Universally Composable Treatment of Multi-Party Isomorphic State Channels

Maxim Jourenko[1]*, Xiangyu Su[1] *,
Adam Blatchley Hansen[2], and Mario Larangeira[1,3]

[1] Institute of Science Tokyo, Japan
{jourenko.m.606a, su.x.4029, rebello.m.f72a}@m.isct.ac.jp
[2] Aarhus University, Denmark
blatchley@cs.au.dk
[3] Input Output Global (IOG), Singapore
mario.larangeira@iohk.io

Abstract. Layer-2 protocols are pivotal in enhancing the scalability of blockchain systems, enabling faster off-chain transactions while maintaining security. These protocols can bridge consensus-based blockchain systems and advanced applications, such as Multiparty Computation (MPC) protocols, often defined within the Universal Composability (UC) Framework. However, despite the existence of some UC-defined protocols, there is currently no comprehensive UC definition for isomorphic multiparty state channels, in particular that are not dependent on the ledger model: account or UTxO based ledger. These protocols depend heavily on timelock mechanisms and the accurate representation of time, which are challenging to model within the UC Framework. Our work addresses this gap by proposing a UC-based definition, *i.e.*, functionality, for isomorphic multi-party state channels that realistically model timely actions of honest users, a critical feature for the security of the channel protocol. Moreover our definition is agnostic with respect to the ledger model. Additionally, we introduce an extended timelock-aided global ledger functionality and demonstrate the security of existing protocols, namely Hydra proposed by Chakravarty et al. (FC'21), under the UC Framework and our proposed functionalities. This contribution provides a robust foundation for developing secure and scalable off-chain protocols in blockchain ecosystems. Finally, we concretely provide the construction of the extension of the Hydra Protocol, only outlined by the original work, and also prove it secure under our framework.

Keywords: Blockchain · Layer 2 · Universal Composability.

* This work was supported by JST CREST Grant Number JPMJCR2113, Japan and Input Output Global.

1 Introduction

A significant challenge for blockchain systems is to address scalability as a higher number of transactions per second (TPS). Ideally, the interaction between two or more participants in the protocol should not depend on the consensus protocol. There is a growing body of work that aims to allow such protocols to increase the TPS ratio, and, in general, they are based on layer-2 alternatives. In a nutshell, layer-2 protocols allow higher TPS despite an initial interaction with the consensus layer for setup. While the initial setup requires a longer time period. This arrangement results in participants freely issuing transactions among themselves directly without requiring any confirmation delay, in contrast to on-chain transactions, *i.e.*, transactions published on the ledger. Hence, a major obstacle to scalability is removed.

Layer-2 protocols come in several "flavors"; payment [23, 12, 32], state [13], and virtual channels [3], with either two or an arbitrary amount of players [11, 22, 10], and such that support general protocols [21, 18]. Despite the off-chain protocols variety, as layer-2 is also named, and assuming their efficiency in providing fast transaction/state layer on the top of blockchain systems, a major concern remains: their composable security.

As outlined earlier, the natural setting requires a stack of protocols, layer-1, *i.e.* the consensus, and layer-2 for scalability. Hence it is paramount to consider the security of the set of composed protocols. In more concrete terms, given a layer-2 protocol, can one positively answer the following question

"Is an arbitrary layer-2 protocol safely composed with the blockchain-based ledger underneath the system?"

This concern can also be extended in another direction.

"Given a particular layer-2 protocol, is it safe to rely on it in order to build increasingly more sophisticated applications on top of it?"

The centerpiece position between the ledger and complex applications urges a closer look into the security guarantees of such a class of protocols, in particular with respect to a wide range of composability requirements of the existing applications. Towards conciliating layer-2 security guarantees and existing applications, the wide range of sophisticated applications available offers clues to a candidate approach. That is, since sophisticated applications are often based on Multiparty Computation (MPC) protocols and their security is frequently defined in terms of the Universal Composability (UC) Framework, then a natural approach to specify a general security for layer2-protocols is to rely on a general UC-based security definition. That is the main goal of this work.

Time, Ledger and Channel Functionalities. We devise (1) a UC definition, *i.e.*, functionality $\mathcal{F}_{CH}$, for fast isomorphic multiplayer state channels. Our definition is designed to be ledger model agnostic, *i.e.* compatible with both the account and UTxO based ledgers. (2) Furthermore, $\mathcal{F}_{CH}$ is designed to be the, arguably, most general class of state channel protocols, namely, multiplayer and isomorphic, which in the context of channels means that parties can interact in the same manner with each other as on Layer 1 without loss of expressiveness.

(3) Furthermore, we define $\mathcal{F}_{CH\text{-}HT}$ as an extension to $\mathcal{F}_{CH}$ where we forego the requirement of being ledger agnostic and, instead, assume the UTxO Model. Thus we show additional security guarantees by utilizing a property of UTxOs which is that any permutation of transactions, that does not violate transaction dependencies, results in the same change of the ledger's state.

Crucially, the security of channel protocols depend on the notion of time, more concretely on time-locks, and the timely action of honest users. Modeling time has been shown somewhat tricky to capture in the UC Framework and it is a topic of research on its own [5, 6, 14]. In our context, "time" is used to implement a "dispute period" when the protocol participants can dispute an attempt to close the channel in a malicious state. Honest users are expected to timely react and present the correct state in order to close the channel safely.

It is not hard to see that the latency of the network should be taken into account when expecting a timely user reaction. Concretely, it is unrealistic to expect that an honest user has a perfectly updated "vision" of the state of the ledger for a timely reaction given the distributed nature of the system. Such setting is challenging for off-chain channels given its extreme dependence on timely dispute. For that reason, our proposed $\mathcal{F}_{CH}$ functionality is defined related to the notion of time via a global *ticker* functionality, along with a global *ledger* functionality which explicitly takes into account ledger *reading* and *writing* time delays. We remark that the latter is a novel definition based on existing work, namely [26], and, therefore, it is also one of our technical contributions.

Related Works. There is a growing body of work tackling several technical challenges of layer-2 [21, 18]: routing [31, 34, 33, 15], privacy [2, 28], permanent online requirement [4], channel capacity management [27], etc. Initially, payment protocols focus on pairwise payment channels. That is, a channel between only two participants typically based on Hashed Timelock Contract (HTLC) aided conditional payments. A practical application of such a technique is the Lightning Protocol (LN) [32] which concatenates channels to construct a *payment network*. LN is based on UTxO but inspired account-based protocols [1]. With the support of smart contracts, a generalization of such channels are *state* and *virtual channels* [13, 12, 3]. Such line of works inspired other protocols that are compatible with Bitcoin [23], *i.e.*, without the aid of smart contracts, and even payment hubs [12]. The use of smart contracts allows more sophisticated applications and protocol constructions. For example, more efficient techniques for closing channels [29] which inspired more efficient protocols [24].

A natural evolution of payment channel networks that are based on pairwise payment channels are *multiplayer* channels. There, players can interact with a set of other players directly instead of having to concatenate channels in a linear fashion, as with the earlier design of payment channel networks. To the best of our knowledge, the first full description of a multiparty state channel with formal UC treatment was introduced by Dziembowski et al. [11], which was followed by other concrete protocols without UC treatment [22, 10]. The dichotomy *account based* versus *UTxO based* ledgers fundamentally interferes with the design of protocols as we noticed that [11] is mainly designed for *account based ledgers*.

Our Contribution. We start by proposing a UC-based isomorphic multiparty state channel functionality which is holistic to the underpinning bookkeeping model of the ledger. One of the main technical challenges of our work is to tackle the security of the state channel construction with respect to modeling time[4]. Time modeling is critical for the security of most channels as the timely actions of honest users within the protocol are equally critical. Consider the case where the players disagree, the most commonly used design introduces a *dispute phase*, *i.e.*, a time window, during which any, possibly malicious, state can be challenged by any player of the protocol. It is not difficult to see that delayed action can result in a malicious state being imposed on the channel, resulting in financial losses. Furthermore, a *delayed vision* of the system state by honest users can result in late action, thus voiding the security guarantees of such actions. These properties of the system must be taken into account, as our work does, for a secure composing setting. Thus we introduce two functionalities: The channel and extended ledger functionality, namely $\mathcal{F}_{CH}$ and $\mathcal{G}_{\mathsf{Constrained\text{-}Ledger}}$ respectively.

Our ledger $\mathcal{G}_{\mathsf{Constrained\text{-}Ledger}}$ derives from $\mathcal{G}^{\delta}_{\mathsf{Ledger}}$ [26], where δ is the time delay parameter, and it is more realistic given that it is explicitly parameterized with $\delta_r, \delta_w, \Lambda$, and Σ_0, respectively, the ledger read and write delay time, the state transition algorithms and the initial state. Whereas $\mathcal{G}^{\delta}_{\mathsf{Ledger}}$ uses δ only to track when a submitted transactions *subjectively* enters a player's view, the separation of δ into δ_r and δ_w allows for the creation of an *objective* ledger time used for allowing the ledger to make decisions based on this objective time. In our work, the read and write time parameters, *i.e.*, δ_r and δ_w, are important given that they can interfere with the dispute phase by triggering late honest actions. Thus, $\mathcal{F}_{CH}$ is defined in a way that, given the separated delay times, it keeps the correct state sequence with $\mathcal{G}_{\mathsf{Constrained\text{-}Ledger}}$ and the state transition algorithms Λ.

In comparison to [11], our proposed description of multiparty state channel offers more flexibility when dealing with network delays: the early outlined ledger read and write time delays. Moreover, we opted for modularity to leave the concrete description of virtual channels out of the definition. Another significant feature of our model is to keep the definition agnostic to the bookkeeping model, *i.e.*, compatible with either *account* and *UTxO* based models. The key observation to propose such channel functionality is the focus on the transition of states only, regardless of the technicalities of the implementation of such states, *i.e.*, the bookkeeping model. In terms of transaction processing, our setting processes transactions in batches, again in contrast with [11] which does it one by one.

Finally, we apply our setting to investigate the security of an existing *isomorphic state multiparty channel protocol*, namely Hydra [10], and we show that, with minor changes, it is secure in our setting. Besides, [10] seems more suitable to our general setting than other multiparty channels protocols. For example, we need batch transaction processing, which exists in [10], to open the channel and our definition supports such a feature.

[4] Typically, the UC formalization does not take into account physical time directly, only more recently works propose techniques to properly model time [5, 6, 14] within the framework.

Summarizing, our contributions are the following

- First, we introduce a constrained ledger functionality, *i.e.*, $\mathcal{G}_{\mathsf{Constrained\text{-}Ledger}}$, compatible with the requirements of time-based disputes mandatory for the security of multiparty state channels, *i.e.*, $\delta_r, \delta_w, \Lambda$, and Σ_0 in **Section 3**;
- Then, we introduce a general security definition for composable multiparty isomorphic state channels by proposing $\mathcal{F}_{CH}$, which can account for full adversarial corruption, in **Section 4**;
- We show that an existing isomorphic state multiparty channel protocol, namely Hydra [10] (reviewed in **Appendix A**), is secure in the UC Framework (reviewed in **Section 2.3**), as shown by the proof in **Section 5**;
- We define $\mathcal{F}_{CH\text{-}HT}$ as an extension to $\mathcal{F}_{CH}$ which assumes the UTxO ledger model and demonstrates additional beneficial properties in **Section 6**; and
- We provide a formal definition of Hydra with hanging transactions in **Section 6.3**, which was previously described only on a high level in [10]. We prove its security and show it realizes $\mathcal{F}_{CH\text{-}HT}$ in **Section 6.4**, thus demonstrating additional beneficial properties.

Our setting seems more realistic regarding the existing time constraints, thus more suitable to a wide class of protocols that rely on timely actions from honest users, which is ubiquitous in layer-2 protocols. The more realistic setting, *i.e.* our novel time-constrained ledger functionality which takes into account time delays, presented a more technically challenging proof scenario. Hence, we consider our proposed setting, along with our security proof for a concrete protocol, the main technical contribution of this work.

Organization. In Section 2, we review basic definitions and recall the overall setting. Our concrete definitions for the *constrained ledger* and the full *channel* functionality description are thoroughly provided, respectively, in Sections 3 and 4, with the full security proof of the protocol is in Section 5. The concrete protocol we study with our framework, *i.e.*, The Hydra Protocol [10], is formally reviewed in Appendix A. We also introduce the extension of [10] that deals with hanging transactions, which was only outlined in that work. This work present its concrete extension and its corresponding security based in our security framework in Section 6.

2 Preliminaries

In the following we review the relevant background to this work. We briefly recall basic definitions: (1) Security of signature scheme; (2) the Universal Composability Framework (UC) and the global setting; (3) global functionalities, *i.e.*, the global ticker (to be used as a clock) and the global ledger; Lastly, as a practical study case, it is convenient, for completeness, to recall (4) the UTxO Model (and its extension, the Extended UTxO Model) and a high level overview of the Hydra Protocol, whereas, for completeness, the formal definition from [10] is deferred to Appendix A.

2.1 Notation

In this work we consider sets $S = \{s_0, \ldots, s_l\}$ and lists $L = [l_0, \ldots, l_m]$, $l, m \in \mathbb{N}$ where $\varnothing$ represents the empty set and ϵ the empty list. We denote union of sets as $S_0 \cup S_1$ and intersection of sets as $S_0 \cap S_1$, whereas $L_0||L_1$ represents concatenation of two lists and $L[i]$, $i \in \mathbb{N}$, $1 \leq i \leq |L|$ represents the i-th list element. For ease of notation, we extend set operations to lists as follows: $L_0 \setminus E$ is a list with all elements in L_0 with exception of all elements in E where E is either a set or a list. $L_0 \cup E$ is a set containing all elements in both L_0 and E. Moreover, we consider tuples of form $A = (a_1, \ldots, a_n)$ where $A.a_j$, $j, n \in \mathbb{N}$, $1 \leq j \leq n$ represents element a_j in tuple A. Lastly, we consider entities such as a party p, formally interactive turing machines as in Section 2.3, with an internal state. If a value b is in the internal state of p it is referred by $p.b$.

