Multi-party Virtual State Channels

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

[1] University of Warsaw, Warsaw, Poland
stefan.dziembowski@crypto.edu.pl
[2] Technische Universität Darmstadt, Darmstadt, Germany
{lisa.eckey,sebastian.faust,julia.hesse,kristina.hostakova}@crisp-da.de

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 s, 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 and security analysis of our construction.

1 Introduction

Blockchain technology emerged recently as a promising technique for distributing trust in security protocols. It was introduced by Satoshi Nakamoto in [22] 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* [26], 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

Y. Ishai and V. Rijmen (Eds.): EUROCRYPT 2019, LNCS 11476, pp. 625–656, 2019.
https://doi.org/10.1007/978-3-030-17653-2_21

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 maximum 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* [2]. 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 for 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 [24]) 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* [5,7,10], 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 [10] on which our work relies. Dziembowski et al. [10] distinguish between two variants of two-party state channels – so-called *ledger* and *virtual* state channels[1]. 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 Sect. 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.

[1] The startup L4 and their project *Counterfactual* [7] 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 [10]. While in ledger state channels disputes are always directly taken to the ledger, in the 2-party virtual state channels from [10] 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,[2] the virtual channel construction of [10] 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 [10]. 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 [10] to less than 25 min 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 [10] 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.

[2] In Ethereum typically Δ equal to 6 min is assumed to be safe.

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 [3] (more precisely, its global variant [4]), and a recursive composition approach similar to [10]. One particular nice feature of our modeling approach is that we are able to re-use the ideal state channel functionality presented in [10]. 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 [8]. Since then, there has been a vast number of different works constructing protocols for off-chain transactions and channel networks with different properties [12,15–18,25]. 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 [7], Celer network [5] and Magmo [6]. The whitepapers 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, non of these works includes a formal modeling of state channels nor a security analysis.

To the best of our knowledge, most related to our work is [10], which we significantly extend (as described above), and the recent work of Sprites [21] and its extensions [19,20] on building multi-party *ledger* state channels. At a high-level in [19–21] 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 [19–21] 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, [19–21] 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, [21] aims at improving the worst case timings in payment channel networks, and [19,20] 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 Sect. 2.1 (we follow the formalism of [10]). In Sect. 2.2 we give an overview of our construction for handling "direct disputes", while in Sect. 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 [10]

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 $\mathsf{C}_{\mathsf{sell}}$ 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).[3] 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

[3] 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.

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 $\mathsf{C}_{\mathsf{sell}}$ consists of functions used to deploy the contract (see footnote 3), 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.

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 [10] 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 8).

Virtual state channels. The main novelty of [10] 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 8). 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.

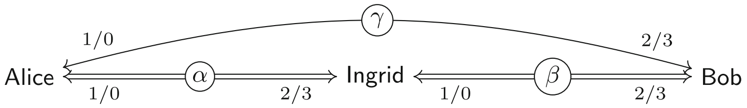

Fig. 1. Virtual channel γ built over ledger channels α and β. The labels "x/y" on the channels denote the fact that a given party locked x coins for the creation of γ, and got y coins as a result of closing γ.

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 [10].

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.[4] Hence, the ledger contract SCC gives Alice time Δ to reply with a more recent signed state (recall that in Sect. 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.[5]

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 [10] 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 [10] 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.

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

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

As discussed in the introduction, one main shortcoming of the recursive approach used by [10] is that even if only one intermediary is malicious[6], the worst-case time needed for dispute resolution is 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 [10] *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 [10] – *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 [21], 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

[6] 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 $P_{n/2}$.

$P_{n/2}$ 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 [10]. 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 8 uppose 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 Sect. 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 $P_1, \ldots P_{t-1}$ 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 $P_1, \ldots, P_{t-1}$ (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 $i \in \{0, \ldots, n-1\}$, 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 $P_1, \ldots, P_5$, 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 $P_1 \leftrightarrow P_3$.

