Foundations of State Channel Networks

Stefan Dziembowski[1,⋆], Sebastian Faust[2,⋆⋆], and Kristina Hostáková[2,⋆⋆]

[1] University of Warsaw, Poland
[2] Technische Universität Darmstadt, Germany

Abstract. One of the main challenges that hinder further adaption of decentralized cryptocurrencies is scalability. Because current cryptocurrencies require that all transactions are processed and stored on a distributed ledger – the so-called blockchain – transaction throughput is inherently limited. An important proposal to significantly improve scalability are *off-chain protocols*, where the massive amount of transactions is executed without requiring the costly interaction with the blockchain. Examples of off-chain protocols include payment channels and networks, which are currently deployed by popular cryptocurrencies such as Bitcoin and Ethereum. A further extension of payment networks envisioned for cryptocurrencies are so-called state channel networks. In contrast to payment networks that only support carrying out off-chain payments between users, state channel networks allow execution of arbitrary complex smart contracts. The main contribution of this work is to give the first full specification for general state channel networks. Moreover, we provide formal security definitions and develop security proofs showing that our construction satisfies security against powerful adversaries. An additional benefit of our construction over most existing payment networks is the use of channel virtualization, which further reduces latency and costs in complex channel networks.

1 Introduction

In recent years we witnessed a growing popularity of distributed cryptocurrencies such as Bitcoin [20] or Ethereum [29]. These systems enable pseudonymous online payments, cheap remittance and many novel applications such as smart contracts, which allow mutually distrusting parties to engage in complex agreements. The underlying main innovation of these currencies is a consensus mechanism that allows special parties – the miners – to maintain the so-called *blockchain*. The blockchain is an append-only ledger on which the transactions of the system are stored, and whose entire content is publicly available, and is checked for consistency by the miners. As the name suggests, blockchain is a chain of data blocks (containing transactions), that are created by the system at some rate. Unfortunately, blockchain-based systems currently face inherent scalability challenges that significantly hinder further adaption. Since each transaction that is processed via the network has to be stored on the blockchain, there is a fundamental limit on how many transactions can be processed per second. For instance, in Bitcoin with its 1MB block size, and a block creation-rate of approx. 10 minutes, the network is currently limited to process up to 7 transactions per second [5].

A natural solution to this problem is to increase the block size, or the block creation rate, but even with these changes it is unlikely that blockchain-based cryptocurrencies can reach the efficiency of centralized payment systems, e.g., the VISA network processes during peak times 56,000 transactions per second [5]. Notice that already at the current transaction rate the blockchain in Bitcoin grows by approximately 5 GB every month – reaching over 150GB in February 2018. Because the miners need to store and verify the entire transaction history, it will be extremely costly to maintain such a system in the long run.

The scalability problem is further amplified by "microtransactions", which is one of the expected "killer applications" when blockchain-based currencies go mainstream [28]. Microtransactions allow users to transfer very small amounts of money, typically less than 1 cent, and can enable many novel business models, e.g., fair sharing of WiFi connection, or devices paying to each other in the "Internet of Things". Besides the scalability issues of microtransactions, there are also several other challenges that need to be addressed by blockchain-based cryptocurrencies before they can handle massive volumes of microtransactions. First, in

[⋆] Supported by the NCN grant 2014/13/B/ST6/03540.

[⋆⋆] Funded by the Emmy Noether Program FA 1320/1-1 of the German Research Foundation (DFG).

many settings microtransactions have to be executed instantaneously (e.g., in the application of sharing the WiFi connection, or when a user wants to read an article on a news webpage). With state-of-the-art cryptocurrencies this is not possible, because current systems require significantly more time to confirm transactions, e.g., in Bitcoin confirmation takes at least around 10 minutes[3]. Secondly, and more importantly, when miners process transactions they can ask for fees. While initially the fees in major cryptocurrencies were relatively low, they are expected to raise when millions of transactions compete for the scarce source of fast processing. Once these fees surpass the actual value assigned to a transaction, micropayments become much less attractive – something which we also witness for traditional online payment systems via credit cards or PayPal. Both systems do not offer real microtransactions due to their fee structure, e.g., PayPal asks for at least 30 cents for each transaction.

Payment and state channels. One of the tools for addressing the above challenges is called *payment channels* [4]. This technique allows two users to rapidly exchange money between each other without sending transactions to the blockchain. This is achieved by keeping the massive bulk of transactions off-chain, and using the blockchain only when parties involved in the payment channel disagree, or when they want to close the channel. Because off-chain transactions can always be fairly settled by the users involved, there is no incentive for them to disagree, and hence honest behavior is enforced. In the normal case, when the two parties involved in the payment channel play honestly, and off-chain transactions never hit the blockchain before the channel is closed, payment channels significantly reduce transaction fees, allow for instantaneous payments and limit the load put on the blockchain.

The concept of payment channels has been extended in several directions. One of the most important extensions are the so-called *payment networks*, which enable users to route transactions via intermediary hubs. To illustrate the concept of payment networks suppose that P_1 has a payment channel with P_2, and P_2 has a payment channel with P_3, while P_1 and P_3 are not directly connected via a channel. A channel network allows P_1 to route payments to P_3 via the intermediate P_2 without the need for P_1 and P_3 to open a direct channel between each other. This reduces the on-chain transaction load even further. An example of such a network using the so-called hash-locked transactions has been designed and implemented by Poon and Dryja over Bitcoin [23]. In a hash-locked based channel network transactions that are sent from P_1 to P_3 have to be confirmed by P_2. At a high-level a transaction is confirmed via the intermediate P_2 by letting P_2 reveal the "lock" of the hash-locked transaction, which technically is achieved by P_2 sending a pre-image of a hash value to P_1. For further details on hash-locked transactions, we refer the reader to, e.g., Lightning network [23].

A further generalization of payment channels are *state channels* [1], which significantly enrich the functionality of payment channels. The users of state channels can, besides payments, execute entire complex smart contracts described in form of self-enforcing programs in an off-chain way. This can, for instance, be used for digital content distribution, online gaming or fast decentralized exchanges. Probably the most prominent project whose final goal is to implement state channels over Ethereum is called *Raiden* [26], but currently it only supports simple payments, and a specification of protocols for full state channel networks has not been provided yet. The main contribution of this work is to address this shortcoming and provide the first construction for building general state channel networks of arbitrary complexity. We next provide further background on state channels, and an overview of our contribution. Further related work is given in the appendix.

State channels and virtual channels. At an informal level a state channel between two parties Alice and Bob provides a method to implement a "virtual 2-party blockchain supporting contracts" in the following sense. Alice and Bob who established a state channel between each other can maintain a "simulated transaction ledger" between themselves and perform the blockchain transactions on it "without registering them on the real blockchain". This happens as long as the parties do not enter into a conflict. The security of this solution comes from the fact that at any time parties can "register" the current off-chain state of the channel on the real blockchain, and let the blockchain fairly finish the execution of a contract. At a technical level state

[3] This is reduced in other cryptocurrencies such as Ethereum, or in Bitcoin via zero-confirmation transactions.

channels (and in fact also payment channels) are implemented using smart contracts. That is, during channel opening the parties deploy a smart contract on the blockchain, which guarantees that during channel closing the money is distributed according to the latest state the parties involved in the channel have agreed on.

Very recently, in [9] it was shown how to virtualize channels using an approach called virtual channels. A virtual channel can be viewed as a result of recursively applying the concept of state channels described above. On a high level, the main observation in [9] is that, since a certain types of state channels between two parties gives us a "virtual blockchain" on top of the real one (denote it by V), then one can further create channels on top of V in a way that is similar to how standard channels are created over the blockchain. Such "second order" channels are called "virtual channels" in [9]. To distinguish the standard channels from the virtual ones, the former ones are also called the *ledger* channels. A virtual channel between P_1 and P_2 will be denoted by $P_1 \leftrightarrow P_2$ and a ledger channel between them will be denoted by $P_1 \Leftrightarrow P_2$.

To explain the idea of [9] in more detail suppose that as in the example already mentioned above P_1 and P_3 are not connected by a ledger channel, but each of them has a ledger channel with an intermediary called P_2. The technique of [9] allows P_1 and P_3 to establish a *virtual payment* channel with the help of P_2 (but without touching the blockchain). Notice that in contrast to creating a ledger channel, a virtual channel can be created just involving the parties P_1, P_2, P_3 and in particular without interacting with the blockchain. The main advantage of virtual channels over the standard "confirmation-based approach" via hash-locked transactions explained above is that as long as everybody is honest, P_1 and P_3 need to interact with P_2 only when the channel is created and when it is closed. Each individual transaction between P_1 and P_3 that goes via this channel does *not* require interacting with P_2.

In [9] the authors construct virtual payment channels of length 2, i.e., for the case described above, where the virtual channel between P_1 and P_3 is created on top of 2 ledger channels. Moreover, the virtual channels that they construct can serve *only for payments*, i.e., they do not have a state and support complex programs. We address these limitations in this work.

1.1 Our contribution

In this paper we construct a system of virtual state channels of arbitrary length, i.e., channels that generalize the construction of [9] in the following sense. Suppose we have m parties, $P_1, \ldots, P_m$ and there is a ledger channel between each pair (P_j, P_{j+1}) of them. Our construction allows to create a virtual channel between P_1 and P_m with $P_2, \ldots, P_{m-1}$ acting as intermediaries. This is done is such a way that each party can be guaranteed that, no matter how the other parties behave, she will not loose her coins locked in the channel. In particular whatever an intermediary P_j has to pay to P_{j+1} (as a result of the execution of our protocol), she is aways ensured to get back the same amount of coins from P_{j-1}.

Our construction is designed by developing a UC-style "state channel theory", in which we develop a formal model for constructing and analyzing state channel networks. The key technique in this theory is a method for combining two (ledger or virtual) channels in order to construct a new virtual state channel. This allows us to construct virtual channels of *arbitrary length* recursively, see Fig. 1 for an example of a channel of length 5 and Section 4.1 for an overview of our modular approach.

Our model is *fully concurrent*, i.e., we allow several virtual channels to be created simultaneously over the same ledger channels. This is possible because our ledger state channels can store and execute several contracts "independently". We provide a rigorous security analysis in a UC-style security model where we analyze our protocols. We believe that our formalization may be of independent interest, especially given the fact that the correct design of state channel networks is very technical and seems to be challenging in real-life (see the discussion on the Raiden system in Appx. A). We moreover emphasize that in the context of cryptocurrencies a sound security analysis is of particular importance, because security flaws have a direct monetary value, and hence unlike in many other settings are guaranteed to be exploited. The later is, e.g., illustrated by the recent attacks on the DAO [25].

While constructing our protocols we will be providing the "optimistic" and "pessimistic" execution times. The "optimistic" ones refer to the standard case when all parties behave honestly. The "pessimistic" case corresponds to the situation when the corrupt parties try to delay the execution as much as they can. We emphasize that we did not attempt to optimize the pessimistic execution times, as this would result in

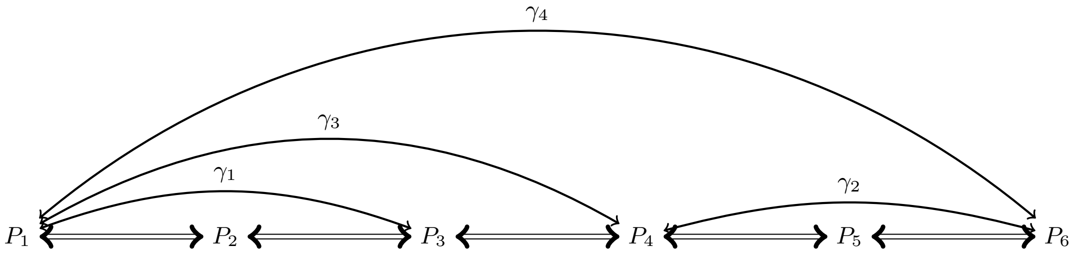

Fig. 1: Example of a recursive construction of a virtual channel γ_4 (of length 5) between P_1 and P_6. First a virtual channel $\gamma_1 := P_1 \leftrightarrow P_3$ is created using ledger channels $P_1 \Leftrightarrow P_2$ and $P_2 \Leftrightarrow P_3$. Then a virtual channel $\gamma_2 := P_4 \leftrightarrow P_6$ is created using ledger channels $P_4 \Leftrightarrow P_5$ and $P_5 \Leftrightarrow P_6$. The other virtual channels are created recursively, as follows: channel $\gamma_3 := P_1 \leftrightarrow P_3$ is created using the virtual channel γ_1 and the ledger channel $P_3 \Leftrightarrow P_4$, and channel $\gamma_4 := P_1 \leftrightarrow P_6$ is created using virtual channels γ_3 and γ_2.

making our protocols even more complicated. An important direction for future work is to fine-tune our construction to optimize the optimistic and pessimistic timings.

2 The model

We follow the model of [9] which is closely related to the one used on works of Kumaresan and Bentov [2, 15, 14, 16] on modeling protocols that operate with *coins*[4] using a synchronous version of the (simplified) UC framework [6, 7]. More precisely, an *n-party protocol* π is run between parties $P_1, \ldots, P_n$, which are modeled as interactive poly-time Turing machines (ITMs). We assume that the P_i's are connected by secure (secret and authentic) communication channels. A protocol is executed in the presence of an *adversary* $\mathcal{A}$ and an *environment* $\mathcal{Z}$ (which are also interactive poly-time Turing machines). The adversary can *corrupt* any party P_i, by which we mean that he takes full control over P_i. We consider static corruption, i.e., the environment $\mathcal{Z}$ can decide at the beginning of the protocol execution which parties to corrupt. In addition to the above entities, a protocol can *have access to ideal functionalities* $\mathcal{G}_1, \ldots, \mathcal{G}_m$ (which are also ITMs), by which we mean that all entities can interact with them. In this case we say that the protocol *works in the* $(\mathcal{G}_1, \ldots, \mathcal{G}_m)$-*hybrid model*. One important difference to the standard UC model is that our ideal functionalities (and also parties) will have to deal with coins. We model this with a special functionality $\mathcal{L}$ which we will describe in more detail below.

Communication model. Since we assume a synchronous communication network, the execution of the protocol happens in rounds (see, e.g, [10, 11, 21, 13] for a formalization of this model and its relation to the model with real time). We assume that if in round i a party sends a message to another party, then it arrives to it at the beginning of round $i + 1$. The adversary is *rushing*, i.e., he can decide about the order in which the messages arrive in a given round.

The parties, the ideal functionalities, the environment and the adversary are always aware of a given round (in practice one can think of it as equipping them with clocks that are synchronized). Let $\mathsf{time} := \mathbb{N} \cup \{\infty\}$ denote the set of all possible round numbers. Whenever we say that some operation (e.g. delivering a message or simply staying in idle state) *takes time at most* $\tau \in \mathsf{time}$ we mean that it is up to the adversary to decide how long this operation takes (as long as it takes at most τ rounds). For simplicity we assume that computation takes no time and is "atomic". The communication between two P_i's takes one round. All other communication – in particular, between the adversary $\mathcal{A}$ and the environment $\mathcal{Z}$ – takes no time.

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

Handling coins. Following [9], the money mechanics is modeled using a special functionality $\mathcal{L}$ that keeps track on how much money the parties have. In some sense it is similar in spirit to the model of Bentov and Kumaresan [2] that model money as a special resource, and the money transfers using a special keyword "coins". Unlike [2], we define the state of the user's accounts as an explicit vector of non-negative (finite precision) real numbers $(x_1, \ldots, x_n)$, where each x_i is the amount of coins that P_i has.[5] The vector is maintained by a special functionality $\mathcal{L}$ (see Fig. 2) which can be realized by a cryptocurrency, for instance Ethereum or Bitcoin.

The state of this functionality is public, i.e., $P_1, \ldots, P_n, \mathcal{Z}$, and $\mathcal{A}$ can freely read all its contents. The functionality $\mathcal{L}$ is initiated by the environment $\mathcal{Z}$ that can also freely add and remove money in user's accounts, via the operations add and remove. The parties $P_1, \ldots, P_n$ *cannot* directly perform any such operations on $\mathcal{L}$. On the other hand, we will have special functionalities that can perform operations on $\mathcal{L}$ (and hence, indirectly, P_i's can also modify $\mathcal{L}$, in a way that is "controlled" by the special functionalities). Every time a special functionality issues an add or remove command, this command is delivered to $\mathcal{L}$ and can be delayed by at most Δ rounds (for some parameter Δ), i.e., the message is given to the adversary who can decide when it arrives to $\mathcal{L}$ (as long as the delay is at most Δ rounds). Special functionalities with access to $\mathcal{L}$ (that can be delayed by Δ rounds) will be denoted with a superscript $\mathcal{L}(\Delta)$ (e.g.: $\mathcal{F}^{\mathcal{L}(\Delta)}$).

By saying that a special functionality *added y coins to P_i's account in ledger $\mathcal{L}$ (with session id sid)* we mean that the special functionality issued a query $(\mathsf{add}, sid, P_i, y)$ to $\mathcal{L}$. Analogously, by saying that a special functionality *removed $y_{i_1}, \ldots, y_{i_t}$ coins from the accounts of $P_{i_1}, \ldots, P_{i_t}$ (respectively) in ledger $\mathcal{L}$ (with session id sid)* we mean the special functionality issued a query $(\mathsf{remove}, sid, \{(P_{i_j}, y_{i_j})\}_{i=1}^{t})$ to $\mathcal{L}$. For all the above instructions, if $\mathcal{L}$ replies with a message $(\mathsf{nofunds}, sid)$ then we say that *the operation has not been performed due to insufficient funds.*

Execution of ideal/real processes. We now describe the process in the ideal/real world and in particular the order of activation. In each round of the protocol/real-world execution the environment $\mathcal{Z}$ is activated first, where the environment $\mathcal{Z}$ can add/remove coins from the ledger via add and remove instructions. Next, $\mathcal{Z}$ provides inputs for the honest parties and for the adversary. Then, it activates the adversary (in the hybrid world), or the simulator (in the ideal world).

Let π be a protocol working in the $\mathcal{G}^{\mathcal{L}(\Delta)}$-hybrid model (where $\mathcal{G}^{\mathcal{L}(\Delta)}$ is a special functionality that has access to the ledger $\mathcal{L}$). The output of an environment $\mathcal{Z}$ interacting with a protocol π and an adversary $\mathcal{A}$ on input 1^λ and auxiliary input z is denoted as $\textsc{exec}^{\mathcal{G}^{\mathcal{L}(\Delta)}}_{\pi,\mathcal{A},\mathcal{Z}}(\lambda, z)$. If π is a trivial protocol in which simply the parties forward their inputs to a special functionality $\mathcal{F}^{\mathcal{L}(\Delta)}$, then we will call the adversary a *simulator* $\mathcal{S}$ and denote the above output as $\textsc{ideal}_{\mathcal{F}^{\mathcal{L}(\Delta)},\mathcal{S},\mathcal{Z}}(\lambda, z)$. It will also be useful to restrict the power of the environment, so that, e.g, $\mathcal{Z}$ will not be allowed to send some inputs to the honest parties in certain moments. For example $\mathcal{Z}$ will be forbidden to initiate some protocol when the parties do not have enough coins for this, or to instruct one party to start a protocol without instructing the other party to start the protocol as well. In this case, we will say that $\mathcal{Z}$ is from some class of environments $\mathcal{E}_{res}$. We will formally define the restrictions that we put on the environment in Appx. C and in Appx. E will argue why it will not affect composability.

Definition 1. *Let $\mathcal{E}_{res}$ be some set of restricted environments $\mathcal{Z}$. We say that a protocol π working in a $\mathcal{G}^{\mathcal{L}(\Delta)}$-hybrid model* emulates a special functionality $\mathcal{F}^{\mathcal{L}(\Delta)}$ against environments from class $\mathcal{E}_{res}$ *if for every adversary $\mathcal{A}$ there exists a simulator $\mathcal{S}$ such that for every environment $\mathcal{Z} \in \mathcal{E}_{res}$ we have*

$$\{\textsc{exec}^{\mathcal{G}^{\mathcal{L}(\Delta)}}_{\pi,\mathcal{A},\mathcal{Z}}(\lambda, z)\}_{\lambda\in\mathbb{N},z\in\{0,1\}^*} \overset{c}{\approx} \{\textsc{ideal}_{\mathcal{F}^{\mathcal{L}(\Delta)},\mathcal{S},\mathcal{Z}}(\lambda, z)\}_{\lambda\in\mathbb{N},z\in\{0,1\}^*}.$$

Session identifiers. To simplify the exposition, we omit the session identifiers and the sub-session identifiers (in other works typically denoted with *sid* and *ssid*, respectively). Instead, we will use expressions like "message m is a reply to message m'" (technically, this would be handled by adding the identifiers to the message). We believe that this approach helps the readability, and does not lead to confusion.

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

Setup assumptions. To enable a simpler exposition we assume that before the protocol starts the following public-key infrastructure setup phase is executed by some trusted party: (1) For every $i = 1, \ldots, n$ let $(pk_{P_i}, sk_{P_i}) \leftarrow_{\$} \mathsf{KGen}(1^\lambda)$, (2) For every $i = 1, \ldots, n$ send $(sk_{P_i}, (pk_{P_1}, \ldots, pk_{P_n}))$ to P_i. The tuple $\Pi := (pk_{P_1}, \ldots, pk_{P_n})$ will also be called the *public key tuple*. We emphasize that the use of a PKI is only an abstraction, and can easily be realized using the blockchain.

Functionality $\mathcal{L}$

Functionality $\mathcal{L}$, running with parties $P_1, \ldots, P_n$ and the environment $\mathcal{Z}$, gets as input $(x_1, \ldots, x_n) \in \mathbb{R}^n_{\geq 0}$ (where $\mathbb{R}_{\geq 0}$ are finite-precision non-negative reals). It stores the vector $(x_1, \ldots, x_n)$ and accepts queries of following types:

Adding money

Upon receiving a message $(\mathsf{add}, sid, P_i, y)$ from $\mathcal{Z}$ (for $y \in \mathbb{R}_{\geq 0}$):

Let $x_i := x_i + y$. We say that *y coins are added to P_i's account in $\mathcal{L}$.*

Removing money

Upon receiving a message $\left(\mathsf{remove}, sid, \{(P_{i_j}, y_{i_j})\}_{j=1}^{t}\right)$ (for some $t \in \{1, \ldots, n\}$) and $y_{i_j} \in \mathbb{R}_{\geq 0}$):

- Check if *for every $j \in \{1, \ldots, t\}$* we have that $x_{i_j} \geq y_{i_j}$; if not then reply with a message $(\mathsf{nofunds}, sid)$ and stop.
- Otherwise for $j \in \{1, \ldots, t\}$ let $x_{i_j} := x_{i_j} - y_{i_j}$. We say that $y_{i_1}, \ldots, y_{i_t}$ *coins were removed from the accounts of $P_{i_1}, \ldots, P_{i_t}$ (resp.) in $\mathcal{L}$.*

Fig. 2: The ledger functionality $\mathcal{L}$.

3 Definitions and notation

We assume that all values like real and natural numbers, poly-time computable functions, tuples of values, etc. are implicitly encoded as binary strings (e.g. when they are sent as messages). We will also use *keywords* which (formally) represent some fixed binary strings. We will frequently present tuples of values using the following convention. The individual values in a tuple T are identified using keywords called *attributes*: $\mathsf{attr1}, \mathsf{attr2}, \ldots$. Strictly speaking an *attribute tuple* is a function from its set of attributes to $\{0,1\}^*$. The *value of an attribute 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. For example when we say that "we add an attribute attr to T and set it to x" it means that T is replaced by T' with and additional attribute attr and $T'.\mathsf{attr} = x$.

3.1 Contract

In this section we introduce the formal terminology concerning smart contracts. We consider contracts between just two parties, as this is the only type of contract that we need in this paper, but our terminology can easily be generalized to contracts between larger groups of users.

A *contract storage* over a set of parties $\mathcal{P}$ is an attribute tuple σ that contains at least the following attributes: $\sigma.\mathsf{user}_\mathsf{L}, \sigma.\mathsf{user}_\mathsf{R} \in \mathcal{P}$ that denote the users that are involved in the contract storage and $\sigma.\mathsf{cash}\colon \{\sigma.\mathsf{user}_\mathsf{L}, \sigma.\mathsf{user}_\mathsf{R}\} \to \mathbb{R}$ that contains information about the amounts of coins that the users invested in the contract storage.

A *contract type* over a set of parties $\mathcal{P}$ is a tuple $\mathtt{C} = (\Lambda, g_1, \ldots, g_r, f_1, \ldots, f_s)$, where Λ is a (possibly infinite) set of contract storages over $\mathcal{P}$ called *admissible contract storages*, $g_1, \ldots, g_r$ are *contract constructors* and $f_1, \ldots, f_s$ are *contract functions*. Each g_i is a poly-time computable function that takes as input a tuple (P, τ, z), with $P \in \mathcal{P}, \tau \in \mathsf{time}$, and $z \in \{0,1\}^*$, and produces as output an admissible contract storage or a special symbol $\perp$ (in which case we say that the storage construction failed). Each f_i is poly-time computable function that takes as input a tuple (σ, P, τ, z), with σ being an admissible contract storage, $P \in \{\sigma.\mathsf{user_L}, \sigma.\mathsf{user_R}\}$, $\tau \in \mathsf{time}$ and $z \in \{0,1\}^*$, and outputs a tuple $(\tilde{\sigma}, add_L, add_R, m)$, where $\tilde{\sigma}$ is an admissible contract storage, values $add_L, add_R \in \mathbb{R}_{\geq 0}$ correspond to the amount of coins that were *unlocked* from the contract storage, and an *output message* $m \in \{0,1\}^* \cup \{\perp\}$. If the output message is $\perp$, we say that the execution *failed* (we assume that the execution always fails if a function is executed on input that does not satisfy the constraints described above, e.g., it is applied to σ that is not admissible). If the output message $m \neq \perp$, then we require that the attributes $\mathsf{user_L}$ and $\mathsf{user_R}$ in $\tilde{\sigma}$ are identical to those in σ. In addition, if $add_L + add_R \geq 0$, then it must hold that $add_L + add_R \leq \sigma.\mathsf{cash}(\sigma.\mathsf{user}_L) - \tilde{\sigma}.\mathsf{cash}(\sigma.\mathsf{user}_L) + \sigma.\mathsf{cash}(\sigma.\mathsf{user}_R) - \tilde{\sigma}.\mathsf{cash}(\sigma.\mathsf{user}_R)$.

A *contract instance* is an attribute tuple ν with attributes $\mathsf{storage}$ and type, where $\nu.\mathsf{type} = (\Lambda, g_1, \ldots, g_r, f_1, \ldots, f_s)$ is a contract type, and $\nu.\mathsf{storage} \in \Lambda$ is a contract storage.

3.2 State channels

We now present our notation for the state channels. It is essentially an extension of the notation used in [9] for payment channels.

Ledger state channel. Formally, a ledger state channel γ over a set of parties $\mathcal{P}$ is defined as an attribute tuple $\gamma := (\gamma.\mathsf{id}, \gamma.\mathsf{Alice}, \gamma.\mathsf{Bob}, \gamma.\mathsf{cash}, \gamma.\mathsf{cspace})$. We call the attribute $\gamma.\mathsf{id} \in \{0,1\}^*$ the identifier of the ledger state channel. Attributes $\gamma.\mathsf{Alice} \in \mathcal{P}$ and $\gamma.\mathsf{Bob} \in \mathcal{P}$ are the identities of parties using the ledger state channel γ. For convenience, we define the set $\gamma.\mathsf{end\text{–}users} := \{\gamma.\mathsf{Alice}, \gamma.\mathsf{Bob}\}$ and the function $\gamma.\mathsf{other\text{–}party}$ as $\gamma.\mathsf{other\text{–}party}(\gamma.\mathsf{Alice}) := \gamma.\mathsf{Bob}$ and $\gamma.\mathsf{other\text{–}party}(\gamma.\mathsf{Bob}) := \gamma.\mathsf{Alice}$. The attribute $\gamma.\mathsf{cash}$ is a function mapping the set $\gamma.\mathsf{end\text{–}users}$ to $\mathbb{R}_{\geq 0}$ such that $\gamma.\mathsf{cash}(T)$ is the amount of coins the party $T \in \gamma.\mathsf{end\text{–}users}$ has locked in the ledger state channel γ. Finally, the attribute $\gamma.\mathsf{cspace}$ is a partial function that takes as input a contract instance identifier $cid \in \{0,1\}^*$ and outputs a contract instance ν such that $\{\nu.\mathsf{storage.user}_L, \nu.\mathsf{storage.user}_R\} = \gamma.\mathsf{end\text{–}users}$. We will refer to $\gamma.\mathsf{cspace}(cid)$ as the *contract instance with identifier cid in the ledger state channel* γ.

We also define a function $\mathtt{Value}$ which on input ledger state channel γ outputs the sum of coins locked in the ledger state channel. More precisely, $\mathtt{Value}(\gamma) := \gamma.\mathsf{cash}(\gamma.\mathsf{Alice}) + \gamma.\mathsf{cash}(\gamma.\mathsf{Bob}) + \sum_{\substack{cid \in \mathbb{N} \\ \gamma.\mathsf{cspace}(cid) \neq \perp}} (c_L^{cid} + c_R^{cid})$, where $c_L^{cid} := \sigma.\mathsf{cash}(\sigma.\mathsf{user}_L)$ and $c_R^{cid} := \sigma.\mathsf{cash}(\sigma.\mathsf{user}_R)$ for $\sigma := \gamma.\mathsf{cspace}(cid).\mathsf{storage}$. In order to update the contract instances off-chain, the users of the ledger state channel will store some additional information in their local copies of γ. To this end, we introduce the following terminology. A *contract instance version* is an attribute tuple ν that in addition to the attributes of contract instance has an attribute $\nu.\mathsf{version} \in \mathbb{N}$. A *contract instance version signed by* $P \in \mathcal{P}$ additionally contains an attribute $\nu.\mathsf{sign}$ which is a signature of P on $(\nu.\mathsf{storage}, \nu.\mathsf{type}, \nu.\mathsf{version})$. In case $\nu.\mathsf{version} = 0$ we allow $\nu.\mathsf{sign} = \perp$. An attribute tuple γ is a *ledger state channel's private version of a party P* if it is defined as the normal ledger state channel, except that every $\gamma.\mathsf{cspace}(cid)$ is a contact instance version signed by $\gamma.\mathsf{other\text{-}party}(P)$.

Virtual state channel. Formally, a virtual state channel γ over a set of parties $\mathcal{P}$ is defined as a tuple $\gamma := (\gamma.\mathsf{id}, \gamma.\mathsf{Alice}, \gamma.\mathsf{Bob}, \gamma.\mathsf{Ingrid}, \gamma.\mathsf{subchan}, \gamma.\mathsf{cash}, \gamma.\mathsf{cspace}, \gamma.\mathsf{length}, \gamma.\mathsf{validity})$. The attributes $\gamma.\mathsf{id}$, $\gamma.\mathsf{Alice}$, $\gamma.\mathsf{Bob}$, $\gamma.\mathsf{cash}$ and $\gamma.\mathsf{cspace}$, are defined as in the case of a ledger state channel. The same holds for the set $\gamma.\mathsf{end\text{–}users}$ and the functions $\gamma.\mathsf{other\text{–}party}$ and $\mathtt{Value}$. The new attribute $\gamma.\mathsf{Ingrid} \in \mathcal{P}$ denotes the identity of the intermediary of the virtual state channel. The attribute $\gamma.\mathsf{subchan}$ is a function mapping the set $\gamma.\mathsf{end\text{–}users}$ to $\{0,1\}^*$. The value $\gamma.\mathsf{subchan}(\gamma.\mathsf{Alice})$ (resp. $\gamma.\mathsf{subchan}(\gamma.\mathsf{Bob})$) equals the identifier of the state channel between $\gamma.\mathsf{Alice}$ and $\gamma.\mathsf{Ingrid}$ (resp. $\gamma.\mathsf{Ingrid}$ and $\gamma.\mathsf{Bob}$). We call the state channels $\gamma.\mathsf{subchan}(\gamma.\mathsf{Alice})$ and $\gamma.\mathsf{subchan}(\gamma.\mathsf{Bob})$ the subchannels of the virtual state channel γ. The attribute $\gamma.\mathsf{validity}$ denotes the round in which the virtual state channel will be closed. And finally, the attribute $\gamma.\mathsf{length} \in \mathbb{N}_{>1}$ refers to the length of the virtual state channel, i.e., the number of ledger state channels over which it is built. For example the state channels on Figure 1 (see Page 4) have the following lengths: $\gamma_1.\mathsf{length} = 2$, $\gamma_2.\mathsf{length} = 2$, $\gamma_3.\mathsf{length} = 3$, $\gamma_4.\mathsf{length} = 5$. Sometimes it will be convenient to refer to ledger state channels as to state channels of length one. Formally, this would require to define ledger state channels such they additionally contain the attribute length whose only possible value would be 1.

Each entity (ideal functionality or party in a protocol), stores and maintains a set of all state channels it is aware of. This set will be called *channel space* and denoted $\varGamma$.

3.3 Abbreviated notation

In order to simplify the notation in the description of ideal functionalities and protocols, we fix the following abbreviated notation. When it is clear from the context which state channel γ we are talking about, we will denote the parties of the state channel $A := \gamma.\mathsf{Alice}$, $B := \gamma.\mathsf{Bob}$ and in case γ is a virtual state channel $I := \gamma.\mathsf{Ingrid}$. In addition, for a party $P \in \gamma.\mathsf{end{-}users}$, we will always denote the other end-user of the state channel by Q, i.e. $Q := \gamma.\mathsf{other{-}party}(P)$.