2.2 Standard Definition for Secure Signature

The correctness and EU-CMA of a signature scheme $\mathsf{DS} = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ is defined as follows, given the security parameter κ.

Definition 1 (Correctness). *A signature scheme is correct if we have that* $\Pr[\mathsf{Verify}(\mathsf{vk}, m, \mathsf{Sign}(\mathsf{sk}, m)) = 1] = 1$ *for any* $\kappa > 0$ *and* $(\mathsf{sk}, \mathsf{vk}) \leftarrow \mathsf{Gen}(1^\kappa)$.

Definition 2 (EU-CMA). *A signature scheme is said to be EU-CMA if for any adversary* $\mathcal{A}$ *that can query a signing oracle* $\mathcal{O}_{\mathsf{Sign}}(\mathsf{sk}, \cdot)$ *for at most* q *(which is polynomial in* κ*) times, such that the queried messages form a set* Q*, then*

$$\Pr\left[m' \notin Q \wedge \mathsf{Verify}(\mathsf{vk}, m', \sigma') = 1 \;\middle|\; (m', \sigma') \leftarrow \mathcal{A}^{\mathcal{O}_{\mathsf{Sign}}}(\mathsf{vk})\right]$$

is negligible in κ*, for any* $\kappa > 0$*, and* $(\mathsf{sk}, \mathsf{vk}) \leftarrow \mathsf{Gen}(1^\kappa)$.

2.3 The Universal Composability and Global Setting [7]

For completeness, we briefly review Canetti's Universal Composability (UC) Framework which defines three components: (1) a protocol π, (2) a functionality $\mathcal{F}$ that is a specification of the desired secure behavior of the protocol and (3) a simulator $\mathcal{S}$. The framework considers two worlds/executions: The real world in which π is executed between potentially n, for $n \in \mathbb{N}$, parties identified through identifiers *pid* and (2) an ideal world in which $\mathcal{F}$ is executed by a centralized trusted third party.

All entities in the UC model, *i.e.*, parties, functionalities are formally Interactive Turing Machines (ITMs) [16, 35, 17, 7] which are Turing machines that include facilities for communication in distributed systems: Each ITM contains additional tapes that can be accessed by other ITMs: An input tape representing received messages, an output tape containing outputs for a caller ITM and a backdoor tape for communication with an adversary.

Sessions of protocols are identified through identifiers *sid*. Then we proceed with an experiment: Both worlds are put into a black box and given to an environment $\mathcal{Z}$, that gives inputs and steers execution of the protocols inside the two black boxes that contain the real and ideal worlds. At the end, $\mathcal{Z}$ outputs a bit to indicate which black box contains the real world and which contains the ideal world. Moreover, $\mathcal{Z}$ is assisted by an adversary $\mathcal{A}$ that can corrupt parties through a backdoor, receiving their internal states and permission to steer their actions within the protocol.

More formally, in this simulation based argument, all parties from π perform the protocol, *i.e.*, the real execution $\mathsf{REAL}_{\pi,\mathcal{A},\mathcal{Z}}$, in the presence of the adversary $\mathcal{A}$ which can see and delay the messages. Whereas the ideal execution, *i.e.* $\mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$, is composed by the functionality $\mathcal{F}$ in the presence of the simulator $\mathcal{S}$, and it is also controlled by $\mathcal{Z}$.

Given the security parameter κ, randomness r and input z, $\mathcal{Z}$ drives both executions $\mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$ and $\mathsf{REAL}_{\pi,\mathcal{A},\mathcal{Z}}$, and outputs either 1 or 0. Therefore, let $\mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$ and $\mathsf{REAL}_{\pi,\mathcal{A},\mathcal{Z}}$ be respectively the ensembles, *i.e.*, ideal and real, $\{\mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(\kappa,z,r)\}_{\kappa\in\mathbb{N},z\in\{0,1\}^*}$ and $\{\mathsf{REAL}_{\pi,\mathcal{A},\mathcal{Z}}(\kappa,z,r)\}_{\kappa\in\mathbb{N},z\in\{0,1\}^*}$ of the outputs of $\mathcal{Z}$ for both executions. Thus, we say that π *realizes* $\mathcal{F}$ when there exists a probabilistic polynomial time (PPT) simulator $\mathcal{S}$, such that for all PPT $\mathcal{Z}$, we have $\mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}} \approx_c \mathsf{REAL}_{\pi,\mathcal{A},\mathcal{Z}}$ (for the standard definition of computationally indistinguishable binary ensembles $\approx_c$).

In more practical terms, the goal of the security proof is to create a simulator $\mathcal{S}$ that is placed in the ideal world and which wraps $\mathcal{A}$. Then, $\mathcal{S}$ simulates a view of π for $\mathcal{A}$. Note that $\mathcal{Z}$, $\mathcal{S}$ and $\mathcal{A}$, have polynomial bound computational resources. Security with an unbound $\mathcal{A}$ or $\mathcal{Z}$ are often much more technically challenging if not impossible, whereas proofs with an unbound $\mathcal{S}$ can be trivial. The proof succeeds if for any $\mathcal{Z}$ and $\mathcal{A}$, $\mathcal{Z}$ distinguishes the real and ideal worlds, *i.e.*, the two black boxes, with at most negligible probability. Then, as it was state earlier, it is said that π realizes the ideal functionality $\mathcal{F}$. The main rationale of the proof is that any attack on π can be translated to an attack on $\mathcal{F}$, which is secure by its definition. Thus, π is at least as secure as $\mathcal{F}$ and no further attacks on π exist in face of a polynomial time bounded adversary.

While this framework originally considers protocol sessions in isolation, global functionalities $\mathcal{G}$ model functionalities that are shared between an arbitrary amount of protocol sessions running arbitrary protocols [8]. In that case, it is said that π *realizes* $\mathcal{F}$ *in the* $\mathcal{G}$*-Global Model*. For a fully understanding of the framework, we refer the reader to both works [7, 8].

2.4 Helper Functions

We write $\mathsf{Call}\ (\mathsf{f}, \mathsf{p}_1, \ldots, \mathsf{p}_\mathsf{m})$ when specifying code that instructs execution of a subroutine f with parameters $\mathsf{p}_1, \ldots, \mathsf{p}_\mathsf{m}$, $m \in \mathbb{N}$. The called function might optionally return a value.

2.5 Global Ticker and Ledger Functionalities

Our channel functionality relies on $\mathcal{G}_{Ticker}$ for tracking progression of time, and an extension of $\mathcal{G}^{\delta}_{\mathsf{Ledger}}$ for the sequence of ledger states. We now review both.

Global Ticker Functionality [6]. The global ticker functionality $\mathcal{G}_{Ticker}$ is illustrated in Figure 1. While it does not track time explicitly, it does track the progression of time which is steered by the environment $\mathcal{Z}$, as is standard in the UC Framework. It allows parties to act and finish their computations in between two ticks. Moreover, it allows functionalities to track relative progression of time through querying message (TICKED?).

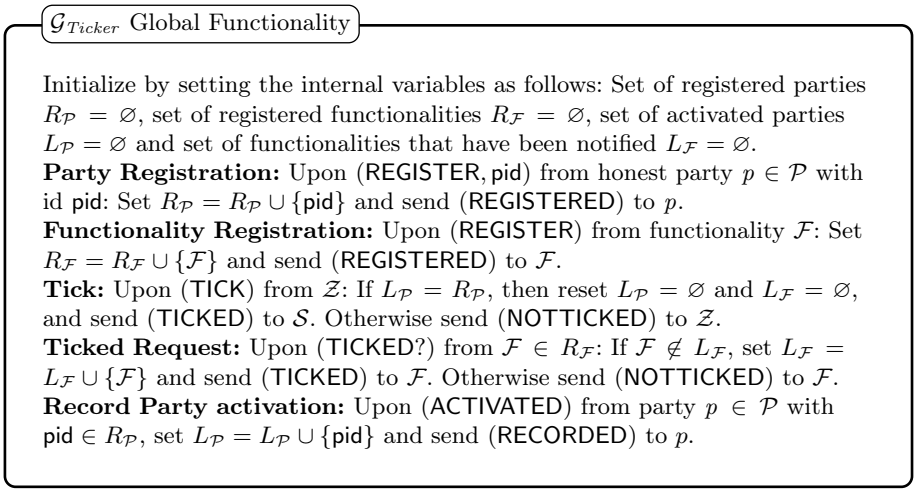

Fig. 1. The ticker functionality tracks progression of time while allowing honest parties to act in between two time ticks, and answering an arbitrary functionality $\mathcal{F}$.

Global Ledger Functionality [26]. The global ledger functionality $\mathcal{G}^{\delta}_{\mathsf{Ledger}}$ is illustrated in Figure 2. Note that the functionality is parameterized with the liveness parameter, *i.e.*, δ, meaning each transaction must enter the ledger's transaction history as well as the viewpoint of all parties within δ rounds subject to $\mathcal{G}_{Ticker}$. It starts in an initial state and maintains an immutable, append-only list of transactions Ω that can be read by parties by sending message (READ). A transaction τ can be appended to Ω through (SUBMIT, τ). However, an adversary $\mathcal{A}$ controls when a transaction enters the history, as well as its order, through sending a message (EXTEND, Ω'). Moreover, $\mathcal{A}$ controls when a transaction enters a party's $p \in \mathcal{P}$ viewpoint, *i.e.*, when a party can read the transaction, through sending message (ADVANCE, p, Ω'). Later we adapt this functionality, by making it receive *ticks* from the $\mathcal{G}_{Ticker}$, instead of interacting with a clock

functionality and keeping an internal counter for time. We note that the **assert** in $\mathcal{G}^{\delta}_{\mathsf{Ledger}}$ is used to limit the actions that can be done by the environment $\mathcal{Z}$ and adversary $\mathcal{A}$, *i.e.*, placing constraints on any security experiment done using $\mathcal{G}^{\delta}_{\mathsf{Ledger}}$. This is in contrast to the Assert (b) helper function defined in Section 2.4 represents a failure state in which $\mathcal{Z}$ is allowed to immediately succeed in distinguishing both worlds.

$\mathcal{G}^{\delta}_{\mathsf{Ledger}}$ Global Functionality [26]

Variables and Initialization: $\Omega = \epsilon$, $M = \lambda p.\epsilon$, and $U = \varnothing$.
Submit: Upon receiving (SUBMIT, τ) from party $p \in \mathcal{P}$, where τ is a transaction, do:

1: Send (READ) to $\mathcal{G}_{Clock}$, receiving t
2: Set $U \leftarrow U \cup \{(\tau, t)\}$
3: Send $(\mathsf{TRANSACTION}, \tau, t)$ to $\mathcal{A}$

Read: Upon receiving (READ) from $p \in \mathcal{P}$:

1: **assert** liveness ▷ This assertion places restrictions on $\mathcal{A}$ and $\mathcal{Z}$
2: **return** $\mathsf{map}(\mathsf{proj}_1, M(p))$

Extend: Upon receiving $(\mathsf{EXTEND}, \Omega')$ from $\mathcal{A}$, do:

1: **if** $\Omega' \subseteq U$ **then**
2: Set $U \leftarrow U \setminus \Omega'$ and $\Omega \leftarrow \Omega || \Omega'$

Advance: Upon receiving $(\mathsf{ADVANCE}, p, \Omega')$ from $\mathcal{A}$, do:

1: **if** $M(p) \prec \Omega' \prec \Omega$ **then**
2: $M(p) \leftarrow \Omega'$

Liveness Procedure:

1: send (READ) to $\mathcal{G}_{Clock}$, receiving t
2: **if** $\exists (\tau, t') \in U : |t - t'| > \delta$ **then**
3: **return** $\bot$ ▷ Write liveness
4: **else**
5: **if** $\exists (\tau, t') \in \Omega : |t - t'| > \delta \wedge \exists p \in \mathcal{H} : (\tau, t') \notin M(p)$ **then**
6: **return** $\bot$ ▷ Read liveness
7: **return** $\top$

Fig. 2. The functionality keeps a transaction history Ω, users views as mappings M and the list of time-stamped unconfirmed transactions U. Here we rely on the λ-expression from [26], *i.e.*, $M = \lambda p.\epsilon$, which means a mapping from parties to ledger states Ω (and they are initially set to ϵ). Moreover, we also rely on the standard common-prefix notation $a \prec b$, which means b includes the state a. Finally we make use of the following functions: a) the higher-order function map b) the tuple projection functions $proj_i$, which returns the ith element of a tuple. Both a) and b) deliver p's view of the ledger to p (as in notation from [26]). The original functionality receives time from $\mathcal{G}_{Clock}$, however we adapt it to receive *ticks* from $\mathcal{G}_{Ticker}$ in later sections.

2.6 Study Case: The UTxO Model and the Hydra Protocol [10]

Our UC modeling is generic enough for both account and UTxO based ledgers. However, as a study case, we will apply our framework to the UTxO based multiplayer channel. Both the UTxO Model and the Hydra Protocol [10] are related in the sense that the latter allows the execution of smart contracts in layer-2 via the extension of the former, *i.e.*, the Extended UTxO [9]. For completeness, as later we apply our framework on [10], it is convenient to review both.