Now the channel γ can be created on top of the subchannels $P_1 \leftrightarrow P_3$, $P_3 \Leftrightarrow P_4$ and $P_4 \Leftrightarrow P_5$.[7] 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.

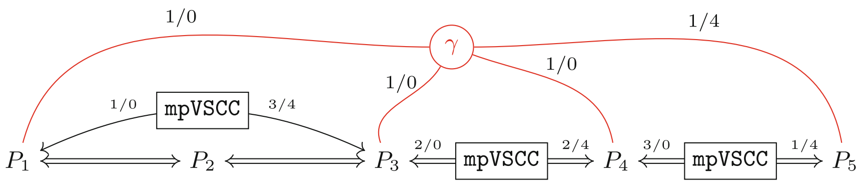

Fig. 2. Example of a multi-party virtual state channel γ between parties P_1, P_3, P_4 and P_5. In each subchannel a contract instance of `mpVSCC` is opened. Initially every party invests one coin and when the channel is closed, party P_5 owns all coins. The figure depicts the initial/final balance of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

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

[7] 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.

our construction uses the direct dispute process outlined in Sect. 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.[8] 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 [9]. The problem with such an approach, however, is that it is well known [13] 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 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.

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

3 Definitions and Notation

We formally model security of our construction in the Universal Composability framework [3]. Coins are handled by a global ledger $\widehat{\mathcal{L}}(\Delta)$, where Δ is an upper bound for the blockchain delay. We will next present the general notation used in this paper. More details about our model and background on it can be found in the full version of this paper [11].

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 [10,12] 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., [14]). The ECDSA scheme used in Ethereum is believed to satisfy this definition.

3.1 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 [10,12].

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 Sect. 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 $P \in \nu.\mathsf{storage}.\mathsf{users}$ 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 $\gamma.\mathsf{end{-}users} := \{\gamma.\mathsf{Alice}, \gamma.\mathsf{Bob}\}$ and the function $\gamma.\mathsf{other{-}party}$ as $\gamma.\mathsf{other{-}party}(\gamma.\mathsf{Alice}) := \gamma.\mathsf{Bob}$ and $\gamma.\mathsf{other{-}party}(\gamma.\mathsf{Bob}) := \gamma.\mathsf{Alice}$. The attribute $\gamma.\mathsf{cash}$ is a function mapping the set $\gamma.\mathsf{end{-}users}$ to $\mathbb{R}_{\geq 0}$ such that $\gamma.\mathsf{cash}(T)$ is the amount of coins the party $T \in \gamma.\mathsf{end{-}users}$ 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 $\nu.\mathsf{storage}.\mathsf{users} = \gamma.\mathsf{end{-}users}$. We refer to $\gamma.\mathsf{cspace}(cid)$ as the *contract instance with identifiercid 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 $\gamma.\mathsf{dispute} =$ 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 $\gamma.\mathsf{end{-}users}$ 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.[9] 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.[10]

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 $\gamma.\mathsf{other{-}party}$ 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. 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 $\gamma.\mathsf{end{-}users}$ are equal.

We demonstrate the introduced definitions on two concrete examples depicted in Fig. 3. 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 $\gamma.\mathsf{split}(\{P_1, P_4\}) = (\{P_1\}, \{P_4, P_5, P_6\})$.

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

[10] Recall from Sect. 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.

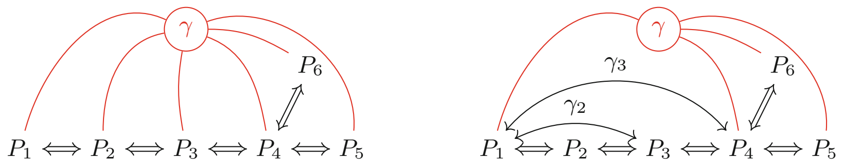

Fig. 3. Examples of multi-party channel setups: A 6-party channel on top of 5 ledger channels (left) and a 4-party channel on 2 ledger and a virtual channel γ_3 (right).

3.2 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 [10].

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 $\gamma.\mathsf{validity} + O(1)$.

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.