When we want to emphasize that we are referring to a local version of a state channel stored by some entity T, we add T to the superscript. So for instance, $\gamma^T := \varGamma^T(id)$ denotes T's local version of the state channel γ as stored in T's channel space $\varGamma^T$. We also introduce symbolic notation for sending and receiving messages. Instead of the instruction "Send the message msg to party P in round τ", we write $msg \overset{\tau}{\hookrightarrow} P$. Instead of the instruction "Send the message msg to all parties in the set $\gamma.\mathsf{end{-}users}$ in round τ", we write $msg \overset{\tau}{\hookrightarrow} \gamma.\mathsf{end{-}users}$. By $msg \overset{\tau}{\hookleftarrow} P$ we mean that an entity (party in a protocol, ideal functionality, simulator etc.) receives a message msg from party P in round τ. And we use $msg \xhookleftarrow{\tau \leq \tau_1} P$ when an entity receives a message msg from party P until round τ_1.

In the protocols and ideal functionalities, entities will frequently update their local versions of a state channel stored in their channel space. Therefore, we define two local update procedure which will shorten the descriptions later in this work. The purpose of the first procedure, `LocalUpdate`, is to update the contract instance and automatically adjust money distribution in the state channel. The other procedure, `LocalUpdateAdd`, also updates the contract instance but in contrast to `LocalUpdate` it gets the amount of money that shall be added back to the state channel explicitly in its input.

```
LocalUpdate($\varGamma, id, cid, \tilde{\sigma}, \mathtt{C}$)

Let $\gamma := \varGamma(id)$ and $\sigma := \gamma.\mathsf{cspace}(cid).\mathsf{storage}$. If $\sigma = \perp$, the set $(x_A, x_B) := (0, 0)$. Else set $(x_A, x_B) := (\sigma.\mathsf{cash}(\gamma.\mathsf{Alice}), \sigma.\mathsf{cash}(\gamma.\mathsf{Bob}))$. Make the following updates:
  1. Add $x_A - \tilde{\sigma}.\mathsf{cash}(\gamma.\mathsf{Alice})$ coins to $\gamma.\mathsf{cash}(\gamma.\mathsf{Alice})$
  2. Add $x_B - \tilde{\sigma}.\mathsf{cash}(\gamma.\mathsf{Bob})$ coins to $\gamma.\mathsf{cash}(\gamma.\mathsf{Bob})$
  3. Set $\gamma.\mathsf{cspace}(cid)$ equal to the tuple $(\tilde{\sigma}, \mathtt{C})$.
Output $\varGamma$ with the updated contract instance $cid$ in the state channel $\gamma$.
```

```
LocalUpdateAdd($\varGamma, id, cid, \tilde{\sigma}, \mathtt{C}, add_A, add_B$)

Let $\gamma := \varGamma(id)$ and $\sigma := \gamma.\mathsf{cspace}(cid).\mathsf{storage}$. Make the following updates:
  1. Add $add_A$ coins to $\gamma.\mathsf{cash}(\gamma.\mathsf{Alice})$
  2. Add $add_B$ coins to $\gamma.\mathsf{cash}(\gamma.\mathsf{Bob})$
  3. Set $\gamma.\mathsf{cspace}(cid)$ equal to the tuple $(\tilde{\sigma}, \mathtt{C})$.
Output $\varGamma$ with the updated contract instance $cid$ in the state channel $\gamma$.
```

Analogously, we define both `LocalUpdate` and `LocalUpdateAdd` in case a party wants to update the private extended version of the contract instance. Notice that in this case procedures will take additional two parameters: the new version number and the signatures created by the parties.

To further simplify the description of the ideal functionalities and the protocols, we will use two "timing functions" TimeExecute(i) and TimeRegister(i). Informally, these functions represent the maximal number of rounds it takes to execute/register a contract instance in a state channel of length $i > 0$. See Section 7.1 for formal definition of these functions.

4 Ideal functionalities for state channels

In this section, we describe the ideal functionality that defines how ledger state channels and virtual state channels are created, maintained and closed. We denote this ideal functionality $\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i, \mathcal{C})$, where $i \in \mathbb{N}$ is the maximal length of a channel that can be opened via the functionality, and $\mathcal{C}$ denotes the set of contract types that can be open in the channels. The ideal functionality $\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i, \mathcal{C})$ communicates with parties from the set $\mathcal{P}$, it has access to the global ideal functionality $\mathcal{L}$ (the ledger). The functionality maintains a channel space, that we denote Γ, containing all the open state channels. The set Γ is initially empty.

Since inputs of parties and the messages they send to the ideal functionality do not contain any private information, we can implicitly assume that the ideal functionality forwards all messages it receives to the simulator $\mathcal{S}$ via the leakage port. The task of the simulator is to instruct the ideal functionality via the leakage port to make changes on the ledger and to output messages to the parties in the correct round (this depends on the choice made by the adversary $\mathcal{A}$ in the real world). We will not explicitly mention the instructions for making changes at the ledger. By saying "wait for at most Δ rounds to remove/add x coins from P's account on the ledger" we mean that the ideal functionality should wait until it is instructed by the simulator, which will happen within Δ rounds, and then request changes of party P's account on the ledger.

Below we first present the formal definition of the $\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i, \mathcal{C})$ functionality and then explain it informally. During the high level explanation, we will also informally introduce the restrictions on the environment (see Section 2) whose full list is given in Appx. C.

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

This functionality accepts messages from parties $\mathcal{P} := \{P_1, \ldots, P_n\}$. We use the abbreviated notation defined in Section 3.3.

Create a ledger channel

Upon $(\text{create}, \gamma) \xhookleftarrow{\tau_0} A$ where γ is a ledger channel:

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

Create a virtual channel

1. In this procedure the ideal functionality waits to receive (create, γ) message from all parties in $\gamma.\mathsf{end\text{–}users} \cup \{\gamma.\mathsf{Ingrid}\}$ (the procedure starts when first such a message is received). These messages are recorded, and additionally the coins from γ's subchannels are removed according to the following rules:
   - Upon $(\text{create}, \gamma) \hookleftarrow P$ where $P \in \gamma.\mathsf{end\text{–}users}$:
     If you have not yet received the message (create, γ) from $\gamma.\mathsf{Ingrid}$, then remove $\gamma.\mathsf{cash}(P)$ coins from P's balance in $\gamma.\mathsf{subchan}(P)$ and $\gamma.\mathsf{cash}(\gamma.\mathsf{other\text{-}party}(P))$ coins from $\gamma.\mathsf{Ingrid}$'s balance in $\gamma.\mathsf{subchan}(P)$. Otherwise do nothing.
   - Upon $(\text{create}, \gamma) \hookleftarrow \gamma.\mathsf{Ingrid}$, then for both $P \in \gamma.\mathsf{end\text{–}users}$:

If you have not yet received (create, γ) from P then remove $\gamma.\mathsf{cash}(P)$ coins from P's balance in $\gamma.\mathsf{subchan}(P)$, and $\gamma.\mathsf{cash}(\gamma.\mathsf{other\text{-}party}(P))$ coins from $\gamma.\mathsf{Ingrid}$'s balance in $\gamma.\mathsf{subchan}(P)$. Otherwise do nothing.

2. If within 3 rounds you record (create, γ) from all users in $\gamma.\mathsf{end\text{–}users} \cup \{\gamma.\mathsf{Ingrid}\}$, then define $\varGamma(\gamma.\mathsf{id}) := \gamma$, send $(\text{created}, \gamma) \hookrightarrow \gamma.\mathsf{end\text{–}users}$ and wait for channel closing in Step 4 (in the meanwhile accepting the update and execute messages that concern γ).
3. Otherwise wait till round $\gamma.\mathsf{validity}$. Then within time $\text{TimeRegister}(j) + 2 \cdot \text{TimeExecute}(\lceil j/2 \rceil)$ rounds, where $j := \gamma.\mathsf{length}$, refund the coins that you removed from the subchannels in Step 1.

Close virtual channel

4. In the round $\gamma.\mathsf{validity} + \text{TimeRegister}(j) + 2 \cdot \text{TimeExecute}(\lceil j/2 \rceil)$ proceed as follows. Let $\hat{\gamma}$ be the current version of the channel, i.e. $\hat{\gamma} := \varGamma(\gamma.\mathsf{id})$ and let $\hat{c}_A := \hat{\gamma}.\mathsf{cash}(A)$ and $\hat{c}_B := \hat{\gamma}.\mathsf{cash}(B)$.
5. Add $\hat{c}_A$ coins to $\gamma.\mathsf{Alice}$'s balance and $\hat{c}_B$ coins to $\gamma.\mathsf{Ingrid}$'s balance in $\gamma.\mathsf{subchan}(\gamma.\mathsf{Alice})$. Add $\hat{c}_A$ coins to $\gamma.\mathsf{Ingrid}$'s balance and $\hat{c}_B$ coins to $\gamma.\mathsf{Bob}$'s balance in $\gamma.\mathsf{subchan}(\gamma.\mathsf{Bob})$.
6. Erase $\hat{\gamma}$ from $\varGamma$ and send $(\text{closed}, \gamma.\mathsf{id}) \hookrightarrow \gamma.\mathsf{end\text{–}users}$.

Update contract instance

Upon $(\text{update}, id, cid, \tilde{\sigma}, \mathtt{C}) \xhookleftarrow{\tau_0} P$:

1. Let $\gamma := \varGamma(id)$ and $j = \gamma.\mathsf{length}$. If $P \notin \gamma.\mathsf{end\text{–}users}$ then stop.
2. Send $(\text{update–requested}, id, cid, \tilde{\sigma}, \mathtt{C}) \xhookrightarrow{\tau_0+1} \gamma.\mathsf{other\text{–}party}(P)$.
3. Let $T = \tau_0 + \text{TimeRegister}(j) + 1$. If $(\text{update–reply}, ok, id, cid) \xhookleftarrow{\tau_1 \leq T} \gamma.\mathsf{other\text{–}party}(P)$, then set $\varGamma := \mathtt{LocalUpdate}(\varGamma, id, cid, \tilde{\sigma}, \mathtt{C})$ and send $(\text{updated}, id, cid) \xhookrightarrow{\tau_1+1} \gamma.\mathsf{end\text{–}users}$ and stop. Otherwise stop.

Execute contract instance

Upon $(\text{execute}, id, cid, f, z) \xhookleftarrow{\tau_0} P$:

1. Let $\gamma := \varGamma(id)$ and $j = \gamma.\mathsf{length}$. If $P \notin \gamma.\mathsf{end\text{–}users}$ then stop.
2. If $j = 1$, then set $T = \tau_0 + 4\varDelta + 5$. Else set $T = \tau_0 + 14 \cdot \text{TimeExecute}(\lceil j/2 \rceil) + 10$.
3. In round $\tau_2 \leq T$, let $\gamma := \varGamma(id)$, $\nu := \gamma.\mathsf{cspace}(cid)$, and $\sigma := \nu.\mathsf{storage}$.
4. Compute $(\tilde{\sigma}, add_L, add_R, m) := f(\sigma, P, \tau_0, z)$. If $m = \perp$, then stop. Else set $\varGamma := \mathtt{LocalUpdate}$ $\mathtt{Add}(\varGamma, id, cid, \tilde{\sigma}, \nu.\mathsf{type}, add_L, add_R)$, send $(\text{executed}, id, cid, \tilde{\sigma}, add_L, add_R, m) \xhookrightarrow{\tau_2} \gamma.\mathsf{end\text{–}users}$ and stop.

Close ledger channel

Upon $(\text{close}, id) \xhookleftarrow{\tau_0} P$, let $\gamma = \varGamma(id)$. If $P \notin \gamma.\mathsf{end\text{–}users}$ then stop. Otherwise wait at most $7\varDelta$ rounds. Then distinguish the following two cases:

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

Let us now provide some intuitions behind this definition. The $\mathcal{F}_{ch}^{\mathcal{L}(\varDelta)}(i, \mathcal{C})$ functionality consists of two "channel opening" procedures: one for ledger channels and one for virtual channels. The ledger channel opening procedure starts with a "create" message from A (without loss of generality we assume that A always initiates the opening process). As a result of this, the functionality removes the coins that A wants to deposit in the channel from A's account on the ledger, and waits for B to also declare that he wants to open the channel. If this happens within time $\varDelta$ then also B's coins are removed form the ledger and the channel is created (which is communicated to the parties with the "created" message), otherwise A can get her money back by sending a "refund" message.

The opening procedure for virtual channel γ works slightly differently since its effects are visible on the subchannels of γ. It works as follows. The intention to open γ is expressed by $P \in \gamma.\mathsf{end\text{–}users} \cup \{\gamma.\mathsf{Ingrid}\}$ by sending a "create" message to the functionality. Once such a message is received from P, the coins that are needed to create γ are locked immediately on *both* sides of the subchannel, i.e. if $P \in \gamma.\mathsf{end\text{–}users}$ sends the "create" message then also the money of $\gamma.\mathsf{Ingrid}$ is locked in her channel with P (and symmetrically when $\gamma.\mathsf{Ingrid}$ sends the "create" message). If the functionality receives the "create" messages from all three parties, then the channel is created, which is communicated to $\gamma.\mathsf{end\text{–}users}$ by the "created" message. Note that $\gamma.\mathsf{Ingrid}$ does not receive this message, since she does not need to know whether γ has been created or not (as she is not going to perform any operations on this channel). After the channel is created $\gamma.\mathsf{end\text{–}users}$ can use it until time $\gamma.\mathsf{validity}$. When this time comes, the parties initiate the closing procedure that has to end in time at most $\gamma.\mathsf{validity} + \mathrm{TimeRegister}(j) + 2 \cdot \mathrm{TimeExecute}(\lceil j/2 \rceil)$. The functionality then looks at the last version of γ and distributes the coins in the subchannels of γ according to it.

In both cases ("ledger" and "virtual") we assume that all the honest parties involved in channel opening initiate the procedure in the same round and that they have enough funds for the new channel. In case of a virtual channel, we additionally assume that the lengths of subchannels differ at most by one. All these assumptions are formally modeled by restricting the environment, see Appx. C.

The procedure for updating a contract instance is identical for ledger and virtual channels (this procedure is also used for creating new contract instances). It is initiated by a party $P \in \gamma.\mathsf{end\text{–}users}$ that sends an "update" message to the ideal functionality. This message has parameters *id* and *cid* that identify a channel γ and a contract instance in this channel (respectively). The other parameters, $\tilde{\sigma}$ and $\mathtt{C}$, denote the new storage and type of the contract instance with identifier *cid* in γ. Party Q confirms the update with an "update-reply" message that has to reach the ideal functionality within time T (which is a function of channel length, see Step 3.). If the update is confirm, then the contract instance with identifier *cid* in γ gets replaced with a contract instance determined by $\tilde{\sigma}, \mathtt{C}$ (if no such contract instance existed before then it gets created). We assume (see Appx. C) that the environment never asks the parties to do obviously illegal things, like updating a channel that does not exits, creating a contract instance when there are not enough coins in the subchannels. Also, changing a type of a contract instance is not permitted. Moreover, we assume that the environment never asks to update a contract instance when it is already being updated or executed and for reasons that will be explained later (see Section 7), we allow only one contract instance to be created in a virtual channel.

The procedure for executing a contract instance is initiated by one of the parties $P \in \gamma.\mathsf{end\text{–}users}$ that sends an "execute" message to the ideal functionality. This message has parameters *id* and *cid* whose meaning is as in the update procedure. Other parameters are: f denoting the function to be executed, and z which is an additional input parameter to the function f. The execution results in updating the contract instance with identifier *cid* according to the result of computing $f(\sigma, P, \tau_0, z)$, where τ_0 is the round when the "execute" message was received, and σ is the current storage of the contract instance. Observe that this storage can be different than the storage in round τ_0. This can sometimes result in a situation when two executions of a contract instance, happening one after another, will have "reversed" information about the rounds. More precisely: the first execution will assume that the current round is τ_0^1, and the second one will assume that it is τ_0^2 with $\tau_0^2 < \tau_0^1$. The users of our protocol should be aware of this asynchronicity when designing the contracts. The restrictions on the environment in case of the contract instance execution are straightforward. In particular, as before, we assume that a given contract instance has to exist.

The procedure for closing a ledger channel γ starts when a party $P \in \gamma.\mathsf{end\text{–}users}$ sends to the ideal functionality a message (close, id) (where *id* is the identifier of γ). The functionality checks (in Step 1) if there are no contract instances that are open over γ. If not, then in Step 2 the functionality distributes the coins from γ to parties' ledger accounts according to γ's latest balance, removes the channel from Γ, and communicates to the parties that channel has been closed. We assume that $\mathcal{Z}$ only asks to close the ledger channels (as the virtual ones are closed "automatically") and that γ exists.[6]

[6] We say that a channel γ exists if the environment received the message $(\mathrm{created}, \gamma)$ but not yet $(\mathrm{closed}, \gamma.\mathsf{id})$.

4.1 Modular approach

Before we define a protocol realizing the ideal functionality $\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i, \mathcal{C})$, let us give a short overview of our approach.

We will first define an ideal functionality $\mathcal{F}_{sc}^{\mathcal{L}(\Delta)}(\mathcal{C})$ which models the behavior of a concrete smart contract on a blockchain that allows two parties to open, maintain and close a ledger state channel. The ideal functionality is parametrized by the set of contract types whose instances can be opened in the ledger channels created via this ideal functionality. The ideal functionality $\mathcal{F}_{sc}^{\mathcal{L}(\Delta)}(\mathcal{C})$ together with the ledger functionality $\mathcal{L}$ can be implemented by a cryptocurrency which supports such a channel smart contract on its blockchain (a candidate cryptocurrency would be, e.g., Ethereum).

As already mentioned in Section 1.1, our technique allows to create virtual state channels of arbitrary length, via applying the virtual channel functionality recursively. This will be modeled by constructing our protocols in the "hybrid model", i.e., a protocol for constructing virtual channels of length up to i (in other words: a protocol realizing functionality $\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i, \mathcal{C})$) will work in a model with access to an ideal functionality for constructing channels of length up to $i - 1$. More formally, for every $i > 1$ we will construct a protocol $\Pi(i, \mathcal{C})$ in the $\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i-1, \mathcal{C}')$-hybrid world, where $\mathcal{C}'$ is a set of contract types defined as $\mathcal{C}' := \mathcal{C} \cup \mathtt{VSC}_i(\mathcal{C})$, and $\mathtt{VSC}_i(\mathcal{C})$ is a contract type that allows to open a virtual state channel of length i in which contract instance of type from the set $\mathcal{C}$ can be opened. The protocol $\Pi(1, \mathcal{C})$ realizing $\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(1, \mathcal{C})$ will be constructed in the $\mathcal{F}_{sc}^{\mathcal{L}(\Delta)}(\mathcal{C})$-hybrid model. This, by applying the composition recursively, will give us a construction of a protocol realizing the $\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i, \mathcal{C})$ functionality in the $\mathcal{F}_{sc}^{\mathcal{L}(\Delta)}(\widehat{\mathcal{C}})$-hybrid model (where $\widehat{\mathcal{C}}$ is a result of applying the "$\mathcal{C} := \mathcal{C} \cup \mathtt{VSC}_i(\mathcal{C})$" equation i times recursively). See Fig. 3 for an example for $i = 3$.

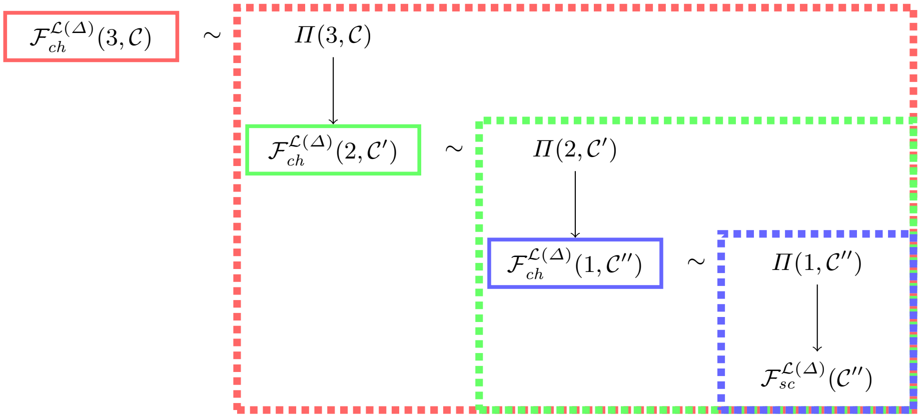
$\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(3, \mathcal{C})$
$\sim$
$\Pi(3, \mathcal{C})$
$\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(2, \mathcal{C}')$
$\sim$
$\Pi(2, \mathcal{C}')$
$\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(1, \mathcal{C}'')$
$\sim$
$\Pi(1, \mathcal{C}'')$
$\mathcal{F}_{sc}^{\mathcal{L}(\Delta)}(\mathcal{C}'')$

Fig. 3: Our modular approach. Above $\mathcal{C}' := \mathcal{C} \cup \mathtt{VSC}_3(\mathcal{C})$, $\mathcal{C}'' := \mathcal{C}' \cup \mathtt{VSC}_2(\mathcal{C}')$.

5 Ledger Channels

In this section, we will first define an ideal functionality $\mathcal{F}_{sc}^{\mathcal{L}(\Delta)}(\mathcal{C})$ which represents the smart contract allowing two parties to open, maintain and close a ledger state channel. Then we will describe the protocol $\Pi(1, \mathcal{C})$ that realizes the state channels ideal functionality $\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(1, \mathcal{C})$ (see Sect. 4) in the hybrid world $\mathcal{F}_{sc}^{\mathcal{L}(\Delta)}(\mathcal{C})$ for any set of contract types $\mathcal{C}$.

5.1 Ideal functionality for Channel smart contract

The ideal functionality $\mathcal{F}_{sc}^{\mathcal{L}(\Delta)}(\mathcal{C})$ is parametrized by a set $\mathcal{C}$ defining the contract types whose instance can be constructed in a ledger state channel. The ideal functionality $\mathcal{F}_{sc}^{\mathcal{L}(\Delta)}(\mathcal{C})$ has access to the global ideal functionality $\mathcal{L}$ (the ledger). The ideal functionality $\mathcal{F}_{sc}^{\mathcal{L}(\Delta)}(\mathcal{C})$ accepts messages from parties $\mathcal{P} := \{P_1, \ldots, P_n\}$. Let us emphasize that since the ideal functionality models a concrete smart contracts on the ledger, each communication session (party sends a message to the ideal functionality which potentially makes some modifications on the ledger and replies) comes with a delay up to Δ rounds. The exact timing (and if applicable, the exact round when transaction on the ledger takes place), is determined by the adversary. Technically, the ideal functionality receives instructions from the adversary via the influence port. In order to shorten the description of the ideal functionality, we do not mention the transact instructions explicitly .

The functionality $\mathcal{F}_{sc}^{\mathcal{L}(\Delta)}(\mathcal{C})$ maintains a space Γ containing all open ledger channels. The set Γ is initially empty. The functionality consists of four parts: "Create a ledger channel", "Contract instance registration", "Contract instance execution" and "Close a ledger channel". These parts will be described and formally defined together with the protocol for ledger state channels in Section 5.2.

5.2 Protocol for Ledger Channels

Create a ledger state channel. In order to create a new ledger state channel γ, the environment sends the message (create, γ) to both parties in $\gamma.\mathsf{end\text{–}users}$. A new ledger state channel can be built only if both users of the ledger channel agree on a smart contract that is published on the ledger. In this work, such smart contracts are modelled via an ideal functionality $\mathcal{F}_{sc}$ which is parametrized by $\mathcal{C}$ – a set of contract types that can be opened in the ledger state channel. The protocol for creating a ledger channel works at a high level as follows.

The initiating party $\gamma.\mathsf{Alice}$ requests construction of the smart contract by sending the message (construct, γ) to the ideal functionality $\mathcal{F}_{sc}^{\mathcal{L}(\Delta)}(\mathcal{C})$. The ideal functionality locks the required amount of coins in her account on the ledger and sends the message $(\text{initializing}, \gamma)$ to both parties. If party $\gamma.\mathsf{Bob}$ confirms the initialization by sending the message $(\text{confirm}, \gamma)$, the ideal functionality $\mathcal{F}_{sc}^{\mathcal{L}(\Delta)}(\mathcal{C})$ outputs $(\text{created}, \gamma)$. In case $\gamma.\mathsf{Bob}$ does not confirm, the channel cannot be created and the initiating party $\gamma.\mathsf{Alice}$ has the option to refund the coins that were locked in her account on the ledger during the first step.

To conclude, creation of a ledger channels takes up to 2Δ rounds since it requires two interactions with the hybrid ideal functionality modelling a smart contract on the ledger. In case the channel is not created but $\gamma.\mathsf{Alice}$'s coins were locked in the first phase of the channel creation, she can receive them back latest after 3Δ rounds.

Formal description of the protocol for ledger channel creation and the corresponding part of the $\mathcal{F}_{sc}$ functionality can be found below.

Protocol $\Pi(1, \mathcal{C})$: Create a ledger channel

We use the abbreviated notation from Section 3.3 and let $\mathcal{F}_{sc} := \mathcal{F}_{sc}^{\mathcal{L}(\Delta)}(\mathcal{C})$.

Party A upon $(\text{create}, \gamma) \xhookleftarrow{\tau_0} \mathcal{Z}$

1. Send $(\text{construct}, \gamma) \xhookrightarrow{\tau_0} \mathcal{F}_{sc}$ and wait.

Party B upon $(\text{create}, \gamma) \xhookleftarrow{\tau_0} \mathcal{Z}$

2. If $(\text{initializing}, \gamma) \xhookleftarrow{\tau_1 \leq \tau_0 + \Delta} \mathcal{F}_{sc}$, then $(\text{confirm}, \gamma) \xhookrightarrow{\tau_1} \mathcal{F}_{sc}$ and wait. Else stop.
3. If $(\text{initialized}, \gamma) \xhookleftarrow{\tau_2 \leq \tau_0 + 2 \cdot \Delta} \mathcal{F}_{sc}$, then set $\Gamma^B(\gamma.\mathsf{id}) := \gamma$, output $(\text{created}, \gamma) \xhookrightarrow{\tau_2} \mathcal{Z}$ and stop. Else stop.

Back to party A

4. If $(\text{initialized}, \gamma) \xhookleftarrow{\tau_2 \leq \tau_0 + 2 \cdot \Delta} \mathcal{F}_{sc}$, then set $\Gamma^A(\gamma.\mathsf{id}) := \gamma$, output $(\text{created}, \gamma) \xhookrightarrow{\tau_2} \mathcal{Z}$ and stop. Else go to next step.
5. If $(\text{refund}, \gamma) \xhookleftarrow{\tau_3 > \tau_0 + 2 \cdot \Delta} \mathcal{Z}$, then $(\text{refund}, \gamma) \xhookrightarrow{\tau_3} \mathcal{F}_{sc}$ and stop.

Functionality $\mathcal{F}_{sc}^{\mathcal{L}(\Delta)}(\mathcal{C})$: Create a ledger channel

We use the abbreviated notation from Section 3.3.

Upon $(\text{construct}, \gamma) \xhookleftarrow{\tau_0} P$**:**

1. If $P \neq A$, there already exists a channel γ' such that $\gamma.\mathsf{id} = \gamma'.\mathsf{id}$, $\gamma.\mathsf{cspace} \neq \emptyset$ or $\mathtt{Value}(\gamma) < 0$, then stop.
2. Within Δ rounds remove $\gamma.\mathsf{cash}(A)$ coins from A's account on the ledger $\mathcal{L}$. If it is impossible due to insufficient funds, then stop. Else $(\text{initializing}, \gamma) \hookrightarrow B$ and store the pair $tamp := (\tau_0, \gamma)$ in memory.

Upon $(\text{confirm}, \gamma) \xhookleftarrow{\tau_1} P$**:**

3. If there is no pair $tamp = (\tau_0, \gamma)$ in the storage, $(\tau_1 - \tau_0) > \Delta$ or $P \neq B$, then stop.
4. Within Δ rounds remove $\gamma.\mathsf{cash}(B)$ coins from B's account on the ledger $\mathcal{L}$. If it is impossible due to insufficient funds, then stop. Else set $\Gamma(\gamma.\mathsf{id}) := \gamma$ and delete $tamp$ from the memory. Thereafter send $(\text{initialized}, \gamma) \hookrightarrow \gamma.\mathsf{end\text{--}users}$.

Upon $(\text{refund}, \gamma) \xhookleftarrow{\tau_2} P$**:**

5. If there is no pair $tamp = (\tau_0, \gamma)$ in the storage, $(\tau_2 - \tau_0) \leq 2\Delta$ or $P \neq A$, then stop.
6. Else within Δ rounds add $\gamma.\mathsf{cash}(A)$ coins to A's account on the ledger $\mathcal{L}$ and delete $tamp$ from the storage.

Register a contract instance in a ledger state channel. As long as both end-users of a ledger state channel behave honestly, they can update, execute and close contract instances running in the ledger state channel off-chain; i.e. without communicating with the ideal functionality $\mathcal{F}_{sc}^{\mathcal{L}(\Delta)}(\mathcal{C})$. However, once the parties run into dispute (e.g., one party does not communicate, sends an invalid message, etc.), parties have to resolve their disagreement on the ledger. We call this process "registration of a contract instance", and will describe its basic functionality below.

The registration of a contract instance might be necessary either when the contract instance is being updated, executed or when a ledger channel is being closed. To prevent repeating the same part of the protocol multiple times in each of the protocols, we state the registration process as a separate procedure $\mathtt{Register}(P, id, cid)$ which can be called by parties running one of the sub-protocols mentioned above. The procedure takes as input party P which initiates the registration and the identifiers defining the contract instance to be registered, i.e. identifier of the ledger channel id and the contract instance identifier cid.

At a high level, the initiating party (assume for now that it is $\gamma.\mathsf{Alice}$) sends her contract instance version to the ideal functionality $\mathcal{F}_{sc}^{\mathcal{L}(\Delta)}(\mathcal{C})$ which first checks the validity of the received version (for example, if it is correctly signed by $\gamma.\mathsf{Bob}$, if the contract type of the instance is in the set $\mathcal{C}$, etc.). If the contract instance version is valid, within Δ rounds the hybrid ideal functionality $\mathcal{F}_{sc}^{\mathcal{L}(\Delta)}(\mathcal{C})$ informs both users that the contract instance is being registered. Party $\gamma.\mathsf{Bob}$ then immediately reacts by sending his own version of the contract instance to $\mathcal{F}_{sc}^{\mathcal{L}(\Delta)}(\mathcal{C})$. The ideal functionality compares the two received versions, registers the one with higher version number and within Δ rounds informs both users which version was registered. In case $\gamma.\mathsf{Bob}$ did not send in his version, $\gamma.\mathsf{Alice}$ can finalize the registration by sending the message "finalize–register" to the hybrid ideal functionality.

In the optimistic case when $\gamma.\mathsf{Bob}$ submits a valid version of the contract instance, the registration procedure takes up to 2Δ rounds since it requires two interactions with the ideal functionality $\mathcal{F}_{sc}^{\mathcal{L}(\Delta)}(\mathcal{C})$. In the pessimistic case when $\gamma.\mathsf{Bob}$ does not react or submits an invalid version, the procedure takes up to 3Δ.

Formal description of this procedure and the corresponding part of the $\mathcal{F}_{sc}^{\mathcal{L}(\Delta)}(\mathcal{C})$ functionality can be found below.

Procedure $\texttt{Register}(P, id, cid)$

We use the abbreviated notation from Section 3.3. In addition, we denote $\mathcal{F}_{sc} := \mathcal{F}_{sc}^{\mathcal{L}(\Delta)}(\mathcal{C})$.

Party P:

1. Let $\gamma^P := \Gamma^P(id)$, $\nu^P := \gamma^P.\mathsf{cspace}(cid)$, and let τ_0 be the current round. Send (instance–register, $id, cid, \nu^P) \xhookrightarrow{\tau_0} \mathcal{F}_{sc}$.
2. If not (instance–registering, $id, cid) \xhookleftarrow{\tau_1 \leq \tau_0 + \Delta} \mathcal{F}_{sc}$, then stop. Else goto step 4.

Party Q upon (instance–registering, $id, cid) \xhookleftarrow{\tau_1} \mathcal{F}_{sc}$

3. Let $\gamma^Q := \Gamma^Q(id)$ and $\nu^Q := \gamma^Q.\mathsf{cspace}(cid)$. Then send (instance–register, $id, cid, \nu^Q) \xhookrightarrow{\tau_1} \mathcal{F}_{sc}$ and goto step 5.

Back to party P:

4. If not (instance–registered, $id, cid, \tilde{\nu}) \xhookleftarrow{\tau_2 \leq \tau_1 + \Delta} \mathcal{F}_{sc}$, then send (finalize–register, $id, cid) \xhookrightarrow{\tau_3 = \tau_1 + \Delta} \mathcal{F}_{sc}$.

End for both $T = A$ and $T = B$

5. Upon (instance–registered, $id, cid, \tilde{\nu}) \hookleftarrow \mathcal{F}_{sc}$, mark (id, cid) as registered in Γ^T and set $\Gamma^T := \texttt{LocalUpdate}(\Gamma^T, id, cid, \tilde{\nu})$.

Functionality $\mathcal{F}_{sc}^{\mathcal{L}(\Delta)}(\mathcal{C})$: Contract instance registration

We use the abbreviated notation from Section 3.3.