The Unspent Transaction Output (UTxO) Model was introduced in Bitcoin [30] as a bookkeeping model for decentralized ledgers. Informally, UTxOs represent the coins that are in circulation on a ledger. Hence the set of UTxO available is the state Σ kept by the ledger. More formally, UTxOs are tuples of form (b, ν) where $b \in \mathbb{N}$ is an amount of coins and $\nu \in \{0,1\}^*$ is a verification script. A UTxO can be accessed through transactions $\tau = (I,\ O,\ t)$ where I is a set of inputs of form (ref, ρ) where ref uniquely identifies a UTxO u to be accessed while $\rho \in \{0,1\}^*$ is a redeemer, and finally O are newly the defined UTxO. In the following let $I.u$ be a shorthand for the UTxO that $I.\mathsf{ref}$ refers. There are several criteria for a transaction to be valid. Of note are that, (1) all verifiers must return $\top$, *i.e.*, $\forall i \in I : i.u.\nu(i.\rho) = \top$, (2) coins are not overspent, *i.e.*, $\sum_{i \in I} i.u.b \geq \sum_{u \in O} u.b$, (3) the timelock is expired, *i.e.*, $\tau.t \geq t_c$ where t_c is received from $\mathcal{G}_{Clock}$ and (4) UTxOs are valid and unique, *i.e.*, $\forall i \in I : i.u \in \mathcal{G}^{\delta}_{\mathsf{Ledger}}.\Sigma$ and $i.u$ was not accessed by a transaction before. Transactions effectively spend UTxOs and they can only be spent once. The EUTxO Model [9] extends the UTxO Model by allowing the execution of state machines on $\mathcal{G}^{\delta}_{\mathsf{Ledger}}$, which, in practice, puts forth means to implement smart contracts.

In the UTxO Model, two transactions τ_0, τ_1 *conflict*, if $\tau_0.I \cap \tau_1.I \neq \varnothing$. Moreover, transaction τ_0 is a *dependency* of τ_1, written $\tau_0 \prec \tau_1$, if $\tau_0.O \cap \tau_1.I \neq \varnothing$. A list of transactions L is a *valid* ordering or permutation of a set of transactions S if, $|S| = |L|, \forall \tau \in S \exists i \in \mathbb{N}, 1 \leq i \leq |L| : \tau = L[i]$ and, moreover, $\forall \tau_0 = L[i], \tau_1 = L[j] \in L, \tau_0 \neq \tau_1 : tx_0$ does not conflict tx_1 and $\tau_0 \prec \tau_1 \Rightarrow i \leq j$.

The UTxO ledger paradigm gives rise to Observation 1 stating that any valid permutation of transactions, when applied on a ledger, will result in the same state. Any valid permutation of transactions must not conflict, i.e. spend the same UTxO, or violate dependencies, *i.e.*, a UTxO must first be created by a transaction, before it can be spent by another. However, the resulting set of UTxOs that are removed and added to the ledger's state does not change, thus applying any valid permutation of a set of transactions they result in the same UTxO set as ledger state.

Observation 1. *Given a state of UTxO Ledger Σ_0 and a set of transactions T. Any valid permutation of the transactions in T when applied on Σ_0 result in the same ledger state Σ_1.*

The Hydra Protocol [10] is a state channel executed by $n \in \mathbb{N}$ parties based on EUTxO Model. In a nutshell, the channel maintains a set of UTxOs $\mathcal{U}$ and allows parties to modify the UTxOs through transactions with a protocol instead of submitting transactions to the ledger. Here, $\mathcal{U}$ is called a *snapshot*, *i.e.*, a collection of UTxOs that represents a state. Beginning at an initial snapshot $\mathcal{U}_0$ the parties create a series of superseding snapshots $\mathcal{U}_i, \ldots, \mathcal{U}_k, k \in \mathbb{N}$ by applying submitted transactions on the channel's managed set of UTxOs. At closure of the channel, the UTxOs that are contained in the latest snapshot $\mathcal{U}_k$ are moved onto the ledger. This allows the parties to submit an arbitrary number of transactions to the channel while at most $\mathcal{O}(n)$ transactions are submitted to the ledger.

There are two main components: (1) A state machine and (2) the *head* protocol. The state machine is executed on $\mathcal{G}^{\delta}_{\mathsf{Ledger}}$ through transactions and has four states as depicted in Figure 3. The *initial* state is created on the ledger via a transaction that sets up the state machine and sets the execution parameters such as the participating parties. After submission of the initial state, each party $p_i, i \in \mathbb{N}, 1 \leq i \leq n$ submits a *commit* transaction τ that allows each p_i to declare a set of UTxO $\mathcal{U}_0^i$ to be included in the initial snapshot $\mathcal{U}_0 = \bigcup_{i=1}^{n} \mathcal{U}_0^i$. On failure to do so, the channel can be aborted into the *final* state. However, otherwise, the channel can be opened by *collecting* the commit transactions and transitioning the state machine into the *open* state. At this point, the parties use the *head* protocol to submit transactions modifying the UTxOs in the channel and, moreover, creating superseding snapshots $\mathcal{U}_k$.

At any point in time the channel can be moved in the *closed* state by submitting a snapshot to the ledger. In this state, each party has one opportunity to *contest* by submitting more recent snapshots, *i.e.*, those with a higher index. This ensures that any honest party can enforce that the latest snapshot is submitted during the contestation phase. Finally, the latest snapshot, concretely, its UTxOs, are made available on the ledger through a *fanout* transaction, moving the state machine into the *final* state.

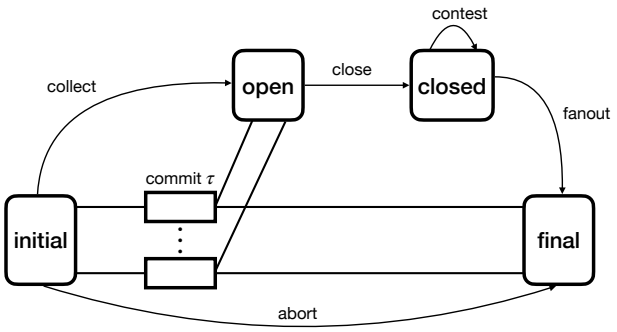

Fig. 3. The Hydra State Machine. Labeled boxes with round edges represent states while arrows are state transitions. Boxes with sharp edges represent transactions.

3 The Overall Setting: The Novel Constrained Ledger

This section is divided in the (1) general setting and the intuition of our design; and (2) the threat model. Regarding (1), we introduce $\mathcal{G}^{\delta_w,\delta_r,\Lambda,\Sigma_0}_{\mathsf{Constrained\text{-}Ledger}}$, a constrained version of the ledger $\mathcal{G}^{\delta}_{\mathsf{Ledger}}$ [26]. Timekeeping, which is crucial for the security of channels, is explicitly handled in $\mathcal{G}^{\delta_w,\delta_r,\Lambda,\Sigma_0}_{\mathsf{Constrained\text{-}Ledger}}$. This definition of ledger is one of our main technical contributions along with the novel channel functionality $\mathcal{F}_{CH}$ (to be presented in Section 4).

3.1 Novel Ledger Design Approach: Overview and Intuition

We consider $n \in \mathbb{N}$ parties executing either a concrete protocol Π in the *Real World* or interfacing the channel functionality $\mathcal{F}_{CH}$ in the *Ideal World*. Both worlds interact with the same global ledger functionality $\mathcal{G}^{\delta_w,\delta_r,\Lambda,\Sigma_0}_{\mathsf{Constrained\text{-}Ledger}}$. The simulator $\mathcal{S}$ simulates the view of Π for adversary $\mathcal{A}$ and is designed to interact only with $\mathcal{F}_{CH}$, notably even when interaction with $\mathcal{G}^{\delta_w,\delta_r,\Lambda,\Sigma_0}_{\mathsf{Constrained\text{-}Ledger}}$ is required. This design consideration ensures that $\mathcal{F}_{CH}$ is not sidelined and can impose the correct behavior expected from the protocol which, furthermore, simplifies the security proof. However, as $\mathcal{G}^{\delta_w,\delta_r,\Lambda,\Sigma_0}_{\mathsf{Constrained\text{-}Ledger}}$ is a global functionality, $\mathcal{Z}$ is free to interact with it directly. The timekeeper functionality $\mathcal{G}_{Ticker}$ allows the functionalities to keep track of a constrained notion of time and ensures all parties have finished acting before time progresses.

Yet another ledger functionality: discussion. As there are plenty ledger functionalities available in the literature, it raises the question on the necessity to propose another ledger functionality, in this case, $\mathcal{G}^{\delta_w,\delta_r,\Lambda,\Sigma_0}_{\mathsf{Constrained\text{-}Ledger}}$. For this, we look at our design considerations and requirements: (1) The channel should not be restricted to work on a specific ledger bookkeeping paradigm, *e.g.*, we require the ledger to be agnostic to whether it is implemented using the UTxO or accounting models; (2) we require that the ledger enforces correct bookkeeping and provides certain transaction logic, *i.e.*, signatures and timelocks. (3) We look for simplicity: The more complex a ledger functionality is, the more it might unnecessarily limit the potential ledger implementations that can realize it. We observe that $\mathcal{G}^{\delta}_{\mathsf{Ledger}}$ [26] is both, agnostic to any bookkeeping model and also simple enough to not limit how the ledger functionality can be implemented. However, as it does not fulfill requirement (2): While it tracks time, it does so only to limit how long it takes for a transaction to enter a party's view. This creates a *subjective* progression of the ledger's state for all parties. However, as we require that the ledger must be able to decide whether a transaction can be included on the ledger and at which point in time through timelocks, we require an *objective* notion of time. As the subjective view of a party is necessary to accurately model a realistic ledger, whereas an objective notion of time is required for implementation of layer-2 protocols that require timelocks, we extend $\mathcal{G}^{\delta}_{\mathsf{Ledger}}$ with our own functionality $\mathcal{G}^{\delta_w,\delta_r,\Lambda,\Sigma_0}_{\mathsf{Constrained\text{-}Ledger}}$.

Our novel functionality $\mathcal{G}^{\delta_w,\delta_r,\Lambda,\Sigma_0}_{\mathsf{Constrained\text{-}Ledger}}$ behaves similarly to $\mathcal{G}^{\delta}_{\mathsf{Ledger}}$; however, it utilizes the state transition algorithms Λ to span a state space created through an initial state and a list of transactions. Transactions that result in an invalid state are rejected by the ledger. While our setting is agnostic to how the state is created and transactions are filtered, we notably require that the ledger enables 1-out-of-n signatures, multisignatures[5] and timelocks, all of which are typically standard properties of ledgers. More concretely, let Σ_0 denote an initial state, Ω be an ordered list of transactions, and τ be a transaction. The state transition algorithms Λ is given by the tuple ($\mathsf{TX_VERIF}, \mathsf{STATE}, \mathsf{SIG}, \mathsf{REQ_T}, \mathsf{MS}, \mathsf{INV}$) that parameterize the functionality and are performed as follows:

- $\mathsf{TX_VERIF}(\Sigma_0, \Omega, \tau)$ returns $\top$, if τ can be appended to Ω according to arbitrary rules. For instance, the rules that enforce correct bookkeeping;
- $\mathsf{STATE}(\Sigma_0, \Omega)$ returns the resulting ledger state Σ from applying all *valid*, as given by procedure $\mathsf{TX_VERIF}$, transactions in Ω, in order, upon Σ_0;
- $\mathsf{SIG}(\Sigma_0, \Omega, \tau)$ returns a list of verification keys L_{vk} and a signature σ of τ; Note that, while σ is part of τ, the signature is on all of τ except for its signature component;
- $\mathsf{REQ_T}(\Sigma_0, \Omega, \tau)$ returns a time $t \in \mathbb{N}$;
- $\mathsf{MS}(\Sigma_0, \Omega, \tau)$ returns a tuple $(m, L_{\mathsf{vk,MS}}, L_\sigma)$, where either $m = \perp$ (*i.e.* multisignature is not required for τ), or $m \in \{0,1\}^*$ being arbitrary data, and $L_{\mathsf{vk,MS}}$ is a list of verification keys and L_σ a list of signatures; and
- $\mathsf{INV}(\tau)$ returns a transaction τ_{inv} which undoes the state transition τ performed, *i.e.*, $\mathsf{STATE}(\Sigma_0, [\tau, \mathsf{INV}(\tau)]) = \mathsf{STATE}(\Sigma_0, \epsilon)$.

The interface of $\mathcal{G}^{\delta_w,\delta_r,\Lambda,\Sigma_0}_{\mathsf{Constrained\text{-}Ledger}}$ differs from $\mathcal{G}^{\delta}_{\mathsf{Ledger}}$ by not accepting messages labeled EXTEND, but instead it has $(\mathsf{C_EXTEND}, \Omega')$ that skips invalid transactions determined by Λ. However, it calls $(\mathsf{EXTEND}, \cdot)$ from the interface of $\mathcal{G}^{\delta}_{\mathsf{Ledger}}$ as a subroutine. Moreover, it keeps time with the help of $\mathcal{G}_{Ticker}$ and an interface to receive TICK commands from $\mathcal{Z}$, enforcing liveness and storing timestamps for all transactions added to the ledger's transaction history. The timestamp of τ can be read by sending $(\mathsf{TIMESTAMP}, \tau)$ to $\mathcal{G}^{\delta_w,\delta_r,\Lambda,\Sigma_0}_{\mathsf{Constrained\text{-}Ledger}}$. The thorough description of $\mathcal{G}^{\delta_w,\delta_r,\Lambda,\Sigma_0}_{\mathsf{Constrained\text{-}Ledger}}$ is given by Figure 4.

