.\mathsf{id}, cid) \xleftarrow{\tau_0+3} Q$ for any $cid \in \{0,1\}^*$ such that $\gamma.\mathsf{cspace}(cid) \neq \bot$, set $\tau_3 := \tau_0 + 3$.
– Else set $\tau_3 := \tau_0 + 3\Delta + \mathrm{TE}_i + 2$.

7. In round τ_3, set $\Gamma^P(\gamma.\mathsf{id}) := \bot$ and output $(\text{closed}, \gamma.\mathsf{id}) \xhookrightarrow{\tau_3} \mathcal{Z}$.

Function $\mathtt{Close}_i^{\mathtt{mp}}(\sigma, P, \tau, z)$

Let $L := \sigma.\mathsf{users}(1), R := \sigma.\mathsf{users}(2)$, $\gamma := \sigma.\mathsf{virtual\text{–}channel}$ and $n = |\gamma.\mathsf{users}|$.

1. Make the following checks: $\gamma \neq \bot$; $P \in \{L, R\}$; and $\tau < \gamma.\mathsf{validity} + 3\Delta + 2$. If one of the checks fails, the output $(\sigma, 0, 0, \bot)$.
2. Let $\sigma^{(0)} := \sigma$, set $(V_L, V_R) := \gamma.\mathsf{split}(\{L, R\})$ and parse $(\nu_1, \ldots, \nu_\ell) := z$.
3. For every $k \in [\ell]$ proceed as follows
   (a) Let $\sigma_n := \nu_i.\mathsf{storage}$ and define $\sigma^{(k+1)} := \sigma^{(k)}$.
   (b) If $\sigma_n.\mathsf{users} = \gamma.\mathsf{users}$, adjust the cash value of L as $\sigma^{(k+1)}.\mathsf{cash}(L) := \sigma^{(k)}.\mathsf{cash}(L) - \sum_{T \in V_L} \sigma_n.\mathsf{cash}(T)$ and of user R: $\sigma^{(k+1)}.\mathsf{cash}(R) := \sigma^{(k)}.\mathsf{cash}(R) - \sum_{T \in V_R} \sigma_n.\mathsf{cash}(T)$.
4. Let $\tilde{\sigma} := \sigma^{(\ell)}$ and the values $\mathit{invest}_L := \sum_{T \in V_L} \gamma.\mathsf{cash}(T)$ and $\mathit{invest}_R := \sum_{T \in V_R} \gamma.\mathsf{cash}(T)$ denote the initial balance of parties in the contract when it was created and let $\mathit{final}_L := \tilde{\sigma}.\mathsf{cash}(L)$ and $\mathit{final}_R := \tilde{\sigma}.\mathsf{cash}(R)$ denote the current balance.
5. Let $X := (\mathit{invest}_L - \mathit{final}_L) + (\mathit{invest}_R - \mathit{final}_R)$. If $X > 0$, then add $|V_L| \cdot \frac{X}{n}$ to final_L and $|V_R| \cdot \frac{X}{n}$ to final_R.
6. Distinguish the following two cases
   – If $\mathit{final}_L \geq 0$ and $\mathit{final}_R \geq 0$ and $X \geq 0$, then set $\tilde{\sigma}.\mathsf{cash}(L) := (\mathit{invest}_L - \mathit{final}_L)$ and $\tilde{\sigma}.\mathsf{cash}(R) := (\mathit{invest}_R - \mathit{final}_R)$ and $\mathsf{add} := (\mathsf{add}(L), \mathsf{add}(R)) := (\mathit{final}_L, \mathit{final}_R)$.
   – Else set $\tilde{\sigma}.\mathsf{cash}(L) := 0$ and $\tilde{\sigma}.\mathsf{cash}(R) := 0$ and $\mathsf{add} := (\mathsf{add}(L), \mathsf{add}(R)) := (\mathit{invest}_L, \mathit{invest}_R)$.
7. Set $\tilde{\sigma}.\mathsf{locked} := 0$, $\tilde{\sigma}.\mathsf{virtual\text{–}channel} := \bot$ and output $(\tilde{\sigma}, \mathsf{add}, \text{contract–closed})$.