		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 $O(1)$		✓	✓	✓

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 [10], 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. 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 Sect. 4.1. As a final step, we define how multi-party channels are created, maintained and closed. This is discussed in Sect. 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 Sect. 3.2. The formal description of the ideal functionality can be found in the full version of this paper [11].

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 [10]), $\mathcal{F}_{dch}(i, \mathcal{C})$ 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,[11] 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 $Q := \gamma.\mathsf{other{-}party}(P)$ 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 $(\text{execute}, \gamma.\mathsf{id}, cid, \tilde{\sigma}, \mathsf{add}, m)$.

[11] 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 $O(\gamma.\mathsf{length} \cdot \Delta)$.

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. Here we provide its high level description and argue that all security and efficiency goals identified in Sect. 3.2 are met.

The ideal functionality $\mathcal{F}_{mpch}(i, \mathcal{C})$ extends the functionality $\mathcal{F}_{dch}(i, \mathcal{C})$, which we described in Sect. 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, 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.

Create and close 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,[12] 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[13] 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.

Update/Execute a contract instance. 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 Sect. 4.1).

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 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 Sect. 5. Thereafter, in Sect. 6, 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. We give a protocol $\Pi_{dch}(i, \mathcal{C}, \pi)$ for building two-party state channels supporting direct dispute which uses three ingredients: (1) a protocol π for virtual state channels with indirect dispute up to length i, which was shown in [10] how to build recursively from subchannels, (2) the ideal functionality $\mathcal{F}_{dch}$

[12] 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 $O(\gamma.\mathsf{length} \cdot \Delta)$.

[13] 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.

for virtual channels with direct dispute up to length $i-1$ and (3) an ideal dispute board. $\Pi_{dch}(i, \mathcal{C}, \pi)$ 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.

The protocol is formally described in the full version of this paper [11]. In particular, there we 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 contract instances whose code is in some set $\mathcal{C}$. On a high level, the functionality models an ideal judge which allows the users to achieve consensus on the latest valid version of a contract instance. For this, $\mathcal{F}_{DB}(\mathcal{C})$ maintains a public "dispute board", which is a list of contract instances available to all parties. $\mathcal{F}_{DB}(\mathcal{C})$ admits two different procedures: *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 version to $\mathcal{F}_{DB}(\mathcal{C})$. The dispute board gives the other party[14] of the contract instance some time to react and send her latest version. $\mathcal{F}_{DB}(\mathcal{C})$ compares both versions and adds the latest valid one to the dispute board. Once a contract instance is registered on the dispute board, a user of the contract instance can execute it via $\mathcal{F}_{DB}(\mathcal{C})$. Upon receiving an execution request, $\mathcal{F}_{DB}(\mathcal{C})$ executes the called function and updates the contract instance on the dispute board according to the outcome. We stress that the other party of the contract instance cannot interfere and merely gets informed about the execution.

Unfortunately, we cannot simply add an ideal dispute board as another hybrid functionality next to one for constructing shorter 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 state sharing by merging both ideal functionalities. Technically, this is done by putting a *wrapper* $\mathcal{W}_{dch}$ around both

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

functionalities, which can be seen just as a piece of code distributing queries to the wrapped functionalities. The formal descriptions of the wrapper as well as the dispute board can be found in the full version of this paper [11].

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 two-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.[15]

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. [11]) 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 $\mathcal{F}_{dch}(i, \mathcal{C}_1)$.*

Remaining technicalities. Remember that our goal is to add direct dispute to a two-party state channel protocol that is itself recursively build from shorter subchannels. We still need to solve two technicalities. Firstly, note that Theorem 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 Theorem 1 recursively, we introduce a technical Lemma 2 which shows how to modify the protocol $\Pi_{dch}(i, \mathcal{C}_0, \pi)$ so that it realizes the wrapped $\mathcal{F}_{dch}$. Secondly, we can apply Theorem 1 on any level *except* for ledger channels. In a nutshell, the reason is that Theorem 1 heavily relies on using subchannels, which simply do not exist in case of ledger channels. Fortunately, this can quite easily be resolved by adding our dispute board to a protocol for ledger channels and to its hybrid ideal functionality. In Lemma 1 we show how to do this with a protocol π_1 from [10]. Their ledger channel protocol assumes an ideal functionality $\mathcal{F}_{scc}$ which models state channel contracts on the blockchain.[16] The description of functionality and protocol wrappers as well as the proofs of both lemmas can be found in the full version of this paper [11].