3.2 The Security and Communication Model

We consider an adversary $\mathcal{A}$ who can *statically* corrupt an arbitrary number of parties at the beginning of the protocol. We consider authenticated channels between parties, however, any messages sent between parties can be arbitrarily delayed by $\mathcal{A}$. Note that this communication and adversarial model is consistent with that used in related works [3, 11, 13], but more general. In contrast to related work, we stress that this model includes the possibility for all messages between parties to be delayed by $\mathcal{A}$ for an arbitrary, but finite, amount of communication rounds, whereas in the related work communication is assumed to be synchronous: Messages sent in one round are guaranteed to be delivered at the beginning of the following round.

[5] These schemes can be instantiated by the EU-CMA secure signature scheme DS given in Section 2.2.

$\mathcal{G}^{\delta_w,\delta_r,\Lambda,\Sigma_0}_{\mathsf{Constrained\text{-}Ledger}}$ Global Functionality

Additional Variables: A set of transaction timestamps $T_\tau = \varnothing$ and ledger time $t = 0$.
When activated for the first time: Send (REGISTER) to $\mathcal{G}_{Ticker}$.
Tick: Upon receiving (TICK) from $\mathcal{Z}$, do

1: Send (TICKED?) to $\mathcal{G}_{Ticker}$ receiving reply m
2: **if** $m = TICKED$ **then**
3: $t \leftarrow t + 1$
4: Assert (liveness)
5: Send (TICKED) to $\mathcal{Z}$
6: **else**
7: Send (NOTTICKED) to $\mathcal{Z}$

Constrained Extend: Upon receiving input $(\mathsf{C_EXTEND}, \Omega')$ from $\mathcal{A}$, do

1: **for** $\tau \in \Omega'$ **do**
2: **if** $(\mathsf{EVAL}, \Sigma_0, \Omega, \tau)$ returns $\top$ **then**
3: $\mathsf{Call}(\mathsf{EXTEND}, \tau)$ ▷ $\Omega \leftarrow \Omega||\{\tau\}$ as in $\mathcal{G}^{\delta}_{\mathsf{Ledger}}$
4: $T_\tau \leftarrow T_\tau \cup \{(\tau, t)\}$

Get Timestamps: Upon receiving input $(\mathsf{TIMESTAMP}, \tau)$, do

1: **if** $\exists (\tau, t) \in T_\tau$ **then**
2: **return** t

Eval: Upon receiving input $(\mathsf{EVAL}, \Sigma_0', \Omega', \tau)$, do

1: **if** $\Omega' \neq \varnothing$ **then** ▷ Induction: Extending Σ_0' by Ω' must be valid
2: $(\Omega_{\mathsf{pre}}||\tau_{top}) \leftarrow \Omega'$ ▷ τ_{top} is last element of Ω' and Ω_{pre} is remainder
3: **if** $(\mathsf{EVAL}, \Sigma_0', \Omega_{\mathsf{pre}}, \tau_{\mathsf{top}})$ returns $\bot$ **then return** $\bot$
4: $(L_{\mathsf{vk,MS}}, \sigma_p) \leftarrow \Lambda.\mathsf{SIG}(\Sigma_0', \Omega', \tau)$ ▷ 1-out-of-n signature
5: **if** $\nexists \mathsf{vk} \in L_{\mathsf{vk,MS}} : \mathsf{DS.Verify}(\mathsf{vk}_p, \tau, \sigma_p) = 1$ **then return** $\bot$
6: $(m, L_{\mathsf{vk,MS}}, L_\sigma) \leftarrow \Lambda.\mathsf{MS}(\Sigma_0', \Omega', \tau)$ ▷ Multisignature
7: **if** $(m \neq \bot \wedge L_{\mathsf{vk,MS}} \neq \varnothing)$ **then** ▷ If $m = \bot$, τ does not require Multisig.
8: **for** $\mathsf{vk_{MS}} \in L_{\mathsf{vk,MS}}$ **do**
9: **if** $\nexists \sigma \in L_\sigma : \mathsf{DS.Verify}(\mathsf{vk_{MS}}, m, \sigma) = 1$ **then return** $\bot$
10: **if** $\Lambda.\mathsf{REQ_T}(\Sigma_0', \Omega', \tau) > t$ **then return** $\bot$ ▷ Timelock
11: **return** $\Lambda.\mathsf{TX_VERIF}(\Sigma_0', \Omega', \tau)$ ▷ Bookkeeping and other rules

Functionality read: Upon receiving (READ) from functionality $\mathcal{F}$ send $(\mathsf{LEDGER}, \Omega)$ to $\mathcal{F}$.
Liveness Procedure:

1: **if** $\exists (\tau, t') \in U : |t - t'| > \delta_w$ **then**
2: **return** $\bot$ ▷ Write liveness
3: **else**
4: **if** $\exists (\tau, t') \in \Omega : |t - t'| > \delta_r \wedge \exists p \in \mathcal{H} : (\tau, t') \notin M(p)$ **then**
5: **return** $\bot$ ▷ Read liveness
6: **return** $\top$

Fig. 4. The functionality is defined as the previous ledger functionality in Figure 2, and it is extended by timestamping transactions and creating a state space through the State Transition Function Λ that can filter invalid transactions.

4 The Channel Functionality $\mathcal{F}_{CH}$

In the following, we fully describe the channel functionality $\mathcal{F}_{CH}$ which is defined atop a ledger $\mathcal{G}_{\mathsf{Constrained\text{-}Ledger}}^{\delta_w,\delta_r,\Lambda,\Sigma_0}$ (Figures 6 through 10). We first introduce the desired properties of $\mathcal{F}_{CH}$ in Section 4.1, in particular we introduce the helper function ζ which represents the *state machine* of the channel. In this section, we also provide the intuition and design considerations that guide the specification of the functionality. These considerations are important to understand the detailed description of $\mathcal{F}_{CH}$ in Sections 4.2 - 4.6. We conclude with a discussion of the overall design in Section 4.7.

4.1 Overview and Intuition

In a nutshell, a channel is a structure deployed on a ledger by a set of parties, which, upon creation, allows them to interact with each other by submitting transactions to the channel instead of to the ledger itself. A channel consists of two components. For one, a *state machine*, *i.e.*, a program, executed on a global ledger which can be considered a trusted third party. For another, a protocol executed by the parties, driving the execution of the state machine. The channel maintains a state, *e.g.*, a distribution of coins between the channel participants which are locked with the channel on the ledger. Submitting transactions changes this state, *e.g.*, it modifies the coin balances on the channel through payments. Upon closure of the channel, the ledger's state changes consistent with sending all transactions that were originally sent to the channel to the ledger itself. In most channel protocols, closing the channel is done with a number of transactions that is constant in the amount of transactions sent to the channel. Considering the above example, the channel can be closed through a transaction that unlocks its coins and distributes them according to the latest coin distribution.

$\mathcal{F}_{CH}$ does not actively drive execution of the channel, but instead acts passively: $\mathcal{F}_{CH}$ forwards all messages from $\mathcal{Z}$ to $\mathcal{S}$, and fowards all messages between $\mathcal{S}$ and the global functionalities $\mathcal{G}_{\mathsf{Constrained\text{-}Ledger}}^{\delta_w,\delta_r,\Lambda,\Sigma_0}$ and $\mathcal{G}_{Ticker}$. Instead, $\mathcal{F}_{CH}$ utilizes the $\mathsf{Assert}\ (\dots)$ procedure, which represents a failure state if its argument equals $\bot$, to place constraints on how the channel can be executed to ensure security guarantees. This, for one, allows for generality, *i.e.* $\mathcal{S}$ is not restrained and can simulate a variety of potential protocols. Furthermore, it also allows easier security proofs as a prover mostly only has to focus on showing that the functionality does not reach a failure state with more than negligible probability.

Channel Lifecycle. The lifecycle of $\mathcal{F}_{CH}$ starts with all participants submitting an initialization message, including the channel parameters, to $\mathcal{F}_{CH}$ as shown in Figure 5. To ensure that all parties agree on the same channel parameters, the functionality drops any initialization message where parameters are inconsistent with previously submitted ones. It only proceeds once all parties submitted a valid initialization message. Afterwards, parties submit transactions to create the channel's initial state and open the channel. At this point, parties execute three

round entity confirmation protocols to (1) acknowledge and confirm transactions and (2) acknowledge and confirm new channel states. While transactions are acknowledged by a party based on their local view of the order of transactions, state transitions create a global ordering of transactions and resolve conflicts that arise from conflicting local views of the parties. Lastly, when the channel is closed, $\mathcal{F}_{CH}$ ensures that it is closed by moving either to (1) the latest channel state to the ledger, or (2) the subsequent state, if it was acknowledged by at least all honest parties. The reason for the latter case is that $\mathcal{A}$ can make corrupted parties to acknowledge the subsequent state or not.

Desired properties and design goals. We aim for $\mathcal{F}_{CH}$ to represent a variety of channel implementations. Thus, we require it to be general as follows: (1) We desire the channel to be state agnostic, *i.e.*, it should be able to maintain an arbitrary state; (2) The channel should be secure, *i.e.*, if there exists a honest party participating in the protocol, then all transactions sent to the channel must be included on channel closure; (3) It should be ledger agnostic, *i.e.*, it should not be limited to a ledger implementation. Note that while the above notion of security in (2) is rather informal, the ideal functionality $\mathcal{F}_{CH}$ represents the formal security definition of a secure channel. In Section 4.7, we discuss whether $\mathcal{F}_{CH}$ achieves these properties.

The State Machine's Transaction Function ζ. The channel is operated through a *state machine* executed on $\mathcal{G}^{\delta_w,\delta_r,\Lambda,\Sigma_0}_{\mathsf{Constrained\text{-}Ledger}}$. On one side, we desire $\mathcal{F}_{CH}$ to be ledger agnostic. However on the other side, we require for $\mathcal{F}_{CH}$ to be able to interact with $\mathcal{G}^{\delta_w,\delta_r,\Lambda,\Sigma_0}_{\mathsf{Constrained\text{-}Ledger}}$ in a meaningful manner in order to submit transactions and to interpret transactions sent to the ledger. We resolve this contradiction utilizing the State Machine Transaction function ζ. This function represents the *state machine*, and it is specific to the ledger implementation, as well as the protocol that attempts to implement the $\mathcal{F}_{CH}$, moreover it must be provided by $\mathcal{S}$ to $\mathcal{F}_{CH}$ during channel setup. The function $\zeta : (\mathsf{H}_\tau, p, \mathsf{param}, S_\sigma) \to \tau$ takes an identifier describing the type of transaction $\mathsf{H}_\tau \in \{\mathsf{INIT}, \mathsf{CLOSE}, \mathsf{COMM}, \mathsf{COLL}, \mathsf{ABORT}, \mathsf{CONTEST}, \mathsf{FANOUT}\}$, a party p, a set of parameters param and a set of signatures S_σ as input. It outputs a transaction τ, created by p, subject to param, and including the required signatures, if they are in S_σ.

4.2 Initialization

During setup, as given by Figure 5, each party submits the parameter they want to operate the channel with, *i.e.*, a list of participating parties γ_p, the duration of the contestation period t_c, and $\Sigma_i \subseteq \Sigma$ where $\Sigma = \bigcup_{p_i \in \gamma_p} \Sigma_i$ is the channel's initial state. Here, the initial state Σ_i is only forwarded to $\mathcal{S}$ to allow for correct simulation of the protocol, however, the actual initial state will be obtained by $\mathcal{F}_{CH}$ by reading the transactions submitted to the ledger as shown in Figure 9. Moreover, the first time $\mathcal{F}_{CH}$ is activated, it asks $\mathcal{S}$ for the initialization parameter, *i.e.*, the parameters Λ and $\delta = \delta_w + \delta_r$ of $\mathcal{G}^{\delta_w,\delta_r,\Lambda,\Sigma_0}_{\mathsf{Constrained\text{-}Ledger}}$, the corrupted parties in $\mathcal{C}$, and the transaction and transition functions ζ and θ. If all parties are corrupted, it executes the procedure $\mathsf{FullAdvControl}$, shown in Figure 6, giving $\mathcal{S}$ full control of $\mathcal{F}_{CH}$, which allows $\mathcal{S}$ to trivially ensure indistinguishability.