H.1 Theorem Statement

For completeness we restate the theorem saying that the protocol described in this section UC-realizes the ideal functionality $\mathcal{F}_{mpch}(i, \mathcal{C}_0)$.

Theorem 1 *Suppose the underlying signature scheme is existentially unforgeable against chosen message attacks. The protocol $\Pi_{mpch}(i, \mathcal{C}_0)$ in the $\mathcal{W}_{dch}(i, \mathcal{C}_1, \mathcal{C}_0)$-hybrid model emulates the ideal functionality $\mathcal{F}_{mpch}(i, \mathcal{C}_0)$ for every set of contract codes $\mathcal{C}_0$, every $i \geq 1$ and every $\Delta \in \mathbb{N}$.*

I Simplifying the protocol descriptions

Let us emphasize that the formal descriptions of both our state channel protocols $\Pi_{dch}(i, \mathcal{C}, \pi)$ and $\Pi_{mpch}(i, \mathcal{C}, \pi)$ are simplified. Similarly as the descriptions for state channel ideal functionalities (see Appx. C.1), our protocol descriptions exclude many natural checks that one would expect a party following the protocol to make. The intuition behind this is that we do not want a state channel protocol to "work" for malformed or invalid requests. Let us give a few examples of such invalid requests of $\mathcal{Z}$ that a party will refuse.

– $\mathcal{Z}$ instructs the party P to ledger channel for which P does not have enough coins on the ledger.
– $\mathcal{Z}$ instructs the party P to create a virtual channel using an intermediary I but there does not exist any state channel between P and I yet.
– $\mathcal{Z}$ instructs the party P to update or execute a contract instance in a channel that was never created by P or that was already closed.
– $\mathcal{Z}$ instructs the party P to execute a non-existing function of a contract instance.
– $\mathcal{Z}$ instruction to party P contains malformed inputs (e.g., missing or unknown parameters).

In order to simplify the already complex protocol description, we define a *wrapper* $\mathcal{W}^{\widehat{\mathcal{L}}(\Delta)}_{checks-P}(i, \mathcal{C}, \Pi)$ that performs all these checks. That is, we write $\Pi := \mathcal{W}^{\widehat{\mathcal{L}}(\Delta)}_{checks-P}(i, \mathcal{C}, \Pi)$ for $\Pi \in \{\Pi_{dch}(i, \mathcal{C}, \cdot), \Pi_{mpch}(i, \mathcal{C})\}$ and $\mathcal{W}^{\widehat{\mathcal{L}}(\Delta)}_{checks-P}$ as defined below.

Wrapper: $\mathcal{W}^{\widehat{\mathcal{L}}(\Delta)}_{checks-P}(i, \mathcal{C}, \Pi)$

Below, we abbreviate notation $A := \gamma.\mathsf{Alice}$, $B := \gamma.\mathsf{Bob}$ and $I := \gamma.\mathsf{Ingrid}$.

Party P

Create

Upon receiving $(\text{create}, \gamma) \xleftarrow{\tau_0} \mathcal{Z}$ make the following checks: $\Gamma^P(\gamma.\mathsf{id}) = \bot$ and there is no state channel γ' with $\gamma.\mathsf{id} = \gamma'.\mathsf{id}$ currently being created; γ is a valid state channel according to the definition given in Sec. 3.2; $\gamma.\mathsf{cspace}(cid) = \bot$ for every $cid \in \{0,1\}^*$. Depending on the type of the channel, additionally checks:

Ledger channel: check that both you and $Q := \gamma.\mathsf{other\text{–}party}(P)$ have enough funds on the ledger for the channel creation;[a]

Two-party virtual channel: $j := \gamma.\mathsf{length} \leq i$; if $\gamma.\mathsf{dispute} = \text{indirect}$, then $\gamma.\mathsf{validity} > \tau_0 + 2 + 4 \cdot \text{TimeExeReq}(\lceil j/2 \rceil)$ and if $\gamma.\mathsf{dispute} = \text{direct}$, then $\gamma.\mathsf{validity} > \tau_0 + 2 + 3\Delta$; and the following holds for the subchannels:

- If $P \in \gamma.\mathsf{end\text{–}users}$, then $\alpha := \Gamma^P(id_P) \neq \bot$, for $id_P := \gamma.\mathsf{subchan}(P)$; $\alpha.\mathsf{end\text{–}users} = \{P, I\}$; $\alpha.\mathsf{length} \leq \lceil j/2 \rceil$; $\alpha.\mathsf{validity} > \gamma.\mathsf{validity} + 2\text{TimeExeReq}(\lceil j/2 \rceil) + 2\text{TimeExe}(\lceil j/2 \rceil)$; if $\alpha.\mathsf{dispute} = \text{indirect}$, then check if $\alpha.\mathsf{cspace}(cid) = \bot$ for every $cid \in \{0,1\}^*$ and if there is no other virtual state channel being created over α; both P and I have enough funds in α.[b]
- If $P = \gamma.\mathsf{Ingrid}$, then $\alpha := \Gamma^P(id_A) \neq \bot$, for $id_A := \gamma.\mathsf{subchan}(P)$ and $\beta := \Gamma^P(id_B) \neq \bot$, for $id_B := \gamma.\mathsf{subchan}(B)$; $\alpha.\mathsf{end\text{–}users} = \{A, I\}$; $\beta.\mathsf{end\text{–}users} = \{B, I\}$; $j = \alpha.\mathsf{length} + \beta.\mathsf{length}$, $\alpha.\mathsf{length} \leq \lceil j/2 \rceil$ and $\beta.\mathsf{length} \leq \lceil j/2 \rceil$; $\min\{\alpha.\mathsf{validity}, \beta.\mathsf{validity}\} > \gamma.\mathsf{validity} + 2\text{TimeExeReq}(\lceil j/2 \rceil) + 2\text{Time Exe}(\lceil j/2 \rceil)$; if $\alpha.\mathsf{dispute} = \text{indirect}$, then check if $\alpha.\mathsf{cspace}(cid) = \bot$ for every $cid \in \{0,1\}^*$ and if there is no other virtual state channel being created over α; (analogously for β); A and I have enough funds in α and B and I have enough funds in β. [b]

Multi-party virtual channel: $\gamma.\mathsf{dispute} = \text{direct}$, $j := \gamma.\mathsf{length} \leq i$; $\gamma.\mathsf{validity} > \tau_0 + 3\Delta + 3$; for every $Q \in \gamma.\mathsf{neighbors}(P)$: $\alpha := \Gamma^P(id_Q) \neq \bot$, where $id_P := \gamma.\mathsf{subchan}(\{P, Q\})$; $\alpha.\mathsf{end\text{–}users} = \{P, Q\}$; $\alpha.\mathsf{length} \leq i$; $\alpha.\mathsf{validity} > \gamma.\mathsf{validity} + 3\Delta + \text{TimeExe}(i) + 2$; if $\alpha.\mathsf{dispute} = \text{indirect}$, then $\alpha.\mathsf{cspace}(cid) = \bot$ for every $cid \in \{0,1\}^*$ and there is no other virtual state channel being opened over α; both P and Q have enough funds in α. [b]

If one of the above checks fail, then drop the message. Otherwise proceed as the in the protocol Π. In addition, if $P = A$, $\gamma.\mathsf{length} = 1$ and id round $\tau_0 + 2\Delta + 1$ it still holds that $\Gamma(\gamma.\mathsf{id}) = \bot$, then act as in the protocol Π upon receiving $(\text{refund}, \gamma) \xleftarrow{\tau_0 + 2\Delta + 1} \mathcal{Z}$.