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.*

[15] For the sake of correctness, in this section we include details about contract sets that 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 can handle all contracts required for building all the longer channels.

[16] Adding the dispute board to any functionality again works by wrapping functionality $\mathcal{F}_x$ and $\mathcal{F}_{DB}$ within a wrapper $\mathcal{W}_x$.

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.*

We finish this section with the complete picture of our approach of building any two-party state channel of length up to 3 (Fig. 4). 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 in the upcoming section.

6 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. Sect. 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, $\texttt{mpVSCC}_i \in \mathcal{C}_1$.

We now discuss case 2 in more detail, by first describing the special contract code $\texttt{mpVSCC}_i$ and then the protocol for multi-party channels. Since case 1 is rather straightforward, we refer the reader to the full version of this paper [11] where also the formal description of our protocol can be found.

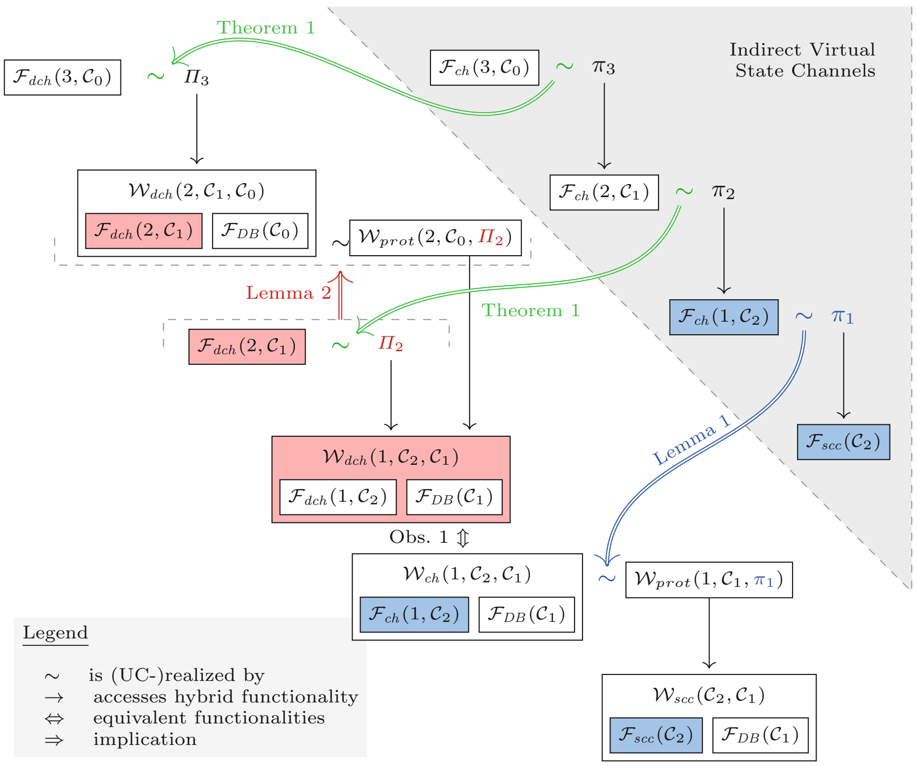
Theorem 1
$\mathcal{F}_{dch}(3, \mathcal{C}_0)$ ∼ Π_3
$\mathcal{F}_{ch}(3, \mathcal{C}_0)$ ∼ π_3
Indirect Virtual State Channels
$\mathcal{W}_{dch}(2, \mathcal{C}_1, \mathcal{C}_0)$
$\mathcal{F}_{dch}(2, \mathcal{C}_1)$
$\mathcal{F}_{DB}(\mathcal{C}_0)$
∼ $\mathcal{W}_{prot}(2, \mathcal{C}_0, \Pi_2)$
$\mathcal{F}_{ch}(2, \mathcal{C}_1)$ ∼ π_2
Lemma 2
Theorem 1
$\mathcal{F}_{dch}(2, \mathcal{C}_1)$ ∼ Π_2
$\mathcal{F}_{ch}(1, \mathcal{C}_2)$ ∼ π_1
$\mathcal{F}_{scc}(\mathcal{C}_2)$
Lemma 1
$\mathcal{W}_{dch}(1, \mathcal{C}_2, \mathcal{C}_1)$
$\mathcal{F}_{dch}(1, \mathcal{C}_2)$
$\mathcal{F}_{DB}(\mathcal{C}_1)$
Obs. 1 ⇕
$\mathcal{W}_{ch}(1, \mathcal{C}_2, \mathcal{C}_1)$
$\mathcal{F}_{ch}(1, \mathcal{C}_2)$
$\mathcal{F}_{DB}(\mathcal{C}_1)$
∼ $\mathcal{W}_{prot}(1, \mathcal{C}_1, \pi_1)$
$\mathcal{W}_{scc}(\mathcal{C}_2, \mathcal{C}_1)$
$\mathcal{F}_{scc}(\mathcal{C}_2)$
$\mathcal{F}_{DB}(\mathcal{C}_1)$
Legend
∼ is (UC-)realized by
→ accesses hybrid functionality
⇔ equivalent functionalities
⇒ implication