$\mathcal{F}_{CH}^{SM}$ Functionality (Initialization)

Internal Variables:

- Channel internal state machine $\gamma_S = \textsc{New}$
- State machine transition function $\theta = \perp$
- State machine transaction function $\zeta = \perp$
- Parties $\gamma_p = \varnothing$, corrupted parties $\mathcal{C} = \varnothing$ and honest parties $\mathcal{H} = \varnothing$
- Whether parties' negotiated channel initialization $b_i = \perp$
- Parties' opening commitments $\Sigma_i = \perp, i \in \mathbb{N}$
- Parties' set of opening transactions $S^{\mathsf{TX}}_{\mathsf{open,i}}$
- The channel's initial state Σ_0^{γ}
- Seen and ordered lists of transactions $L^{\mathsf{TX}}_{\mathsf{seen},i} = L^{\mathsf{ST}}_{\mathsf{seen},i} = \epsilon$ for each $p_i \in \mathcal{H}$
- Set of rejected transactions $S^{\mathsf{TX}}_{\mathsf{Reject},i} = \varnothing$ for each party $p_i \in \mathcal{H}$
- Sets of rejected $S^{\mathsf{TX}}_{\mathsf{Reject}} = \varnothing$ and confirmed transactions $S^{\mathsf{TX}}_{\mathsf{conf}} = \varnothing$
- Ordered list of transactions seen by all honest parties $L^{\mathsf{ST}}_{\mathsf{pending}} = \epsilon$
- Set of transactions seen by all honest parties $S^{\mathsf{TX}}_{\mathsf{reject,pending}} = \varnothing$
- An ordered list of confirmed and ordered lists of transactions $L^{\mathsf{ST}}_{\mathsf{conf}} = \epsilon$
- Duration of the contesting phase $t_c = -1$ and its start time $t_e = -1$
- Channel time $t = 0$
- Views of the ledger state $\Omega_i = \epsilon$ for each $p_i \in \mathcal{H}$
- A bit $b_{\mathsf{fail}} = \perp$ indicating $\mathcal{F}_{CH}$ is in the failure state

---

Setup: Upon receiving message $(\mathsf{SETUP}, sid, \gamma'_p, t'_c, \Sigma_i, i)$ from a party p

1: $p_i \leftarrow p$
2: **if** $\gamma_p = \varnothing$ **then**
3: $\quad \gamma_p \leftarrow \gamma_{p'}$
4: $\quad$ Send $(\mathsf{FUNC_SETUP}, sid)$ to $\mathcal{S}$, receiving $(\zeta', \theta', \delta'_r, \mathcal{C}')$
5: $\quad \mathcal{C} \leftarrow \mathcal{C}'$, $\mathcal{H} \leftarrow \gamma_p \setminus \mathcal{C}'$, $\zeta \leftarrow \zeta'$, $\theta \leftarrow \theta'$, $t_c \leftarrow t'_c$
6: $\quad$ Send $(\mathsf{REGISTER})$ to $\mathcal{G}_{Ticker}$
7: $\quad$ **if** $\mathcal{C} = \gamma_p$ **then** $\mathsf{Call}(\mathsf{FullAdvControl})$
8: **if** $b_i = \perp \wedge \gamma_p = \gamma'_p \wedge t_c = t'_c$ **then**
9: $\quad$ Send $(\mathsf{PARTY_SETUP}, sid, p_i, \gamma'_p, t'_c, \Sigma_i, i)$ to $\mathcal{S}$
10: $\quad b_i \leftarrow \top$
11: $\quad$ **if** $p_i \in \mathcal{H}$ **then**
12: $\quad\quad$ Send $(\mathsf{REGISTER}, p_i)$ to $\mathcal{G}_{Ticker}$ on behalf of p_i receiving reply m
13: $\quad\quad$ Send m on behalf of $\mathcal{G}_{Ticker}$ to p_i
14: **if** $\forall p_i \in \gamma_p : b_i = \top$ **then** $\gamma_S \leftarrow \textsc{NotYetOpen}$

Fig. 5. During the setup, the functionality differentiate between three cases 1) when the user started the channel creation from start; 2) some initiation has already occurred by another player; or 3) all have already negotiated the channel creation. $\mathcal{F}_{CH}$ receives parameter from $\mathcal{S}$ by accessing its output tape upon sending message $\mathsf{FUNC_SETUP}$.

While we do not concretely specify the channel state machine transaction and transition functions ζ and θ respectively, $\mathcal{S}$ provides the functions, *i.e.*, the ones that implement the desired channel, to fulfill the $\mathcal{F}_{CH}$ constraints.

4.3 Behavior on Functionality Activation

Figure 6 displays the functionality's behavior whenever it is activated, e.g. upon receipt of a message. Before it proceeds with reading and processing a message, it updates its internal clock by interacting with $\mathcal{G}_{Ticker}$. Moreover, if all parties are corrupt, $\mathcal{F}_{CH}$ suppresses evaluation of all assumptions as no security guarantees can be provided in this case. To aid simulation it grants $\mathcal{S}$ full control.

Moreover, $\mathcal{F}_{CH}$ has the procedure Assert (b), $b \in \{\bot, \top\}$. If $b = \top$ the instruction does nothing, whereas $b = \bot$ represents $\mathcal{F}_{CH}$ entering failure state: After entering the failure state, whenever $\mathcal{F}_{CH}$ receives a message, instead of processing the message it sends (ASSERT_FAILURE, *sid*) to all protocol participants and terminates.

$\mathcal{F}_{CH}^{SM}$ Functionality (Behavior on activation)

On Activation: When activated with (instr, *sid*, ...), before proceeding

1: **if** $b_{\mathsf{fail}} \wedge \gamma_p \neq \mathcal{C}$ **then** Assert ($\bot$) and drop message
2: **if** sid is not current session id **then** drop message
3: Send (TICKED?) to $\mathcal{G}_{Ticker}$
4: **if** reply is (TICKED) **then** $t \leftarrow t + 1$

On FullAdvControl: On message (FullAdvControl, *sid*) from $\mathcal{S}$: If $\mathcal{C} = \gamma_p \neq \varnothing$, change channel state to allow $\mathcal{S}$ full control of the functionality, including when to output messages to $\mathcal{Z}$, giving $\mathcal{S}$ a trivial way to ensure indistinguishability. Skip execution of any Assert (...) instructions.

Procedure Assert (b)**:**

1: **if** $b = \bot \wedge \gamma_p \neq \mathcal{C}$ **then** send (ASSERT_FAILURE, *sid*) to all $p \in \gamma_p$
2: $b_{\mathsf{fail}} \leftarrow \top$

Fig. 6. The functionality's behavior upon activation, executed whenever activated before proceeding. Moreover, if all parties are corrupted, allow $\mathcal{S}$ to control $\mathcal{F}_{CH}$ and suppress all assertions, as no security guarantees can be provided in this case.

4.4 Interaction with Global Functionalities

Figure 7 describes the functionality backdoor interface with $\mathcal{S}$ for interactions with $\mathcal{G}^{\delta_w,\delta_r,\Lambda,\Sigma_0}_{\mathsf{Constrained\text{-}Ledger}}$, *i.e.*, reading the ledger and submitting transactions, and $\mathcal{G}_{Ticker}$. This makes sure $\mathcal{S}$ only interacts with $\mathcal{F}_{CH}$ and no global functionality, simplifying its construction. Moreover, $\mathcal{F}_{CH}$ only forwards the communication between $\mathcal{S}$ and the global functionalities.

$\mathcal{F}^{SM}_{CH}$ Functionality (Backdoor tape: global functionality interaction)

Party Activation: Upon receiving $(\mathsf{ACTIVATED}, sid, p_i)$ from $\mathcal{S}$, $p_i \in \mathcal{H}$:

1: Send $(\mathsf{ACTIVATED})$, on behalf of p_i, to $\mathcal{G}_{Ticker}$ receiving the reply m
2: Send m, on behalf of $\mathcal{G}_{Ticker}$, to p_i

Read Ledger: Upon $(\mathsf{READ}, sid, p_i)$ from $\mathcal{S}$, do

1: Send (READ), on behalf of p_i, to $\mathcal{G}^{\delta_w,\delta_r,\Lambda,\Sigma_0}_{\mathsf{Constrained\text{-}Ledger}}$ receiving Ω'
2: **if** $\Omega_i \subseteq \Omega'$ **then**
3: Let D be as follows: $\Omega' = \Omega_i || D$
4: **for** $\tau \in D$ **do**
5: $\mathsf{OnTx}(\tau, p_i)$
6: Send $(\mathsf{LEDGER_READ}, sid, p_i, \Omega')$ to $\mathcal{S}$

Submit Transaction: Upon $(\mathsf{S_TX}, sid, \tau)$ from $\mathcal{S}$, send (SUBMIT, τ) to $\mathcal{G}^{\delta_w,\delta_r,\Lambda,\Sigma_0}_{\mathsf{Constrained\text{-}Ledger}}$.

Fig. 7. The backdoor accessed by $\mathcal{S}$ for interactions with $\mathcal{G}^{\delta_w,\delta_r,\Lambda,\Sigma_0}_{\mathsf{Constrained\text{-}Ledger}}$. This indirect access to the global ledger simplifies the framework allowing simpler security proofs.

4.5 Channel Transaction Handling and Functionality State Lifecycle

Figure 8 describes the protocol interface, *i.e.*, what messages ideal parties sent: (1) Issuing new transactions via $(\mathsf{NEW}, sid, \tau)$ and channel state updates through $(\mathsf{NEW_ST}, sid, \mathcal{U})$; (2) aborting channel creation through (ABORT, sid); (3) closing the channel through (CLOSE, sid); and (4) resolving the channel on the ledger through (FANOUT, sid). When these messages are received. the functionality notifies $\mathcal{S}$ which decides whether and at what time transactions are issued as shown in Figure 7 and detailed in the next section.

The bulk of the complexity in Figure 8 is regarding the transaction and channel state lifecycles: (1) When transactions are issued, they are unordered. Parties process transactions in the order in which they arrive and only *acknowledge* an them, if it is consistent with their current view, *e.g.* if a transaction double spends some coins a previous transaction already spent, then that transaction will be dropped. This implies that each honest party might have a different ordering of acknowledged transactions; (2) Channel states are updated and *confirmed* to create a consistent ordering of transactions for all parties; (3) Once all honest parties *acknowledge* an entity and there is at least one corrupted party, the adversary can decide whether the entity can be confirmed by selecting whether

corrupted parties sign or do not sign it, and (4) Since $\mathcal{F}_{CH}$ is aimed to be agnostic to the ledger's bookkeeping paradigm, a transaction can only be considered to affect the channel if it was included in a channel state update. The reason for this is that the outcome of a transaction in the account based model highly depends on the ordering of transactions.

$\mathcal{F}_{CH}^{SM}$ Functionality (Backdoor tape: transaction and state lifecycle)

Instructions: Upon receiving input $(\mathsf{instr}, sid, \mathsf{param})$ from party p where $\mathsf{instr} \in \{\mathsf{ABORT}, \mathsf{NEW}, \mathsf{NEW_ST}, \mathsf{CLOSE}, \mathsf{FANOUT}\}$, do

1: **if** $\mathsf{instr} = \mathsf{CLOSE} \wedge \gamma_S = \textsc{Open} \wedge t_e < 0$ **then** $t_e \leftarrow t$
2: Send $(\mathsf{instr}, sid, p, \mathsf{param})$ to $\mathcal{S}$

State Request: Upon receiving input (STATE, sid) from party $p_i \in \gamma_p$ do

1: Send $(\mathsf{STATE}, sid, L^{\mathsf{ST}}_{\mathsf{conf},i}, L^{\mathsf{ST}}_{\mathsf{seen},i}, S^{\mathsf{TX}}_{\mathsf{reject},i}, S^{\mathsf{TX}}_{\mathsf{conf},i}, L^{\mathsf{TX}}_{\mathsf{seen},i})$ to p

Acknowledge TX: Upon input $(\mathsf{ACK_TX}, sid, \tau, p_i)$, $p_i \in \mathcal{H}$ from $\mathcal{S}$, do

1: Submit $(\mathsf{EVAL}, \Sigma_0^\gamma, L^{\mathsf{ST}}_{\mathsf{conf}} || L^{\mathsf{ST}}_{\mathsf{seen},i} || L^{\mathsf{TX}}_{\mathsf{seen},i}, \tau)$ to $\mathcal{G}^{\delta_w,\delta_r,\Lambda,\Sigma_0}_{\mathsf{Constrained\text{-}Ledger}}$ receiving b
2: Assert $((\gamma_S = \textsc{Open} \vee t \leq t_e + \delta_r) \wedge b \wedge \tau \notin S^{\mathsf{TX}}_{\mathsf{Reject},i})$
3: $L^{\mathsf{TX}}_{\mathsf{seen},i} \leftarrow L^{\mathsf{TX}}_{\mathsf{seen},i} || \tau$

Confirm TX: Upon input $(\mathsf{CONF_TX}, sid, \tau, p_i)$, $p_i \in \mathcal{H}$ from $\mathcal{S}$, do

1: Assert $((\gamma_S = \textsc{Open} \vee t \leq t_e + \delta_r) \wedge \tau \notin S^{\mathsf{TX}}_{\mathsf{Reject},i} \wedge \forall p_i \in \mathcal{H} : \tau \in L^{\mathsf{ST}}_{\mathsf{seen},i} || L^{\mathsf{TX}}_{\mathsf{seen},i})$
2: $S^{\mathsf{TX}}_{\mathsf{conf}} \leftarrow S^{\mathsf{TX}}_{\mathsf{conf}} \cup \tau$, $S^{\mathsf{TX}}_{\mathsf{conf},i} \leftarrow S^{\mathsf{TX}}_{\mathsf{conf},i} \cup \tau$

Acknowledge ST: Upon input $(\mathsf{ACK_ST}, sid, T, R, p_i)$, $p_i \in \mathcal{H}$ from $\mathcal{S}$, do

1: Send $(\mathsf{EVAL}, \Sigma_0^\gamma, L^{\mathsf{ST}}_{\mathsf{conf}} || T)$ to $\mathcal{G}^{\delta_w,\delta_r,\Lambda,\Sigma_0}_{\mathsf{Constrained\text{-}Ledger}}$ receiving b
2: Assert $((\gamma_S = \textsc{Open} \vee t \leq t_e + \delta_r) \wedge \mathsf{b} \wedge T \subseteq S^{\mathsf{TX}}_{\mathsf{conf}} \wedge L^{\mathsf{ST}}_{\mathsf{seen},i} = \epsilon \wedge (T \cap (S^{\mathsf{TX}}_{\mathsf{Reject},i} \cup R)) = \varnothing)$
3: $L^{\mathsf{ST}}_{\mathsf{seen},i} \leftarrow T$; $L^{\mathsf{TX}}_{\mathsf{seen,i}} \leftarrow L^{\mathsf{TX}}_{\mathsf{seen,i}} \setminus (T \cup R)$, $S^{\mathsf{TX}}_{\mathsf{Reject},i} \leftarrow S^{\mathsf{TX}}_{\mathsf{Reject},i} \cup R$
4: **if** $\forall p_i \in \mathcal{H} : L^{\mathsf{ST}}_{\mathsf{seen,i}} = T$ **then** $L^{\mathsf{ST}}_{\mathsf{pending}} \leftarrow L^{\mathsf{ST}}_{\mathsf{seen,i}}$; $S^{\mathsf{TX}}_{\mathsf{reject,pending}} \leftarrow S^{\mathsf{TX}}_{\mathsf{reject,pending}} \cup R$