Upon (instance–register, $id, cid, \nu) \xhookleftarrow{\tau_0} P$**,** let $\gamma := \Gamma(id)$ and do:

1. If party $P \notin \gamma.\mathsf{end\text{–}users}$, one of the signatures $\nu.\mathsf{sign}(A), \nu.\mathsf{sign}(B)$ is invalid, $\nu.\mathsf{type} \notin \mathcal{C}$, $\gamma.\mathsf{cspace}(cid) \neq \bot$, $\nu.\mathsf{storage} \notin \nu.\mathsf{type}.\Lambda$ or the sum of locked coins in the contract instance is negative, then stop.
2. Else let $Q := \gamma.\mathsf{other\text{–}party}(P)$ and consider the following three cases:
   - If your memory contains a tuple $(P, id, cid, \widehat{\nu}, \widehat{\tau}_0)$, then ignore this call.
   - If your memory contains a tuple $(Q, id, cid, \widehat{\nu}, \widehat{\tau}_0)$, then first compare the version number, i.e. if $\nu.\mathsf{storage.version} > \widehat{\nu}.\mathsf{storage.version}$, then set $\tilde{\nu} := (\nu.\mathsf{storage}, \nu.\mathsf{type})$ and otherwise set $\tilde{\nu} := (\widehat{\nu}.\mathsf{storage}, \widehat{\nu}.\mathsf{type})$. Thereafter wait for at most Δ rounds to send (instance–registered, $id, cid, \tilde{\nu}) \xhookrightarrow{\tau_1 \leq \tau_0 + \Delta} \gamma.\mathsf{end\text{–}users}$, update $\Gamma := \texttt{LocalUpdate}(\Gamma, id, cid, \tilde{\nu})$ and erase $(Q, id, cid, \widehat{\nu}, \widehat{\tau}_0)$ from your memory.
   - Else save $(P, id, cid, \nu, \tau_0)$ to your memory and send (instance–registering, $id, cid) \xhookrightarrow{\tau_1 \leq \tau_0 + \Delta} \gamma.\mathsf{end\text{–}users}$.

Upon (finalize–register, $id, cid) \xhookleftarrow{\tau_2} P$**,** let $\gamma := \Gamma(id)$ and do

- If $P \in \gamma.\mathsf{end\text{–}users}$ and your memory contains a value $(P, id, cid, \widehat{\nu}, \widehat{\tau}_0)$ such that $\tau_2 - \widehat{\tau}_0 \geq 2\Delta$, then set $\tilde{\nu} := (\widehat{\nu}.\mathsf{storage}, \widehat{\nu}.\mathsf{type})$, send (instance–registered, $id, cid, \tilde{\nu}) \xhookrightarrow{\tau_3 \leq \tau_2 + \Delta} \gamma.\mathsf{end\text{–}users}$, set $\Gamma := \texttt{LocalUpdate}(\Gamma, id, cid, \tilde{\nu})$ and erase $(P, id, cid, \widehat{\nu}, \widehat{\tau}_0)$ from your memory.
- Else ignore this call.

Update a contract instance in a ledger state channel. In order to update the storage of a contract instance in a ledger state channel, we assume that the environment sends the message (update, $id, cid, \tilde{\sigma}, \mathtt{C})$

to the initiating party $P \in \gamma.\mathsf{end\text{–}users}$. The input parameters $\tilde{\sigma}$ and $\mathtt{C}$ define the new contract instance that should be stored in $\gamma.\mathsf{cspace}$ under the identifier cid.

The update protocol works on high level as follows. The initiating party P signs the new contract instance version with increased version number (i.e. if ν^P is the contract instance version stored by P until now, then the new contract instance version ν will be such that $\nu.\mathsf{version} = \nu^P.\mathsf{version} + 1$). Party P then sends her signature on this value to the party $Q := \gamma.\mathsf{other\text{–}party}(P)$. The other party verifies the signature and informs the environment that the update was requested. If the environment confirms the update, the party Q signs the updated contract version and sends the signature to P. In this optimistic case, the update takes 2 rounds.

Let us discuss how parties behave in case the environment does not confirm the update. If Q simply aborts in this situation, P does not know if update failed because Q is malicious or because the environment did not confirm the update. Therefore, Q has to inform P about the failure. This is, however, still not sufficient. Note that Q holds P's signature of the new contract instance. If Q is corrupt, it can register the updated contract instance on the ledger at any later point. Thus, in case the environment does not confirm the update, party Q must sign the original contract instance but with version number increased by 2 and send the signature to party P. Now, if P does not receive a valid signature on either the updated contract instance version or the original contract instance with increased version number from Q, it is clear that Q is malicious and therefore P initiates the registration of the contract instance on the ledger by calling the procedure $\mathtt{Register}(P, id, cid)$. Note that Q can still register the updated contract instance (the one that was signed by P). But importantly, after at most $2 + 3\Delta$ rounds it will be clear to both parties what the current contract instance version is. Formal description of the protocol for updating a contract instance in a ledger channel can be found below.

Protocol $\Pi(1, \mathcal{C})$: Contract instance update

We use the abbreviated notation from Section 3.3 .

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

1. Let $\gamma^P := \Gamma^P(id)$ and $\nu^P := \gamma^P.\mathsf{cspace}(cid)$. If $\nu^P = \perp$, then set $v^P := 0$, else set $v^P := \nu^P.\mathsf{version}$.
2. Compute $s_P := \mathtt{Sign}_{sk_P}(id, cid, \tilde{\sigma}, \mathtt{C}, v^P + 1)$.
3. Send (update, $s_P, id, cid, \tilde{\sigma}, \mathtt{C}$) $\overset{\tau_0}{\hookrightarrow} Q$ and wait.

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

4. Let $\gamma^Q := \Gamma^Q(id)$ and $\nu^Q := \gamma^Q.\mathsf{cspace}(cid)$. If $\nu^Q = \perp$, then set $v^Q := 0$, else set $v^Q := \nu^Q.\mathsf{version}$.
5. If $\mathtt{Vfy}_{pk_P}(id, cid, \tilde{\sigma}, \mathtt{C}, v^Q + 1; s_P) \neq 1$, then mark (id, cid) as corrupt and stop. Else output (update–requested, id, cid) $\overset{\tau_1}{\hookrightarrow} \mathcal{Z}$ and consider the following two cases
   - If (update–reply, ok, id, cid) $\overset{\tau_1}{\hookleftarrow} \mathcal{Z}$, then compute the signature $s_Q := \mathtt{Sign}_{sk_Q}(id, cid, \tilde{\sigma}, \mathtt{C}, v^Q + 1)$, send (update–ok, s_Q) $\overset{\tau_1}{\hookrightarrow} P$, set $\Gamma^Q := \mathtt{LocalUpdate}(\Gamma^Q, id, cid, \tilde{\sigma}, \mathtt{C}, v^Q + 1, \{s_P, s_Q\})$ and output (updated, id, cid) $\xhookrightarrow{\tau_1+1} \mathcal{Z}$.
   - Else compute $s_Q := \mathtt{Sign}_{sk_Q}(id, cid, \nu^Q.\mathsf{storage}, \nu^Q.\mathsf{type}, v^Q + 2)$, send (update–not–ok, s_Q) $\overset{\tau_1}{\hookrightarrow}$ P and stop.

Back to party P

6. Distinguish the following three cases:
   - If (update–ok, s_Q) $\xhookleftarrow{\tau_2=\tau_0+2} Q$, where $\mathtt{Vfy}_{pk_Q}(id, cid, \tilde{\sigma}, \mathtt{C}, v^P + 1; s_Q) = 1$, then set $\Gamma^P := \mathtt{Local} \mathtt{Update}(\Gamma^P, id, cid, \tilde{\sigma}, \mathtt{C}, v^P + 1, \{s_P, s_Q\})$, output (updated, id, cid) $\overset{\tau_2}{\hookrightarrow} \mathcal{Z}$ and stop.

- If (update–not–ok, s_Q) $\xhookleftarrow{\tau_2=\tau_0+2}$ Q, where $\mathtt{Vfy}_{pk_Q}(id, cid, \nu^P.\mathsf{storage}, \nu^P.\mathsf{type}, v^P + 2; s_Q) = 1$, then compute $s_P := \mathtt{Sign}_{sk_P}(id, cid, \nu^P.\mathsf{storage}, \nu^P.\mathsf{type}, v^P + 2)$, set $\mathit{\Gamma}^P := \mathtt{LocalUpdate}(\mathit{\Gamma}^P, id, cid, \nu^P.\mathsf{storage}, \nu^P.\mathsf{type}, v^P + 2, \{s_P, s_Q\})$ and stop.
- Else mark (id, cid) as corrupt and in round $\tau_0 + 2$ call the subprocedure $\mathtt{Register}(P, id, cid)$. After the subrprocedure execution (in round $\tau_3 \leq \tau_0 + 3\mathit{\Delta} + 2$), if $\gamma^P.\mathsf{cspace}(cid) = (\tilde{\sigma}, \mathtt{C})$, then output (updated, id, cid) $\xhookrightarrow{\tau_3}$ $\mathcal{Z}$.

Execute a contract instance in a ledger state channel. In order to execute a contact instance stored in a ledger channel γ, the environment sends the message (execute, $\gamma.\mathsf{id}, cid, f, z$) to the initiating party $P \in \gamma.\mathsf{end\text{–}users}$. The parameter cid points to the contract instance, f is the contact function to be applied to the contract instance and z are additional input values of the function f.

The protocol works on a high level as follows (let us for now assume that $P = \gamma.\mathsf{Alice}$). If the parties never registered the contract instance with identifier cid, then $\gamma.\mathsf{Alice}$ first tries to execute the contract instance "peacefully". This means that she locally executes f on the contract version she stores in $\mathit{\Gamma}^{\gamma.\mathsf{Alice}}$, signs the new contract instance and sends the signature to $\gamma.\mathsf{Bob}$. Party $\gamma.\mathsf{Bob}$ also executes f locally on his own version of the contract instance stored in $\mathit{\Gamma}^{\gamma.\mathsf{Bob}}$ and thereafter verifies $\gamma.\mathsf{Alice}$'s signature. If the signature is valid, $\gamma.\mathsf{Bob}$ immediately confirms the execution by sending his signature on the new contract instance to party $\gamma.\mathsf{Alice}$.

A technical challenge occurs when both parties want to peacefully execute the same contract instance in the same round τ since it becomes unclear what is the new contract instance. To overcome this technical difficulty, $\gamma.\mathsf{Alice}$ peacefully executes only if $\tau = 1 \mod 4$ and $\gamma.\mathsf{Bob}$ only when $\tau = 3 \mod 4$. So, for example, if $\gamma.\mathsf{Alice}$ receives the message "execute" in round $\tau = 2 \mod 4$, then she waits for three rounds and only then starts the peaceful execution. Hence, in the optimistic case when both users are honest, the execution protocol takes up to 5 rounds.

In case the contract instance with identifier cid has already been registered on the ledger or the peaceful execution fails, the initiating party executes the contract instance "forcefully". By this we mean that $\gamma.\mathsf{Alice}$ first initiates registration of the contract instance by calling the procedure $\mathtt{Register}(P, id, cid)$, and then instructs the ideal functionality $\mathcal{F}_{sc}^{\mathcal{L}(\mathit{\Delta})}(\mathcal{C})$ to execute the contract instance. The $\mathtt{Register}$ procedure can take up to $3\mathit{\Delta}$ rounds and the contract instance execution on the ledger can take up to $\mathit{\Delta}$ rounds. Thus, pessimistic time complexity of the execution protocol is equal to $4\mathit{\Delta} + 5$ rounds. Formal description of the protocol for executing a contract instance in a ledger channel and the corresponding part of the functionality $\mathcal{F}_{sc}^{\mathcal{L}(\mathit{\Delta})}(\mathcal{C})$ can be found below.

Protocol $\mathit{\Pi}(1, \mathcal{C})$: Contract instance execution

We use the abbreviated notation from Section 3.3 and denote $\mathcal{F}_{sc} := \mathcal{F}_{sc}^{\mathcal{L}(\mathit{\Delta})}(\mathcal{C})$.

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

1. Let $\gamma^P := \mathit{\Gamma}^P(id), \nu^P := \gamma^P.\mathsf{cspace}(cid)$, $\sigma^P := \nu^P.\mathsf{storage}$, $\mathtt{C}^P := \nu^P.\mathsf{type}$ and $v^P := \nu^P.\mathsf{version}$.
2. Set $\tau_1 := \tau_0 + x$, where x is the smallest offset such that $\tau_1 = 1 \mod 4$ if $P = \gamma^P.\mathsf{Alice}$ and $\tau_1 = 3 \mod 4$ if $P = \gamma^P.\mathsf{Bob}$. Wait till round τ_1.
3. If (id, cid) is not marked as corrupt in $\mathit{\Gamma}^P$, then compute $(\tilde{\sigma}, add_L, add_R, m) := f(\sigma^P, P, \tau_0, z)$. If $m = \perp$, then stop. Otherwise compute $s_P := \mathtt{Sign}_{sk_P}(id, cid, \tilde{\sigma}, \mathtt{C}^P, v^P + 1)$, send (peaceful–request, $id, cid, f, z, s_P, \tau_0$) $\xhookrightarrow{\tau_1}$ Q and goto step 10.
4. If (id, cid) is marked as corrupt, proceed as follows. If (id, cid) not marked as registered, then run $\mathtt{Register}(P, id, cid)$. Let τ_3 be the current round. Then send (instance–execute, id, cid, f, z) $\xhookrightarrow{\tau_3}$ $\mathcal{F}_{sc}$ and goto step 11.

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

5. Let $\gamma^Q := \Gamma^Q(id), \nu^Q := \gamma^Q.\mathsf{cspace}(cid)$, $\sigma^Q := \nu^Q.\mathsf{storage}$, $\mathtt{C}^Q := \nu^Q.\mathsf{type}$, $v^Q := \nu^Q.\mathsf{version}$. If $\gamma^Q = \bot$ or $P, Q \notin \gamma^Q.\mathsf{end\text{–}users}$ or $\nu^Q = \bot$, then goto step 9.
6. If $P = \gamma^Q.\mathsf{Alice}$ and $\tau_Q \mod 4 \neq 2$ or if $P = \gamma.\mathsf{Bob}$ and $\tau_Q \mod 4 \neq 0$, then goto step 9.
7. If $\tau_0 \notin [\tau_Q - 4, \tau_Q - 1]$, then goto step 9.
8. If (id, cid) is not marked as corrupt in Γ^Q, do:
   (a) Compute $(\tilde{\sigma}, add_L, add_R, m) := f(\sigma^Q, P, \tau_0, z)$.
   (b) If $m = \bot$ or $\mathtt{Vfy}_{pk_P}(id, cid, \tilde{\sigma}, \mathtt{C}^Q, v^Q + 1; s_P) \neq 1$, then goto step 9.
   (c) Else sign $s_Q := \mathtt{Sign}_{sk_Q}(id, cid, \tilde{\sigma}, \mathtt{C}^Q, v^Q + 1)$, send (peaceful–confirm, $id, cid, f, z, s_Q) \xhookrightarrow{\tau_Q} P$, set $\Gamma^Q := \mathtt{LocalUpdateAdd}(\Gamma^Q, id, cid, \tilde{\sigma}, \mathtt{C}^Q, add_A, add_R, v^Q + 1, \{s_P, s_Q\})$, output (executed, $id, cid, \tilde{\sigma}, add_L, add_R, m) \xhookrightarrow{\tau_Q+1} \mathcal{Z}$ and stop.
9. Mark (id, cid) as corrupt in Γ^Q. Then goto step 11.

Back to party P

10. Distinguish the following two cases
    – If (peaceful–confirm, $id, cid, f, z, s_Q) \xhookleftarrow{\tau_2 = \tau_1 + 2} Q$ such that $\mathtt{Vfy}_{pk_Q}(id, cid, \tilde{\sigma}, \mathtt{C}^P, v^P + 1; s_Q) = 1$, then set $\Gamma^P := \mathtt{LocalUpdateAdd}(\Gamma^P, id, cid, \tilde{\sigma}, \mathtt{C}^P, v^P + 1, \{s_P, s_Q\})$, output (executed, $id, cid, \tilde{\sigma}, add_L, add_R, m) \xhookrightarrow{\tau_2} \mathcal{Z}$ and stop.
    – Else mark (id, cid) as corrupt in Γ^P and execute the $\mathtt{Register}(P, id, cid)$. Once the procedure is executed (in round $\tau_3 \leq \tau_0 + 3\Delta + 5$), distinguish the following two cases:
      - If $\Gamma^P(id).\mathsf{cspace}(cid).\mathsf{storage} = \tilde{\sigma}$, then output (executed, $id, cid, \tilde{\sigma}, add_L, add_R, m) \xhookrightarrow{\tau_3} \mathcal{Z}$ and stop.
      - Else send (instance–execute, $id, cid, f, z) \xhookrightarrow{\tau_3} \mathcal{F}_{sc}$ and goto step 11.

End for both parties $T = A, B$

11. If (instance–executed, $id, cid, \widehat{\sigma}, add_L, add_R, m) \xhookleftarrow{\tau_4 \leq \tau_0 + 4\Delta + 5} \mathcal{F}_{sc}$, set $\Gamma^T := \mathtt{LocalUpdateAdd}(\Gamma^T, id, cid, \widehat{\sigma}, \mathtt{C}^T, add_L, add_R)$, output (executed, $id, cid, \widehat{\sigma}, add_L, add_R, m) \xhookrightarrow{\tau_4} \mathcal{Z}$ and stop. Else stop.

Functionality $\mathcal{F}_{sc}^{\mathcal{L}(\Delta)}(\mathcal{C})$: Contract instance execution

We use the abbreviated notation from Section 3.3.

Upon (instance–execute, $id, cid, f, z) \xhookleftarrow{\tau_0} P$, within Δ rounds proceed as follows. Let $\gamma := \Gamma(id)$. If $\gamma = \bot$, then stop. Else set $\nu := \gamma.\mathsf{cspace}(cid)$ and $\sigma := \nu.\mathsf{storage}$. If $P \neq \gamma.\mathsf{end\text{–}users}$ or $\nu = \bot$, then stop. Else compute $(\widehat{\sigma}, add_L, add_R, m) := f(\sigma, P, \tau_0, z)$. If $m = \bot$, then stop. Else update the channel space $\Gamma := \mathtt{LocalUpdateAdd}(\Gamma, id, cid, \widehat{\sigma}, \nu.\mathsf{type}, add_L, add_R)$, send (instance–executed, id, $cid, \widehat{\sigma}, add_L, add_R, m) \xhookrightarrow{\tau_1 \leq \tau_0 + \Delta} \gamma.\mathsf{end\text{–}users}$ and stop.

Close a ledger state channel. In order to close a ledger channel with identifier id by party $P \in \gamma.\mathsf{end\text{–}users}$, the environment sends the message (close, id) to the initiating party P. Before a ledger channel can be closed, the end-users of the channel have the chance to register all the contract instances that they have constructed off-chain. Thus, the initiating party P first (in parallel) registers all the contract instances which have been updated/peacefully executed but not registered at the ledger yet. This takes up to 3Δ rounds. Next, P asks the ideal functionality $\mathcal{F}_{sc}^{\mathcal{L}(\Delta)}(\mathcal{C})$ representing the smart contract on the ledger to close the channel. Within Δ rounds, the ideal functionality informs both parties that the channel is being closed and gives the other end-user of the channel time 3Δ to register contract instances that were not registered by P. If after 3Δ

rounds all registered contract instances are terminated (the locked amount of coins is equal to zero), the ideal functionality adds $\gamma.\mathsf{cash}(\gamma.\mathsf{Alice})$ coins to $\gamma.\mathsf{Alice}$'s account on the ledger, and $\gamma.\mathsf{cash}(\gamma.\mathsf{Bob})$ coins to $\gamma.\mathsf{Bob}$'s account on the ledger, deletes the channel from its channel space and within Δ rounds informs both parties that the channel was successfully closed. If there exists at least one unterminated contract instance, the channel can not be closed in which case the $\mathcal{F}_{sc}^{\mathcal{L}(\Delta)}(\mathcal{C})$ simply aborts. To conclude our description, it can take up to 8Δ rounds to successfully close a ledger state channel. The protocol and the contract functionality for the ledger channel closing are presented formally below.

Protocol $\Pi(1, \mathcal{C})$: Close a ledger channel

We use the abbreviated notation from Section 3.3 and denote $\mathcal{F}_{sc} := \mathcal{F}_{sc}^{\mathcal{L}(\Delta)}(\mathcal{C})$

Party P upon $(\text{close}, id) \xhookleftarrow{\tau_0} \mathcal{Z}$

1. Let $\gamma^P := \Gamma^P(id)$. For each $cid \in \{0,1\}^*$ such that $\gamma^P.\mathsf{cspace}(cid) \neq \perp$ and (id, cid) is not marked as registered, execute $\mathtt{Register}(P, id, cid)$ in round τ_0. Then send $(\text{contract–close}, id) \xhookrightarrow{\tau_1 \leq \tau_0 + 3\Delta} \mathcal{F}_{sc}$ and wait.

Party Q upon $(\text{contract–closing}, id) \xhookleftarrow{\tau_2 \leq \tau_0 + 4\Delta} \mathcal{F}_{sc}$

2. Let $\gamma^Q := \Gamma^Q(id)$. For each $cid \in \mathbb{N}$ such that (id, cid) is not marked as registered in Γ^Q and $\gamma^Q.\mathsf{cspace}(cid) \neq \perp$, call $\mathtt{Register}(Q, id, cid)$ in round τ_2

Rest of the protocol for $T = P, Q$ (respectively):

3. If $(\text{contract–closed}, id) \xhookleftarrow{\tau_3 \leq \tau_0 + 8\Delta} \mathcal{F}_{sc}$, then set $\Gamma^T(id) := \perp$ and output $(\text{closed}, id) \xhookrightarrow{\tau_3} \mathcal{Z}$.

Functionality $\mathcal{F}_{sc}^{\mathcal{L}(\Delta)}(\mathcal{C})$: Close a ledger channel

We use the abbreviated notation from Section 3.3.

Upon $(\text{contract–close}, id) \xhookleftarrow{\tau_0} P$ let $\gamma := \Gamma(id)$ and proceed as follows:

1. Within Δ rounds send $(\text{contract–closing}, id) \xhookrightarrow{\tau_1 \leq \tau_0 + \Delta} \gamma.\mathsf{end\text{–}users}$.
2. Wait for next at most 3Δ rounds. If in round $\tau_2 \leq \tau_0 + 4\Delta$ there exists $cid \in \{0,1\}^*$ such that $\gamma.\mathsf{cspace}(cid) \neq \perp$ but the contract instance is not terminated, i.e. $\sigma_{cid}.\mathsf{cash}(A) + \sigma_{cid}.\mathsf{cash}(B) \neq 0$, where $\sigma_{cid} := \gamma.\mathsf{cspace}(cid).\mathsf{storage}$, then stop.
3. Else wait for at most Δ round to add $\gamma.\mathsf{cash}(A)$ coins to A's account and $\gamma.\mathsf{cash}(B)$ coins to B's account on the ledger and set $\Gamma(id) = \perp$. Then send $(\text{contract–closed}, id) \xhookrightarrow{\tau_3 \leq \tau_0 + 5\Delta} \gamma.\mathsf{end\text{–}users}$.

We can now state the theorem showing that our construction for ledger channels emulates the ideal functionality $\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(1, \mathcal{C})$ from Section 4. The proof is given in the Appx. D.

Theorem 1. *Let $\mathcal{E}_{res}$ be the class of restricted environments defined in Appx. C. The protocol $\Pi(1, \mathcal{C})$ working in $\mathcal{F}_{sc}^{\mathcal{L}(\Delta)}(\mathcal{C})$-hybrid model emulates the ideal functionality $\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(1, \mathcal{C})$ against environments from class $\mathcal{E}_{res}$ for every set of contract types $\mathcal{C}$ and every $\Delta \in \mathbb{N}$.*

6 Virtual State Channel Contract

We now define a concrete contract type whose instances can be used to create virtual state channels γ. We denote this contract type as $\mathtt{VSC}_i(\mathcal{C})$, where the parameter $i > 1$ is the length of γ and the parameter $\mathcal{C}$ is a set of contract types that can be constructed in γ. Consider three parties: Alice, Bob, and Ingrid, and suppose that Alice and Ingrid have opened a state channel α, and Bob and Ingrid have created a state channel β. During the creation of the virtual channel γ between Alice and Bob, the parties Alice and Ingrid agree on updating α such that it contains the contract instance $(\sigma_A, \mathtt{VSC}_i(\mathcal{C}))$. Here, σ_A denotes the initial contract storage created by calling $\mathtt{Init}_i^{\mathcal{C}}$, the constructor of $\mathtt{VSC}_i(\mathcal{C})$, on input tuple $(\text{Alice}, \tau, \gamma)$. On a very informal level, one may think of the contract storage $\sigma_A := \mathtt{Init}_i^{\mathcal{C}}(\text{Alice}, \tau, \gamma)$ as being a "copy" of the virtual channel description γ, where Ingrid plays the role of Bob. This "copy" of the virtual channel γ will be stored in $\alpha.\mathsf{cspace}$ under the identifier $cid_A := \text{Alice}||\gamma.\mathsf{id}$. Symmetrically, Ingrid and Bob agree on updating their channel β such that it contains the contract instance $(\sigma_B, \mathtt{VSC}_i(\mathcal{C}))$, where $\sigma_B := \mathtt{Init}_i^{\mathcal{C}}(\text{Bob}, \tau, \gamma)$ is the initial state representing γ. This "copy" of the virtual channel γ will be stored in $\beta.\mathsf{cspace}$ under the identifier $cid_B := \text{Bob}||\gamma.\mathsf{id}$.

The contract functions of $\mathtt{VSC}_i(\mathcal{C})$ are defined in such a way that they provide Ingrid with enough time to react on possible changes in cid_A or cid_B and to always keep both channel "copies" in the same state. Since Ingrid plays the role of Bob in contract cid_A, and the role of Alice in contract cid_B, in order to prevent her from loosing money, she has to react to events happening in one of the contracts and mimic them in the corresponding other contract. For example, consider the situation when Alice executes a function f via the "contract instance execute protocol". If Alice calls f with input parameters z in time τ, then Ingrid immediately executes f with the same input parameters z and the same timings in the contract instance referred to by cid_B. In some sense, for the users in $\gamma.\mathsf{end\!-\!users}$, the contracts referred to by cid_A and cid_B are now representing the contracts running on the ledger. They guarantee that as long as the parties $\gamma.\mathsf{end\!-\!users}$ behave honestly, they will never loose money.

Before we move to the formal description of our construction, we will now take a look at a simple example for the case when $i = 3$ (see Fig. 4). Suppose that each two consecutive parties $P_1, \ldots, P_4$ have *ledger* state channel with each other. If P_1 and P_4 want to create a virtual state channel using the underlying *ledger* state channels, they can proceed recursively as follows. First, P_1 and P_3 create a virtual state channel γ' of length 2 between each other, where P_2 takes the role of Ingrid. This is done by creating a contract instance from type $\mathtt{VSC}_2(\mathcal{C} \cup \mathtt{VSC}_3(\mathcal{C}))$ in the ledger state channel between P_1 and P_2, resp. between P_2 and P_3. Let us take a closer look at the meaning of the contract type $\mathtt{VSC}_2(\mathcal{C} \cup \mathtt{VSC}_3(\mathcal{C}))$. Very informally, this contract type says that the virtual state channel is of length 2 (this is the reason for $\mathtt{VSC}_2$), and that γ' can be used by its end-users to create contracts of type $\mathcal{C}$ and $\mathtt{VSC}_3(\mathcal{C})$. The later are contracts that represent virtual state channels of length 3, which allows its end-users (of the length 3 virtual channel) to open contracts from type $\mathcal{C}$. Next, parties P_1 and P_4 can open the virtual state channel of length 3, where party P_3 will take the role of Ingrid. To this end, P_1 and P_3 will use their previously created virtual state channel γ', and P_3 and P_4 will update their ledger state channel. The contract instance in these two channels is from type $\mathtt{VSC}_3(\mathcal{C})$.

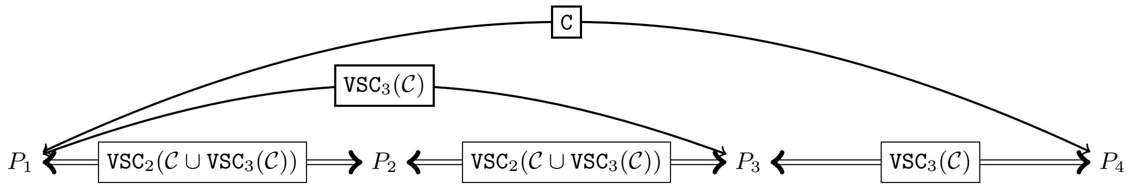
C
$\mathtt{VSC}_3(\mathcal{C})$
P_1
$\mathtt{VSC}_2(\mathcal{C} \cup \mathtt{VSC}_3(\mathcal{C}))$
P_2
$\mathtt{VSC}_2(\mathcal{C} \cup \mathtt{VSC}_3(\mathcal{C}))$
P_3
$\mathtt{VSC}_3(\mathcal{C})$
P_4

Fig. 4: The contracts opened in state channels in order to create a virtual channel of length 3 in which a contract $\mathtt{C} \in \mathcal{C}$ was opened.

7 Protocol Virtual Channels

We will now describe the protocol $\Pi(i, \mathcal{C})$ that $\mathcal{E}_{res}$-realizes the ideal functionality $\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i, \mathcal{C})$ for $i > 1$. The protocol is in the hybrid world with the hybrid ideal functionality which allows to create, update, execute and close state channels of lengths up to $i-1$ in which contract instances of type from the set $\texttt{VSC}_i(\mathcal{C}) \cup \mathcal{C}$ can be constructed, i.e. the functionality $\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i-1, \texttt{VSC}_i(\mathcal{C}) \cup \mathcal{C})$.

The protocol consists of four subprotocols: Create a virtual state channel, Contract instance update, Contract instance execute and Close a virtual state channel. Similarly as for ledger state channels, we will additionally define a procedure $\texttt{Register}_i(P, id, cid)$ that registers a contract instance in a virtual state channel of length i and can be called by parties of the protocol $\Pi(i, \mathcal{C})$.

The protocol $\Pi(i, \mathcal{C})$ has to handle messages about channels of any length j, where $1 \leq j \leq i$. If a party P of the protocol $\Pi(i, \mathcal{C})$ is instructed by the environment to create, update, execute or close a channel of length $1 \leq j < i$, the party forwards this message to the hybrid ideal functionality $\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i-1, \texttt{VSC}_i(\mathcal{C}) \cup \mathcal{C})$.[7] Concretely, in case of create and close P acts as a dummy party and directly forwards the message without any modifications. In case of update and execute, for technical reasons the party adds a prefix "external" to the contract instance identifier. So for example, if the party P receives the message $(\text{execute}, id, cid, f, z)$, where id refers to a channel of length $j < i$, the forwarded message is $(\text{execute}, id, external||cid, f, z)$. The purpose of the prefix is to ensure that the contract instance identifiers which are of the form $\text{Alice}||id$ or $\text{Bob}||id$ for $id \in \{0,1\}^*$ are reserved for contract instances corresponding to the virtual channel of length i with the identifier id. These contract instances can be updated or executed only by parties of the protocol $\Pi(i, \mathcal{C})$ creating, maintaining or closing the corresponding virtual channel of length i.

From now on we will focus on the protocol $\Pi(i, \mathcal{C})$ in case of channels of length exactly i.

Create a virtual state channel. To create the virtual state channel γ of length i in which contract instances of type from set $\mathcal{C}$ can be constructed, the environment sends a message (create, γ) to all three parties $\gamma.\mathsf{Alice}, \gamma.\mathsf{Bob}$ and $\gamma.\mathsf{Ingrid}$ in the same round τ_0. The creation of γ then works at a high level as follows.

As already explained in Section 6, end-users of the channel, $\gamma.\mathsf{Alice}$ and $\gamma.\mathsf{Bob}$, both need to construct a new contract instance of type $\texttt{VSC}_i(\mathcal{C})$ in the subchannels they each have with $\gamma.\mathsf{Ingrid}$. Let us denote these channels by α, β in the outline that follows below. To create these contract instances, party $\gamma.\mathsf{Alice}$ first locally computes the constructor $\texttt{Init}_i^{\mathcal{C}}(\gamma.\mathsf{Alice}, \tau, \gamma)$ to obtain the initial admissible contract storage of type $\texttt{VSC}_i(\mathcal{C})$. Recall that informally this contract storage can be viewed as a "copy" of the virtual channel γ. Thereafter, she sends an update request of the channel α to the hybrid ideal functionality $\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i-1, \texttt{VSC}_i(\mathcal{C}) \cup \mathcal{C})$. At the same time, $\gamma.\mathsf{Bob}$ analogously requests the update of the channel β. If $\gamma.\mathsf{Ingrid}$ receives update requests of both channels α and β from the hybrid ideal functionality, she immediately confirms both of them. As already mentioned before, it is crucial for $\gamma.\mathsf{Ingrid}$ that either both contract instances are created (meaning both her channels α and β are updated) or none of them. Only then she has a guarantee that if she looses coins in the subchannel α because of the virtual channel γ, she can claim these coins back from the subchannel β.