Fig. 4. 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**). Theorem 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 Theorem 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.

6.1 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 Sect. 2.3). We denote the code of these instances $\mathtt{mpVSCC}_i$, where $i \in \mathbb{N}$ is the maximal length of the channel in which an instance of $\mathtt{mpVSCC}_i$ can be opened. A contract instance of $\mathtt{mpVSCC}_i$ 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 $\mathtt{mpVSCC}_i$ 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.[17]

The contract has in addition to the mandatory attributes $\mathsf{users}, \mathsf{locked}, \mathsf{cash}$ (see Sect. 3.1) one additional attribute $\mathsf{virtual{-}channel}$ storing the initial version of the multi-party channel γ. The contract has one constructor $\mathtt{Init}_i^{\mathtt{mp}}$ which given a multi-party channel γ and identities of two parties P and Q as input, creates a "copy" of γ as described above. The only contract function, $\mathtt{Close}_i^{\mathtt{mp}}$, is discussed together with the protocol $\Pi_{dch}(i, \mathcal{C}_0, \pi)$ later in this section.

6.2 Protocol Description

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 $\mathtt{mpVSCC}_i$ 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 Sect. 3.1 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 $\mathtt{Init}_i^{\mathtt{mp}}(P, Q, \tau_0, \gamma)$ and contract code $\mathtt{mpVSCC}_i$. Recall that $\mathtt{Init}_i^{\mathtt{mp}}$ is the constructor of the special contract $\mathtt{mpVSCC}_i$. 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 $\mathtt{mpVSCC}_i$. However, note that a party $P \in \gamma.\mathsf{users}$ only has information about subchannels it is part of, i.e. about subchannels $S_P := \{\alpha \in \gamma.\mathsf{subchan} \mid P \in \alpha.\mathsf{end{-}users}\}$. 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 $\mathtt{mpVSCC}_i$ 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 (created, γ) 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 $\tau_0 + 4$.

[17] 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 $\gamma.\mathsf{validity}$.

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 $(\text{update}, id, cid, \tilde{\sigma}, \mathsf{C})$ 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 $Q \in \gamma.\mathsf{other-party}(P)$ 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 $(\text{update−not−ok}, \sigma, w + 2)$ 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[18] 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.

[18] We assume a fixed ordering on peaceful execution requests.

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 $\gamma.\mathtt{cspace}$. In this case, users have to register all contract instances of γ on the dispute board. This guarantees a fixed global view on $\gamma.\mathtt{cspace}$. 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 $\gamma.\mathtt{cspace}$ 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.