Confirm ST: Upon input $(\mathsf{CONF_ST}, sid, p_i)$, $p_i \in \mathcal{H}$ from $\mathcal{S}$, do

1: **for** $\tau \in (S^{\mathsf{TX}}_{\mathsf{Reject},i} \cup L^{\mathsf{ST}}_{\mathsf{seen},i})$ **do** Remove τ from $L^{\mathsf{TX}}_{\mathsf{seen},i}$
2: Assert $((\gamma_S = \textsc{Open} \vee t \leq t_e + \delta_r) \wedge L^{\mathsf{ST}}_{\mathsf{seen},i} \neq \epsilon \wedge (L^{\mathsf{ST}}_{\mathsf{seen},i} = L^{\mathsf{ST}}_{\mathsf{pending}} \vee \exists L_{\mathsf{prefix}} : L^{\mathsf{ST}}_{\mathsf{conf}} = L_{\mathsf{prefix}} || L^{\mathsf{ST}}_{\mathsf{seen},i}))$
3: **if** $L^{\mathsf{ST}}_{\mathsf{pending}} = L^{\mathsf{ST}}_{\mathsf{seen},i}$ **then**
4: $\quad L^{\mathsf{ST}}_{\mathsf{conf}} \leftarrow L^{\mathsf{ST}}_{\mathsf{conf}} || L^{\mathsf{ST}}_{\mathsf{pending}}$; $L^{\mathsf{ST}}_{\mathsf{conf},i} \leftarrow L^{\mathsf{ST}}_{\mathsf{conf}}$; $S^{\mathsf{TX}}_{\mathsf{Reject}} \leftarrow S^{\mathsf{TX}}_{\mathsf{Reject}} \cup S^{\mathsf{TX}}_{\mathsf{Reject,pending}}$; $L^{\mathsf{ST}}_{\mathsf{pending}} \leftarrow \epsilon$; $S^{\mathsf{TX}}_{\mathsf{conf}} \leftarrow S^{\mathsf{TX}}_{\mathsf{conf}} \setminus (S^{\mathsf{TX}}_{\mathsf{Reject},i} \cup L^{\mathsf{ST}}_{\mathsf{seen},i})$; $S^{\mathsf{TX}}_{\mathsf{conf},i} \leftarrow S^{\mathsf{TX}}_{\mathsf{conf},i} \setminus (S^{\mathsf{TX}}_{\mathsf{Reject},i} \cup L^{\mathsf{ST}}_{\mathsf{seen},i})$
5: $L^{\mathsf{ST}}_{\mathsf{seen},i} \leftarrow \epsilon$

Fig. 8. The functionality's interface, and backdoor for entity confirmation. For either transactions (TX) and states (ST), the design is based in separating the "acknowledge" and "confirm" cases, resulting in four interfaces.

Lifecycle of a Transaction. To aid the intuition of transaction confirmations and updates to the channel state as outlined above we consider the lifecycle of a transaction in the following. When the channel is open, at any point in time a party can issue a transaction by sending a message of form $(\mathsf{NEW}, sid, \mathsf{param})$ to $\mathcal{F}_{CH}$ as outlined in Figure 8 under *instructions* which is forwarded to $\mathcal{S}$. Each transaction proceeds through a three round confirmation protocol. Note that $\mathcal{F}_{CH}$ does not place any assumptions on the protocol simulated by $\mathcal{S}$, however, if we take the Hydra protocol as an example, $\mathcal{S}$ proceeds to have parties to sign transactions and broadcast the signatures among the parties. If a transaction is signed by a party, $\mathcal{S}$ notifies $\mathcal{F}_{CH}$ by sending the message $(\mathsf{ACK_TX}, sid, \dots)$ and $\mathcal{F}_{CH}$ verifies that the transaction does not conflict with the signed party's view of the order of transactions. Note that all parties track the order of transactions in the order they arrive, thus these views can differ. Once a party observes a valid signature from all parties, for a given transaction, $\mathcal{S}$ notifies $\mathcal{F}_{CH}$ by sending the message $(\mathsf{CONF_TX}, sid, \dots)$ confirming the transaction.

Lifecycle of the Channel State. $\mathcal{F}_{CH}$ keeps two ordered transaction lists $L^{\mathsf{ST}}_{\mathsf{conf}}$ and $L^{\mathsf{ST}}_{\mathsf{pending}}$ which form the *channel state*. $L^{\mathsf{ST}}_{\mathsf{conf}}$ represents the confirmed channel state, while $L^{\mathsf{ST}}_{\mathsf{pending}}$ the channel state that was acknowledged by at least all honest parties. As with transactions, the state maintained by the channel is modified through a three round confirmation protocol.

4.6 Interpreting Transactions as State Machine Transitions

The channel itself is operated through a program, a *state machine*, executed on the ledger through submission of transactions.

The procedure OnTx (*i.e.*, Figure 7) places assertions on how the state machine operates and, thereby, it implicitly constrains which transactions can be created through ζ. This procedure is executed whenever a honest party sends (READ) to $\mathcal{G}^{\delta_w,\delta_r,\Lambda,\Sigma_0}_{\mathsf{Constrained\text{-}Ledger}}$ (Figure 9), upon which the functionality processes the received transactions one at a time. The function θ takes a transaction as input and outputs an ordered list of state machine transitions. This design allows to capture a wider range of protocols by selecting ζ and θ appropriately: Any transaction can execute one state transition of the underlying state machine, or even none or multiples. Yet, in this design, transactions from corrupted parties are not necessarily submitted through $\mathcal{F}_{CH}$, but by $\mathcal{Z}$ even through separate protocol sessions or arbitrary protocols.

By design, $\mathcal{F}_{CH}$ *guarantees* the following by checking $\mathsf{Assert}\,(\dots)$ instructions when processing a FANOUT transaction: When the channel is closed, the state of $\mathcal{G}^{\delta_w,\delta_r,\Lambda,\Sigma_0}_{\mathsf{Constrained\text{-}Ledger}}$ is changed corresponding to (1) closing the channel and (2) either submitting $L^{\mathsf{ST}}_{\mathsf{conf}}$ or $L^{\mathsf{ST}}_{\mathsf{conf}}||L^{\mathsf{ST}}_{\mathsf{pending}}$, which is the concatenation of both lists, to $\mathcal{G}^{\delta_w,\delta_r,\Lambda,\Sigma_0}_{\mathsf{Constrained\text{-}Ledger}}$. This is, because the adversary can choose whether to have the remaining, corrupted parties to acknowledge $L^{\mathsf{ST}}_{\mathsf{pending}}$ thus confirming it or refrain from doing so. Note that this does not imply that any transaction in $L^{\mathsf{ST}}_{\mathsf{conf}}$ or $L^{\mathsf{ST}}_{\mathsf{pending}}$ are submitted to the ledger directly. In fact most channel constructions submit only an amount of transactions constant in the size of transactions in the channel's state, *i.e.*, $|L^{\mathsf{ST}}_{\mathsf{conf}}||L^{\mathsf{ST}}_{\mathsf{pending}}|$, which instead sum up the channel state.

4.7 Design Discussion and Generality of $\mathcal{F}_{CH}$

From now, we discuss how far $\mathcal{F}_{CH}$ satisfies our desired properties as laid out in Section 4.1, its generality and the adherence with a well defined state machine.

The functionality states. Instead of storing explicit state, *e.g.*, a coin balance, it stores the transactions that were submitted to open the channel, as well as all transaction submitted to the channel which affect its state. Functionality $\mathcal{F}_{CH}$ is *secure* by guaranteeing that the channel can only be closed in one of two states, either in $L^{\mathsf{ST}}_{\mathsf{conf}}$ or $L^{\mathsf{ST}}_{\mathsf{conf}}||L^{\mathsf{ST}}_{\mathsf{pending}}$ which are the history of submitted transactions and the channel state. Moreover, through use of Transaction and Transition Functions, respectively, ζ and θ, $\mathcal{F}_{CH}$ is ledger implementation agnostic. Moreover, $\mathcal{F}_{CH}$ does not restrict which transactions are submitted by $\mathcal{S}$ keeping it protocol agnostic. Lastly, we argue about the functionality's generality, meaning that $\mathcal{F}_{CH}$ can be implemented by a variety of channel protocols.

Generality of $\mathcal{F}_{CH}$. The Functionality $\mathcal{F}_{CH}$ is designed with leeway to allow other channel implementations to realize $\mathcal{F}_{CH}$ for the cost of additional design complexity. For one, channels do not strictly require to adhere to the lifecycle of transaction and channel state update generation as laid out in Figure 8. We note

$\mathcal{F}^{SM}_{CH}$ Functionality (Interpreting transactions)

Procedure **OnTx(τ, p_i):**

1: Set inputs (about the functionality) for the state machine representation
   $\mathsf{Fdata} \leftarrow (p_i, \mathsf{param}, S_\sigma, t_c, \Sigma^\gamma_0, \Sigma_i, S^{\mathsf{TX}}_{\mathsf{open,i}})$ ▷ Functional. data
2: Set bit for condition to safe state machine abortion
   // Is there a chance of wrong abort condition, i.e., different state?
   $L_{\mathsf{INV}} \leftarrow$ an arbitrary permutation of $\{\Lambda.INV(\tau) \mid \tau \in S^{\mathsf{TX}}_{\mathsf{open,i}}\}$
   $\mathsf{b}_{AbortCond} \leftarrow \mathbf{S_EQ}(\Omega_i||L_{\mathsf{INV}}, \Omega_i||\tau)$
3: Send $(\mathsf{TIMESTAMP}, \tau)$ to $\mathcal{G}^{\delta_w,\delta_r,\Lambda,\Sigma_0}_{\mathsf{Constrained\text{-}Ledger}}$ receiving t
4: **if** $\exists x, \mathsf{param}$: $\tau = \zeta(x, p_i, \mathsf{param}, S_\sigma)$ **then**
5: **for** $\mathsf{tr} \in \theta(\tau)$ **do** ▷ Process state transitions τ triggers 1-by-1
      // Single transition in the state machine for tr
6: Let $\mathsf{output} \leftarrow SM(\tau, \mathsf{tr}, \mathsf{Fdata})$ ▷ Transitioning SM
7: **HandleTransitions**(output)
8: $\Omega_i \leftarrow \Omega_i||\tau$

Procedure **S_EQ(Ω_1, Ω_2):**

1: **if** $\mathcal{G}^{\delta_w,\delta_r,\Lambda,\Sigma_0}_{\mathsf{Constrained\text{-}Ledger}}.\Lambda.\mathsf{STATE}(\Sigma_0, \Omega_1) = \mathcal{G}^{\delta_w,\delta_r,\Lambda,\Sigma_0}_{\mathsf{Constrained\text{-}Ledger}}.\Lambda.\mathsf{STATE}(\Sigma_0, \Omega_2)$ **then return** $\top$. Otherwise, **return** $\bot$

Fig. 9. The **OnTX Procedure** captures, via the internal "for" loop, all the state transitions in the $ProtoSM$, the state machine representation of the protocol, in accordance with each transaction $\mathsf{tr} \in \theta(\tau)$ provided by p_i.

that while new transactions and channel state updates are issued through the protocol interface accessed by $\mathcal{Z}$, the remainder of the transaction and channel state lifecycle is controlled by $\mathcal{S}$. Thus, we abstract away from concrete protocol implementations by delegating these details to each simulator $\mathcal{S}$ in the security proof of the individual protocol. Moreover, we note that any instruction to create a new transaction or channel state update from $\mathcal{Z}$ to $\mathcal{F}_{CH}$ is directly forwarded to $\mathcal{S}$ without performing any state change in $\mathcal{F}_{CH}$. This implies that $\mathcal{S}$ can create channel state updates that were not issued by $\mathcal{Z}$. Thus, protocols that only confirm transaction one-by-one without a designated channel state update

$\mathcal{F}_{CH}^{SM}$ Functionality (Handling State Transitions)

Procedure **HandleTransitions(**output**):**

Parse $(\mathsf{msgs}, \mathsf{newFST}, \mathsf{newST}, \mathsf{b}, \mathsf{b}_{Open}, \mathsf{b}_{Abt}, \mathsf{b}_{Dpt}) \leftarrow \mathsf{output}$
Assert (b) ▷ $\mathcal{F}_{CH}$ fails on invalid state transition in case of $\mathsf{b} = \bot$
if $\gamma_S = \text{NotYetOpen} \wedge (\mathsf{b}_{Abt} = \bot) \wedge \mathsf{b}_{Open}$ **then**
    $\gamma_S \leftarrow \text{Open}$
    Assert $(\Sigma_0^\gamma = \bigcup_{i=1}^n \Sigma_i)$ ▷ *All parties have commitment in the initial channel state?*
    // In state machine abort, necessary to check conditions
if $\gamma_S = \text{NotYetOpen} \wedge \mathsf{b}_{Abt}$ **then**
    Assert $(\mathsf{b}_{AbortCond})$ and $\gamma_S \leftarrow \text{End}$
if $\gamma_S = \text{Open} \wedge \mathsf{b}_{Dpt} = \top$ **then**
    if $(t_e < 0)$ **then** $t_e \leftarrow t$ ▷ *Set dispute start time, if not yet.*
    $\gamma_S \leftarrow \text{NotYetEnd}$
    Assert $(t \leq t_e + \Delta_{Close})$ ▷ *Hardwired CLOSE time bound.*
if $\gamma_S = \text{NotYetEnd}$ **then** $\gamma_S \leftarrow \text{End}$ ▷ *Only way valid tr.*
    Assert $(t \geq t_e + t_c \wedge t \leq t_e + t_c + \Delta_{End})$ ▷ *Hardwired END time bound.*
    //Does τ sum up the channel state?
    Assert $(\mathbf{S_EQ}(\Omega_i || L_{\mathsf{conf}}^{\mathsf{ST}}, \Omega_i || \tau) \vee \mathbf{S_EQ}(\Omega_i || L_{\mathsf{conf}}^{\mathsf{ST}} || L_{\mathsf{pending}}^{\mathsf{ST}}, \Omega_i || \tau))$
Set new states for functionality and party view
    $\Sigma_0^\gamma \leftarrow \mathsf{newFST}$ and $S_{\mathsf{open,i}}^{\mathsf{TX}} \leftarrow \mathsf{newST}$
for $\mathsf{msg} \in \mathsf{msgs}$ **do** ▷ Some state transitions result in messaging
    Parse $(\mathsf{type}, \mathsf{attr}, \mathsf{receiver}) \leftarrow \mathsf{msg}$ ▷ *Not sanitized msg's.*
    if $\mathsf{attr} = \cdot$ **then** send (type, sid) to $\mathsf{receiver}$
    else send $(\mathsf{type}, sid, \mathsf{attr})$ to $\mathsf{receiver}$