To ensure that at the end of the protocol two honest users $\gamma.\mathsf{Alice}$ and $\gamma.\mathsf{Bob}$ can conclude whether the virtual channel γ was successfully created, there is one additional technicality in our protocol. Notice that if $\gamma.\mathsf{Alice}$ would know whether $\gamma.\mathsf{Ingrid}$ is honest, once she receives a confirmation that her update request of α was successfully competed, she could conclude that the virtual channel is created. However, $\gamma.\mathsf{Alice}$ does not have any information about the channel between $\gamma.\mathsf{Ingrid}$ and $\gamma.\mathsf{Bob}$, and, in particular, whether it was updated for creating γ. It might be the case that malicious $\gamma.\mathsf{Ingrid}$ did not confirm the update request of the channel β which led $\gamma.\mathsf{Bob}$ to conclude that the virtual channel γ was not created. To guarantee that when both $\gamma.\mathsf{Alice}$ and $\gamma.\mathsf{Bob}$ are honest they will agree on whether γ was opened, they exchange confirmation messages at the end of the protocol. Thus, if creation of a virtual state channel is successful, both end-users output $(\text{created}, \gamma)$ to the environment after 3 rounds. Notice that internally when something during virtual channel creation went wrong (e.g., Ingrid misbehaved) the parties may still run registration of contract instances in the subchannels. This is internally handled by the hybrid functionality and does not require the end-users of the virtual channel to wait for in order to reach an agreement whether γ was created or not.

[7] Recall that we assume with our restrictions on the environment that the environment never lies to an honest party about the length of a virtual state channel, so this forwarding can be implemented in a trivial way.

We emphasize that creating a virtual channel runs in constant time – independent of the ledger processing time Δ and length of the channel. This is in contrast to the *ledger* state channels with require always 2Δ time for creation. Formal description of the protocol for ledger channel creation and the corresponding part of the contract type $\mathtt{VSC}_i(\mathcal{C})$ can be found below.

Protocol $\Pi(i, \mathcal{C})$: Create a virtual channel

We use the notation established in Section 3.3 and denote the hybrid functionality as $\mathcal{F}_{ch} := \mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i - 1, \mathtt{VSC}_i(\mathcal{C}) \cup \mathcal{C})$. In addition, let $\mathtt{C} := \mathtt{VSC}_i(\mathcal{C})$.

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

1. Compute $\tilde{\sigma}_T := \mathtt{Init}_i^{\mathcal{C}}(T, \tau_0; \gamma)$ and send $(\text{update}, id_T, cid_T, \tilde{\sigma}_T, \mathtt{C}) \overset{\tau_0}{\hookrightarrow} \mathcal{F}_{ch}$, where $cid_T := T||\gamma.\mathsf{id}$ and $id_T := \gamma.\mathsf{subchan}(T)$.

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

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

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

4. If $(\text{updated}, id_T, cid_T) \xleftarrow{\tau_0+2} \mathcal{F}_{ch}$, then send $(\text{create–ok}, \gamma) \xrightarrow{\tau_0+2} \gamma.\mathsf{other\text{–}party}(T)$. If $(\text{create–ok}, \gamma) \xleftarrow{\tau_0+3} \gamma.\mathsf{other\text{–}party}(T)$, then set $\Gamma^T(\gamma.\mathsf{id}) := \gamma$ and output $(\text{created}, \gamma) \xrightarrow{\tau_0+3} \mathcal{Z}$.
5. Wait till time $\gamma.\mathsf{validity}$.

Contract $\mathtt{VSC}_i(\mathcal{C})$: constructor $\mathtt{Init}_i^{\mathcal{C}}(P, \tau, \gamma)$

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

$$\begin{aligned}
(\sigma.\mathsf{user_L}, \sigma.\mathsf{user_R}) &:= \begin{cases} (\gamma.\mathsf{Alice}, \gamma.\mathsf{Ingrid}), & \text{if } P = \gamma.\mathsf{Alice}, \\ (\gamma.\mathsf{Ingrid}, \gamma.\mathsf{Bob}), & \text{if } P = \gamma.\mathsf{Bob}, \end{cases} \\
(\sigma.\mathsf{cash}(\sigma.\mathsf{user}_L), \sigma.\mathsf{cash}(\sigma.\mathsf{user}_R)) &:= (\gamma.\mathsf{cash}(\gamma.\mathsf{Alice}), \gamma.\mathsf{cash}(\gamma.\mathsf{Bob})), \\
\sigma.\mathsf{virtual\text{–}channel} &:= \gamma, \\
\sigma.\mathsf{cspace}(cid) &:= \bot, \text{ for all } cid \in \{0,1\}^*, \\
(\sigma.\mathsf{aux}_R, \sigma.\mathsf{aux}_E) &:= (\emptyset, \emptyset).
\end{aligned}$$

Resister a contract instance in a virtual state channel. Similarly to the procedure `Register` defined for ledger channels, the subprotocol $\mathtt{Register}_i$ is called with parameters (P, id, cid) the first time end-users of a virtual channel γ with identifier id disagree on a contract instance $\nu := \gamma.\mathsf{cspace}(cid)$. Intuitively, we need the intermediate party $\gamma.\mathsf{Ingrid}$ to play the role of the ledger and resolve the dispute between $\gamma.\mathsf{Alice}$ and $\gamma.\mathsf{Bob}$. If the intermediary would be trusted, then both end-users could simply send their latest contract instance version to $\gamma.\mathsf{Ingrid}$, who would then decide whose contract instance version is the latest valid one. Unfortunately, the situation is more complicated since $\gamma.\mathsf{Ingrid}$ is not a trusted party. She might, for example, stop communicating or collude with one of the end-users. This is the point where the contract instances of type $\mathtt{VSC}_i(\mathcal{C})$ created in the underlying subchannels during the virtual channel creation play an important role. Parties instead of sending versions of ν directly to each other send them indirectly by executing the contract instances in their subchannels with $\gamma.\mathsf{Ingrid}$ on the contract function $\mathtt{RegisterInstance}_i^{\mathcal{C}}$. Since this execution of the contract instance in the subchannel cannot be stopped (i.e., in the worst case it may involve the ledger which will resolve the conflict), this guarantees that the end-users eventually can settle the latest state on which they both have agreed on.

Let us now take a closer look at how this is achieved by $\mathtt{VSC}_i(\mathcal{C})$. Let $cid_A := \gamma.\mathsf{Alice}||\gamma.\mathsf{id}$ be the contract instance of type $\mathtt{VSC}_i^{\mathcal{C}}$ stored in the channel $\alpha := \gamma.\mathsf{subchan}(\gamma.\mathsf{Alice})$ and $cid_B := \gamma.\mathsf{Bob}||\gamma.\mathsf{id}$ the contract instance of type $\mathtt{VSC}_i^{\mathcal{C}}$ stored in the channel $\beta := \gamma.\mathsf{subchan}(\gamma.\mathsf{Bob})$. The initiating party (assume for now that it is $\gamma.\mathsf{Alice}$) first executes cid_A on the function $\mathtt{RegisterInstance}_i^{\mathcal{C}}$ with input parameters (cid, ν^A), where ν^A is $\gamma.\mathsf{Alice}$'s current off-chain contract instance version. Notice that this execution is in a channel of length strictly less than i and hence will be handled by the trusted hybrid ideal functionality $\mathcal{F}_{ch} := \mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i-1, \mathtt{VSC}_i(\mathcal{C}) \cup \mathcal{C})$.

The contract function $\mathtt{RegisterInstance}_i^{\mathcal{C}}$ is defined in such a way that it first verifies the validity of $\gamma.\mathsf{Alice}$'s contract instance version (it, for example, verifies the signatures, admissibility of the storage with respect to the type of the contract instance, if the amount of coins locked in the contract instance is not negative etc.). If all these checks pass, it stores ν^A together with a time-stamp in the auxiliary attribute aux_R. Then, it outputs $\gamma.\mathsf{Alice}$'s contract instance version ν^A and the identifier cid in the output message. The intermediary $\gamma.\mathsf{Ingrid}$ upon receiving information about the execution of cid_A can now symmetrically execute cid_B with inputs (cid, ν^A) via the ideal functionality $\mathcal{F}_{ch}$. Once $\gamma.\mathsf{Bob}$ is notified about the execution of cid_B, he immediately reacts by executing cid_B again on the contract function $\mathtt{RegisterInstance}_i^{\mathcal{C}}$ but with input parameters (cid, ν^B), where ν^B is Bob's contract instance version. If $\gamma.\mathsf{Bob}$'s version of the contract instance with identifier cid is valid as well, the contract function $\mathtt{RegisterInstance}_i^{\mathcal{C}}$ compares $\gamma.\mathsf{Bob}$'s and $\gamma.\mathsf{Alice}$'s versions and stores the one with higher version number in the attribute $\mathsf{cspace}(cid)$. It outputs the registered version and the identifier cid in the output message. In case $\gamma.\mathsf{Bob}$'s version was registered, $\gamma.\mathsf{Ingrid}$ can complete the registration procedure by symmetrically executing cid_A on input (cid, ν^B). In case $\gamma.\mathsf{Alice}$'s version was registered, then $\gamma.\mathsf{Ingrid}$ only needs to confirm that $\gamma.\mathsf{Alice}$'s version of cid should be registered. She does so by executing cid_A on function $\mathtt{EndRegisterInstance}_i^{\mathcal{C}}$ on the input parameter cid.

In case $\gamma.\mathsf{Bob}$ is corrupt and does not submit a valid contract instance version in time, $\gamma.\mathsf{Ingrid}$ can finalize the registration procedure by executing both cid_A and cid_B on function $\mathtt{EndRegisterInstance}_i^{\mathcal{C}}$ on the input parameter cid. Similarly, if $\gamma.\mathsf{Ingrid}$ is corrupt and stops communicating, $\gamma.\mathsf{Alice}$ can after certain time finalize the registration by executing cid_A on function $\mathtt{EndRegisterInstance}_i^{\mathcal{C}}$ with the input parameter cid. The registration procedure of a virtual channel of length i can take up to

$$\text{TimeRegister}(i) := 5 \cdot \text{TimeExecute}(\lceil i/2 \rceil) \tag{1}$$

rounds (this corresponds to the pessimistic case when $\gamma.\mathsf{Alice}$ has to finalize the registration).

Before we give the full description of the registration protocol and the corresponding contract parts of $\mathtt{VSC}_i(\mathcal{C})$, let us explain here the reason why we restrict the number of contract instances in a virtual channel (although the syntax as defined in Section 3.2 supports infinitely many contract instances as in the ledger channel).

Assume the following scenario. Alice and Bob open a virtual channel on top of two ledger channels which they each have with Ingrid and thereafter they create (off-line) a large amount of contract instances in this virtual channel. At some point Alice starts registering all the contract instances by executing the subchannel she has with Ingrid. According to the protocol, Ingrid has to symmetrically execute the subchannel she has with Bob otherwise she might loose money. If Bob is corrupt and does not react on peaceful execution requests, Ingrid has to execute all the requests forcefully on the blockchain. While in our theoretical model this is not an issue, in practice, this step would be very expensive for Ingrid due to the large amount of fees Ingrid would have to pay to the miners in common cryptocurrencies such as the Ethereum network.

Thus, if Ingrid has no control on the amount of contract instances that Alice and Bob can create, the two parties can force Ingrid to pay arbitrary amount of money in fees. Therefore, we restrict the number of contract instance that Alice and Bob can open in a virtual channel and hence give Ingrid the ability to estimate the costs in fees that might result from the virtual channel (recall that Ingrid had to agree with the virtual channel creation; in particular, with the channel length and the contract types that can be open in the virtual channel).

Procedure $\texttt{Register}_i(P, id, cid)$

We use the notation from Section 3.3 and denote $\mathcal{F}_{ch} := \mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i-1, \texttt{VSC}_i(\mathcal{C}) \cup \mathcal{C})$ and $\text{TE}_{sub} := \text{TimeExecute}(\lceil i/2 \rceil)$.

Party P:

1. Let $\gamma^P := \Gamma^P(id)$, $\nu^P := \gamma^P.\mathsf{cspace}(cid)$, $id_P := \gamma^P.\mathsf{subchan}(P)$, $cid_P := P||id$ and let τ_0^P be the current round. Then send $(\text{execute}, id_P, cid_P, \texttt{RegisterInstance}_i^{\mathcal{C}}, (cid, \nu^P)) \xhookrightarrow{\tau_0^P} \mathcal{F}_{ch}$ and goto step 7.

Party I:

2. Upon $(\text{executed}, id_P, cid_P, \sigma_P, L_P, R_P, m_P) \xhookleftarrow{\tau_1^I} \mathcal{F}_{ch}$, where $m_P = (\text{instance–registering}, cid, \nu^P)$ proceed as follows. Set $\gamma^I := \sigma_P.\mathsf{virtual\text{–}channel}$, $P := \gamma^I.\mathsf{end\text{–}users} \cap \{\sigma_P.\mathsf{user}_L, \sigma_P.\mathsf{user}_R\}$, $Q := \gamma^I.\mathsf{other\text{-}party}(P)$, $id_Q := \gamma^I.\mathsf{subchan}(Q)$ and $cid_Q := Q||\gamma^I.\mathsf{id}$. Then send $(\text{execute}, id_Q, cid_Q,$ $\texttt{RegisterInstance}_i^{\mathcal{C}}, (cid, \nu^P)) \xhookrightarrow{\tau_1^I} \mathcal{F}_{ch}$ and goto step 5.

Party Q:

3. Upon $(\text{executed}, id_Q, cid_Q, \sigma_Q, L_Q, R_Q, m_Q) \xhookleftarrow{\tau_1^Q} \mathcal{F}_{ch}$, where $m_Q := (\text{instance–registering}, cid, \nu^P)$, proceed as follows. Parse $Q||id := cid_Q$, set $\gamma^Q := \Gamma^Q(id)$, $\nu^Q := \gamma^Q.\mathsf{cspace}(cid)$ and then send $(\text{execute}, id_Q, cid_Q, \texttt{RegisterInstance}_i^{\mathcal{C}}, (cid, \nu^Q)) \xhookrightarrow{\tau_1^Q} \mathcal{F}_{ch}$.
4. Upon $(\text{executed}, id_Q, cid_Q, \tilde{\sigma}_Q, \tilde{L}_Q, \tilde{R}_Q, \tilde{m}_Q) \xhookleftarrow{\tau_2^Q \leq \tau_1^Q + \text{TE}_{sub}} \mathcal{F}_{ch}$, where $m = (\text{instance–registered},$ $cid, \widehat{\nu})$, set $\Gamma^Q := \texttt{LocalUpdate}(\Gamma^Q, id, cid, \widehat{\nu})$.

Party I:

5. If you receive $(\text{executed}, id_Q, cid_Q, \tilde{\sigma}_Q, \tilde{L}_Q, \tilde{R}_Q, \tilde{m}_Q) \xhookleftarrow{\tau_2^I \leq \tau_1^I + 2 \cdot \text{TE}_{sub}} \mathcal{F}_{ch}$, where $\tilde{m}_Q = (\text{instance–registered}, cid, \widehat{\nu})$, then send $(\text{execute}, id_P, cid_P, \texttt{RegisterInstance}_i^{\mathcal{C}}, (cid, \widehat{\nu})) \xhookrightarrow{\tau_2^I} \mathcal{F}_{ch}$.
6. Else send messages $(\text{execute}, id_P, cid_P, \texttt{EndRegisterInstance}_i^{\mathcal{C}}, cid) \xhookrightarrow{\tau_1^I + 2 \cdot \text{TE}_{sub}} \mathcal{F}_{ch}$ and $(\text{execute},$ $id_Q, cid_Q, \texttt{EndRegisterInstance}_i^{\mathcal{C}}, cid) \xhookrightarrow{\tau_1^I + 2 \cdot \text{TE}_{sub}} \mathcal{F}_{ch}$.

Party P:

7. If not $(\text{executed}, id_P, cid_P, \tilde{\sigma}_P, \tilde{L}_P, \tilde{R}_P, \tilde{m}_P) \xhookleftarrow{\leq \tau_0^P + 4 \cdot \text{TE}_{sub}} \mathcal{F}_{ch}$, where $\tilde{m}_P = (\text{instance–registered},$ $cid, \widehat{\nu})$, then send $(\text{execute}, id_P, cid_P, \texttt{EndRegisterInstance}_i^{\mathcal{C}}, cid) \xhookrightarrow{\tau_0^P + 4 \cdot \text{TE}_{sub}} \mathcal{F}_{ch}$.
8. Upon $(\text{executed}, id_P, cid_P, \tilde{\sigma}_P, \tilde{L}_P, \tilde{R}_P, \tilde{m}_P) \xhookleftarrow{\leq \tau_0^P + 5 \cdot \text{TE}_{sub}} \mathcal{F}_{ch}$, where $\tilde{m}_P = (\text{instance–registered},$ $cid, \widehat{\nu})$, then set $\Gamma^P := \texttt{LocalUpdate}(\Gamma^P, id, cid, \widehat{\nu})$.

Contract $\texttt{VSC}_i(\mathcal{C})$

Function $\texttt{RegisterInstance}_i^{\mathcal{C}}(\sigma, P, \tau; (cid, \nu_n))$

Let $\gamma := \sigma.\mathsf{virtual\text{–}channel}, id := \gamma.\mathsf{id}, A := \gamma.\mathsf{Alice}, B := \gamma.\mathsf{Bob}, I := \gamma.\mathsf{Ingrid}$. Let $\mathtt{C} := \nu_n.\mathsf{type}$, $\sigma_n := \nu_n.\mathsf{storage}$, $v := \nu_n.\mathsf{version}$, $s_A := \nu_n.\mathsf{sign}(A)$, $s_B := \nu_n.\mathsf{sign}(B)$ and $\mathrm{TE}_{sub} := \mathrm{TimeExecute}(\lceil i/2 \rceil)$.

1. If $P \notin \{\sigma.\mathsf{user}_L, \sigma.\mathsf{user}_R\}$, then output $(\sigma, 0, 0, \bot)$.
2. If $\mathtt{Vfy}_{pk_A}(id, cid, \sigma_n, \mathtt{C}, v; s_A) \neq 1$ or $\mathtt{Vfy}_{pk_B}(id, cid, \sigma_n, \mathtt{C}, v; s_B) \neq 1$ or $\{\sigma_n.\mathsf{user}_L, \sigma_n.\mathsf{user}_R\} \neq \{A, B\}$ or $\mathtt{C} \notin \mathcal{C}$ or $\sigma_n \notin \mathtt{C}.\Lambda$ or $\sigma_n \mathsf{cash}(\sigma_n.\mathsf{user}_L) + \sigma_n.\mathsf{cash}(\sigma_n.\mathsf{user}_R) < 0$ or there exists $cid' \in \{0,1\}^*$ such that $\sigma.\mathsf{cspace}(cid') \neq \bot$, then output $(\sigma, 0, 0, \bot)$.
3. Else define $\tilde{\sigma} := \sigma$ and consider the following cases:
   - If there is no tuple $(Q, \tau^Q; cid, \nu_n^Q)$ in $\sigma.\mathsf{aux}_R$, i.e. none of the parties has registered the contract instance with identifier cid, and

     $$\tau \leq \begin{cases} \gamma.\mathsf{validity} & \text{if } P \in \{A, B\}, \\ \gamma.\mathsf{validity} + \mathrm{TE}_{sub} & \text{if } P = I, \end{cases}$$

     then add $(P, \tau; cid, \nu_n)$ to $\tilde{\sigma}.\mathsf{aux}_R$ and output $(\tilde{\sigma}, 0, 0, m)$, where $m := (\text{instance–registering}, cid, \nu_n)$.
   - If there is $(Q, \tau^Q; cid, \nu_n^Q)$ in $\sigma.\mathsf{aux}_R$, where $Q \neq P$, and

     $$0 < \tau - \tau^Q \leq \begin{cases} \mathrm{TE}_{sub}, & \text{if } P \in \{A, B\}, \\ 3 \cdot \mathrm{TE}_{sub}, & \text{if } P = I, \end{cases}$$

     then set $\widehat{\nu}_n := \nu_n$ if $v \geq \nu_n^Q.\mathsf{version}$ and otherwise let $\widehat{\nu}_n := \nu_n^Q$. Then make the following changes: set $\tilde{\sigma}.\mathsf{cspace}(cid) := (\widehat{\nu}_n.\mathsf{storage}, \widehat{\nu}_n.\mathsf{type})$ and modify the cash attributes accordingly (for example, assuming that $\sigma.\mathsf{user}_\mathsf{L} = \sigma_n.\mathsf{user}_L$ and denoting the registered instance $\tilde{\sigma}_n := \tilde{\sigma}.\mathsf{cspace}(cid).\mathsf{storage}$, the value $\tilde{\sigma}.\mathsf{cash}(\tilde{\sigma}.\mathsf{user}_L)$ will be set to the value of $\sigma.\mathsf{cash}(\sigma.\mathsf{user}_L) - \tilde{\sigma}_n.\mathsf{cash}(\sigma_n.\mathsf{user}_L)$). Finally, delete $(Q, \tau^Q; cid, \nu_n^Q)$ from $\tilde{\sigma}.\mathsf{aux}_R$ and output $(\tilde{\sigma}, 0, 0, m)$, where $m := (\text{instance–registered}, cid, \widehat{\nu}_n)$.

Function $\texttt{EndRegisterInstance}_i^{\mathcal{C}}(\sigma, P, \tau; cid)$

Let $\gamma := \sigma.\mathsf{virtual\text{–}channel}$, $I := \gamma.\mathsf{Ingrid}$, $\mathrm{TR}_i := \mathrm{TimeRegister}(i)$ and $\mathrm{TE}_{sub} := \mathrm{TimeExecute}(\lceil i/2 \rceil)$

1. If $P \notin \{\sigma.\mathsf{user}_L, \sigma.\mathsf{user}_R\}$ or $\tau > \gamma.\mathsf{validity} + \mathrm{TR}_i$, then output $(\sigma, 0, 0, \bot)$.
2. If for every $cid' \in \{0,1\}^*$ it holds that $\sigma.\mathsf{cspace}(cid') = \bot$ and there is $(Q, \tau^Q; cid, \nu_n^Q)$ in $\sigma.\mathsf{aux}_R$, such that either $P \neq Q$ or $P = Q$ and

   $$\tau - \tau^Q > \begin{cases} 4 \cdot \mathrm{TE}_{sub}, & \text{if } P = \{A, B\}, \\ 2 \cdot \mathrm{TE}_{sub}, & \text{if } P = I, \end{cases}$$

   then let $\tilde{\sigma} := \sigma$ and make the following changes. Delete $(Q, \tau^Q; cid, \nu_n^Q)$ from $\tilde{\sigma}.\mathsf{aux}_R$ and set $\tilde{\sigma}.\mathsf{cspace}(cid) := (\nu_n^Q.\mathsf{storage}, \nu_n^Q.\mathsf{type})$. Thereafter modify the attribute $\tilde{\sigma}.\mathsf{cash}$ accordingly and output $(\tilde{\sigma}, 0, 0, m)$, where $m := (\text{instance–registered}, cid, \nu_n^Q)$.
3. Else output $(\sigma, 0, 0, \bot)$.

Update a contract instance in a virtual state channel. As long as both end-users of a virtual state channel are honest, they can update a contract instance exactly the same way as if it would be a ledger state channel. That means that parties exchange signatures on the new contract instance version (see Section 5.2 for more details). The differences between update of a ledger channel and a virtual channel appears only when end-users of the channel run into dispute, i.e., when the parties run the contract instance registration procedure, which was defined above. The pessimistic time complexity of updating a virtual state channel of length i is equal to $\text{TimeRegister}(i) + 2$.

Execute a contract instance in a virtual channel. In order to execute a contract instance in a virtual state channel, the environment sends a message $(\text{execute}, id, cid, f, z)$ to one of the end-users of the virtual channel. Let us assume for now that this party is $\gamma.\mathsf{Alice}$. The party $\gamma.\mathsf{Alice}$ first tries to execute the contract instance "peacefully", exactly as if γ would be a ledger state channel (see Section 5.2 in the "Execute a contract instance" protocol). In case the peaceful execution fails, $\gamma.\mathsf{Alice}$ registers the contract instance by calling the subproderure $\mathtt{Register}_i(\gamma.\mathsf{Alice}, id, cid)$. Next, $\gamma.\mathsf{Alice}$ has to execute the contract instance "forcefully" via the intermediary of the channel; $\gamma.\mathsf{Ingrid}$. Since the intermediary is not trusted, execution must be performed by executing the contract instances of type $\mathtt{VSC}_i(\mathcal{C})$ stored in the underlying subchannels via the hybrid ideal functionality $\mathcal{F}_{ch}$ (recall that the contract instance in the subchannel α between $\gamma.\mathsf{Alice}$ and $\gamma.\mathsf{Ingrid}$ is stored under the identifier $cid_A := \gamma.\mathsf{Alice}||\gamma.\mathsf{id}$ and the contract instance in the channel β between $\gamma.\mathsf{Bob}$ and $\gamma.\mathsf{Ingrid}$ is stored under the identifier $cid_B := \gamma.\mathsf{Bob}||\gamma.\mathsf{id}$).

The first attempt would be to let $\gamma.\mathsf{Alice}$ execute cid_A on the function $\mathtt{ExecuteInstance}_i^{\mathcal{C}}$ with parameters $param = (cid, \gamma.\mathsf{Alice}, \tau, f, z, s_A)$, where τ is the round in which $\gamma.\mathsf{Alice}$ received the message from the environment and s_A is $\gamma.\mathsf{Alice}$'s signature on the tuple $(cid, \gamma.\mathsf{Alice}, \tau, f, z)$. The contract function $\mathtt{ExecuteInstance}_i^{\mathcal{C}}$ would be defined such that it verifies $\gamma.\mathsf{Alice}$'s signature and then internally executes the contract instance with identifier cid. After successful execution of cid_A, $\gamma.\mathsf{Ingrid}$ would symmetrically execute cid_B on the same contract function $\mathtt{ExecuteInstance}_i^{\mathcal{C}}$ and the same input parameters $param = (cid, \gamma.\mathsf{Alice}, \tau, f, z, s_A)$. The entire process of force execution would then take $5 + \text{TimeRegister}(i) + 2 \cdot \text{TimeExecute}(\lceil i/2 \rceil)$.

Let us explain with the following example why this straightforward solution does not work, which is due to allowing that parties interact fully concurrently. Assume that while the execution between $\gamma.\mathsf{Alice}$ and $\gamma.\mathsf{Ingrid}$ is running, $\gamma.\mathsf{Bob}$ wants to forcefully execute the contract instance with identifier cid in round $\tau' = \tau + 1$ on different inputs. This means that before $\gamma.\mathsf{Ingrid}$ has time to execute cid_B on $\gamma.\mathsf{Alice}$'s request, $\gamma.\mathsf{Bob}$ executes cid_B on the function $\mathtt{ExecuteInstance}_i^{\mathcal{C}}$ with his own parameters $param' = (cid, B, \tau', f', z', s_B)$. Consequently, the order of internal execution of the contract instance cid is different in cid_A and cid_B. Depending on the contract type of cid, this asymmetry may lead to $\gamma.\mathsf{Ingrid}$ loosing money.

To overcome this difficulty, we define the contract function $\mathtt{ExecuteInstance}_i^{\mathcal{C}}$ in such a way that when it is executed by $\gamma.\mathsf{Bob}$ on some parameters $param'$, the request is stored in the attribute aux_E but the internal execution of the contract instance cid is not performed yet. The function outputs details of the request in its output message. In other words, execution of the contract instance cid_B on the function $\mathtt{ExecuteInstance}_i^{\mathcal{C}}$ with $param' = (cid, \gamma.\mathsf{Bob}, \tau', f, z, s_A)$ only informs $\gamma.\mathsf{Ingrid}$ about $\gamma.\mathsf{Bob}$'s intention to internally execute the contract instance cid and gives her time to execute potential $\mathtt{ExecuteInstance}$ requests with the same cid which were made earlier by $\gamma.\mathsf{Alice}$. Once $\gamma.\mathsf{Ingrid}$ successfully executes $\gamma.\mathsf{Bob}$'s request in cid_A, she can finalize the execution of cid_B via the function $\mathtt{EndExecuteInstance}$. It works analogously for the contract instance cid_A in the channel between $\gamma.\mathsf{Alice}$ and $\gamma.\mathsf{Ingrid}$. If $\gamma.\mathsf{Bob}$'s execution request was not finalized by $\gamma.\mathsf{Ingrid}$ within certain amount of time, $\gamma.\mathsf{Bob}$ can finalize his execution himself via the function $\mathtt{EndExecute Instance}$. To conclude, the contract instance execution protocol of a virtual channel of length i can take up to $\text{TimeExecute}(i) := 2 \cdot (5 + \text{TimeRegister}(i) + 2 \cdot \text{TimeExecute}(\lceil i/2 \rceil))$ rounds. Using the Equation (1) we obtain

$$\text{TimeExecute}(i) := 10 + 14 \cdot \text{TimeExecute}(\lceil i/2 \rceil)). \tag{2}$$

The previous description omits many technicalities and we refer the reader for further details to the full specification below.

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

We use the notation established in Section 3.3 and denote $\mathcal{F}_{ch} := \mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i-1, \texttt{VSC}_i(\mathcal{C}) \cup \mathcal{C})$. In addition, let $\mathrm{TE}_{sub} := \mathrm{TimeExecute}(\lceil i/2 \rceil)$ and $\mathrm{TR}_i := \mathrm{TimeRegister}(i)$.

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

1. Let $\gamma^P := \mathit{\Gamma}^P(id), \nu^P := \gamma^P.\mathsf{cspace}(cid)$, $\sigma^P := \nu^P.\mathsf{storage}$, $\mathtt{C}^P := \nu^P.\mathsf{type}$, $v^P := \nu^P.\mathsf{version}$.
2. Set $\tau_1 := \tau_0 + x$, where x is the smallest offset such that $\tau_1 = 1 \mod 4$ if $P = \gamma^P.\mathsf{Alice}$ and $\tau_1 = 3 \mod 4$ if $P = \gamma^P.\mathsf{Bob}$. Wait till round τ_1.
3. If (id, cid) is not marked as corrupt in $\mathit{\Gamma}^P$, then compute $(\tilde{\sigma}, add_L, add_R, m) := f(\sigma^P, P, \tau_0, z)$. If $m = \perp$, then stop. Otherwise compute $s_P := \mathtt{Sign}_{sk_P}(id, cid, \tilde{\sigma}, \mathtt{C}^P, v^P + 1)$, send (peaceful–request, $id, cid, f, z, s_P, \tau_0) \xhookrightarrow{\tau_1} Q$ and goto step 10.
4. If (id, cid) is marked as corrupt, proceed as follows. If (id, cid) not marked as registered, then run $\mathtt{Register}_i(P, id, cid)$. Goto step 11.

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

5. Let $\gamma^Q := \mathit{\Gamma}^Q(id), \nu^Q := \gamma^Q.\mathsf{cspace}(cid)$, $\sigma^Q := \nu^Q.\mathsf{storage}$, $\mathtt{C}^Q := \nu^Q.\mathsf{type}$, $v^Q := \nu^Q.\mathsf{version}$. If $\gamma^Q = \perp$ or $P, Q \notin \gamma^Q.\mathsf{end–users}$ or $\nu^Q = \perp$, then goto step 9.
6. If $P = \gamma^Q.\mathsf{Alice}$ and $\tau_Q \mod 4 \neq 2$ or if $P = \gamma.\mathsf{Bob}$ and $\tau_Q \mod 4 \neq 0$, then goto step 9.
7. If $\tau_0 \notin [\tau_Q - 4, \tau_Q - 1]$, then goto step 9.
8. If (id, cid) is not marked as corrupt in $\mathit{\Gamma}^Q$, do:
   (a) Compute $(\tilde{\sigma}, add_L, add_R, m) := f(\sigma^Q, P, \tau_0, z)$.
   (b) If $m = \perp$ or $\mathtt{Vfy}_{pk_P}(id, cid, \tilde{\sigma}, \mathtt{C}^Q, v^Q + 1; s_P) \neq 1$, then goto step 9.
   (c) Else sign $s_Q := \mathtt{Sign}_{sk_Q}(id, cid, \tilde{\sigma}, \mathtt{C}^Q, v^Q + 1)$, send (peaceful–confirm, $id, cid, f, z, s_Q) \xhookrightarrow{\tau_Q} P$, set $\mathit{\Gamma}^Q := \mathtt{LocalUpdateAdd}(\mathit{\Gamma}^Q, id, cid, \tilde{\sigma}, \mathtt{C}^Q, add_L, add_R, v^Q + 1, \{s_P, s_Q\})$, output (executed, $id, cid, \tilde{\sigma}, add_L, add_R, m) \xhookrightarrow{\tau_Q + 1} \mathcal{Z}$ and stop.
9. Mark (id, cid) as corrupt in $\mathit{\Gamma}^Q$. Then goto step 14.