Let us emphasize that the high level description provided in this section excludes some technicalities which are explained in the full version of the paper.

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 $\mathcal{F}_{mpch}(i, \mathcal{C}_0)$.*

7 Conclusion

We presented the first full specification and construction of a state channel network that supports multi-party 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 [23]. 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, Polish NCBiR Prokrym project.

References

1. Allison, I.: Ethereum's Vitalik Buterin explains how state channels address privacy and scalability (2016)
2. Bitcoin Wiki: Payment Channels (2018). https://en.bitcoin.it/wiki/Payment_channels

3. Canetti, R.: Universally composable security: a new paradigm for cryptographic protocols. In: 42nd FOCS (2001)
4. Canetti, R., Dodis, Y., Pass, R., Walfish, S.: Universally composable security with global setup. In: Vadhan, S.P. (ed.) TCC 2007. LNCS, vol. 4392, pp. 61–85. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-70936-7_4
5. Celer Network (2018). https://www.celer.network
6. Close, T.: Nitro protocol. Cryptology ePrint Archive, Report 2019/219 (2019). https://eprint.iacr.org/2019/219
7. Counterfactual (2018). https://counterfactual.com
8. Decker, C., Wattenhofer, R.: A fast and scalable payment network with bitcoin duplex micropayment channels. In: Pelc, A., Schwarzmann, A.A. (eds.) SSS 2015. LNCS, vol. 9212, pp. 3–18. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21741-3_1
9. Dolev, D., Strong, H.R.: Authenticated algorithms for Byzantine agreement. SIAM J. Comput. **12**(4), 656–666 (1983)
10. Dziembowski, S., et al.: General state channel networks. In: ACM CCS 2018 (2018)
11. Dziembowski, S., et al.: Multi-party virtual state channels. Cryptology ePrint Archive (2019). https://eprint.iacr.org/2019
12. Dziembowski, S., et al.: Perun: virtual payment hubs over cryptographic currencies. In: Conference Version Accepted to the 40th IEEE Symposium on Security and Privacy (IEEE S&P) 2019 (2017)
13. Garay, J.A., et al.: Round complexity of authenticated broadcast with a dishonest majority. In: 48th FOCS (2007)
14. Katz, J., Lindell, Y.: Introduction to Modern Cryptography (Chapman & Hall/Crc Cryptography and Network Security Series) (2007)
15. Khalil, R., Gervais, A.: NOCUST - a non-custodial 2nd-layer financial intermediary. Cryptology ePrint Archive, Report 2018/642 (2018). https://eprint.iacr.org/2018/642
16. Khalil, R., Gervais, A.: Revive: rebalancing off-blockchain payment networks. In: ACM CCS 2017 (2017)
17. Lind, J., 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 (2018)
18. Malavolta, G., et al.: Concurrency and privacy with payment-channel networks. In: ACM CCS 2017 (2017)
19. McCorry, P., et al.: Pisa: arbitration outsourcing for state channels. Cryptology ePrint Archive, Report 2018/582 (2018). https://eprint.iacr.org/2018/582
20. McCorry, P., et al.: You sank my battleship! A case study to evaluate state channels as a scaling solution for cryptocurrencies (2018)
21. Miller, A., et al.: Sprites: payment channels that go faster than lightning. CoRR (2017)
22. Nakamoto, S.: Bitcoin: A Peer-to-Peer Electronic Cash System (2009). http://bitcoin.org/bitcoin.pdf
23. Poon, J., Buterin, V.: Plasma: Scalable Autonomous Smart Contracts (2017)
24. Poon, J., Dryja, T.: The bitcoin lightning network: scalable off-chain instant payments. Draft version 0.5.9.2 (2016). https://lightning.network/lightning-network-paper.pdf
25. Roos, S., et al.: Settling payments fast and private: efficient decentralized routing for path-based transactions. In: NDSS (2018)
26. Szabo, N.: Smart contracts: building blocks for digital markets. Extropy Mag. (1996)