Fig. 10. The procedure handles the $\mathcal{F}_{CH}^{SM}$ internal states, which are {NotYetOpen, Open, NotYetEnd, End} and the transitions are defined by the concrete state machine output bits. It also relies on the hardwired time limits Δ_{Close} and Δ_{End} for, respectively, to close and end the channel. They can be also specified particularly for each protocol. Finally, the functionality deliver message for the parties. For readability, here we simplified and the messages are not sanitized (*i.e.*, there is no checking for consistency with respect to the internal state γ_S of $\mathcal{F}_{CH}$), but it can be easily added.

phase can be implemented as follows: Whenever a transaction τ is confirmed, $\mathcal{S}$ afterwards creates a new channel state update that only contains τ as a new transaction and notifies $\mathcal{F}_{CH}$ that it was acknowledged and confirmed. While $\mathcal{F}_{CH}$ uses the latest channel state to verify whether the channel was closed correctly, there does not need to be a transaction that submits the channel state to the ledger, thus no signature for the channel state itself is required.

Account based Ledgers. We observe that in general, channel closing in the Account based model might not result in a predictable state as the effect of transactions is not known beforehand. This is due to not knowing the result of a transaction interacting with a smart contract before the respective transaction interacted with it. Thus, it might not be possible to close the $\mathcal{F}_{CH}$ without triggering a failed assertion, *i.e.*, Figure 10/ Line 5. However, this issue can be averted by limiting closure of $\mathcal{F}_{CH}$ to transferring coins. In fact, to our knowledge all account based model Layer 2 protocols adhere to this limitation [11, 12, 13, 25].

State machine adherence. Channels do not strictly require to adhere to the state machine included in $\mathcal{F}_{CH}$ and described in Figure 9. While $\mathcal{F}_{CH}$ requires progressing through its internal state machine, which does reflect the Hydra protocol's state machine, progression through that state machine is controlled through function θ (Line 3), provided by $\mathcal{S}$ during the setup of $\mathcal{F}_{CH}$ shown in Figure 6. Transactions that affect the channel can opt to not perform a state transition, or to perform multiple state transition with one and the same transaction. For instance, a simple payment channel might have two on-ledger transactions, one to fund it and lock coins of the channel participants requiring a multisignature to spend these coins, and a final closing transaction that takes the channel's coins and returns coins to the parties according to the channel's latest state. In that case, the funding transaction implicitly performs initialization of the channel, commitment of coins as well as collection of these coins, transitioning the channel straight into the open state. Moreover, in this example, the closing transaction moves the channel straight to the end state. However, the invariant that all channels have to adhere, to in order to implement $\mathcal{F}_{CH}$ and what is enforced through the state machine in Figure 9, is that parties collaborate to open the channel with an initial state, *e.g.*, an initial amount of coins and their distribution among the parties, and that the channel is eventually closed in a final state that is consistent with the transactions and confirmed channel state.

5 Security Proof of Π_{Hydra} with respect to $\mathcal{F}_{CH}$

In this section we analyze the security of the Hydra Protocol Π_{Hydra} [10] with respect to the previous functionality in the $(\mathcal{G}^{\delta_w,\delta_r,\Lambda,\Sigma_0}_{\mathsf{Constrained\text{-}Ledger}}, \mathcal{G}_{Ticker})$-Global Model. Namely we show that Π_{Hydra} UC-realizes the functionality $\mathcal{F}_{CH}$. For completeness, we review Π_{Hydra} in Appendix A.

5.1 The Setup of $\mathcal{F}_{CH}$

In order to provide our main security claim it is necessary to provide a concrete construction for the state machine verifier to be used as a parameter in $\mathcal{F}_{CH}$.

The concrete state machine verifier. The Hydra State Machine Verifier HydraSM is used in Figure 9 to capture the Hydra State Machine, as presented in Figure 3. It completes $\mathcal{F}_{CH}$ definition, since SM is set as HydraSM.

Hydra State Machine Verifier $\mathbf{HydraSM}(\tau, \mathsf{tr}, \mathsf{Fdata})$

2: Set $\mathsf{b} \leftarrow \bot$ ▷ Set to $\top$ if τ performs any valid state transition
3: Set $\mathsf{b}_{Abt} \leftarrow \bot$ ▷ Set to $\top$ if the valid transition was for abort
4: Set $\mathsf{b}_{Dpt} \leftarrow \bot$ ▷ Set to $\top$ if in dispute
5: Set $\mathsf{b}_{Open} \leftarrow \bot$ ▷ Set to $\top$ if channel is open
6: Define $\mathsf{newFST} = \mathsf{newST} = \cdot$ ▷ Initially empty variables
7: Define $\mathsf{FState} = \mathsf{SETUP}$ ▷ Internal State initially in 'Setup'
8: Define empty list msgs ▷ To allow correct messaging later
9: Parse Fdata as $(\mathsf{party}, \mathsf{param}, S_\sigma, t_c, \mathsf{ChInitState}, \mathsf{OpenCommits}, \mathsf{OpenTXs})$
// From here, it looks for the correct state transition according to tr
10: **if** $\mathbf{SM}(\mathsf{SETUP}, \mathsf{INIT}, \mathsf{INIT}, \mathsf{tr}) = \top$ **then** $b \leftarrow \top$
11: **if** $\mathbf{SM}(\mathsf{INIT}, \mathsf{COMM}, \mathsf{INIT}, \mathsf{tr}) = \top \wedge \tau = \zeta_H(\mathsf{COMM}, \mathsf{party}, \mathsf{param}, S_\sigma)$ **then**
12: Assert ($\Sigma_i \notin \Sigma_0^\gamma$, for $\Sigma_i \in \mathsf{param}$) ▷ *Is i-th party's only commitment?*
13: $\mathsf{b} \leftarrow \top$, $\mathsf{newST} \leftarrow \mathsf{OpenTXs} \cup \{\tau\}$,
14: $\mathsf{newFST} \leftarrow \mathsf{ChInitState} \cup \mathsf{OpenCommits}$
15: **if** $\mathbf{SM}(\mathsf{INIT}, \mathsf{COLL}, \mathsf{OPEN}, \mathsf{tr}) = \top$ **then**
16: Set $\mathsf{b} \leftarrow \top$ and $\mathsf{b}_{Open} \leftarrow \top$, and
17: add $(\mathsf{OPEN}, \mathsf{ChInitState}, \mathsf{party})$ to msgs
18: **if** $\mathbf{SM}(\mathsf{OPEN}, \mathsf{CLOSE}, \mathsf{DISPUTE}, \mathsf{tr}) = \top$ **then**
19: Set $\mathsf{b} \leftarrow \top$, $\mathsf{b}_{Dpt} \leftarrow \top$ and $\mathsf{b}_{Open} \leftarrow \bot$,
20: add $(\mathsf{CLOSING}, \cdot, \mathsf{party})$ to msgs
21: **if** $\mathbf{SM}(\mathsf{DISPUTE}, \mathsf{CONTEST}, \mathsf{DISPUTE}, \mathsf{tr}) = \top \wedge t \leq t_e + t_c$ **then** $\mathsf{b} \leftarrow \top$
22: **if** $\mathbf{SM}(\mathsf{DISPUTE}, \mathsf{FANOUT}, \mathsf{END}, \mathsf{tr}) = \top \wedge t > t_e + t_c$ **then**
23: Set $b \leftarrow \top$, $\mathsf{b}_{Dpt} \leftarrow \bot$ and $\mathsf{b}_{Open} \leftarrow \bot$,
24: add $(\mathsf{CLOSED}, \cdot, \mathsf{party})$ to msgs
25: **if** $\mathbf{SM}(\mathsf{INIT}, \mathsf{ABORT}, \mathsf{END}, \mathsf{tr}) = \top$ **then**
26: Set $\mathsf{b} \leftarrow \top$ and $\mathsf{b}_{Abt} \leftarrow \top$ ▷ *Channel not even opened.*
27: Add $(\mathsf{CLOSED}, \cdot, \mathsf{party})$ to msgs
// Report on the state transition, including new state, messages, etc
28: Return $(\mathsf{msgs}, \mathsf{newFST}, \mathsf{newST}, \mathsf{b}, \mathsf{b}_{Open}, \mathsf{b}_{Abt}, \mathsf{b}_{Dpt})$
// Verifies whether such transition exists

Procedure $\mathbf{SM}(\mathsf{state}, \mathsf{edge}, \mathsf{next_state}, \mathsf{tr})$: If $\mathsf{FState} = \mathsf{state} \wedge \mathsf{tr} = \mathsf{edge}$ then $\mathsf{FState} \leftarrow \mathsf{next_state}$, and return $\top$. Otherwise, return $\bot$.

Fig. 11. HydraSM checks the adherence to the representation in Figure 3, by relying on the state machine transaction function ζ_H for Π_{Hydra}.

5.2 The Main Security Claim

In the following sections we prove Hydra Protocol is secure with respect to our proposed $\mathcal{F}_{CH}$ functionality when instantiated with HydraSM. That is equivalent to say that Π_{Hydra} *UC-realizes* $\mathcal{F}_{CH}^{HydraSM}$ *in the* $(\mathcal{G}_{\mathsf{Constrained\text{-}Ledger}}^{\delta_w,\delta_r,\Lambda,\Sigma_0},\mathcal{G}_{Ticker})$*-Global Model*. Our security analysis relies on the following claim regarding Π_{Hydra}.

Claim. Let Π_{Hydra} be the Hydra Protocol as in [10], and consider the state machine presented in Figure 3, therefore HydraSM does not return $\perp$ for transactions from Π_{Hydra}.

We prove this claim later. In the meanwhile we state our main claim.

Theorem 1. *Let HydraSM be the state machine verifier for* Π_{Hydra}*, the function* ζ_H *to implement* Π_{Hydra}*. Given the earlier Claim, and assuming the EU-CMA security of the signature scheme, then* Π_{Hydra} *UC-realizes* $\mathcal{F}_{CH}^{HydraSM}$ *in the* $(\mathcal{G}_{\mathsf{Constrained\text{-}Ledger}}^{\delta_w,\delta_r,\Lambda,\Sigma_0},\mathcal{G}_{Ticker})$*-Global Model.*

The security proof is composed by three auxiliary lemmas (Lemmas 1, 2 and 3) with respect to the construction for the yet to be presented simulator $\mathcal{S}_{\mathsf{H}}$ (as given by the UC Framework outlined in Section 2.3). Therefore, in the following sections, we introduce the considered assumptions over the environment $\mathcal{Z}$, the concrete construction of $\mathcal{S}_{\mathsf{H}}$, the formal presentation of Lemmas 1, 2 and 3, and, finally, the complete proof for early presented theorem.

Assumptions. We have three assumptions that have to hold for the theorem to hold true: (1) We require that the underlying signature scheme used by $\mathcal{G}_{\mathsf{Constrained\text{-}Ledger}}^{\delta_w,\delta_r,\Lambda,\Sigma_0}$ is EU-CMA secure to be able to bound the probability for $\mathcal{Z}$ to forge a signature to be negligible. (2) We require that $\mathcal{G}_{\mathsf{Constrained\text{-}Ledger}}^{\delta_w,\delta_r,\Lambda,\Sigma_0}$ *liveness* assumption is not violated as shown in Figure 4, i.e. $\mathcal{A}$ must extend the ledger and advance each party's view of the ledger within the time bounds dictated by the liveness property. Otherwise, if *write liveness* does not hold, i.e. if extending the ledger with a transaction happens more than δ_w after the transaction is submitted, then $\mathcal{Z}$ can circumvent constraints that are implemented through timelocks of the channel's state machine dictated by ζ. If *read liveness* does not hold, i.e. a party's view of the ledger does not contain transactions that were included in the ledger time δ_r ago, then it will prevent the party to react to changes of the on-chain state machine in time, e.g. the party might not be able to contest closing the channel in an outdated state. (3) Lastly, we require that the ledger's state transition function Λ is expressive enough such that there exists Π_{Hydra} for $\mathcal{G}_{\mathsf{Constrained\text{-}Ledger}}^{\delta_w,\delta_r,\Lambda,\Sigma_0}$ utilizing Λ, i.e. we must be able to implement the Hydra on-chain state machine [10] for the given ledger. The Hydra state machine was defined for the EUTxO model. If Λ is Turing complete, then an implementation of the Hydra on-chain state machine must exist, even if considering other ledger paradigms, e.g. the account model. However, if Λ is not Turing complete, e.g. Bitcoin's scripting language, then we might not be able to implement Hydra's on-chain state machine such there does not exist Π_{Hydra} and $\mathcal{S}_{\mathsf{H}}$ could neither provide Π_{Hydra} to $\mathcal{F}_{CH}$ nor simulate Π_{Hydra} for $\mathcal{A}$.