Back to party P

10. Distinguish the following two cases
    – If (peaceful–confirm, $id, cid, f, z, s_Q) \xhookleftarrow{\tau_2 = \tau_1 + 2} Q$ such that $\mathtt{Vfy}_{pk_Q}(id, cid, \tilde{\sigma}, \mathtt{C}^P, v^P + 1; s_Q) = 1$, then set $\mathit{\Gamma}^P := \mathtt{LocalUpdateAdd}(\mathit{\Gamma}^P, id, cid, \tilde{\sigma}, \mathtt{C}^P, add_L, add_R, v^P + 1, \{s_P, s_Q\})$, output (executed, $id, cid, \tilde{\sigma}, add_L, add_R, m) \xhookrightarrow{\tau_2} \mathcal{Z}$ and stop.
    – Else mark (id, cid) as corrupt in $\mathit{\Gamma}^P$ and execute the $\mathtt{Register}_i(P, id, cid)$. Once the procedure is executed (in round $\tau_3 \leq \tau_1 + \mathrm{TR}_i + 2$), distinguish the following two cases:
      - If $\sigma^P = \tilde{\sigma}$, then output (executed, $id, cid, \tilde{\sigma}, add_L, add_R, m) \xhookrightarrow{\tau_3} \mathcal{Z}$ and stop.
      - Else goto step 11.
11. Let τ_4 be the current round, $id_P := \gamma^P.\mathsf{subchan}(P)$, $cid_P := P || id$, $s_n := \mathtt{Sign}_{sk_P}(cid, P, \tau_0, f, z)$ and $p_n := (P, \tau_0, f, z, s_n)$. Then send (execute, $id_P, cid_P, \mathtt{ExecuteInstance}_i^{\mathcal{C}}, (cid, p_n)) \xhookrightarrow{\tau_4} \mathcal{F}_{ch}$.

Party I:

12. Upon receiving (executed, $id_P, cid_P, \sigma_P, L_P, R_P, m_P) \xhookleftarrow{\tau_0^I} \mathcal{F}_{ch}$, where $m_P =$ (instance–executing, cid, p_n, m), proceed as follows. Define $\gamma^I := \sigma_P.\mathsf{virtual–channel}$, $P := \gamma^I.\mathsf{end–users} \cap \{\sigma_P.\mathsf{user}_L, \sigma_P.\mathsf{user}_R\}$, $Q := \gamma^I.\mathsf{other\text{-}party}(P)$, $id_Q := \gamma^I.\mathsf{subchan}(Q)$ and $cid_Q := Q || \gamma^I.\mathsf{id}$.
13. Send (execute, $id_Q, cid_Q, \mathtt{ExecuteInstance}_i^{\mathcal{C}}, (cid, p_n)) \xhookrightarrow{\tau_0^I} \mathcal{F}_{ch}$.

Party Q:

14. Upon receiving (executed, $id_Q, cid_Q, \sigma_Q, L_Q, R_Q, M_Q$) $\xleftarrow{\tau_0^Q}$ $\mathcal{F}_{ch}$, where the message M_Q contains (instance–executed, $cid, \tilde{\sigma}_n, p_n, add_L, add_R, m_n$), then parse $Q||id := cid_Q$, output (executed, id, $cid, \tilde{\sigma}_n, add_L, add_R, m_n$) $\xhookrightarrow{\tau_0^Q}$ $\mathcal{Z}$, set $\mathit{\Gamma}^Q$:= $\texttt{LocalUpdateAdd}(\mathit{\Gamma}^Q, id, cid, \tilde{\sigma}_n, \texttt{C}^Q, add_L, add_R)$ and stop.

Back to party I:

15. Upon receiving (executed, $id_Q, cid_Q, \sigma_Q, L_Q, R_Q, M_Q$) $\xleftarrow{\tau_1^I \leq \tau_0^I + \mathrm{TE}_{sub}}$ $\mathcal{F}_{ch}$, where the message M_Q contains (instance–executed, $cid, \tilde{\sigma}_n, p_n, add_L, add_R, m_n$), send the message (execute, id_P, cid_P, $\texttt{EndExecuteInstance}_i^{\mathcal{C}}$, (cid, p_n)) $\xhookrightarrow{\tau_1^I}$ $\mathcal{F}_{ch}$.

Back to party P:

16. Let $\tau_6 := \tau_4 + 4 + \mathrm{TR}_i + 2 \cdot \mathrm{TE}_{sub}$. If you receive (executed, $id_P, cid_P, \tilde{\sigma}_P, \tilde{L}_P, \tilde{R}_P, M_P$) $\xleftarrow{\tau_5 \leq \tau_6}$ $\mathcal{F}_{ch}$, where M_P contains (instance–executed, $cid, \tilde{\sigma}_n, p_n, add_L, add_R, m_n$), output (executed, $id, cid, \tilde{\sigma}_n$, add_L, add_R, m_n) $\xhookrightarrow{\tau_5}$ $\mathcal{Z}$, set $\mathit{\Gamma}^P := \texttt{LocalUpdateAdd}(\mathit{\Gamma}^P, id, cid, \tilde{\sigma}_n, \texttt{C}^P, add_L, add_R)$ and stop.
17. Else send (execute, id_P, cid_P, $\texttt{EndExecuteInstance}_i^{\mathcal{C}}$, (cid, p_n)) $\xhookrightarrow{\tau_6}$ $\mathcal{F}_{ch}$ and when you receive the message (executed, $id_P, cid_P, \tilde{\sigma}_P, \tilde{L}_P, \tilde{R}_P, M_P$) $\xleftarrow{\tau_7 \leq \tau_6 + \mathrm{TE}_{sub}}$ $\mathcal{F}_{ch}$, where M_P contains (instance–executed, $cid, \tilde{\sigma}_n, p_n, add_L, add_R, m_n$), then output (executed, $id, cid, \tilde{\sigma}_n, add_L, add_R, m_n$) $\xhookrightarrow{\tau_7}$ $\mathcal{Z}$, set $\mathit{\Gamma}^P := \texttt{LocalUpdateAdd}(\mathit{\Gamma}^P, id, cid, \tilde{\sigma}_n, \texttt{C}^P, add_L, add_R)$ and stop.

We will now define the contract functions $\texttt{ExecuteInstance}_i^{\mathcal{C}}$ and $\texttt{EndExecuteInstance}_i^{\mathcal{C}}$ of the contract type $\texttt{VSC}_i(\mathcal{C})$. Both of these functions have to internally execute contract functions. Not to repeat the same code several times, we separately define an auxiliary procedure $\texttt{Evaluate}$.

Contract $\texttt{VSC}_i(\mathcal{C})$

Procedure $\texttt{Evaluate}(\sigma, cid, P_n, \tau_n, f_n, z_n)$

Let $\gamma := \sigma.\mathsf{virtual\text{–}channel}, I := \gamma.\mathsf{Ingrid}$, $\nu := \sigma.\mathsf{cspace}(cid)$, $\sigma_n := \nu.\mathsf{storage}$ and $P := \gamma.\mathsf{end\text{–}users} \cap \{\sigma.\mathsf{user}_L, \sigma.\mathsf{user}_R\}$

1. Compute $(\tilde{\sigma}_n, add_L, add_R, m_n) = f_n(\sigma_n, P_n, \tau_n, z_n)$
2. If $m_n = \perp$, then output $(\sigma, 0, 0, \perp)$.
3. Otherwise let $\tilde{\sigma} := \sigma$ and make the following changes:
   (a) Set $\tilde{\sigma}.\mathsf{cspace}(cid) := (\tilde{\sigma}_n, \nu.\mathsf{type})$
   (b) If $P = \sigma_n.\mathsf{user}_L$, then $\tilde{\sigma}.\mathsf{cash}(P) := \sigma.\mathsf{cash}(P) + add_L$ and $\tilde{\sigma}.\mathsf{cash}(I) := \sigma.\mathsf{cash}(I) + add_R$.
   (c) If $P = \sigma_n.\mathsf{user}_R$, then $\tilde{\sigma}.\mathsf{cash}(P) := \sigma.\mathsf{cash}(P) + add_R$ and $\tilde{\sigma}.\mathsf{cash}(I) := \sigma.\mathsf{cash}(I) + add_L$.
4. Output $(\tilde{\sigma}, add_L, add_R, m_n)$.

Function $\texttt{ExecuteInstance}_i^{\mathcal{C}}(\sigma, P, \tau, (cid, P_n, \tau_n, f_n, z_n, s_n))$

Let $\gamma := \sigma.\mathsf{virtual\text{–}channel}, A := \gamma.\mathsf{Alice}, B := \gamma.\mathsf{Bob}, I := \gamma.\mathsf{Ingrid}$, $\nu := \sigma.\mathsf{cspace}(cid)$ and $\sigma_n := \nu.\mathsf{storage}$. In addition, let $\mathrm{TE}_{sub} := \mathrm{TimeExecute}(\lceil i/2 \rceil)$ and $\mathrm{TR}_i := \mathrm{TimeRegister}(i)$.

1. If $P \notin \{\sigma.\mathsf{user}_L, \sigma.\mathsf{user}_R\}$, then output $(\sigma, 0, 0, \perp)$.

2. If $\nu = \bot$, $P_n \notin \{A, B\}$, $\mathtt{Vfy}_{pk_{P_n}}(cid, P_n, \tau_n, f_n, z_n; s_n) \neq 1$ or f_n is not a contract function with respect to $\nu.\mathsf{C}$, then output output $(\sigma, 0, 0, \bot)$.
3. Distinguish the following two situations:
   - $P \in \{A, B\}$:
     If $\tau - \tau_n > 5 + \mathrm{TR}_i$ or $P \neq P_n$, then output $(\sigma, 0, 0, \bot)$. Else let $\tilde{\sigma} := \sigma$, add $(\tau; cid, P_n, \tau_n, f_n, z_n)$ to $\tilde{\sigma}.\mathsf{aux}_E$ and output $(\tilde{\sigma}, 0, 0, m)$ for $m := (\text{instance–executing}, cid, P_n, \tau_n, f_n, z_n, s_n)$.
   - $P = I$:
     If $\tau - \tau_n > 5 + \mathrm{TR}_i + \mathrm{TE}_{sub}$, then output $(\sigma, 0, 0, \bot)$.
     Else proceed as follows
     (a) Let $Q_n := \{A, B\} \cap \{\sigma.\mathsf{user}_L, \sigma.\mathsf{user}_R\}$ and $\tilde{\sigma}^{(0)} := \sigma$.
     (b) Let $E \subseteq \sigma.\mathsf{aux}_E$ consisting of all tuples $(\tau'; cid, Q_n, \tau'_n, f'_n, z'_n)$, where $\tau'_n \leq \tau_n$.
     (c) Let $|E| = \ell$ and $(e^{(1)}, \ldots, e^{(\ell)})$ be such that $e^{(k)} = (\tau^{(k)}; cid, Q_n, \tau_n^{(k)}, f_n^{(k)}, z_n^{(k)}) \in E$ for every $k \in [1, \ell]$ and $\tau_n^{(1)} \leq \cdots \leq \tau_n^{(\ell)}$.
     (d) For $k = 1$ to ℓ
        i. Compute $(\tilde{\sigma}^{(k)}, add_L^{(k)}, add_R^{(k)}, m^{(k)}) := \mathtt{Evaluate}(\tilde{\sigma}^{(k-1)}, cid, Q_n, \tau_n^{(k)}, f_n^{(k)}, z_n^{(k)})$.
        ii. Delete $e^{(k)}$ from $\tilde{\sigma}^{(k)}.\mathsf{aux}_E$.
     (e) Compute $(\tilde{\sigma}, add_R, add_R, m) := \mathtt{Evaluate}(\tilde{\sigma}^{(\ell)}, cid, P_n, \tau_n, f_n, z_n)$.
     (f) Output $(\tilde{\sigma}, 0, 0, M||M^{(1)}||\ldots||M^{(\ell)})$, for $M := (\text{instance–executed}, \tilde{\sigma}.\mathsf{cspace}(cid), p, add_L, add_R, m)$ for $p = (cid, P_n, \tau_n, f_n, z_n)$, and $M^{(k)} := (\text{instance–executed}, \tilde{\sigma}.\mathsf{cspace}(cid), p^{(k)}, add_L^{(k)}, add_R^{(k)}, m^{(k)})$ for $p^{(k)} = (cid, Q_n, \tau_n^{(k)}, f_n^{(k)}, z_n^{(k)})$ for every $k \in [\ell]$.

Function $\mathtt{EndExecuteInstance}_i^{\mathcal{C}}(\sigma, P, \tau, (cid, P_n, \tau_n, f_n, z_n))$

Let $\gamma := \sigma.\mathsf{virtual–channel}, A := \gamma.\mathsf{Alice}, B := \gamma.\mathsf{Bob}$, $\nu := \mathsf{cspace}(cid)$, $\sigma := \nu.\mathsf{storage}$, $\mathrm{TE}_{sub} := \mathrm{TimeExecute}(\lceil i/2 \rceil)$ and $\mathrm{TR}_i := \mathrm{TimeRegister}(i)$.

1. If $P \notin \{\sigma.\mathsf{user}_L, \sigma.\mathsf{user}_R\}$, then output $(\sigma, 0, 0, \bot)$.
2. If there is no entry $(\tau'; cid, P_n, \tau_n, f_n, z_n)$ in $\sigma.\mathsf{aux}_E$, then output $(\sigma, 0, 0, \bot)$.
3. If $P = P_n$ and $\tau - \tau' < 5 + \mathrm{TR}_i + 2 \cdot \mathrm{TE}_{sub}$, then output $(\sigma, 0, 0, \bot)$.
4. Else compute $(\tilde{\sigma}, add_L, add_R, m) := \mathtt{Evaluate}(\sigma, cid, P_n, \tau_n, f_n, z_n)$, and output $(\tilde{\sigma}, 0, 0, (\text{instance–executed}, \tilde{\sigma}.\mathsf{cspace}(cid), p, add_L, add_R, m))$ for $p = (cid, P_n, \tau_n, f_n, z_n)$.

Close a virtual state channel. Recall that in case of ledger state channels, the environment instructs one party to close the channel. The parties of the ledger channel have some time to register all contract instances that were opened in the channel offline. If thereafter there is a contract instance in the channel which is not terminated (the amount of coins locked in the instance is greater than zero), then the channel is not closed.

The situation is different for virtual state channels. We require that the closing procedure of a virtual channel γ always starts in round $\gamma.\mathsf{validity}$ and always results in γ being closed. In other words, both contract instances with type $\mathtt{VSC}_i^{\mathcal{C}}$ that were opened in the subchannels of γ must be terminated (also in the case when the virtual channel was not created). Let us now explain how the protocol "Close a virtual channel" works.

In round $\gamma.\mathsf{validity}$ both end-users of the channel start registering the contract instance (if it has been created in the virtual channel γ but have never been registered before). This takes up to TimeRegister(i) rounds. Afterwards, $\gamma.\mathsf{Alice}$ requests execution of the contract instance $cid_A := \gamma.\mathsf{Alice}||\gamma.\mathsf{id}$ stored in the subchannel $id_A := \gamma.\mathsf{subchan}(\gamma.\mathsf{Alice})$, on the contact function $\mathtt{Close}_i^{\mathcal{C}}$. Party $\gamma.\mathsf{Bob}$ behaves analogously. In case one of the end-users of the channels does not request the closure of the underlying contract instance, $\gamma.\mathsf{Ingrid}$ can request it herself after certain time has passed.

The contract function $\mathtt{Close}_i^{\mathcal{C}}$ first checks if there is a registered but unterminated contract instance in the virtual channel γ. The first idea would be to let $\mathtt{Close}_i^{\mathcal{C}}$ ignore such contract instance. However, this would lead to the problem that the intermediary of the channel, $\gamma.\mathsf{Ingrid}$, looses money (because some money may still be locked in the contract) without ever having the chance to react to channel closing. Instead, the contract function $\mathtt{Close}_i^{\mathcal{C}}$ fairly distributes the locked coins to accounts of the users. For example, if user_L's cash balance in the contract instance with identifier cid is 3 and user_R's balance is -2, then 1 coin is added to user_L's account.

Next, the contract function verifies that the current value of the attribute cash is non-negative for both users and that the amount of coins that were originally invested into the virtual channel is equal to the current amount of coins in the channel. If this is the case, $\mathtt{Close}_i^{\mathcal{C}}$ unlocks for each user the current amount of coins it holds in the channel contract. If one of the users have negative balance in the channel or the amount of invested coins is not equal to the current amount of coins, then any trading that happened between the end-users is reverted by $\mathtt{Close}_i^{\mathcal{C}}$. This again guarantees that $\gamma.\mathsf{Ingrid}$ cannot loose money when $\gamma.\mathsf{Alice}$ and $\gamma.\mathsf{Bob}$ are malicious. The time complexity of closing a virtual channel of length i can be computed as $\mathrm{TimeRegister}(i) + 2 \cdot \mathrm{TimeExecute}(\lceil i/2 \rceil)$.

Before we provide the full specification of the protocol and the corresponding part of $\mathtt{VSC}_i(\mathcal{C})$, let us briefly explain one additional technicality. Recall that in case $\gamma.\mathsf{Ingrid}$ is corrupt, it can happen that the contract instances of type $\mathtt{VSC}_i(\mathcal{C})$ are opened in the subchannels of γ although the virtual channel γ was is not successfully created. This in particular means that the coins needed to create γ are locked in the subchannels and can be unlocked only after round $\gamma.\mathsf{validity}$ by executing the contact function $\mathtt{Close}_i^{\mathcal{C}}$.

Protocol $\Pi(i, \mathcal{C})$: Close a virtual channel

We use the notation established in Section 3.3 and denote $\mathcal{F}_{ch} := \mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i-1, \mathtt{VSC}_i(\mathcal{C}) \cup \mathcal{C})$. In addition, let $\mathrm{TV} := \gamma.\mathsf{validity}$, $\mathrm{TR}_{sub} := \mathrm{TimeRegister}(\lceil i/2 \rceil), \mathrm{TR}_i := \mathrm{TimeRegister}(i)$ and $\mathrm{TE}_{sub} := \mathrm{TimeExecute}(\lceil i/2 \rceil)$.

Party $T \in \gamma.\mathsf{end\text{–}users}$ in round TV

1. If $\Gamma^T(\gamma.\mathsf{id}) = \perp$ and you received $(\mathrm{updated}, id_T, cid_T) \xhookleftarrow{\le \tau_0 + 2 + \mathrm{TR}_{sub}} \mathcal{F}_{ch}$, then goto step 3.
2. If $\gamma^T := \Gamma^T(\gamma.\mathsf{id}) \neq \perp$, then for $cid \in \{0,1\}^*$ such that $\gamma^T.\mathsf{cspace}(cid) \neq \perp$ and (id, cid) is not marked as registered in Γ^T, call $\mathtt{Register}_i(T, id, cid)$. Thereafter, goto step 3.
3. Send $(\mathrm{execute}, id_T, cid_T, \mathtt{Close}_i^{\mathcal{C}}, \emptyset) \xhookrightarrow{\mathrm{TV} + \mathrm{TR}_i} \mathcal{F}_{ch}$.

Party I

For both $T \in \{A, B\}$ behave as follows:

4. If you did not receive $(\mathrm{executed}, id_T, cid_T, \sigma_T, L_T, R_T, m_T) \xhookleftarrow{\le \mathrm{TV} + \mathrm{TR}_i + \mathrm{TE}_{sub}} \mathcal{F}_{ch}$ where $m_T = (\mathrm{contract\text{–}closed}, final_A, final_B)$, then send $(\mathrm{execute}, id_T, cid_T, \mathtt{Close}_i^{\mathcal{C}}, \emptyset) \xhookrightarrow{\mathrm{TV} + \mathrm{TR}_i + \mathrm{TE}_{sub}} \mathcal{F}_{ch}$.

Party $T = A, B$

5. Upon $(\mathrm{executed}, id_T, cid_T, \sigma_T, L_T, R_T, m_T) \xhookleftarrow{\le \mathrm{TV} + \mathrm{TR}_i + 2\mathrm{TE}_{sub}} \mathcal{F}_{ch}$ where $m_T = (\mathrm{contract\text{–}closed}, final_A, final_B)$, delete γ^T from Γ^T and output $(\mathrm{closed}, id, final_A, final_B) \xhookrightarrow{\mathrm{TV} + \mathrm{TR}_i + 2\mathrm{TE}_{sub}} \mathcal{Z}$.

Contract $\mathtt{VSC}_i(\mathcal{C})$: function $\mathtt{Close}_i^{\mathcal{C}}(\sigma, P, \tau)$

Let $L := \sigma.\mathsf{user}_L, R := \sigma.\mathsf{user}_\mathsf{R}$, $\gamma := \sigma.\mathsf{virtual\text{–}channel}, A := \gamma.\mathsf{Alice}, B := \gamma.\mathsf{Bob}$.

1. If $P \notin \{L, R\}$, $\tau < \gamma.\mathsf{validity} + \mathrm{TimeRegister}(i)$ or $\gamma = \perp$, then output $(\sigma, 0, 0, \perp)$.
2. Let $\tilde{\sigma} := \sigma$. If there exists $cid \in \{0,1\}^*$ such that $\sigma.\mathsf{cspace}(cid) \neq \perp$ and we have that $\sigma_n.\mathsf{cash}(L) + \sigma_n.\mathsf{cash}(R) > 0$, where $\sigma_n := \sigma.\mathsf{cspace}(cid).\mathsf{storage}$ (i.e. the contract instance with identifier cid still has some locked coins), then distribute the coins fairly between the users as follows:

  - If $\sigma_n.\mathsf{cash}(L) > 0$ and $\sigma_n.\mathsf{cash}(R) > 0$, then set $\tilde{\sigma}.\mathsf{cash}(L) := \sigma.\mathsf{cash}(L) + \sigma_n.\mathsf{cash}(L)$ and $\tilde{\sigma}.\mathsf{cash}(R) := \sigma.\mathsf{cash}(R) + \sigma_n.\mathsf{cash}(R)$.
  - If $\sigma_n.\mathsf{cash}(L) > 0$ and $\sigma_n.\mathsf{cash}(R) \leq 0$, then set $\tilde{\sigma}.\mathsf{cash}(L) := \sigma.\mathsf{cash}(L) + (\sigma_n.\mathsf{cash}(L) + \sigma_n.\mathsf{cash}(R))$.
  - If $\sigma_n.\mathsf{cash}(L) \leq 0$ and $\sigma_n.\mathsf{cash}(R) > 0$, then set $\tilde{\sigma}.\mathsf{cash}(R) := \sigma.\mathsf{cash}(R) + (\sigma_n.\mathsf{cash}(L) + \sigma_n.\mathsf{cash}(R))$.

3. Let $invest_L := \gamma.\mathsf{cash}(A)$, $invest_R := \gamma.\mathsf{cash}(B)$ denote the balance when the contract was opened and let $final_L := \tilde{\sigma}.\mathsf{cash}(L)$ and $final_R := \tilde{\sigma}.\mathsf{cash}(R)$ denote the current balance. Distinguish the following two situations
   - If $(invest_L - final_L) + (invest_R - final_R) = 0$, then set $\tilde{\sigma}.\mathsf{cash}(L) := (invest_L - final_L)$ and $add_L := final_L$. Analogously for $\tilde{\sigma}.\mathsf{cash}(R)$ and add_R.
   - Otherwise set both $\tilde{\sigma}.\mathsf{cash}(L) := 0$ and $\tilde{\sigma}.\mathsf{cash}(R) := 0$ and $(add_L, add_R) := (invest_L, invest_R)$.
4. Set $\tilde{\sigma}.\mathsf{virtual\text{–}channel} := \perp$ and output $(\tilde{\sigma}, add_L, add_R, m)$, where $m = (\text{contract–closed}, add_L, add_R)$.

We can now state the final theorem showing that our constructions emulates the ideal functionality from Section 4. The proof is given in the Appx. E.

Theorem 2. *Let $\mathcal{E}_{res}$ be the class of restricted environments defined in Appx. C and let* $\mathtt{VSC}$ *be the contract type defined in Section 6. The protocol $\Pi(i, \mathcal{C})$ working in $\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i-1, \mathtt{VSC}_i(\mathcal{C}) \cup \mathcal{C})$-hybrid model emulates the ideal functionality $\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i, \mathcal{C})$ against environments from class $\mathcal{E}_{res}$ for every set of contract types $\mathcal{C}$, every $i > 1$ and every $\Delta \in \mathbb{N}$.*

7.1 Time complexity

Let us summaries the complexities of the protocol $\Pi(i, \mathcal{C})$ and define the timing functions $\mathrm{TimeCreate}(i, \Delta)$, $\mathrm{TimeUpdate}(i, \Delta)$, $\mathrm{TimeRegister}(i, \Delta)$, $\mathrm{TimeExecute}\ (i, \Delta)$ and $\mathrm{TimeClose}(i, \Delta)$. These functions, informally speaking, on input the channel length $i \in \mathbb{N}$ and the delay parameter $\Delta \in \mathbb{N}$, output the maximal number of rounds the corresponding part of the protocol $\Pi(i, \mathcal{C})$ takes.

Recall that for protocol parts that do not require interaction with the ledger (in case all parties behave honestly), we define *optimistic* time complexity in addition to the pessimistic time complexity. The optimistic time complexity of updating a contract instance in a state channel is equal to 2 rounds. Executing a contract instance in a state channel takes in the optimistic case at most 5 rounds. Let us emphasize that the optimistic time complexity of these protocol parts is *independent of the channel length.* This is also the case for virtual channel creation which takes at most 3 rounds for any $i > 1$.

The pessimistic time complexities of the protocol $\Pi(1, \mathcal{C})$, i.e. for ledger state channels, are the following. It takes at most 2Δ rounds to create a ledger state channel, i.e. $\mathrm{TimeCreate}(1, \Delta) = 2\Delta$. The pessimistic time complexity for registering a contract instance in a ledger channel is $\mathrm{TimeRegister}(1, \Delta) := 3\Delta$ rounds. The pessimistic time complexity for updating a contract instance is $\mathrm{TimeUpdate}(1, \Delta) := 2 + 3\Delta$ rounds. Execution of a contract instance in a ledger channel takes in the pessimistic case up to $\mathrm{TimeExecute}(1, \Delta) := 5 + 4\Delta$ rounds and closing a ledger channel takes $\mathrm{TimeClose}(1, \Delta) := 8\Delta$ rounds.

The pessimistic time complexities of the protocol $\Pi(i, \mathcal{C})$ for a virtual state channel of length i can be expressed in terms of the time complexities to execute its subchannels (which are channels of length $\lceil i/2 \rceil$), using recursively Equation (2). After solving the recurrence we obtain

$$\mathrm{TimeExecute}(i, \Delta) := \frac{14^{\lceil \log_2 i \rceil} \cdot (75 + 52\Delta)}{13} - 10.$$

Registering a contact instance in a virtual channel of length i takes at most $\mathrm{TimeRegister}(i, \Delta) := 5 \cdot \mathrm{TimeExecute}(\lceil i/2 \rceil, \Delta)$ rounds. Updating a contract instance in a virtual channel of length i is upper bounded by $\mathrm{TimeUpdate}(i, \Delta) := 2 + 5 \cdot \mathrm{TimeExecute}(\lceil i/2 \rceil, \Delta)$ and closing a virtual channel of length i takes in the pessimistic case $\mathrm{TimeClose}(i, \Delta) := 7 \cdot \mathrm{TimeExecute}(\lceil i/2 \rceil, \Delta)$.

Acknowledgment

We thank Jeff Coleman for several useful comments and in particular for pointing out a weakness of an earlier version of our protocol when taking fees into account.

References

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

[2] I. Bentov and R. Kumaresan. "How to Use Bitcoin to Design Fair Protocols". In: *Advances in Cryptology – CRYPTO 2014, Part II.* Ed. by J. A. Garay and R. Gennaro. Vol. 8617. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, 2014, pp. 421–439. DOI: 10.1007/978-3-662-44381-1_24.

[3] I. Bentov, R. Kumaresan, and A. Miller. "Instantaneous Decentralized Poker". In: *Advances in Cryptology – ASIACRYPT 2017.* Ed. by T. Takagi and T. Peyrin. Cham: Springer International Publishing, 2017, pp. 410–440. ISBN: 978-3-319-70697-9.

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

[5] *Bitcoin Wiki: Scalability.* https://en.bitcoin.it/wiki/Scalability.

[6] R. Canetti. "Universally Composable Security: A New Paradigm for Cryptographic Protocols". In: *42nd Annual Symposium on Foundations of Computer Science.* Las Vegas, Nevada, USA: IEEE Computer Society Press, 2001, pp. 136–145.

[7] R. Canetti, A. Cohen, and Y. Lindell. "A Simpler Variant of Universally Composable Security for Standard Multiparty Computation". In: *Advances in Cryptology – CRYPTO 2015, Part II.* Ed. by R. Gennaro and M. J. B. Robshaw. Vol. 9216. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, 2015, pp. 3–22. DOI: 10.1007/978-3-662-48000-7_1.

[8] C. Decker and R. Wattenhofer. "A Fast and Scalable Payment Network with Bitcoin Duplex Micropayment Channels". In: *Stabilization, Safety, and Security of Distributed Systems: 17th International Symposium, SSS 2015, Edmonton, AB, Canada, August 18-21, 2015, Proceedings.* Ed. by A. Pelc and A. A. Schwarzmann. Cham: Springer International Publishing, 2015, pp. 3–18. ISBN: 978-3-319-21741-3. DOI: 10.1007/978-3-319-21741-3_1. URL: http://dx.doi.org/10.1007/978-3-319-21741-3_1.

[9] S. Dziembowski et al. "Perun: Virtual Payment Hubs over Cryptographic Currencies". In: *IACR Cryptology ePrint Archive* 2017 (2017), p. 635. URL: http://eprint.iacr.org/2017/635.

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

[11] Y. T. Kalai, Y. Lindell, and M. Prabhakaran. "Concurrent Composition of Secure Protocols in the Timing Model". In: *Journal of Cryptology* 20.4 (Oct. 2007), pp. 431–492.

[12] A. Kate. "Introduction to Credit Networks: Security, Privacy, and Applications". In: *ACM CCS 16: 23rd Conference on Computer and Communications Security.* ACM Press, 2016, pp. 1859–1860.

[13] J. Katz et al. "Universally Composable Synchronous Computation". In: *TCC 2013: 10th Theory of Cryptography Conference.* Ed. by A. Sahai. Vol. 7785. Lecture Notes in Computer Science. Tokyo, Japan: Springer, Heidelberg, Germany, 2013, pp. 477–498. DOI: 10.1007/978-3-642-36594-2_27.

[14] R. Kumaresan and I. Bentov. "How to Use Bitcoin to Incentivize Correct Computations". In: *ACM CCS 14: 21st Conference on Computer and Communications Security.* Ed. by G.-J. Ahn, M. Yung, and N. Li. Scottsdale, AZ, USA: ACM Press, 2014, pp. 30–41.

[15] R. Kumaresan, T. Moran, and I. Bentov. "How to Use Bitcoin to Play Decentralized Poker". In: *ACM CCS 15: 22nd Conference on Computer and Communications Security.* Ed. by I. Ray, N. Li, and C. Kruegel: Denver, CO, USA: ACM Press, 2015, pp. 195–206.

[16] R. Kumaresan, V. Vaikuntanathan, and P. N. Vasudevan. "Improvements to Secure Computation with Penalties". In: *ACM CCS 16: 23rd Conference on Computer and Communications Security.* ACM Press, 2016, pp. 406–417.

[17] J. Lind et al. "Teechain: Scalable Blockchain Payments using Trusted Execution Environments". In: *CoRR* abs/1707.05454 (2017). arXiv: 1707.05454. URL: http://arxiv.org/abs/1707.05454.

[18] S. Micali and R. L. Rivest. "Micropayments Revisited". In: *Topics in Cryptology – CT-RSA 2002*. Ed. by B. Preneel. Vol. 2271. Lecture Notes in Computer Science. San Jose, CA, USA: Springer, Heidelberg, Germany, 2002, pp. 149–163.

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

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

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

[22] R. Pass and A. Shelat. "Micropayments for Decentralized Currencies". In: *ACM CCS 15: 22nd Conference on Computer and Communications Security.* Ed. by I. Ray, N. Li, and C. Kruegel: Denver, CO, USA: ACM Press, 2015, pp. 207–218.

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

[24] R. L. Rivest. "Electronic Lottery Tickets as Micropayments". In: *FC'97: 1st International Conference on Financial Cryptography.* Ed. by R. Hirschfeld. Vol. 1318. Lecture Notes in Computer Science. Anguilla, British West Indies: Springer, Heidelberg, Germany, 1997, pp. 307–314.

[25] D. Siegel. *Understanding The DAO Attack.* CoinDesk, http://www.coindesk.com/understanding-dao-hack-journalists/. 2016.

[26] *Update from the Raiden team on development progress, announcement of raidEX.* https://tinyurl.com/z2snp9e. Feb. 2017.

[27] D. Wheeler. "Transactions Using Bets". In: *Proceedings of the International Workshop on Security Protocols.* London, UK, UK: Springer-Verlag, 1997, pp. 89–92. ISBN: 3-540-62494-5. URL: http://dl.acm.org/citation.cfm?id=647214.720381.

[28] *Wikipedia: Microtransaction.* https://en.wikipedia.org/wiki/Microtransaction.

[29] G. Wood. *ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER.* http://gavwood.com/paper.pdf.

A Further related work

The most widely discussed recent proposals for the channel networks are *Lightning* and *Raiden*. Both of them are routing payments using the interactive mechanism based on the hash-locked transactions (we explain the main ideas of this mechanism in Appx. B). Lightning is purely a payment network. Raiden, that uses the Ethereum cryptocurrency, can potentially provide state channels. However, this system is still in development and, according to the latest reports, the first experimental transactions that were performed concerned the simple payment channels (not the *state* channels). According to other recent reports [26], the developers of this product have focused on constructing a fully functional payment channel under the name *Raiden Minimum Viable Product*, and renamed the full implementation of state channels to "Raiden 2.0" that will be implemented later. State channels have also been recently defined and constructed in [19]. Compared to our work, they do not seem to have an easy interface for handling multiple states in parallel, and for having the two-stage channel update (since this is not needed in [19] for their application). Payment channels were also constructed in [8]. Other micropayment systems that have been proposed are based on probabilistic payments (see [27, 24, 18, 22]). Our technique for channel virtualization could potentially be used to create such probabilistic payment channels.