Update

Upon receiving $(\text{update}, id, cid, \tilde{\sigma}, \mathsf{C}) \xleftarrow{\tau_0} \mathcal{Z}$ make the following checks: $\gamma := \Gamma^P(id) \neq \bot$; $\tau_0 < \gamma.\mathsf{validity}$; $\mathsf{C} \in \mathcal{C}$; $\tilde{\sigma} \in \mathsf{C}.\Lambda$; $\tilde{\sigma}.\mathsf{locked} = \sum_{Q \in \gamma.\mathsf{users}} \tilde{\sigma}.\mathsf{cash}(Q)$, all parties have enough cash in the state channel for the contract instance update. If $\nu := \gamma.\mathsf{cspace}(cid) \neq \bot$, then the following must hold: $\nu.\mathsf{code} = \mathsf{C}$; $\sigma.\mathsf{user} = \tilde{\sigma}.\mathsf{user}$, where $\sigma := \nu.\mathsf{storage}$; $\sigma.\mathsf{locked} = \sum_{Q \in \gamma.\mathsf{users}} \sigma.\mathsf{cash}(Q)$. If $\gamma.\mathsf{dispute} = \text{indirect}$, then $\gamma.\mathsf{cspace}(cid^*) = \bot$ for every $cid^* \in \{0,1\}$. If one of the above checks fails, then drop the message. Otherwise proceed as the protocol Π.

Upon receiving $(\text{update–reply}, ok, id, cid) \xleftarrow{\tau_0} \mathcal{Z}$ make the following checks: the message is a reply to your message $(\text{update–requested}, id, cid, \tilde{\sigma}, \mathsf{C})$ sent to $\mathcal{Z}$ in round τ_0; there is no other update or execution of the contract instance cid in channel $\gamma := \Gamma(id)$ currently going on. In addition, if $\gamma.\mathsf{dispute} = \text{indirect}$, then check is there is no virtual state channel currently being created over γ. If one of the above checks fails, then drop the message. Otherwise proceed as in the protocol Π.

Execute

Upon receiving $(\text{execute}, id, cid, f, z) \xleftarrow{\tau_0} \mathcal{Z}$, make the following checks $\gamma := \Gamma^P(id) \neq \bot$; $\tau_0 < \gamma.\mathsf{validity}$; $\gamma.\mathsf{cspace}(cid) \neq \bot$, $f \in \gamma.\mathsf{cspace}(cid).\mathsf{code}$. If one of the above checks fails, then drop the message. Otherwise proceed as in the protocol Π.

Close

Upon $(\text{close}, id) \xleftarrow{\tau_0} \mathcal{Z}$, check if $\gamma := \Gamma^P(id) \neq \bot$ and $\gamma.\mathsf{length} = 1$. If not, then drop the message. Otherwise proceed as in the protocol Π.

[a] In case more ledger state channels are being created at the same time, both parties have enough funds for all ledger state channels that are being created.

[b] In case more virtual state channels are being created at the same time, all parties have enough funds for all of them.

Recall that our protocol $\Pi_{dch}(i, \mathcal{C}, \pi)$, that was formally defined in Appx. G, is parameterized by a protocol π which must be a protocol that UC-realizes the ideal functionality $\mathcal{W}^{\hat{\mathcal{L}}(\Delta)}_{checks}(i, \mathcal{C}, \mathcal{F}_{ch}(i, \mathcal{C}))$.

Authors of [12] define a protocol realizing the ideal functionality $\mathcal{F}_{ch}(i, \mathcal{C})$ with respect to a set of restricted environments $\mathcal{E}_{res}$. The reason for restricting the environment is to reduce the complexity of the functionality and protocol description. As discuss in Appx. C.1, it is straightforward to verify that restrictions defining the set $\mathcal{E}_{res}$ translate to checks performed by our functionality wrapper. It is also straightforward to see that parties *following the wrapped protocol* directly drop all queries that a restricted environment $\mathcal{E}_{res}$ would not be allowed to make. On the other hand, corrupted parties can of course send any message they want, but note that also $\mathcal{E}_{res}$ only was restricted in sending messages to honest parties. Hence, we conclude the protocol $\mathcal{W}^{\hat{\mathcal{L}}(\Delta)}_{checks-P}(i, \mathcal{C}, \Pi_{ch}(i, \mathcal{C}))$, where $\Pi_{ch}(i, \mathcal{C})$ is the protocol from [12], realizes the ideal functionality $\mathcal{W}^{\hat{\mathcal{L}}(\Delta)}_{checks}(i, \mathcal{C}, \mathcal{F}_{ch}(i, \mathcal{C}))$.