Channel constructions based on the sequence number maturity (that we use in this paper) have been mentioned already in [23], and recently described in more detail (as "stateful duplex off-chain micropayment channels") in [3]. Payment channels bare some resemblance with the *credit networks* (see, e.g., [12]). It would be interesting to see if our techniques also apply in this area.

A recent idea for micropayments over Bitcoin is based on the "trusted environments" [17]. It would be interesting to see if our protocols can be implemented along the same lines.

B Routing payments using hash-locked transactions

Consider the situation when Alice has a payment channel with Ingrid and Ingrid has a payment channel with Bob. Assume that Alice wants to send one coin to Bob and route the payment via Ingrid. The first idea would be to let Alice update the channel with Ingrid such that Alice pays one coin to Ingrid and then let Ingrid symmetrically update the channel with Bob such that Ingrid pays one coin to Bob. However, this naive solution allows a malicious Ingrid to abort after receiving the coin from Alice and never pay anything to Bob.

Let us briefly explain how to solve the above problem using *hash-locked transactions*. Let H be some fixed hash function. Bob first picks a random value $x \in \{0,1\}^*$ and sends the hash value $h = H(x)$ to Alice who creates a hash-locked transaction HLT_A. Informally, this transaction promises to update the channel between Alice and Ingrid such that Ingrid earns one coin if she publishes a preimage of h before a timeout t_A. Ingrid, upon receiving the hash-locked transaction HLT_A from Alice, creates a hash-locked transaction HLT_B which promises to update the channel between Ingrid and Bob such that Bob earns one coin if he publishes a preimage of h before the timeout $t_B < t_A$. Hence, if Bob reveals x before time t_B, he gets one coin from Ingrid. Since $t_A > t_B$, Ingrid has time to use the value x to get one coin from Alice and thus finalize the payment. In case Bob does not reveal x to Ingrid before the timeout t_B, Ingrid can refund her coin locked in HLT_B. Analogously, in case Ingrid does not reveal x, Alice can refund her coin locked in HLT_A after round t_A.

C Restrictions on the Environment

In order to simplify the description of the ideal functionality $\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i,\mathcal{C})$ and the protocol $\Pi(i,\mathcal{C})$ realizing it, we define a set of restricted environments $\mathcal{E}_{res}$. Every $\mathcal{Z} \in \mathcal{E}_{res}$ has to satisfy the following

- $\mathcal{Z}$ never sends the same message to the same party twice.
- $\mathcal{Z}$ sends a message (create, γ), where γ is a ledger channel, to all honest parties in the set $\gamma.\mathsf{end\text{–}users}$ in the same round τ_0 (and it never send this message to any other honest party). In addition, we assume the following: there does not exist a channel γ' with $\gamma.\mathsf{id} = \gamma'.\mathsf{id}$ (and no such channel is currently being created); parties of the channels are from the set $\mathcal{P}$; $\mathtt{Value}(\gamma.\mathsf{id}) \geq 0$; both parties of the channel have enough funds on the ledger for the channel creation;[8] the set of contract instances is empty; and $\gamma.\mathsf{length} = 1$.
- $\mathcal{Z}$ sends the message (create, γ), where γ is a virtual channel, to all honest parties in the set $\gamma.\mathsf{end\text{–}users} \cup \{\gamma.\mathsf{Ingrid}\}$ in the same round τ_0 (and it never send this message to any other honest party). In addition, we assume the following: there does not exists a channel γ' with $\gamma.\mathsf{id} = \gamma'.\mathsf{id}$ (and no such channel is currently being created); parties of the channels are from the set $\mathcal{P}$; $\mathtt{Value}(\gamma.\mathsf{id}) \geq 0$; the set of contract instances is empty; $\gamma.\mathsf{validity} < \tau_0 + 3$. Additionally, we assume the following about the subchannels of γ:
  - if honest $P \in \gamma.\mathsf{end\text{–}users}$ receives the message (create, γ), then the following must be satisfied: the subchannel $\alpha := \gamma.\mathsf{subchan}(P)$ must exist; $\alpha.\mathsf{end\text{–}users} = \{P, \gamma.\mathsf{Ingrid}\}$; $\alpha.\mathsf{length} \leq \lceil \gamma.\mathsf{length}/2 \rceil$; $\gamma.\mathsf{validity} > \alpha.\mathsf{validity} + \mathrm{TimeClose}(\gamma.\mathsf{length})$; $\gamma.\mathsf{cspace}(cid) = \perp$ for every $cid \in \{0,1\}^*$ and both P and $\gamma.\mathsf{Ingrid}$ have enough funds in α.
  - if honest $\gamma.\mathsf{Ingrid}$ receives the message (create, γ), then both subchannels $\alpha := \gamma.\mathsf{subchan}(\gamma.\mathsf{Alice})$, $\beta := \gamma.\mathsf{subchan}(\gamma.\mathsf{Bob})$ exist; $\alpha.\mathsf{end\text{–}users} = \{\gamma.\mathsf{Alice}, \gamma.\mathsf{Ingrid}\}$ and $\beta.\mathsf{end\text{–}users} = \{\gamma.\mathsf{Bob}, \gamma.\mathsf{Ingrid}\}$; $\alpha.\mathsf{length} \leq \lceil \gamma.\mathsf{length}/2 \rceil$, $\beta.\mathsf{length} \leq \lceil \gamma.\mathsf{length}/2 \rceil$ and $\gamma.\mathsf{length} = \alpha.\mathsf{length} + \beta.\mathsf{length}$; $\gamma.\mathsf{validity} > \max\{\alpha.\mathsf{validity}, \beta.\mathsf{validity}\} + \mathrm{TimeClose}(\gamma.\mathsf{length})$; $\gamma.\mathsf{cspace}(cid) = \perp$ for every $cid \in \{0,1\}^*$; $\gamma.\mathsf{Alice}$ and $\gamma.\mathsf{Ingrid}$ have enough funds in α and $\gamma.\mathsf{Bob}$ and $\gamma.\mathsf{Ingrid}$ have enough funds in β.
- If $\mathcal{Z}$ sends the message $(\text{update}, id, cid, \tilde{\sigma}, \mathtt{C})$ or $(\text{update–reply}, ok, id, cid)$ to an honest party P, then a channel γ with identifier id exists in Γ; $P \in \gamma.\mathsf{end\text{–}users}$, the channel supports the contract type; if the contract instance has already been updated before, then the contract type remains the same, i.e. if

[8] In case the environment requests opening more channels at the same time, we require that all parties have enough funds for all channels that are being created.

$\nu := \gamma.\mathsf{cspace}(cid) \neq \bot$, then $\nu.\mathsf{type} = \mathsf{C}$; the new contract instance $\tilde{\sigma}$ is admissible with respect to C, i.e. $\tilde{\sigma} \in \mathsf{C}.\Lambda$; and both parties have enough cash in the channel for the contract instance update.[9] $\mathcal{Z}$ never asks to update a contract instance that is currently being updated or executed. In addition, if $\Gamma(id)$ is a virtual channel, then we assume that there is no other contract instance in the channel (and no other instance is being created).

– If $\mathcal{Z}$ sends the message $(\text{execute}, id, cid, f, z)$ to an honest party P, then a channel γ with identifier id exists in Γ, $P \in \gamma.\mathsf{end\text{–}users}$, the contract instance cid has already been defined in γ, i.e. $\gamma.\mathsf{cspace}(cid) \neq \bot$, and f is a contract function with respect to $\gamma.\mathsf{cspace}(cid).\mathsf{type}$.

– If $\mathcal{Z}$ send the message (close, id) to honest party P, then channel γ with identifier id exists in Γ, γ is a ledger channel and $P \in \gamma.\mathsf{end\text{–}users}$.

D Security analysis of the ledger channel protocol

In this section, we will prove Theorem 1, i.e. show that for any set of contract types $\mathcal{C}$, the protocol $\Pi(1, \mathcal{C})$ $\mathcal{E}_{res}$-emulates the ideal functionality $\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(1, \mathcal{C})$ in $\mathcal{F}_{sc}^{\mathcal{L}(\Delta)}(\mathcal{C})$-hybrid world. In order to do so, we need to construct a simulator $\mathcal{S}_1$ that operates in the $\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(1, \mathcal{C})$ world and simulates the $\mathcal{F}_{sc}^{\mathcal{L}(\Delta)}(\mathcal{C})$-hybrid world for any PPT adversary $\mathcal{A}$ and any environment $\mathcal{Z} \in \mathcal{E}_{res}$.

The two main challenges of our analysis are the following: (i) ensure the consistency of timings (if an honest party P outputs a message m in round τ in the hybrid world, then P must output the same message m in the same round τ in the ideal world as well) and (ii) ensure the consistency of balances of parties on the ledger (i.e. if the state of accounts on the ledger in round τ is equal $(x_1, \dots, x_n)$ in the hybrid world, then the state of user's accounts in round τ must be $(x_1, \dots, x_n)$ in the ideal world as well).

Since the simulator $\mathcal{S}_1$ internally runs the hybrid ideal functionality $\mathcal{F}_{sc}^{\mathcal{L}(\Delta)}(\mathcal{C})$ and gets the instructions for the adversary from the environment, it can instruct (via the influence port) the ideal functionality $\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(1, \mathcal{C})$ to make changes on the ledger in the same round as the adversary $\mathcal{A}$ would do in the hybrid world. To simplify the description of the simulator, we do not write these instructions explicitly. Also recall that there are no private inputs or messages being sent, thus we assume that the ideal functionality $\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(1, \mathcal{C})$ on receiving a message m from party P immediately leaks (P, m) to the simulator $\mathcal{S}_1$ via the leakage port. Analogously, if the ideal functionality sends a message m to an honest party P. The simulator $\mathcal{S}_1$ constructed in this section will maintain a channel space Γ^T for every honest party T and the channel space Γ for the hybrid ideal functionality $\mathcal{F}_{sc}^{\mathcal{L}(\Delta)}(\mathcal{C})$. In addition, the simulator will generate a key pair $(pk_T, sk_T) \leftarrow_{\$} \mathsf{KGen}(1^\lambda)$ for every honest party T during the setup phase.

Create a ledger channel. Let us begin with the simulation of the "Create a ledger state channel" protocol. In the case when both parties of the ledger channel are honest, the simulator only has to instruct the ideal functionality to remove coins from ledger accounts in the correct round. This can be done by any simulator that internally runs the adversary $\mathcal{A}$ as discussed above. In addition, the simulator updates the channel space sets, i.e. defines $\Gamma^A(\gamma.\mathsf{id}) = \Gamma^B(\gamma.\mathsf{id}) = \Gamma(\gamma.\mathsf{id}) = \gamma$.

The simulator for the remaining cases, when one or both parties are corrupt, is more interesting and its description can be found in below.

Simulator $\mathcal{S}_1$: Create a ledger channel

We use the abbreviated notation from Section 3.3. Let $\mathcal{F}_{ch} := \mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(1, \mathcal{C})$.

Case A is honest and B is corrupt:

[9] In case the environment requests constructing more contract instances at the same time, we require that both parties have enough funds in the channel for all of them.

Upon $(A, \text{create}, \gamma) \xhookleftarrow{\tau_0} \mathcal{F}_{ch}$, proceed as follows:

1. Wait till round $\tau_1 \leq \tau_0 + \Delta$ to send $(\text{initializing}, \gamma) \xhookrightarrow{\tau_1} B$.
2. If $(\text{confirm}, \gamma) \xhookleftarrow{\tau_1} B$, then send $(\text{create}, \gamma) \xhookrightarrow{\tau_1} \mathcal{F}_{ch}$ on behalf of B. Else stop.
3. Send $(\text{initialized}, \gamma) \xhookrightarrow{\tau_2 \leq \tau_1 + \Delta} B$ and set $\Gamma^A(\gamma.\mathsf{id}) := \gamma, \Gamma(\gamma.\mathsf{id}) := \gamma$ and stop.

Case A is corrupt and B is honest:

Upon $(\text{construct}, \gamma) \xhookleftarrow{\tau_0} A$ proceed as follows:

1. If A does not have enough funds on the ledger, there already exists a channel γ' such that $\gamma.\mathsf{id} = \gamma'.\mathsf{id}$ in Γ, $\gamma.\mathsf{cspace} \neq \emptyset$, or $\texttt{Value}(\gamma) < 0$, then stop.
2. Else send $(\text{create}, \gamma) \xhookrightarrow{\tau_0} \mathcal{F}_{ch}$ on behalf of A and in round $\tau_1 \leq \tau_0 + \Delta$ send $(\text{initializing}, \gamma) \xhookrightarrow{\tau_1} A$.
3. Distinguish the following two situations:
   - If $(B, \text{create}, \gamma) \xhookleftarrow{\tau_0} \mathcal{F}_{ch}$, then send $(\text{initialized}, \gamma) \xhookrightarrow{\tau_0 + 2\Delta} A$ and set $\Gamma^B(\gamma.\mathsf{id}) := \gamma$, $\Gamma(\gamma.\mathsf{id}) := \gamma$ and stop.
   - Else wait. If $(\text{refund}, \gamma) \xhookleftarrow{\tau_3 > \tau_0 + 2\Delta} A$, then send $(\text{refund}, \gamma) \xhookrightarrow{\tau_3} \mathcal{F}_{ch}$.

Case A and B are corrupt:

Upon $(\text{construct}, \gamma) \xhookleftarrow{\tau_0} A$ proceed as follows:

1. If A does not have enough funds on the ledger, there already exists a channel γ' such that $\gamma.\mathsf{id} = \gamma'.\mathsf{id}$, $\gamma.\mathsf{cspace} \neq \emptyset$ or $\texttt{Value}(\gamma) < 0$, then stop.
2. Else send $(\text{create}, \gamma) \xhookrightarrow{\tau_0} \mathcal{F}_{ch}$ on behalf of A and in round $\tau_1 \leq \tau_0 + \Delta$ send $(\text{initializing}, \gamma) \xhookrightarrow{\tau_1} \gamma.\mathsf{end\text{–}users}$.
3. Distinguish the following two situations:
   - If $(\text{confirm}, \gamma) \xhookleftarrow{\tau_1} B$ and B has sufficient funds on the ledger, then $(\text{create}, \gamma) \xhookrightarrow{\tau_1} \mathcal{F}_{ch}$ and on behalf of B and wait till round $\tau_2 \leq \tau_0 + 2\Delta$ to send $(\text{initialized}, \gamma) \xhookrightarrow{\tau_2} \gamma.\mathsf{end\text{–}users}$. Then set $\Gamma(\gamma.\mathsf{id}) := \gamma$ and stop.
   - Else wait if $(\text{refund}, \gamma) \xhookleftarrow{\tau_3 > \tau_0 + 2\Delta} A$. In such a case send $(\text{refund}, \gamma) \xhookrightarrow{\tau_3} \mathcal{F}_{ch}$ and stop.

Registration of a contract instance in a ledger channel. Since registration of a contract instance is defined as a separate procedure that can be called by parties of the protocol $\Pi(1, \mathcal{C})$, we define a "subsimulator" $\texttt{SimRegister}(P, id, cid)$ which can be called as a procedure by the simulator $\mathcal{S}_1$. Before we define the subsimulator formally, let us discuss one technicality.

As already mentioned, one of the main challenges of the simulation is to ensure the consistency of the ledger accounts in the ideal and hybrid world. In particular, if two parties created a ledger channel between them (i.e. their coins were subtracted from their ledger accounts), the simulator has to ensure that once this channel is closed, the amount of coins returned to each party's account on the ledger is the same in the real and hybrid world. In case at least one party of the channel is honest, every time the channel is updated or executed, the ideal functionality $\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i, \mathcal{C})$ receives the corresponding message from the honest party and thus has the same view on the channel's state as the honest party in the hybrid world. The situation is more tricky in case both parties are corrupt.

If two corrupt parties have a ledger channel between them, they can update its state arbitrarily (even to an invalid state). As long as these updates are done off-chain (parties exchange messages with each other and do not send any message to the hybrid ideal functionality $\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i-1, \texttt{VSC}_i(\mathcal{C}) \cup \mathcal{C})$), no changes in the channel space Γ of ideal functionality $\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i, \mathcal{C})$ are needed. Only when parties successfully register a contract instance with the hybrid ideal functionality $\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i-1, \texttt{VSC}_i(\mathcal{C}) \cup \mathcal{C})$, the update of the channel resulting from the new contract instance becomes "official". Thus, the simulator has to ensure that these changes to the ledger channel are also made in the ideal functionality $\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i, \mathcal{C})$. This is the reason, why the simulator has to send update message to the ideal functionality on behalf of the corrupt parties, in case they successfully register a contract instance in the hybrid world.

Sub-simulator : $\texttt{SimRegister}(P, id, cid)$

We use the abbreviated notation from Section 3.3. Let $\mathcal{F}_{ch} := \mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(1, \mathcal{C})$.

Case P and Q are honest:

1. Let $\gamma^P := \Gamma^P(id)$, $\nu^P := \gamma^P.\mathsf{cspace}(cid)$ and $\gamma^Q := \Gamma^Q(id)$, $\nu^Q := \gamma^Q.\mathsf{cspace}(cid)$.
2. Wait upto 2Δ rounds and then proceed as follows. If $\nu^P.\mathsf{version} \geq \nu^Q.\mathsf{version}$, then set $\tilde{\nu} := (\nu^P.\mathsf{storage}, \nu^P.\mathsf{type})$. Else set $\tilde{\nu} := (\nu^Q.\mathsf{storage}, \nu^Q.\mathsf{type})$.
3. Mark (id, cid) as registered in $\Gamma, \Gamma^P, \Gamma^Q$ and update all three sets, i.e. set $\Gamma := \texttt{LocalUpdate}(\Gamma, id, cid, \tilde{\nu})$, $\Gamma^P := \texttt{LocalUpdate}(\Gamma^P, id, cid, \tilde{\nu})$, $\Gamma^Q := \texttt{LocalUpdate}(\Gamma^Q, id, cid, \tilde{\nu})$.

Case P is honest and Q is corrupt:

1. Let $\gamma^P := \Gamma^P(id)$, $\nu^P := \gamma^P.\mathsf{cspace}(cid)$, $\sigma^P := \nu^P.\mathsf{storage}$.
2. Set τ_0 be the current round. Send $(\text{instance–registering}, id, cid, \nu^P) \xhookrightarrow{\tau_1 \leq \tau_0 + \Delta} Q$.
3. If $(\text{instance–register}, id, cid, \nu^Q) \xhookleftarrow{\tau_1} Q$ where ν^Q is a valid contract instance (both signatures $\nu^Q.\mathsf{sign}(A)$ and $\nu^Q.\mathsf{sign}(B)$ are valid, the amount of locked coins in ν^Q is non-negative, users of the contract instance are A and B, the contract instance storage is admissible and the contract type is from the set $\mathcal{C}$), then proceed as follows. If $\nu^P.\mathsf{version} \geq \nu^Q.\mathsf{version}$, then $\tilde{\nu} := (\nu^P.\mathsf{storage}, \nu^P.\mathsf{type})$ and otherwise set $\tilde{\nu} := (\nu^Q.\mathsf{storage}, \nu^Q.\mathsf{type})$. Thereafter $(\text{instance–registered}, id, cid, \tilde{\nu}) \xhookrightarrow{\tau_2 \leq \tau_1 + \Delta} Q$ and goto step 5.
4. Else define $\tilde{\nu} := (\nu^P.\mathsf{storage}, \nu^P.\mathsf{type})$, send $(\text{instance–registered}, id, cid, \tilde{\nu}) \xhookrightarrow{\tau_2 \leq \tau_0 + 3\Delta} Q$ and goto step 5.
5. Mark (id, cid) as registered in Γ^P, Γ and set $\Gamma := \texttt{LocalUpdate}(\Gamma, id, cid, \tilde{\nu})$ and $\Gamma^P := \texttt{LocalUpdate}(\Gamma^P, id, cid, \tilde{\nu})$.

Case P is corrupt and Q is honest:

Upon $(\text{instance–register}, id, cid, \nu^P) \xhookleftarrow{\tau_0} P$, s.t. $\Gamma(id) \neq \perp$, $\Gamma(id).\mathsf{cspace}(cid) = \perp$, ν^P is a valid contract instance (both signatures $\nu^P.\mathsf{sign}(A)$ and $\nu^P.\mathsf{sign}(B)$ are valid, the amount of locked coins in ν^P is non-negative, users of the contract instance are A and B, the contract instance storage is admissible and the contract type is from the set $\mathcal{C}$), then do:

1. Within Δ rounds, send $(\text{instance–registering}, id, cid, \nu^P) \xhookrightarrow{\tau_1 \leq \tau_0 + \Delta} P$.
2. Let $\gamma^Q := \Gamma^Q(id)$, $\nu^Q := \gamma^Q.\mathsf{cspace}(cid)$. If $\nu^P.\mathsf{version} \geq \nu^Q.\mathsf{version}$, then $\tilde{\nu} := (\nu^P.\mathsf{storage}, \nu^P.\mathsf{type})$ and otherwise set $\tilde{\nu} := (\nu^Q.\mathsf{storage}, \nu^Q.\mathsf{type})$.
3. Send $(\text{instance–registered}, id, cid, \tilde{\nu}) \xhookrightarrow{\tau_2 \leq \tau_1 + \Delta} P$, mark (id, cid) as registered in Γ^Q and Γ and then set $\Gamma := \texttt{LocalUpdate}(\Gamma, id, cid, \tilde{\nu})$ and $\Gamma^Q := \texttt{LocalUpdate}(\Gamma^Q, id, cid, \tilde{\nu})$.

Case P and Q are corrupt :

Upon $(\text{instance–register}, id, cid, \nu^P) \xhookleftarrow{\tau_0} P$, s.t. $\Gamma(id) \neq \perp$, $\Gamma(id).\mathsf{cspace}(cid) = \perp$, ν^P is a valid contract instance (both signatures $\nu^P.\mathsf{sign}(A), \nu^P.\mathsf{sign}(B)$ are valid, the amount of locked money in ν^P is non-negative, users of the contract instance are A and B, the contract instance storage is admissible and the contract type is from the set $\mathcal{C}$), then do:

1. Within Δ rounds, send $(\text{instance–registering}, id, cid, \nu^P) \xhookrightarrow{\tau_1 \leq \tau_0 + \Delta} \Gamma(id).\mathsf{end–users}$.

2. If (instance–register, id, cid, ν^Q) $\xhookleftarrow{\tau_1}$ Q s.t. ν^Q is a valid contract instance (both $\nu^Q.\mathsf{sign}(A)$ and $\nu^Q.\mathsf{sign}(B)$ are valid signatures, the amount of locked money in ν^Q is non-negative, users of the contract instance are A and B, the contract instance storage is admissible and the contract type is from the set $\mathcal{C}$), then proceed as follows. If $\nu^P.\mathsf{version} \geq \nu^Q.\mathsf{version}$, then $\tilde{\nu} := (\nu^P.\mathsf{storage}, \nu^P.\mathsf{type})$ and otherwise set $\tilde{\nu} := (\nu^Q.\mathsf{storage}, \nu^Q.\mathsf{type})$. Thereafter send (instance–registered, $id, cid, \tilde{\nu}$) $\xhookrightarrow{\tau_2 \leq \tau_1 + \Delta}$ $\Gamma(id).\mathsf{end\text{–}users}$ and goto step 4.
3. Else proceed as follows. If (finalize–register, id, cid) $\xhookleftarrow{\tau_0 + 2\Delta}$ P, then define $\tilde{\nu} := (\nu^P.\mathsf{storage}, \nu^P.\mathsf{type})$, send (instance–registered, $id, cid, \tilde{\nu}$) $\xhookrightarrow{\tau_2 \leq \tau_0 + 3\Delta}$ $\Gamma(id).\mathsf{end\text{–}users}$ and goto step 4.
4. Mark (id, cid) as registered Γ and update the channel space $\Gamma := \mathtt{LocalUpdate}(\Gamma, id, cid, \tilde{\nu})$. Then send (update, $id, cid, \tilde{\nu}.\mathsf{storage}, \tilde{\nu}.\mathsf{type}$) $\hookrightarrow \mathcal{F}_{ch}$ on behalf of P and (update–reply, ok, id, cid) $\hookrightarrow \mathcal{F}_{ch}$ on behalf of Q.

Update a contract instance in a ledger channel If both parties are honest, the simulator does not need to give any instructions to the ideal functionality and only updates the sets Γ^P, Γ^Q, when the messages (P, update, $id, cid, \tilde{\sigma}, \mathtt{C}$) and ($Q$, update–reply, ok, id, cid) are leaked by the ideal functionality.

In case both parties are corrupt, the simulator can internally simulate the communication of the two corrupt parties and in case the registration procedure is started by one of them, it executes the subsimulator `SimRegister` for the case when both parties are corrupt. Note that if the registration procedure is successful (a contract instance gets registered), the subsimulator `SimRegister` instructs the ideal functionality to update the contract instance accordingly.

Below we define the simulator $\mathcal{S}_1$ for the remaining two case; i.e. when only the initiating party is corrupt and if only the reacting party is corrupt.

Simulator $\mathcal{S}_1$: Contract instance update

We use the abbreviated notation from Section 3.3. Let $\mathcal{F}_{ch} := \mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(1, \mathcal{C})$.

Case P is honest and Q is corrupt:

Upon (P, update, $id, cid, \tilde{\sigma}, \mathtt{C}$) $\xhookleftarrow{\tau_0}$ $\mathcal{F}_{ch}$ do:

1. Let $\gamma^P := \Gamma^P(id)$, $\nu^P := \gamma^P.\mathsf{cspace}(cid)$, $\sigma^P := \nu^P.\mathsf{storage}$. If $\nu^P = \perp$, then set $v^P := 0$, else set $v^P := \nu^P.\mathsf{version}$.
2. Sign $s_P := \mathtt{Sign}_{sk_P}(id, cid, \tilde{\sigma}, \mathtt{C}, v^P + 1)$ and send (update, $s_P, id, cid, \tilde{\sigma}, \mathtt{C}$) $\xhookrightarrow{\tau_0 + 1}$ Q of behalf of P.
3. Distinguish the following cases:
   - If (update–ok, s_Q) $\xhookleftarrow{\tau_1 \leq \tau_0 + 1}$ Q such that $\mathtt{Vfy}_{pk_Q}(id, cid, \tilde{\sigma}, \mathtt{C}, v^P + 1; s_B) = 1$, then send (update–reply, ok, id, cid) $\xhookrightarrow{\tau_1}$ $\mathcal{F}_{ch}$ on behalf of Q and set $\Gamma^P := \mathtt{LocalUpdate}(\Gamma^P, id, cid, \tilde{\sigma}, \mathtt{C}, v^P + 1, \{s_P, s_Q\})$.
   - If (update–not–ok, s_Q) $\xhookleftarrow{\tau_1 \leq \tau_0 + 1}$ Q such that $\mathtt{Vfy}_{pk_B}(id, cid, \sigma^P, \mathtt{C}, v^P + 2; s_Q) = 1$, then compute $s_P := \mathtt{Sign}_{sk_P}(id, cid, \sigma^P, \mathtt{C}, v^P + 2)$ and set $\Gamma^P := \mathtt{LocalUpdate}(\Gamma^P, id, cid, \sigma^P, \mathtt{C}, v^P + 2, \{s_P, s_Q\})$.
   - Else execute $\mathtt{SimRegister}(P, id, cid)$. If after the sub-simulator is executed (in round $\tau_2 \leq \tau_0 + 3\Delta + 1$) it holds that $\Gamma^P(id).\mathsf{cspace}(cid) = (\tilde{\sigma}, \mathtt{C})$, then (update–reply, ok, id, cid) $\xhookrightarrow{\tau_2}$ $\mathcal{F}_{ch}$ on behalf of Q.

Case P is corrupt and Q is honest:

Upon (update, $s_P, id, cid, \tilde{\sigma}, \mathtt{C}$) $\xhookleftarrow{\tau_0}$ P do:

1. Let $\gamma^Q := \Gamma^Q(id)$. If $\gamma^Q = \perp$ or there exists $cid' \neq cid$ such that $\gamma.\mathsf{cspace}(cid) \neq \perp$, then stop; else let $\nu^Q := \gamma^Q.\mathsf{cspace}(cid)$. If $\nu^Q = \perp$, then set $v^Q := 0$, else set $v^Q := \nu^Q.\mathsf{version}$.
2. If $\mathtt{Vfy}_{pk_P}(id, cid, \tilde{\sigma}, \mathtt{C}, v^Q + 1; s_P) \neq 1$, then mark (id, cid) as corrupt in Γ^Q and stop. Else send $(\text{update}, id, cid, \tilde{\sigma}, \mathtt{C}) \overset{\tau_0}{\hookrightarrow} \mathcal{F}_{ch}$ on behalf of P.
3. Distinguish the following cases:
   - If $(Q, \text{update–reply}, ok, id, cid) \xhookleftarrow{\tau_1 \leq \tau_0 + 1} \mathcal{F}_{ch}$, then compute $s_Q := \mathtt{Sign}_{sk_Q}(id, cid, \tilde{\sigma}, \mathtt{C}, v^Q + 1)$, set $\Gamma^Q := \mathtt{LocalUpdate}(\Gamma^Q, id, cid, \tilde{\sigma}, \mathtt{C}, v^Q + 1, \{s_P, s_Q\})$ and send $(\text{update–ok}, s_Q) \xhookrightarrow{\tau_0 + 2} P$ on behalf of Q and stop.
   - Else compute $s_Q := \mathtt{Sign}_{sk_Q}(id, cid, \nu^Q.\mathsf{storage}, \nu^Q.\mathsf{type}, v^Q + 2)$ and on behalf of Q send $(\text{update–not–ok}, s_Q) \xhookrightarrow{\tau_0 + 2} P$.

Execute a contract instance in a ledger cannels In case both parties are honest, the simulator only has to instruct the ideal functionality to output the result in the correct round. Let τ_0 be the round in which the environment instructed the initiating party P to execute. Then the simulator sets $\tau_1 := \tau_0 + x$, where x is the smallest offset such that $\tau_1 = 1 \mod 4$ if $P = \gamma.\mathsf{Alice}$ and $\tau_1 = 3 \mod 4$ if $P = \gamma.\mathsf{Bob}$ and waits till round τ_1 to instruct the ideal functionality to output the result. The it updates both channel space Γ^P and Γ^Q accordingly.

Below we describe in detail the situation when one or two parties are corrupt.

Simulator $\mathcal{S}_1$: Contract instance execution.

We use the abbreviated notation from Section 3.3. Let $\mathcal{F}_{ch} := \mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(1, \mathcal{C})$.

Case P is honest and Q is corrupt:

Upon $(P, \text{execute}, id, cid, f, z) \overset{\tau_0}{\hookleftarrow} \mathcal{F}_{ch}$, let $\gamma^P := \Gamma^P(id)$, $\nu^P := \gamma^P.\mathsf{cspace}(cid)$, $\sigma^P := \nu^P.\mathsf{storage}$ and $v^P := \nu^P.\mathsf{version}$. In addition, set $\tau_1 := \tau_0 + x$, where x is the smallest offset such that $\tau_1 = 1 \mod 4$ if $P = \gamma^P.\mathsf{Alice}$ and $\tau_1 = 3 \mod 4$ if $P = \gamma^P.\mathsf{Bob}$. Wait till round τ_1 and then proceed as follows:

1. If (id, cid) is not marked as corrupt in Γ^P, do:
   (a) Set $(\tilde{\sigma}, add_L, add_R, m) := f(\sigma^P, P, \tau_0, z)$. If $m = \perp$, then stop.
   (b) Else compute $s_P := \mathtt{Sign}_{sk_P}(id, cid, \tilde{\sigma}, \nu^P.\mathsf{type}, v^P + 1)$ and send $(\text{peaceful–request}, id, cid, f, z, s_P, \tau_0) \xhookrightarrow{\tau_1 + 1} Q$.
   (c) If $(\text{peaceful–confirm}, id, cid, f, z, s_Q) \xhookleftarrow{\tau_1 + 1} Q$ such that $\mathtt{Vfy}_{pk_Q}(id, cid, \tilde{\sigma}, \nu^P.\mathsf{type}, v^P + 1; s_Q) = 1$, then set $\Gamma^P := \mathtt{LocalUpdateAdd}(\Gamma^P, id, cid, \tilde{\sigma}, \nu^P.\mathsf{type}, add_L, add_R, v^P + 1, \{s_P, s_Q\})$ and instruct the ideal functionality to output the result. Else execute $\mathtt{SimRegister}(P, id, cid)$ in round $\tau_1 + 2$. If after the execution of the sub-simulator (in round $\tau_1 \leq \tau_0 + 3\Delta + 5$) it holds that $\sigma^P = \tilde{\sigma}$, then set $\Gamma^P := \mathtt{LocalUpdateAdd}(\Gamma^P, id, cid, \tilde{\sigma}, \nu^P.\mathsf{type}, add_L, add_R)$, instruct the ideal functionality to output the result and stop. Else goto step 2e.
2. If (id, cid) is marked as corrupt
   (d) If (id, cid) is not marked as registered in Γ^P, then execute the sub-simulator $\mathtt{SimRegister}(P, id, cid)$.
   (e) Let τ_3 be the current round. If $(\text{executed}, id, cid, \sigma, add_L, add_R, m) \xhookleftarrow{\tau_4 \leq \tau_3 + \Delta} \mathcal{F}_{ch}$, then update the channel space Γ^P and Γ and send $(\text{instance–executed}, id, cid, \sigma, add_L, add_R, m) \overset{\tau_4}{\hookrightarrow} Q$ and stop. Else stop.

Case P is corrupt and Q is honest:

Upon (peaceful–request, $id, cid, f, z, s_P, \tau_0) \xhookleftarrow{\tau_1} P$

1. Let $\gamma^Q := \Gamma^Q(id)$, $\nu^Q := \gamma^Q.\mathsf{cspace}(cid)$, $\sigma^Q := \nu^Q.\mathsf{storage}$, $v^Q := \nu^Q.\mathsf{version}$. If $\gamma^Q = \bot$, $P \notin \gamma^Q.\mathsf{end\text{–}users}$, $\nu^Q = \bot$ or $f \notin \nu^Q.\mathsf{type}$, then goto step 4.
2. If $P = \gamma^Q.\mathsf{Alice}$ and $\tau_1 \mod 4 \neq 1$ or if $P = \gamma.\mathsf{Bob}$ and $\tau_1 \mod 4 \neq 3$, then goto step 4.
3. If (id, cid) is not marked as corrupt in Γ^Q, do:
   (a) Compute $(\tilde{\sigma}, add_L, add_R, m) := f(\sigma^Q, P, \tau_0, z)$.
   (b) If $m = \bot$ or $\mathtt{Vfy}_{pk_P}(id, cid, \tilde{\sigma}, \nu^Q.\mathsf{type}, v^Q + 1; s_P) \neq 1$, then goto step 4.
   (c) Send (execute, $id, cid, f, z) \xhookrightarrow{\tau_0} \mathcal{F}_{ch}$ on behalf of P and instruct the functionality to deliver the result.
   (d) Compute the signature $s_Q := \mathtt{Sign}_{sk_Q}(id, cid, \tilde{\sigma}, \nu^Q.\mathsf{type}, v^Q + 1)$, send (peaceful–confirm, id, $cid, f, z, s_Q) \xhookrightarrow{\tau_1+1} P$, set $\Gamma^Q := \mathtt{LocalUpdateAdd}(\Gamma^Q, id, cid, \tilde{\sigma}, \nu^Q.\mathsf{type}, add_L, add_R)$ and stop.
4. Mark (id, cid) as corrupt in Γ^Q and stop.

Upon P starting the registration procedure for id, cid, then execute the sub-simulator $\mathtt{SimRegister}(P, id, cid)$.

Upon (instance–execute, $id, cid, f, z) \xhookleftarrow{\tau_2} P$, then

1. Let $\gamma := \Gamma(id)$. If $\gamma = \bot$ or $P \notin \gamma.\mathsf{end\text{–}users}$, then stop. Else let $\nu := \gamma.\mathsf{cspace}(cid)$, $\sigma := \nu.\mathsf{storage}$. If $\nu = \bot$ or $f \notin \nu.\mathsf{type}$, then stop.
2. Else send (execute, $id, cid, f, z) \xhookrightarrow{\tau_2} \mathcal{F}_{ch}$ on behalf of P and within Δ round instruct the functionality to output the result.
3. If (executed, $id, cid, \sigma, add_L, add_R, m) \xhookleftarrow{\tau_3 \leq \tau_2 + \Delta} \mathcal{F}_{ch}$, then update the sets Γ^Q and Γ, send (instance–executed, $id, cid, \sigma, add_L, add_R, m) \xhookrightarrow{\tau_3} P$ and stop.

Case P and Q are corrupt:

Internally simulate the communication of the corrupt parties. If P starting the registration procedure for id, cid, then execute the sub-simulator $\mathtt{SimRegister}(P, id, cid)$ for the case when both parties are corrupt. Note that if the registration procedure is successful (a contract instance gets registered), the subsimulator $\mathtt{SimRegister}$ instructs the ideal functionality to update the contract instance accordingly.

If (instance–execute, $id, cid, f, z) \xhookleftarrow{\tau_2} P$, then

1. Let $\gamma := \Gamma(id)$. If $\gamma = \bot$ or $P \notin \gamma.\mathsf{end\text{–}users}$, then stop. Else let $\nu := \gamma.\mathsf{cspace}(cid)$, $\sigma := \nu.\mathsf{storage}$. If $\nu = \bot$ or $f \notin \nu.\mathsf{type}$, then stop.
2. Else send (execute, $id, cid, f, z) \xhookrightarrow{\tau_2} \mathcal{F}_{ch}$ on behalf of P and within Δ round instruct the functionality to output the result.
3. If (executed, $id, cid, \sigma, add_L, add_R, m) \xhookleftarrow{\tau_3 \leq \tau_2 + \Delta} \mathcal{F}_{ch}$, then update Γ, send (instance–executed, id, $cid, \sigma, add_L, add_R, m) \xhookrightarrow{\tau_3} P$ and stop.

Close a ledger state channel. Below we describe the simulator in case of ledger channel closure. We discuss all four possible situations.

Simulator $\mathcal{S}_1$: Close a ledger channel

We use the abbreviated notation from Section 3.3. Let $\mathcal{F}_{ch} := \mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(1, \mathcal{C})$.

Case P, Q are honest

Upon $(P, \text{close}, id) \xhookleftarrow{\tau_0} \mathcal{F}_{ch}$, proceed as follows. Let $\gamma^P := \Gamma^P(id)$. For every $cid \in \{0,1\}^*$ such that $\gamma^P.\mathsf{cspace}(cid) \neq \bot$, execute $\texttt{SimRegister}(P, id, cid)$ for the case when both parties are honest. In round $\tau_1 \leq \tau_0 + 8\Delta$ instruct the ideal functionality to output the result. If $(\text{closed}, id) \xhookleftarrow{\tau_1 \leq \tau_0 + 8\Delta} \mathcal{F}_{ch}$, set $\Gamma(id) := \bot$, $\Gamma^P(id) := \bot$, $\Gamma^Q(id) := \bot$ and stop.

Case P is honest and Q is corrupt:

Upon $(P, \text{close}, id) \xhookleftarrow{\tau_0} \mathcal{F}_{ch}$, do:

1. Let $\gamma^P := \Gamma^P(id)$. For every cid such that $\gamma^P.\mathsf{cspace}(cid) \neq \bot$ but the contract instance has never been registered, execute $\texttt{SimRegister}(P, id, cid)$.
2. After the execution of the subsimulator, wait for at most Δ rounds to send the message $(\text{contract–closing}, id) \xhookrightarrow{\tau_2 \leq \tau_0 + 4\Delta} Q$.
3. Execute the sub-simulator $\texttt{SimRegister}(Q, id, cid)$ if registration started by Q for some cid.
4. In round $\tau_3 \leq \tau_0 + 8\Delta$ instruct the ideal functionality to output the result. If $(\text{closed}, id) \xhookleftarrow{\tau_3} \mathcal{F}_{ch}$, set $\Gamma(id) := \bot$ $\Gamma^P(id) := \bot$ and send $(\text{contract–closed}, id) \xhookrightarrow{\tau_3} Q$. Then stop.

Case P is corrupt and Q is honest:

1. Execute the sub-simulator $\texttt{SimRegister}(P, id, cid)$ if registration started by P for some cid in round τ_0.
2. After the execution (in round $\tau_1 \leq \tau_0 + 2\Delta$), if $(\text{contract–close}, id) \xhookleftarrow{\tau_1} P$, where $\Gamma(id) \neq \bot$, then send $(\text{close}, id) \xhookrightarrow{\tau_1} \mathcal{F}_{ch}$ on behalf of P.
3. Wait at most Δ rounds to $(\text{contract–closing}, id) \xhookrightarrow{\tau_2 \leq \tau_0 + 3\Delta} P$.
4. Let $\gamma^Q := \Gamma^Q(id)$. If there exists cid such that $\gamma^Q.\mathsf{cspace}(cid) \neq \bot$ but the contract instance has never been registered, execute the sub-simulator $\texttt{SimRegister}(Q, id, cid)$.
5. Upon $(\text{closed}, id) \xhookleftarrow{\tau_5 \leq \tau_0 + 8\Delta} \mathcal{F}_{ch}$, $\Gamma^Q(id) := \bot$ and $(\text{contract–closed}, id) \xhookrightarrow{\tau_5} P$ and stop.

Case P and Q are corrupt:

1. Execute the sub-simulator $\texttt{SimRegister}(P, id, cid)$ if registration started by P for some cid. Note that if the registration procedure is successful (a contract instance gets registered), the subsimulator $\texttt{SimRegister}$ instructs the ideal functionality to update the contract instance accordingly.
2. After the execution (in round $\tau_1 \leq \tau_0 + 2\Delta$), if $(\text{contract–close}, id) \xhookleftarrow{\tau_1} P$, where $\Gamma(id) \neq \bot$, then send $(\text{close}, id) \xhookrightarrow{\tau_1} \mathcal{F}_{ch}$ on behalf of P.
3. Wait at most Δ rounds to $(\text{contract–closing}, id) \xhookrightarrow{\tau_2 \leq \tau_0 + 4\Delta} \Gamma(id).\mathsf{end–users}$.
4. Execute the sub-simulator $\texttt{SimRegister}(Q, id, cid)$ if registration started by Q for some cid. Again, if the registration procedure is successful, the subsimulator $\texttt{SimRegister}$ instructs the ideal functionality to update the contract instance accordingly.
5. In round $\tau_3 \leq \tau_0 + 8\Delta$ instruct the ideal functionality to output the result. If $(\text{closed}, id) \xhookleftarrow{\tau_1 \leq \tau_0 + 8\Delta} \mathcal{F}_{ch}$, set $\Gamma(id) := \bot$ and stop.

E Security analysis of the virtual channel

The purpose of this section is to prove Theorem2, i.e. show that for any $i > 1$ and any set $\mathcal{C}$ of contract types, the protocol $\Pi(i, \mathcal{C})$ emulates the ideal functionality $\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i, \mathcal{C})$ in $\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i-1, \texttt{VSC}_i(\mathcal{C}) \cup \mathcal{C})$-hybrid world against environments from the set $\mathcal{E}_{res}$.

The proof consists of two parts. First, we need to prove an auxiliary lemma stating that an instance of the protocol $\Pi(i, \mathcal{C})$ called by an environment $\mathcal{Z} \in \mathcal{E}_{res}$ is $\mathcal{E}_{res}$-respecting. This is because the hybrid ideal functionality $\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i-1, \texttt{VSC}_i(\mathcal{C}) \cup \mathcal{C})$ is emulated by the protocol $\Pi(i-1, \texttt{VSC}_i(\mathcal{C}) \cup \mathcal{C})$ only against environments from the set $\mathcal{E}_{res}$. This proves that the hybrid world is well defined and the composition of state channel protocols is possible. Thereafter we can construct the simulator $\mathcal{S}_i$ in order to prove that the protocol $\Pi(i,\mathcal{C})$ in the hybrid world of $\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i-1,\mathtt{VSC}_i(\mathcal{C})\cup\mathcal{C})$ emulates the ideal functionality $\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i,\mathcal{C})$ against environments from the set $\mathcal{E}_{res}$.

Lemma 1. *For any $i > 1$, set of contract types $\mathcal{C}$, PPT adversary $\mathcal{A}$ and environment $\mathcal{Z} \in \mathcal{E}_{res}$, the protocol $\Pi(i,\mathcal{C})$ is $\mathcal{E}_{res}$-respecting.*

Proof. We need to prove that for any PPT adversary $\mathcal{A}$ and any environment $\mathcal{Z} \in \mathcal{E}_{res}$, honest parties of the protocol $\Pi(i,\mathcal{C})$ make calls to the hybrid ideal functionality $\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i-1,\mathtt{VSC}_i(\mathcal{C})\cup\mathcal{C})$ according to the restrictions defining the set $\mathcal{E}_{res}$. In other words, honest parties of the protocol jointly represent an environment from the set $\mathcal{E}_{res}$.

If the environment $\mathcal{Z}$ sends a message to a honest party in the protocol regarding a state channel of length $j < i$, then the party simply forwards the message to the hybrid ideal functionality. Since $\mathcal{Z} \in \mathcal{E}_{res}$, no invalid calls can be made to the hybrid functionality in this way. It remains to show that the protocol is $\mathcal{E}_{res}$-respecting even if the environments sends a message regarding a state channel of length i.

First note that honest parties in the protocol $\Pi(i,\mathcal{C})$ upon receiving a message about a channel of length i only ask the hybrid ideal functionality to update or execute a contract instance in a state channel but never to create or close a state channel. Thus, none of the restrictions regarding creating or closing a state channel can be violated.

Parties of the protocol send messages regarding update of a contract instance to the hybrid ideal functionality $\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i-1,\mathtt{VSC}_i(\mathcal{C})\cup\mathcal{C})$ only during the protocol "Create a virtual channel". Since we assume that parties of the protocol receive messages from an environment $\mathcal{Z} \in \mathcal{E}_{res}$, we have the guarantee that they all receive the message (create,γ) in the same round τ_0. According to the protocol, party $\gamma.\mathsf{Alice}$ sends in round τ_0 the message $(\text{update}, id_A, cid_A, \tilde{\sigma}_A, \mathtt{VSC}_i(\mathcal{C}))$, where $\tilde{\sigma}_A := \mathtt{Init}_i^{\mathcal{C}}(\gamma.\mathsf{Alice},\tau_0,\gamma)$, $id_A := \gamma.\mathsf{subchan}(\gamma.\mathsf{Alice})$ and $cid_A := \gamma.\mathsf{Alice}||\gamma.\mathsf{id}$. Hence clearly $\tilde{\sigma}_A$ is admissible with respect to $\mathtt{VSC}_i(\mathcal{C})$. We can argue similarly with the update of the subchannel between $\gamma.\mathsf{Ingrid}$ and $\gamma.\mathsf{Bob}$. Since $\mathcal{Z} \in \mathcal{E}_{res}$, we know that both subchannels of the channel γ exist, that they contain no contract instances and that they have enough funds. In addition, the subchannels do support contracts of type $\mathtt{VSC}_i(\mathcal{C})$ since they were created via the hybrid ideal functionality $\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i-1,\mathtt{VSC}_i(\mathcal{C})\cup\mathcal{C})$.

Parties of the protocol send messages regarding execution of a contract instance to the hybrid ideal functionality $\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i-1,\mathtt{VSC}_i(\mathcal{C})\cup\mathcal{C})$ during (i) the protocol "Update a contract instance in a virtual channel" (more specifically in the procedure $\mathtt{Register}_i$), during (ii) the protocol "Execute a contract instance in a virtual channel" and (iii) during the protocol "Close virtual channel". Since $\mathcal{Z} \in \mathcal{E}_{res}$, we know that none these protocols is ever called for a channel that does not exists. This in particular implies that the contract instance that is being executed by parties of the protocol in the underlying subchannels must have been constructed and could not have been closed yet. In other words, we know that $\alpha.\mathsf{cspace}(cid_A) \neq \perp$ and $\beta.\mathsf{cspace}(cid_B) \neq \perp$, where $cid_A := \gamma.\mathsf{Alice}||\gamma.\mathsf{id}$, $\alpha := \Gamma^A(\gamma.\mathsf{subchan}(\gamma.\mathsf{Alice}))$ and $cid_B := \gamma.\mathsf{Bob}||\gamma.\mathsf{id}$, $\beta := \Gamma^B(\gamma.\mathsf{subchan}(\gamma.\mathsf{Bob}))$, where Γ^A and Γ^B are the channel space of Alice and Bob, respectively. □

In order to complete the proof that $\Pi(i,\mathcal{C})$ protocol $\mathcal{E}_{res}$-emulates the ideal functionality $\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i,\mathcal{C})$ in $\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i-1,\mathtt{VSC}_i(\mathcal{C})\cup\mathcal{C})$-hybrid world for any set of contract types $\mathcal{C}$, we need to construct a simulator $\mathcal{S}_i$ that simulates the real world adversary $\mathcal{A}$ for any environment $\mathcal{Z} \in \mathcal{E}_{res}$.

The simulator $\mathcal{S}_i$ constructed in this section will maintain a channels space Γ^T for every honest party $T \in \mathcal{P}$ and Γ for the hybrid ideal functionality $\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i-1,\mathtt{VSC}_i(\mathcal{C})\cup\mathcal{C})$. In addition, the simulator will generate a key pair $(pk_T, sk_T) \leftarrow\!\!\$\, \mathsf{KGen}(1^\lambda)$ for every honest party T during the setup phase. Recall that there are no private inputs or messages being sent, thus we assume that the ideal functionality $\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i,\mathcal{C})$ upon receiving a message m from party P immediately leaks (P,m) to the simulator $\mathcal{S}_i$ via the leakage port.

We will discuss in detail the most interesting case, when the ideal functionality $\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i,\mathcal{C})$ leaks a message about a channel of length exactly i or when a corrupt party P is instructed by the environment to update or execute a subchannel of a virtual channel of length exactly i, where the other user of the subchannel is not corrupt. The simulation in the remaining cases is straightforward. Let us describe it here only briefly.

If the ideal functionality $\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i,\mathcal{C})$ leaks a message about a channel of length j, where $1 \leq j < i$, the simulator internally executes the hybrid ideal functionality $\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i-1, \mathtt{VSC}_i(\mathcal{C}) \cup \mathcal{C})$ on the received message and sends the result to the adversary $\mathcal{A}$ (recall that honest parties in the protocol $\Pi(i,\mathcal{C})$ act like dummy parties and only forward messages to the hybrid ideal functionality $\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i-1, \mathtt{VSC}_i(\mathcal{C}) \cup \mathcal{C})$). If the corrupt parties are instructed to send valid replies to the hybrid ideal functionality $\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i-1, \mathtt{VSC}_i(\mathcal{C}) \cup \mathcal{C})$, the simulator $\mathcal{S}_i$ sends the messages to the ideal functionality $\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i,\mathcal{C})$ on their behalf and further influences the ideal functionality $\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i,\mathcal{C})$ as the simulator $\mathcal{S}_j$ would do. Thus specially, if all parties of a channel are honest, then the simulator $\mathcal{S}_i$ is defined exactly as the simulator $\mathcal{S}_j$. Let us give one example on how the simulator is defined in case there are corrupt parties.

Let us consider the situation when $\gamma.\mathsf{Alice}$ and $\gamma.\mathsf{Ingrid}$ are honest, $\gamma.\mathsf{Bob}$ is corrupt and the ideal functionality $\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i,\mathcal{C})$ leaks the messages $(\gamma.\mathsf{Alice}, \text{create}, \gamma)$ and $(\gamma.\mathsf{Ingrid}, \text{create}, \gamma)$, where $1 < \gamma.\mathsf{length} < i$, in round τ_0. Then the simulator waits till round $\tau_0 + 3$ if the corrupt party $\gamma.\mathsf{Bob}$ is instructed to send (create, γ) to the hybrid ideal functionality $\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i-1, \mathtt{VSC}_i(\mathcal{C}) \cup \mathcal{C})$. In that case, $\mathcal{S}_i$ forwards the message to the ideal functionality $\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i,\mathcal{C})$ on behalf of $\gamma.\mathsf{Bob}$, adds the new virtual channel γ to the channel spaces Γ^A and Γ. The simulator then waits till round $\gamma.\mathsf{validity}$.

The simulator $\mathcal{S}_i$ is defined similarly in the remaining case when it does not receive any leakage from the ideal functionality $\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i,\mathcal{C})$ but a corrupt party is instructed to send a message to the hybrid ideal functionality $\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i-1, \mathtt{VSC}_i(\mathcal{C}) \cup \mathcal{C})$ about a channel of length $1 \leq j < i$. This happens if a corrupt party is the initiator of execute or update procedure or when all parties of the channel are corrupt. In this situation, the simulator $\mathcal{S}_i$ internally executes the hybrid ideal functionality $\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i-1, \mathtt{VSC}_i(\mathcal{C}) \cup \mathcal{C})$. In case the message satisfies the restrictions on the environment, $\mathcal{S}_i$ forwards it to the ideal functionality $\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i,\mathcal{C})$ on behalf of the corrupt party and further influences the ideal functionality $\mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i,\mathcal{C})$ via the influence port as the simulator $\mathcal{S}_j$ would do.

From now on, we will focus on the simulator $\mathcal{S}_i$ for the most challenging case when at least one party of a virtual channel of length exactly i is honest.

Create a virtual channel. We begin with the definition of the simulator for virtual channel creation.

Simulator $\mathcal{S}_i$: Create a virtual channel

We use the notation established in Section 3.3 and denote the ideal functionality $\mathcal{F}_{ch}(i) := \mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i,\mathcal{C})$ and $\mathcal{F}_{ch}(i-1) := \mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i-1,\mathcal{C})$.

Case A, I, B are honest

Upon receiving $(A, \text{create}, \gamma) \xhookleftarrow{\tau_0} \mathcal{F}_{ch}(i)$, $(B, \text{create}, \gamma) \xhookleftarrow{\tau_0} \mathcal{F}_{ch}(i)$ and $(I, \text{create}, \gamma) \xhookleftarrow{\tau_0} \mathcal{F}_{ch}(i)$ proceed as follows:

1. Set $id_A := \gamma.\mathsf{subchan}(\gamma.\mathsf{id}), cid_A := A||\gamma.\mathsf{id}$ and $id_B := \gamma.\mathsf{subchan}(B)$, $cid_B := B||\gamma.\mathsf{id}$. Compute $\tilde{\sigma}_A := \mathtt{Init}_i^{\mathcal{C}}(A, \tau_0, \gamma)$ and $\tilde{\sigma}_B = \mathtt{Init}_i^{\mathcal{C}}(B, \tau_0, \gamma)$.
2. For both $T \in \{A, B\}$, internally simulate $\mathcal{F}_{ch}(i-1)$ upon receiving the message $(\text{update}, id_T, cid_T, \tilde{\sigma}_T, \mathtt{VSC}_i(\mathcal{C})) \xhookleftarrow{\tau_0} T$.
3. For both $T \in \{A, B\}$ internally simulate $\mathcal{F}_{ch}(i-1)$ upon receiving the message $(\text{update–reply}, ok, id_T, cid_T) \xhookleftarrow{\tau_0+1} I$.
4. Set $\Gamma^A(\gamma.\mathsf{id}) := \gamma$, $\Gamma^B(\gamma.\mathsf{id}) := \gamma$ and wait till round $\gamma.\mathsf{validity}$.

Case A, B are honest and I is corrupt:

Upon receiving $(A, \text{create}, \gamma) \xhookleftarrow{\tau_0} \mathcal{F}_{ch}(i)$ and $(B, \text{create}, \gamma) \xhookleftarrow{\tau_0} \mathcal{F}_{ch}(i)$ proceed as follows:

1. Set $id_A := \gamma.\mathsf{subchan}(\gamma.\mathsf{id}), cid_A := A||\gamma.\mathsf{id}$ and $id_B := \gamma.\mathsf{subchan}(B)$, $cid_B := B||\gamma.\mathsf{id}$. Compute $\tilde{\sigma}_A := \mathtt{Init}_i^{\mathcal{C}}(A, \tau_0, \gamma)$ and $\tilde{\sigma}_B = \mathtt{Init}_i^{\mathcal{C}}(B, \tau_0, \gamma)$.
2. For both $T \in \{A, B\}$, internally simulate $\mathcal{F}_{ch}(i-1)$ upon receiving the message (update, id_T, cid_T, $\tilde{\sigma}_T, \mathtt{VSC}_i(\mathcal{C}) \xhookleftarrow{\tau_0} T$ and forward the result to I.
3. If (update–reply, ok, id_T, cid_T) $\xhookleftarrow{\tau_0+1} I$ for $T \in \{A, B\}$, then internally simulate $\mathcal{F}_{ch}(i-1)$ upon receiving this message and forward the result to I.
4. If in round $\tau_0 + 1$, party I confirms both updates, then send (create, γ) $\xhookrightarrow{\tau_0+1} \mathcal{F}_{ch}(i)$ on behalf of I, set $\mathit{\Gamma}^A(\gamma.\mathsf{id}) := \gamma$, $\mathit{\Gamma}^B(\gamma.\mathsf{id}) := \gamma$
5. Wait till round $\gamma.\mathsf{validity}$.

Case A, I are honest and B is corrupt:

Upon $(A, \text{create}, \gamma) \xhookleftarrow{\tau_0} \mathcal{F}_{ch}(i)$ and $(I, \text{create}, \gamma) \xhookleftarrow{\tau_0} \mathcal{F}_{ch}(i)$, proceed as follows:

1. Set $id_A := \gamma.\mathsf{subchan}(\gamma.\mathsf{id}), cid_A := A||\gamma.\mathsf{id}$ and $id_B := \gamma.\mathsf{subchan}(B)$, $cid_B := B||\gamma.\mathsf{id}$. Compute $\tilde{\sigma}_A := \mathtt{Init}_i^{\mathcal{C}}(A, \tau_0, \gamma)$ and $\tilde{\sigma}_B = \mathtt{Init}_i^{\mathcal{C}}(B, \tau_0, \gamma)$.
2. In round τ_0, internally simulate $\mathcal{F}_{ch}(i-1)$ upon receiving (update, $id_A, cid_A, \tilde{\sigma}_A, \mathtt{VSC}_i(\mathcal{C})) \xhookleftarrow{\tau_0} A$.
3. If (update, $id_B, cid_B, \tilde{\sigma}_B, \mathtt{VSC}_i(\mathcal{C})) \xhookleftarrow{\tau_0} B$, then internally simulate $\mathcal{F}_{ch}(i-1)$ upon receiving this message and proceed. Else stop.
4. For both $T \in \{A, B\}$ internally simulate $\mathcal{F}_{ch}(i-1)$ upon receiving the message (update–reply, ok, $id_T, cid_T) \xhookleftarrow{\tau_0+1} I$ and forward the result of updating id_B to B.
5. Send (create–ok, γ) $\xhookrightarrow{\tau_0+3} B$ on behalf of A.
6. If (create–ok, γ) $\xhookleftarrow{\tau_0+2} B$, then send (create, γ) $\xhookrightarrow{\tau_0+3} \mathcal{F}_{ch}(i)$ on behalf of B, add γ to $\mathit{\Gamma}^A$.
7. Wait till round $\gamma.\mathsf{validity}$.

Case I, B are honest and A is corrupt:

Analogous to the case when only B is corrupt.

Case I, B are corrupt and A is honest:

Upon receiving $(A, \text{create}, \gamma) \xhookleftarrow{\tau_0} \mathcal{F}_{ch}(i)$ proceed as follows:

1. Set $id_A := \gamma.\mathsf{subchan}(\gamma.\mathsf{id}), cid_A := A||\gamma.\mathsf{id}$ and $id_B := \gamma.\mathsf{subchan}(B)$, $cid_B \neq B||\gamma.\mathsf{id}$. Compute $\tilde{\sigma}_A := \mathtt{Init}_i^{\mathcal{C}}(A, \tau_0, \gamma)$.
2. In round τ_0, internally simulate $\mathcal{F}_{ch}(i-1)$ upon receiving (update, $id_A, cid_A, \tilde{\sigma}_A, \mathtt{VSC}_i(\mathcal{C})) \xhookleftarrow{\tau_0} A$ and forward the result to I.
3. If (update–reply, ok, id_A, cid_A) $\xhookleftarrow{\tau_0+1} I$, then internally simulate $\mathcal{F}_{ch}(i-1)$ upon receiving this message, send (create, γ) $\xhookrightarrow{\tau_0+1} \mathcal{F}_{ch}(i)$ on behalf of I and send (create–ok, γ) $\xhookrightarrow{\tau_0+3} B$ on behalf of A.
4. If (create–ok, γ) $\xhookleftarrow{\tau_0+2} B$, then send (create, γ) $\xhookrightarrow{\tau_0+3} \mathcal{F}_{ch}(i)$ on behalf of B and add γ to $\mathit{\Gamma}^A$.
5. Wait till round $\gamma.\mathsf{validity}$.

Case A, I are corrupt and B is honest:

Analogous to the case when only A is honest.

Case A, B are corrupt and I is honest:

Upon receiving $(I, \text{create}, \gamma) \xhookleftarrow{\tau_0} \mathcal{F}_{ch}(i)$ proceed as follows:

1. Set $id_A := \gamma.\mathsf{subchan}(\gamma.\mathsf{id}), cid_A := A||\gamma.\mathsf{id}$ and $id_B := \gamma.\mathsf{subchan}(B)$, $cid_B \neq B||\gamma.\mathsf{id}$. Compute $\tilde{\sigma}_A := \mathtt{Init}_i^{\mathcal{C}}(A, \tau_0, \gamma)$ and $\tilde{\sigma}_B := \mathtt{Init}_i^{\mathcal{C}}(B, \tau_0, \gamma)$.

2. If $(\text{update}, id_A, cid_A, \tilde{\sigma}_A, \texttt{VSC}_i(\mathcal{C})) \overset{\tau_0}{\hookleftarrow} A$ and in the same round $(\text{update}, id_B, cid_B, \tilde{\sigma}_B, \texttt{VSC}_i(\mathcal{C})) \overset{\tau_0}{\hookleftarrow} B$, then proceed. Else stop.
3. For both $T \in \{A, B\}$, internally simulate $\mathcal{F}_{ch}(i-1)$ upon receiving the message $(\text{update}, id_T, cid_T, \tilde{\sigma}_T, \texttt{VSC}_i(\mathcal{C})) \overset{\tau_0}{\hookleftarrow} T$.
4. For both $T \in \{A, B\}$ internally simulate $\mathcal{F}_{ch}(i-1)$ upon receiving the message $(\text{update–reply}, ok, id_T, cid_T) \overset{\tau_0+1}{\hookleftarrow} I$ and forward the result to T.
5. Wait till round $\gamma.\mathsf{validity}$.

Register a contact instance in a virtual channel. Similarly as for the ledger state channels, we will separately define a sub-simulator $\texttt{SimRegister}_i$ which can be called as a procedure by the simulator $\mathcal{S}_i$. The subsimulator is defined below.

Subsimulator: $\texttt{SimRegister}_i(P, id, cid)$

We use the notation established in Section 3.3. In addition, let $\text{TE}_{sub} := \text{TimeExecute}(\lceil i/2 \rceil)$, $\mathcal{F}_{ch}(i) := \mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i, \mathcal{C})$ and $\mathcal{F}_{ch}(i-1) := \mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i-1, \mathcal{C})$.

All parties are honest:

1. Let $\gamma := \Gamma^P(id)$, $id_P := \gamma.\mathsf{subchan}(P)$, $cid_P := P||\gamma.\mathsf{id}$, $id_Q := \gamma.\mathsf{subchan}(Q)$, $cid_Q := Q||\gamma^P.\mathsf{id}$, $\nu^P := \gamma.\mathsf{cspace}(cid)$, $\nu^Q := \Gamma^Q(id).\mathsf{cspace}(cid)$ and let τ_0 be the current round.
2. In round τ_0, internally simulate $\mathcal{F}_{ch}(i-1)$ upon receiving $(\text{execute}, id_P, cid_P, \texttt{RegisterInstance}_i^{\mathcal{C}}, (cid, \nu^P)) \overset{\tau_0}{\hookleftarrow} P$.
3. In round $\tau_1 \leq \tau_0 + 5$, internally simulate $\mathcal{F}_{ch}(i-1)$ upon receiving $(\text{execute}, id_Q, cid_Q, \texttt{RegisterInstance}_i^{\mathcal{C}}, (cid, \nu^P)) \overset{\tau_1}{\hookleftarrow} I$.
4. In round $\tau_2 \leq \tau_1 + 5$, internally simulate $\mathcal{F}_{ch}(i-1)$ upon receiving $(\text{execute}, id_Q, cid_Q, \texttt{RegisterInstance}_i^{\mathcal{C}}, (cid, \nu^Q)) \overset{\tau_2}{\hookleftarrow} Q$.
5. In round $\tau_3 \leq \tau_2 + 5$, internally simulate $\mathcal{F}_{ch}(i-1)$ upon receiving $(\text{execute}, id_Q, cid_Q, \texttt{RegisterInstance}_i^{\mathcal{C}}, (cid, \nu^Q)) \overset{\tau_3}{\hookleftarrow} I$.

Case P, I are honest and Q is corrupt:

1. Let $\gamma := \Gamma^P(id)$, $id_P := \gamma.\mathsf{subchan}(P)$, $cid_P := P||\gamma.\mathsf{id}$, $id_Q := \gamma.\mathsf{subchan}(Q)$, $cid_Q := Q||\gamma.\mathsf{id}$, $\nu^P := \gamma.\mathsf{cspace}(cid)$ and let τ_0 be the current round.
2. In round τ_0, internally simulate $\mathcal{F}_{ch}(i-1)$ upon receiving $(\text{execute}, id_P, cid_P, \texttt{RegisterInstance}_i^{\mathcal{C}}, (cid, \nu^P)) \overset{\tau_0}{\hookleftarrow} P$.
3. In round $\tau_1 \leq \tau_0 + 5$, internally simulate $\mathcal{F}_{ch}(i-1)$ upon receiving $(\text{execute}, id_Q, cid_Q, \texttt{RegisterInstance}_i^{\mathcal{C}}, (cid, \nu^P)) \overset{\tau_1}{\hookleftarrow} I$ and forward the result of the execution to Q.
4. Let $\tau_2 \leq \tau_1 + \text{TE}_{sub}$ be the current round. Distinguish the following two cases
   - If $(\text{execute}, id_Q, cid_Q, \texttt{RegisterInstance}_i^{\mathcal{C}}, (cid, \nu^Q)) \overset{\tau_2}{\hookleftarrow} Q$, where ν^Q is a valid contract instance, then
     (a) Internally simulate $\mathcal{F}_{ch}(i-1)$ upon receiving $(\text{execute}, id_Q, cid_Q, \texttt{RegisterInstance}_i^{\mathcal{C}}, (cid, \nu^Q)) \overset{\tau_2}{\hookleftarrow} I$ and forward the result of execution to Q.
     (b) Let $\tilde{\nu}$ be the internally registered version of the contract instance cid in the channel id_Q. In round $\tau_3 \leq \tau_2 + \text{TE}_{sub}$, internally simulate $\mathcal{F}_{ch}(i-1)$ upon receiving $(\text{execute}, id_P, cid_P, \texttt{RegisterInstance}_i^{\mathcal{C}}, (cid, \tilde{\nu})) \overset{\tau_3}{\hookleftarrow} I$.
   - Otherwise in round $\tau_3 := \tau_1 + 3 \cdot \text{TE}_{sub}$ proceed as follows

(a) Internally simulate $\mathcal{F}_{ch}(i-1)$ upon receiving $(\text{execute}, id_Q, cid_Q, \texttt{EndRegisterInstance}_i, cid) \overset{\tau_3}{\hookleftarrow} I$ and forward the result of execution to Q.

(b) Internally simulate $\mathcal{F}_{ch}(i-1)$ upon receiving $(\text{execute}, id_P, cid_P, \texttt{EndRegisterInstance}_i, cid) \overset{\tau_3}{\hookleftarrow} I$.

Case P, Q are honest and I is corrupt:

1. Let $\gamma := \Gamma^P(id)$, $id_P := \gamma.\mathsf{subchan}(P)$, $cid_P := P||\gamma.\mathsf{id}$, $id_Q := \gamma.\mathsf{subchan}(Q)$, $cid_Q := Q||\gamma.\mathsf{id}$.
2. Let $\nu^P := \gamma.\mathsf{cspace}(cid)$ and $\nu^Q := \Gamma^Q(id).\mathsf{cspace}(cid)$.
3. Internally simulate $\mathcal{F}_{ch}(i-1)$ upon receiving the message $(\text{execute}, id_P, cid_P, \texttt{RegisterInstance}_i^{\mathcal{C}}, (cid, \nu^P)) \overset{\tau_0}{\hookleftarrow} P$ and forward the result of the execution to I.
4. Let $\tau_1 \leq \tau_0 + \text{TE}_{sub}$ be the current round. The distinguish the following two situations
   - If $(\text{execute}, id_Q, cid_Q, \texttt{RegisterInstance}_i^{\mathcal{C}}, (cid, \nu^P)) \overset{\tau_1}{\hookleftarrow} I$, then
     (a) Internally simulate $\mathcal{F}_{ch}(i-1)$ upon receiving $(\text{execute}, id_Q, cid_Q, \texttt{RegisterInstance}_i^{\mathcal{C}}, (cid, \nu^P)) \overset{\tau_1}{\hookleftarrow} I$ and forward the result of execution to I.
     (b) Let $\tau_2 \leq \tau_1 + \text{TE}_{sub}$ be the current round. Internally simulate $\mathcal{F}_{ch}(i-1)$ upon receiving $(\text{execute}, id_Q, cid_Q, \texttt{RegisterInstance}_i^{\mathcal{C}}, (cid, \nu^Q)) \overset{\tau_2}{\hookleftarrow} Q$ and forward the result of execution to I and goto step 5.
   - Else go to step 5.
5. Distinguish the following two cases
   - If $(\text{execute}, id_P, cid_P, \texttt{RegisterInstance}_i^{\mathcal{C}}, (cid, \tilde{\nu})) \xhookleftarrow{\tau_3 \leq \tau_0 + 3 \cdot \text{TE}_{sub}} I$ for some $\tilde{\nu}$ or if $(\text{execute}, id_P, cid_P, \texttt{EndRegisterInstance}_i, cid) \xhookleftarrow{\tau_3 \leq \tau_0 + 3 \cdot \text{TE}_{sub}} I$, then internally simulate $\mathcal{F}_{ch}(i-1)$ upon receiving one of these messages and forward the result of execution to I .
   - Otherwise in round $\tau_4 := \tau_1 + 3 \cdot \text{TE}_{sub}$, internally simulate $\mathcal{F}_{ch}(i-1)$ upon receiving $(\text{execute}, id_P, cid_P, \texttt{EndRegisterInstance}_i, cid) \overset{\tau_4}{\hookleftarrow} P$ and forward the result of execution to I.

Case I, Q are honest and P is corrupt:

Upon $(\text{execute}, id_P, cid_P, \texttt{RegisterInstance}_i^{\mathcal{C}}, (cid, \nu^P)) \overset{\tau_0}{\hookleftarrow} P$, such that $\alpha \neq \perp$ for $\alpha := \Gamma(id_P)$, $\nu \neq \perp$ for $\nu := \alpha.\mathsf{cspace}(cid_P)$, $\nu.\mathsf{type} = \mathtt{VSC}_i(\mathcal{C})$, $\nu.\mathsf{storage.cspace}(cid') = \perp$ for every $cid' \in \{0,1\}^*$ and ν^P is a valid contract instance, proceed as follows

1. Set $\gamma := \nu.\mathsf{storage.virtual\text{–}channel}$, $Q := \gamma.\mathsf{other\text{-}party}(P)$, $id_Q := \gamma.\mathsf{subchan}(Q)$, $cid_Q := Q||\gamma.\mathsf{id}$ and $\nu^Q := \Gamma^Q(\gamma.\mathsf{id}).\mathsf{cspace}(cid)$.
2. In round τ_0, internally simulate $\mathcal{F}_{ch}(i-1)$ upon receiving $(\text{execute}, id_P, cid_P, \texttt{RegisterInstance}_i^{\mathcal{C}}, (cid, \nu^P)) \overset{\tau_0}{\hookleftarrow} P$ and forward the result of the execution to P.
3. Let $\tau_1 \leq \tau_0 + \text{TE}_{sub}$ be the current round. Internally simulate $\mathcal{F}_{ch}(i-1)$ upon receiving $(\text{execute}, id_Q, cid_Q, \texttt{RegisterInstance}_i^{\mathcal{C}}, (cid, \nu^P)) \overset{\tau_1}{\hookleftarrow} I$.
4. In round $\tau_2 \leq \tau_1 + 5$, internally simulate $\mathcal{F}_{ch}(i-1)$ upon receiving $(\text{execute}, id_Q, cid_Q, \texttt{Register Instance}_i^{\mathcal{C}}, (cid, \nu^Q)) \overset{\tau_2}{\hookleftarrow} Q$.
5. In round $\tau_3 \leq \tau_2 + 5$, internally simulate $\mathcal{F}_{ch}(i-1)$ upon receiving $(\text{execute}, id_P, cid_P, \texttt{Register Instance}_i^{\mathcal{C}}, (cid, \nu^Q)) \overset{\tau_3}{\hookleftarrow} I$ and forward the result of the execution to P.

Case I is honest and P, Q are corrupt:

Upon $(\text{execute}, id_P, cid_P, \texttt{RegisterInstance}_i^{\mathcal{C}}, (cid, \nu^P)) \overset{\tau_0}{\hookleftarrow} P$, such that $\alpha \neq \perp$ for $\alpha := \Gamma(id_P)$, $\nu \neq \perp$ for $\nu := \alpha.\mathsf{cspace}(cid_P)$, $\nu.\mathsf{type} = \mathtt{VSC}_i(\mathcal{C})$, $\nu.\mathsf{storage.cspace}(cid') = \perp$ for every $cid' \in \{0,1\}^*$ and ν^P is a valid contract instance, proceed as follows:

1. Set $\gamma := \nu.\mathsf{storage}.\mathsf{virtual{-}channel}$, $Q := \gamma.\mathsf{other\text{-}party}(P)$, $id_Q := \gamma.\mathsf{subchan}(Q)$, $cid_Q := Q||\gamma.\mathsf{id}$.
2. Internally simulate $\mathcal{F}_{ch}(i-1)$ upon receiving the message $(\text{execute}, id_P, cid_P, \texttt{RegisterInstance}^{\mathcal{C}}_i, (cid, \nu^P)) \overset{\tau_0}{\hookleftarrow} P$ and forward the result of execution to P.
3. Let $\tau_1 \leq \tau_0 + \mathrm{TE}_{sub}$ be the current round. Internally simulate $\mathcal{F}_{ch}(i-1)$ upon receiving $(\text{execute}, id_Q, cid_Q, \texttt{RegisterInstance}^{\mathcal{C}}_i, (cid, \nu^P)) \overset{\tau_1}{\hookleftarrow} I$ and forward the result of execution to Q.
4. Let $\tau_2 \leq \tau_1 + \mathrm{TE}_{sub}$ be the current round. Then distinguish two cases
   - If $(\text{execute}, id_Q, cid_Q, \texttt{RegisterInstance}^{\mathcal{C}}_i, (cid, \nu^Q)) \overset{\tau_2}{\hookleftarrow} Q$, then proceed as follows
     (a) Internally simulate $\mathcal{F}_{ch}(i-1)$ upon receiving $(\text{execute}, id_Q, cid_Q, \texttt{RegisterInstance}^{\mathcal{C}}_i, (cid, \nu^Q)) \overset{\tau_2}{\hookleftarrow} Q$ and forward the result of the execution to Q. Let $\tilde{\nu}$ be the registered contract instance.
     (b) In round $\tau_3 := \tau_2 + \mathrm{TE}_{sub}$, internally simulate $\mathcal{F}_{ch}(i-1)$ upon receiving $(\text{execute}, id_P, cid_P, \texttt{RegisterInstance}^{\mathcal{C}}_i, (cid, \tilde{\nu})) \overset{\tau_3}{\hookleftarrow} I$ and forward the result of the execution to P.
   - Otherwise set $\tilde{\nu} := \nu^P$ and in round $\tau_3 := \tau_1 + 2 \cdot \mathrm{TE}_{sub}$ do the following
     (a) Internally simulate $\mathcal{F}_{ch}(i-1)$ upon receiving $(\text{execute}, id_Q, cid_Q, \texttt{EndRegisterInstance}_i, cid) \overset{\tau_3}{\hookleftarrow} I$ and forward the result of execution to Q.
     (b) Internally simulate $\mathcal{F}_{ch}(i-1)$ upon receiving $(\text{execute}, id_P, cid_P, \texttt{EndRegisterInstance}_i, cid) \overset{\tau_3}{\hookleftarrow} I$ and forward the result of execution to P.
5. Let τ_4 be the current round. Then $(\text{update}, id, cid, \tilde{\nu}.\mathsf{storage}, \tilde{\nu}.\mathsf{type}) \overset{\tau_4}{\hookrightarrow} \mathcal{F}_{ch}(i)$ on behalf of P and $(\text{update–reply}, ok, id, cid) \xhookrightarrow{\tau_4+1} \mathcal{F}_{ch}(i)$ on behalf of Q.

Case Q is honest and P, I are corrupt:

1. If $(\text{execute}, id_Q, cid_Q, \texttt{RegisterInstance}^{\mathcal{C}}_i, (cid, \nu^P)) \overset{\tau_0}{\hookleftarrow} I$, such that $\beta \neq \bot$ for $\beta := \mathit{\Gamma}(id_Q)$, $\nu \neq \bot$ for $\nu := \beta.\mathsf{cspace}(cid_Q)$, $\nu.\mathsf{type} = \mathsf{VSC}_i(\mathcal{C})$, $\nu.\mathsf{storage}.\mathsf{cspace}(cid') = \bot$ for every $cid' \in \{0,1\}^*$ and ν^P is a valid contract instance, then proceed. Otherwise stop.
2. Set $\gamma := \nu.\mathsf{storage}.\mathsf{virtual{-}channel}$, $Q := \gamma.\mathsf{other\text{-}party}(P)$ and $\nu^Q := \mathit{\Gamma}^Q(\gamma.\mathsf{id}).\mathsf{cspace}(cid)$.
3. Internally simulate $\mathcal{F}_{ch}(i-1)$ upon receiving the message $(\text{execute}, id_Q, cid_Q, \texttt{RegisterInstance}^{\mathcal{C}}_i, (cid, \nu^P)) \overset{\tau_0}{\hookleftarrow} I$ and forward the result of execution to I.
4. Let $\tau_1 \leq \tau_0 + \mathrm{TE}_{sub}$ be the current round. Internally simulate $\mathcal{F}_{ch}(i-1)$ upon receiving $(\text{execute}, id_Q, cid_Q, \texttt{RegisterInstance}^{\mathcal{C}}_i, (cid, \nu^P)) \overset{\tau_1}{\hookleftarrow} I$ and forward the result of execution to I.

Case P is honest and Q, I are corrupt:

1. Let $\gamma := \mathit{\Gamma}^P(id)$, $id_P := \gamma.\mathsf{subchan}(P)$, $cid_P := P||\gamma.\mathsf{id}$, $id_Q := \gamma.\mathsf{subchan}(Q)$, $cid_Q := Q||\gamma.\mathsf{id}$ and $\nu^P := \gamma.\mathsf{cspace}(cid)$.
2. Internally simulate $\mathcal{F}_{ch}(i-1)$ upon receiving the message $(\text{execute}, id_P, cid_P, \texttt{RegisterInstance}^{\mathcal{C}}_i, (cid, \nu^P)) \overset{\tau_0}{\hookleftarrow} P$ and forward the result of execution to I.
3. Then distinguish the following two situations
   - If $(\text{execute}, id_P, cid_P, \texttt{RegisterInstance}^{\mathcal{C}}_i, (cid, \widehat{\nu})) \xhookleftarrow{\tau_1 \leq \tau_0 + 3 \cdot \mathrm{TE}_{sub}} I$, then internally simulate $\mathcal{F}_{ch}(i-1)$ upon receiving $(\text{execute}, id_P, cid_P, \texttt{RegisterInstance}^{\mathcal{C}}_i, (cid, \widehat{\nu})) \overset{\tau_1}{\hookleftarrow} I$ and forward the result of execution to I.
   - Else, in round $\tau_2 := \tau_0 + 4 \cdot \mathrm{TE}_{sub}$, internally simulate $\mathcal{F}_{ch}(i-1)$ upon receiving $(\text{execute}, id_P, cid_P, \texttt{EndRegisterInstance}^{\mathcal{C}}_i, cid) \overset{\tau_2}{\hookleftarrow} I$ and forward the result of execution to I.

Update a contract instance in a virtual channel. The description of the simulator $\mathcal{S}_i$ for the contract instance update in a virtual channel of length i will be very similar to the simulator $\mathcal{S}_1$. Therefore, we refer the reader to the described in Appx. D and discuss here only the main differences. Firstly, the simulator $\mathcal{S}_i$ internally calls the subsimulator $\texttt{SimRegister}_i$ instead of the subsimulator $\texttt{SimRegister}$ and secondly, in case the initiating party P is corrupt the simulator $\mathcal{S}_i$ also checks if there is no other contract instance cid' already created in the virtual channel (recall that we allow only one contract instance to be opened in each virtual channel).

Execute a contract instance in a virtual channel. In case both end-users of the virtual channel are honest, the simulator $\mathcal{S}_i$ is defined exactly as the simulator $\mathcal{S}_1$, see Appx. D. Let us below define the simulator for the cases when at least one of the end-users is corrupt.

Simulator $\mathcal{S}_i$: Contract instance execution

We use the abbreviated notation from Section 3.3. Let $\mathrm{TE}_{sub} := \mathrm{TimeExecute}(\lceil i/2 \rceil)$, $\mathcal{F}_{ch}(i) := \mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i, \mathcal{C})$ and $\mathcal{F}_{ch}(i-1) := \mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i-1, \mathtt{VSC}_i(\mathcal{C}) \cup \mathcal{C})$.

$\boxed{\textbf{Case } P \textbf{ and } I \textbf{ are honest and } Q \textbf{ is corrupt:}}$

Upon $(P, \text{execute}, id, cid, f, z) \xhookleftarrow{\tau_0} \mathcal{F}_{ch}$, let $\gamma^P := \Gamma^P(id)$, $\nu^P := \gamma^P.\mathsf{cspace}(cid)$, $\sigma^P := \nu^P.\mathsf{storage}$ and $v^P := \nu^P.\mathsf{version}$. In addition, set $\tau_1 := \tau_0 + x$, where x is the smallest offset such that $\tau_1 = 1 \mod 4$ if $P = \gamma^P.\mathsf{Alice}$ and $\tau_1 = 3 \mod 4$ if $P = \gamma^P.\mathsf{Bob}$. Wait till round τ_1 and then proceed as follows:

1. If (id, cid) is not marked as corrupt in Γ^P, do:
   (a) Set $(\tilde{\sigma}, add_L, add_R, m) := f(\sigma^P, P, \tau_0, z)$. If $m = \bot$, then stop.
   (b) Else compute $s_P := \mathtt{Sign}_{sk_P}(id, cid, \tilde{\sigma}, \nu^P.\mathsf{type}, v^P + 1)$ and send $(\text{peaceful–request}, id, cid, f, z, s_P, \tau_0) \xhookrightarrow{\tau_1+1} Q$.
   (c) If $(\text{peaceful–confirm}, id, cid, f, z, s_Q) \xhookleftarrow{\tau_1+1} Q$ such that $\mathtt{Vfy}_{pk_Q}(id, cid, \tilde{\sigma}, \nu^P.\mathsf{type}, v^P + 1; s_Q) = 1$, then set $\Gamma^P := \mathtt{LocalUpdateAdd}(\Gamma^P, id, cid, \tilde{\sigma}, \nu^P.\mathsf{type}, add_L, add_R, v^P + 1, \{s_P, s_Q\})$ and instruct the ideal functionality to output the result. Else execute $\mathtt{SimRegister}_i(P, id, cid)$ in round $\tau_1 + 2$. If after the execution of the sub-simulator (in round $\tau_1 \leq \tau_0 + \mathrm{TE}_{sub} + 5$) it holds that $\sigma^P = \tilde{\sigma}$, then set $\Gamma^P := \mathtt{LocalUpdateAdd}(\Gamma^P, id, cid, \tilde{\sigma}, \nu^P.\mathsf{type}, add_L, add_R)$, instruct the ideal functionality to output the result and stop. Else goto step 3.
2. If (id, cid) is marked as corrupt and (id, cid) is not marked as registered in Γ^P, then execute the sub-simulator $\mathtt{SimRegister}_i(P, id, cid)$.
3. Let τ_3 be the current round and let $\gamma := \Gamma^P(id)$, $id_P := \gamma.\mathsf{subchan}(P)$, $cid_P := P || \gamma.\mathsf{id}$. Compute $s_n := \mathtt{Sign}_{sk_P}(cid, P, \tau_0, f, z)$ and set $p_n := (P, \tau_0, f, z, s_n)$. Then internally simulate $\mathcal{F}_{ch}(i-1)$ upon receiving $(\text{execute}, id_P, cid_P, \mathtt{ExecuteInstance}, (cid, p_n)) \xhookleftarrow{\tau_3} P$.
4. Let $Q := \gamma.\mathsf{other–party}(P)$, $id_Q := \gamma.\mathsf{subchan}(Q)$, $cid_Q := Q || \gamma.\mathsf{id}$. In round $\tau_4 \leq \tau_3 + 5$ internally simulate $\mathcal{F}_{ch}(i-1)$ upon receiving $(\text{execute}, id_Q, cid_Q, \mathtt{ExecuteInstance}, (cid, p_n)) \xhookleftarrow{\tau_4} I$ and forward the result to Q.
5. In round $\tau_5 \leq \tau_4 + \mathrm{TE}_{sub}$ internally simulate $\mathcal{F}_{ch}(i-1)$ upon receiving $(\text{execute}, id_P, cid_P, \mathtt{End\ ExecuteInstance}, (cid, p_n)) \xhookleftarrow{\tau_5} I$
6. Let $\tau_6 \leq \tau_5 + 5$ be the current round. Then instruct the ideal functionality to output the result and update the channel space Γ^P.

$\boxed{\textbf{Case } Q \textbf{ and } I \textbf{ are honest and } P \textbf{ is corrupt:}}$

Upon $(\text{peaceful–request}, id, cid, f, z, s_P, \tau_0) \xhookleftarrow{\tau_1} P$

1. Let $\gamma^Q := \Gamma^Q(id)$, $\nu^Q := \gamma^Q.\mathsf{cspace}(cid)$, $\sigma^Q := \nu^Q.\mathsf{storage}$, $v^Q := \nu^Q.\mathsf{version}$. If $\gamma^Q = \bot$ or $P \notin \gamma^Q.\mathsf{end–users}$ or $\nu^Q = \bot$ or $f \notin \nu^Q.\mathsf{type}$, then goto step 4.
2. If $P = \gamma^Q.\mathsf{Alice}$ and $\tau_1 \mod 4 \neq 1$ or if $P = \gamma.\mathsf{Bob}$ and $\tau_1 \mod 4 \neq 3$, then goto step 4.
3. If (id, cid) is not marked as corrupt in Γ^Q, do:
   (a) Compute $(\tilde{\sigma}, add_L, add_R, m) := f(\sigma^Q, P, \tau_0, z)$.

(b) If $m = \bot$ or $\texttt{Vfy}_{pk_P}(id, cid, \tilde{\sigma}, \nu^Q.\mathsf{type}, v^Q + 1; s_P) \neq 1$, then goto step 4.

(c) Send $(\text{execute}, id, cid, f, z) \overset{\tau_0}{\hookrightarrow} \mathcal{F}_{ch}$ on behalf of P and instruct the functionality to deliver the result.

(d) Compute the signature $s_Q := \texttt{Sign}_{sk_Q}(id, cid, \tilde{\sigma}, \nu^Q.\mathsf{type}, v^Q + 1)$, send (peaceful–confirm, id, $cid, f, z, s_Q) \overset{\tau_1+1}{\hookrightarrow} P$, set $\varGamma^Q := \texttt{LocalUpdateAdd}(\varGamma^Q, id, cid, \tilde{\sigma}, \nu^Q.\mathsf{type}, add_L, add_R)$ and stop.

4. Mark (id, cid) as corrupt in $\varGamma^Q$ and stop.

Upon P starting the registration procedure for id, cid, then execute the sub-simulator $\texttt{SimRegister}_i(P, id, cid)$.

Upon $(\text{execute}, id_P, cid_P, \texttt{ExecuteInstance}, (cid, p_n)) \overset{\tau_2}{\hookleftarrow} P$ proceed as follows:

1. Let $\alpha := \varGamma(id_P)$, $\nu_P := \alpha.\mathsf{cspace}(cid_P)$, $\gamma := \nu_P.\mathsf{storage}.\mathsf{virtual\text{–}channel}$, $Q := \gamma.\mathsf{other\text{–}party}(P)$, $id_Q := \gamma.\mathsf{subchan}(Q)$ and $cid_Q := Q||\gamma.\mathsf{id}$. In addition, parse $p_n := (P, \tau_0, f, z, s_n)$.
2. Send $(\text{execute}, \gamma.\mathsf{id}, cid, f, z) \overset{\tau_3}{\hookrightarrow} \mathcal{F}_{ch}(i)$ and internally simulate $\mathcal{F}_{ch}(i-1)$ upon receiving (execute, $id_P, cid_P, \texttt{ExecuteInstance}, (cid, p_n)) \overset{\tau_2}{\hookleftarrow} P$ and forward the result to P. If the result of execution is $(\text{executed}, id_P, cid_P, \tilde{\sigma}_P, L_P, R_P, m_P)$, where $m_P = (\text{instance–executing}, cid, p_n, m)$, proceed. Else stop.
3. In round $\tau_3 \leq \tau_2 + \mathrm{TE}_{sub}$, internally simulate $\mathcal{F}_{ch}(i-1)$ upon receiving (execute, id_Q, cid_Q, $\texttt{ExecuteInstance}, (cid, p_n) \overset{\tau_3}{\hookleftarrow} I$ and update the set $\varGamma^Q$.
4. In round $\tau_4 \leq \tau_3 + 5$, internally simulate $\mathcal{F}_{ch}(i-1)$ upon receiving (execute, $id_P, cid_P, \texttt{EndExecuteInstance}, (cid, p_n) \overset{\tau_4}{\hookleftarrow} I$ and forward the result to P.
5. Let $\tau_5 \leq \tau_4 + \mathrm{TE}_{sub}$ be the current round. Instruct the ideal functionality to output the result.

Case P and Q are corrupt and I is honest:

Internally simulate the communication of the corrupt parties. If P starting the registration procedure for id, cid, then execute the sub-simulator $\texttt{SimRegister}(P, id, cid)$ for the case when both parties are corrupt. Note that if the registration procedure is successful (a contract instance gets registered), the subsimulator $\texttt{SimRegister}_i$ instructs the ideal functionality to update the contract instance accordingly.

Upon $(\text{execute}, id_P, cid_P, \texttt{ExecuteInstance}, (cid, p_n)) \overset{\tau_2}{\hookleftarrow} P$ proceed as follows:

1. Let $\alpha := \varGamma(id_P)$, $\nu_P := \alpha.\mathsf{cspace}(cid_P)$, $\gamma := \nu_P.\mathsf{storage}.\mathsf{virtual\text{–}channel}$,$Q := \gamma.\mathsf{other\text{–}party}(P)$, $id_Q := \gamma.\mathsf{subchan}(Q)$ and $cid_Q := Q||\gamma.\mathsf{id}$. In addition, parse $p_n := (P, \tau_0, f, z, s_n)$.
2. Send $(\text{execute}, \gamma.\mathsf{id}, cid, f, z) \overset{\tau_3}{\hookrightarrow} \mathcal{F}_{ch}(i)$ and internally simulate $\mathcal{F}_{ch}(i-1)$ upon receiving (execute, $id_P, cid_P, \texttt{ExecuteInstance}, (cid, p_n)) \overset{\tau_2}{\hookleftarrow} P$ and forward the result to P. If the result of execution is $(\text{executed}, id_P, cid_P, \tilde{\sigma}_P, L_P, R_P, m_P)$, where $m_P = (\text{instance–executing}, cid, p_n, m)$, proceed. Else stop.
3. In round $\tau_3 \leq \tau_2 + \mathrm{TE}_{sub}$, internally simulate $\mathcal{F}_{ch}(i-1)$ upon receiving (execute, id_Q, cid_Q, $\texttt{ExecuteInstance}, (cid, p_n) \overset{\tau_3}{\hookleftarrow} I$ and forward the result to Q.
4. In round $\tau_4 \leq \tau_3 + \mathrm{TE}_{sub}$, internally simulate $\mathcal{F}_{ch}(i-1)$ upon receiving (execute, $id_P, cid_P, \texttt{End ExecuteInstance}, (cid, p_n) \overset{\tau_4}{\hookleftarrow} I$ and forward the result to P.
5. Let $\tau_5 \leq \tau_4 + \mathrm{TE}_{sub}$ be the current round. Instruct the ideal functionality to output the result.

Closing a virtual channel. Finally, we finalize the definition of the simulator $\mathcal{S}_i$ by defining its behavior in time $\gamma.\mathsf{validity}$, where γ is a virtual channel of length i whose creation environment requested earlier.

Simulator $\mathcal{S}_i$: Closing a virtual state channel

We use the abbreviated notation from Section 3.3. Let $\mathrm{TE}_{sub} := \mathrm{TimeExecute}(\lceil i/2 \rceil)$, $\mathcal{F}_{ch}(i) := \mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i, \mathcal{C})$ and $\mathcal{F}_{ch}(i-1) := \mathcal{F}_{ch}^{\mathcal{L}(\Delta)}(i-1, \mathtt{VSC}_i(\mathcal{C}) \cup \mathcal{C})$.

Case A, B, I are honest

Let γ the virtual channel to be closed. In round $\gamma.\mathsf{validity}$ proceed as follows for both $T \in \{A, B\}$.

1. Set $id_T := \gamma.\mathsf{subchan}(\gamma.\mathsf{id}), cid_T := T||\gamma.\mathsf{id}$.
2. In parallel, run the subsimulator $\mathtt{SimRegister}_i$ with parameters $T, \gamma.\mathsf{id}, cid$ for every $cid \in \{0,1\}^*$ such that $\Gamma^T(\gamma.\mathsf{id}).\mathsf{cspace}(cid) \neq \perp$.
3. After the subsimulator is executed, then internally simulate $\mathcal{F}_{ch}(i-1)$ upon receiving (execute, id_T, $cid_T, \mathtt{Close}_i^{\mathcal{C}}, \emptyset) \hookleftarrow T$.
4. After the execution, set $\Gamma^T(\gamma.\mathsf{id}) := \perp$.

Case A, B are honest and I is corrupt

In round $\gamma.\mathsf{validity}$ for both $T \in \{A, B\}$ proceed as follows.

1. Set $id := \gamma.\mathsf{id}$, $id_T := \gamma.\mathsf{subchan}(T)$, $cid_T := T||id$.
2. If $\Gamma^T(id) = \perp$ and $\Gamma(id_T).\mathsf{cspace}(cid_T) = \perp$, then stop.
3. If $\Gamma^T(id) = \perp$ but $\Gamma(id_T).\mathsf{cspace}(cid_T) \neq \perp$, then goto step 5.
4. If $\Gamma^T(id) \neq \perp$, then in parallel, run the subsimulator $\mathtt{SimRegister}_i$ with parameters $T, \gamma.\mathsf{id}, cid$ for every $cid \in \{0,1\}^*$ such that $\Gamma^T(\gamma.\mathsf{id}).\mathsf{cspace}(cid) \neq \perp$ and after the execution goto step 5.
5. In round $\tau_1 := \gamma.\mathsf{validity} + \mathrm{TimeRegister}(i) + \mathrm{TE}_{sub}$, internally simulate $\mathcal{F}_{ch}(i-1)$ upon receiving (execute, $id_T, cid_T, \mathtt{Close}_i^{\mathcal{C}}, \emptyset) \xhookleftarrow{\tau_1} T$.
6. After the execution, set $\Gamma^T(\gamma.\mathsf{id}) := \perp$.

Case A, I are honest and B is corrupt

In round $\gamma.\mathsf{validity}$ proceed as follows.

1. Set $id := \gamma.\mathsf{id}$, $id_A := \gamma.\mathsf{subchan}(A)$, $cid_A := A||id$, $id_B := \gamma.\mathsf{subchan}(B)$, $cid_B := B||id$.
2. If $\Gamma^A(id) = \perp$, then stop.
3. If $\Gamma^A(id) \neq \perp$, then in parallel, run the subsimulator $\mathtt{SimRegister}_i$ with parameters $A, \gamma.\mathsf{id}, cid$ for every $cid \in \{0,1\}^*$ such that $\Gamma^A(\gamma.\mathsf{id}).\mathsf{cspace}(cid) \neq \perp$ and (id, cid) is not marked as registered in Γ^A. After the execution goto step 5.
4. If B starts the registration procedure with parameters B, id, cid for some $cid \in \{0,1\}^*$, then execute the subsimulator $\mathtt{SimRegister}_i$ with the same parameters.
5. In round $\tau_1 := \gamma.\mathsf{validity} + \mathrm{TimeRegister}(i) + \mathrm{TE}_{sub}$, internally simulate $\mathcal{F}_{ch}(i-1)$ upon receiving (execute, $id_A, cid_A, \mathtt{Close}_i^{\mathcal{C}}, \emptyset) \xhookleftarrow{\tau_1} A$. After the execution, set $\Gamma^A(\gamma.\mathsf{id}) := \perp$.
6. Upon receiving (execute, $id_B, cid_B, \mathtt{Close}_i^{\mathcal{C}}, \emptyset) \xhookleftarrow{\tau_1} B$, internally simulate the functionality $\mathcal{F}_{ch}(i-1)$ upon receiving this message and forward the result to B.
7. If B does not initiate the execution, then in round $\tau_2 := \tau_1 + \mathrm{TE}_{sub}$ internally simulate the functionality $\mathcal{F}_{ch}(i-1)$ upon receiving (execute, $id_B, cid_B, \mathtt{Close}_i^{\mathcal{C}}, \emptyset) \xhookleftarrow{\tau_2} I$ and forward the result to I.

Case B, I are honest and A is corrupt

Analogous to the previous case.

Case A is honest and I, B are corrupt

In round $\gamma.\mathsf{validity}$ proceed as follows.

1. Set $id := \gamma.\mathsf{id}$, $id_A := \gamma.\mathsf{subchan}(A)$, $cid_A := A||id$
2. If $\Gamma^A(id) = \bot$ and $\Gamma(id_A).\mathsf{cspace}(cid_A) = \bot$, then stop.
3. If $\Gamma^A(id) = \bot$ but $\Gamma(id_A).\mathsf{cspace}(cid_A) \neq \bot$, then goto step 6.
4. If $\Gamma^A(id) \neq \bot$, then in parallel, run the subsimulator $\mathtt{SimRegister}_i$ with parameters $A, \gamma.\mathsf{id}, cid$ for every $cid \in \{0,1\}^*$ such that $\Gamma^A(\gamma.\mathsf{id}).\mathsf{cspace}(cid) \neq \bot$ and (id, cid) is not marked as registered in Γ^A. After the execution goto step 6.
5. If B starts the registration procedure with parameters B, id, cid for some $cid \in \{0,1\}^*$, then execute the subsimulator $\mathtt{SimRegister}_i$ with the same parameters.
6. In round $\tau_1 := \gamma.\mathsf{validity} + \mathrm{TimeRegister}(i) + \mathrm{TE}_{sub}$, internally simulate $\mathcal{F}_{ch}(i-1)$ upon receiving $(\text{execute}, id_A, cid_A, \mathtt{Close}_i^{\mathcal{C}}, \emptyset) \overset{\tau_1}{\hookleftarrow} A$. After the execution, set $\Gamma^A(\gamma.\mathsf{id}) := \bot$.

Case B is honest and I, A are corrupt

Analogous to the previous case.