The Simulator $\mathcal{S}_{\mathsf{H}}$ (Figures 12 and 13) receives inputs from $\mathcal{F}_{CH}$ and internally simulates the protocol Π_{Hydra} for $\mathcal{A}$. Of note is that $\mathcal{S}_{\mathsf{H}}$ exclusively interacts with $\mathcal{F}_{CH}$ and does not send messages directly to $\mathcal{G}^{\delta_w,\delta_r,\Lambda,\Sigma_0}_{\mathsf{Constrained\text{-}Ledger}}$ or $\mathcal{G}_{Ticker}$. In a nutshell, $\mathcal{S}_{\mathsf{H}}$ does the following:

1. internally simulates the execution of the Π_{Hydra} Protocol for $\mathcal{A}$;
2. forwards events occurring in the simulation, *e.g.*, a honest party acknowledges a transaction, to $\mathcal{F}_{CH}$. As the timing of these events depends on the communication delays incurred by $\mathcal{A}$, doing so ensures that these events will occur with the correct timing in the ideal world; and
3. acts as an intermediary between corrupted parties, and $\mathcal{F}_{CH}$ and the ledger $\mathcal{G}^{\delta_w,\delta_r,\Lambda,\Sigma_0}_{\mathsf{Constrained\text{-}Ledger}}$ by forwarding transactions and requests to read the ledger from corrupted parties, and returning any results.

Simulator $\mathcal{S}_{\mathsf{H}}$ (Part 1): Variables and party setup

Variables:
- Indexed parties p_i, $i \in \mathbb{N}$ and $i \leq n$. ▷ Index for a n-party protocol.
- Sets of channel participants $\gamma_p = \varnothing$, corrupted and honest parties, respectively, $\mathcal{C} = \varnothing$ and $\mathcal{H} = \{p_1, \ldots, p_n\}$.

Behavior: At start, initialize simulated parties sunning Π_{Hydra}. Each party performs Π_{Hydra} as in Fig. 20, 21, 22, 23 and 24 (Appendix A).

Corruption: Upon receiving $(\mathsf{CORRUPT}, sid, \mathcal{C}')$ from $\mathcal{A}$, at the start, do

1: $\mathcal{C} \leftarrow \mathcal{C}'$ and $\mathcal{H} \leftarrow \mathcal{H} \setminus \mathcal{C}'$
2: **for** $p \in \mathcal{C}$ **do**
3: Deliver the p's internal state to $\mathcal{A}$
4: From this point, any input message from $\mathcal{A}$ controlling $p \in \mathcal{C}$, simulate p accordingly

Party Setup: On message $(\mathsf{PARTY_SETUP}, sid, p, \gamma'_p, t_c, \Sigma_i, i)$ from $\mathcal{F}_{CH}$

1: $p_i \leftarrow p$
2: **if** $\gamma_p = \varnothing$ **then** $\gamma_p \leftarrow \gamma'_p$
3: **if** $p_i \notin \mathcal{C}$ **then**
4: Internally execute p_i with input $(\mathsf{SETUP}, sid, \gamma_p, t_c, \Sigma_i, i)$ ▷ Fig. 21.
5: Share with $\mathcal{A}$ all the messages ReqSetup, AckSetup, between all p_i and p, and the SUBMIT ones, from p_i to the ledger ▷ Figures 21 and 20.
6: Leak $p_i.\mathsf{vk}$ to $\mathcal{A}$ ▷ $\mathcal{A}$ receives honest verification keys.

Send Transaction: Upon simulated party p sending $(\mathsf{SUBMIT}, sid, \tau)$ to $\mathcal{G}^{\delta_w,\delta_r,\Lambda,\Sigma_0}_{\mathsf{Constrained\text{-}Ledger}}$, send $(\mathsf{S_TX}, sid, \tau)$ to $\mathcal{F}_{CH}$. ▷ Figure 20.

Party Activation: Upon simulated $p \in \mathcal{H}$ sending $(\mathsf{ACTIVATED})$ to $\mathcal{G}_{Ticker}$, send $(\mathsf{ACTIVATED}, sid, p)$ to $\mathcal{F}_{CH}$.

Fig. 12. How the simulator $\mathcal{S}_{\mathsf{H}}$ handles messages from $\mathcal{F}_{CH}$. This part presents how is the general behavior and internal variables. Moreover, it details the initial setup while Part 2 next, details the channel setup.

Simulator $\mathcal{S}_{\mathsf{H}}$ (Part 2)

Functionality Setup: Upon message $(\mathsf{FUNC_SETUP}, sid)$ from $\mathcal{F}_{CH}$, then

1: **if** $\gamma_p = \mathcal{C}$ **then**
2: Take control of $\mathcal{F}_{CH}$. Make it behave normally but prevent it from sending any messages except to $\mathcal{S}_{\mathsf{H}}$, $\mathcal{G}^{\delta_w,\delta_r,\Lambda,\Sigma_0}_{\mathsf{Constrained\text{-}Ledger}}$ and $\mathcal{G}_{Ticker}$, *i.e.*, $\mathcal{F}_{CH}$ does not send messages to $\mathcal{Z}$, or any party, and only forwards messages between $\mathcal{S}_{\mathsf{H}}$ and the global functionalities
3: Select ζ_H such that it implements the Hydra on-chain state machine [10] on $\mathcal{G}^{\delta_w,\delta_r,\Lambda,\Sigma_0}_{\mathsf{Constrained\text{-}Ledger}}$
4: Select θ such that for a transaction $\tau = \zeta_H(x, \dots)$, $\theta(\tau) = \{x\}$ holds
5: $\delta \leftarrow \delta_r + \delta_w$
6: **return** $(\zeta, \theta, \delta, \mathcal{C})$ to $\mathcal{F}_{CH}$

Read Ledger: Upon simulated $p \in \mathcal{H}$ sending (READ, sid) to $\mathcal{G}^{\delta_w,\delta_r,\Lambda,\Sigma_0}_{\mathsf{Constrained\text{-}Ledger}}$, do

1: Send (READ, sid, p) to $\mathcal{F}_{CH}$ receiving output Ω
2: Simulate p receiving Ω as output

Abort: Upon receiving (ABORT, sid, p) from $\mathcal{F}_{CH}$,

1: **if** $p_i \in (\gamma_p \setminus \mathcal{C})$ **then** simulate p executing $\Pi_{Hydra}.\mathsf{Abort}$

Fanout: Upon receiving $(\mathsf{FANOUT}, sid, p)$ from $\mathcal{F}_{CH}$,

1: **if** $p_i \in (\gamma_p \setminus \mathcal{C})$ **then** simulate p executing $\Pi_{Hydra}.\mathsf{Fanout}$

New Transaction: Upon receiving $(\mathsf{ABORT}, sid, p, \tau)$ from $\mathcal{F}_{CH}$,

1: **if** $p_i \in (\gamma_p \setminus \mathcal{C})$ **then** simulate p executing $\Pi_{Hydra}.\mathsf{New}(\tau)$

Submit State: Upon receiving $(\mathsf{NEW_ST}, sid, p, \mathsf{param})$ from $\mathcal{F}_{CH}$,

1: **if** $T \notin \mathsf{param} \vee R \notin \mathsf{param}$ **then return**
2: **if** $p_i \in (\gamma_p \setminus \mathcal{C})$ **then**
3: simulate p executing $\Pi_{Hydra}.\mathsf{NewSt}(\mathsf{param.T}, \mathsf{param.R})$

Close: Upon receiving the message (CLOSE, sid, p) from $\mathcal{F}_{CH}$,

1: **if** $p_i \in (\gamma_p \setminus \mathcal{C})$ **then** simulate p executing $\Pi_{Hydra}.\mathsf{CLOSE}$

Fig. 13. How the simulator $\mathcal{S}_{\mathsf{H}}$ handles messages from $\mathcal{F}_{CH}$. In particular, it offers the interface for the channel setup relying on the Hydra State Machine Transaction Function ζ_H.

The auxiliary lemmas. Consider that there exists a function ζ_H (as described in Section 4) such that it implements the protocol Π_{Hydra} state machine. Such ζ_H must be implemented in a way that no assertion will be violated when $\mathcal{F}_{CH}$ reads the content of the ledger with procedure OnTx, shown in Figure 9. Moreover, assume that at least one party is honest as in the case where all parties are malicious, $\mathcal{S}_\mathsf{H}$ takes over $\mathcal{F}_{CH}$ which allows it to simulate Π_{Hydra} trivially.

To create a proof for Theorem 1, we must show that the executions of an ideal world with simulator $\mathcal{S}_\mathsf{H}$ and functionality $\mathcal{F}_{CH}$ is indistinguishable to an execution of real protocol Π_{Hydra} for the environment $\mathcal{Z}$. We observe that $\mathcal{Z}$ receives messages, thus it can potentially distinguish between functionality and protocol, from three sources: (1) The adversary $\mathcal{A}$'s view; (2) the global functionalities $\mathcal{G}_{Ticker}$ and $\mathcal{G}^{\delta_w,\delta_r,\Lambda,\Sigma_0}_{\mathsf{Constrained\text{-}Ledger}}$; and (3) the outputs from the functionality $\mathcal{F}_{CH}$ and the protocol Π_{Hydra}. Our proof strategy is to analyze each of these sources and argue for indistinguishability of messages originating from them. Thus, we have the following three lemmas.

Lemma 1. *Let ζ_H implement Π_{Hydra} such that HydraSM implements the state machine given in Figure 3. Assuming the underlying signature scheme is EU-CMA, then $\mathcal{S}_\mathsf{H}$ simulates Π_{Hydra} such that $\mathcal{Z}$ can distinguish through the view of $\mathcal{A}$ with at most negligible probability.*

Proof. The simulator $\mathcal{S}_\mathsf{H}$ provides a view of Π_{Hydra} execution, by running as as defined in Figure 12 and 13, to create a view of Π_{Hydra} for $\mathcal{A}$. We stress that $\mathcal{A}$ cannot witness messages between $\mathcal{S}_\mathsf{H}$ and $\mathcal{F}_{CH}^{\text{HydraSM}}$ but only messages between the simulated parties offered by $\mathcal{S}_\mathsf{H}$. The simulated parties send messages indistinguishably to parties in the ideal world, if they

1. are initialized correctly;
2. receive the same inputs and messages;
3. execute the same protocol;
4. $\mathcal{A}$ has full control of corrupted parties and messaging in the simulation is subject to the delays imposed by $\mathcal{A}$; and
5. $\mathcal{A}$ reads the expected ledger state through corrupted parties reading the ledger.

Regarding the concrete protocol execution, we observe the only messaging $\mathcal{A}$ witnesses are the 3-round confirmation protocols for channel initialization, transaction- / and channel state update confirmation. Hence, it is sufficient to show that simulated parties behave the same as parties in ideal world for executing the 3-round confirmation protocols (as given by Hydra Protocol in Figures 23 and 24).

Thus, we argue that each of the early cited items hold with respect to $\mathcal{S}_\mathsf{H}$.

1. $\mathcal{F}_{CH}$ forwards all setup parameters to $\mathcal{S}_\mathsf{H}$ directly, as shown in Figure 5, via the simulator's interfaces **Party Setup** and **Functionality Setup**, and as $\mathcal{S}_\mathsf{H}$ simulates Π_{Hydra}, thus (1) follows;
2. We observe that $\mathcal{F}_{CH}^{\text{HydraSM}}$ at no point impedes messages between parties through simulator $\mathcal{S}_\mathsf{H}$ and the global functionalities. Also, $\mathcal{F}_{CH}^{\text{HydraSM}}$ does not create and submit any transactions to the ledger other than forwarding

transactions created by $\mathcal{S}_{\mathsf{H}}$. Moreover, $\mathcal{F}_{CH}^{\text{HydraSM}}$ also directly forwards all instructions from $\mathcal{Z}$, along with the party who is recipient of that message, and without altering or withholding contents in these messages such that it follows that (2) holds;

3. As $\mathcal{S}_{\mathsf{H}}$ simulates Π_{Hydra}, then, by the simulator construction, *i.e.*, Figures 12 and 13, it follows that (3) holds;
4. In case of corruption, $\mathcal{S}_{\mathsf{H}}$ reveals all internal state of simulated parties to $\mathcal{A}$ upon corruption, and $\mathcal{S}_{\mathsf{H}}$ only simulates that a party received a message after $\mathcal{A}$ has it delivered. Therefore, it follows that (4) holds; and
5. $\mathcal{F}_{CH}^{\text{HydraSM}}$ does not impede messages between parties (through simulator $\mathcal{S}_{\mathsf{H}}$) and the global functionalities. Neither does $\mathcal{F}_{CH}^{\text{HydraSM}}$ create any transactions. Then, $\mathcal{A}$ witnesses transactions upon reading $\mathcal{G}_{\mathsf{Constrained\text{-}Ledger}}^{\delta_w,\delta_r,\Lambda,\Sigma_0}$ that deviate from a correct execution of Π_{Hydra} only if $\mathcal{Z}$ forges a signature for a transaction that would not be signed by a honest party $p \in \mathcal{H}$. However, as the signatures scheme is EU-CMA (as in Definition 1), the probability of $\mathcal{Z}$ creating a valid signature is negligible. Thus (5) holds.

In summary, it follows that $\mathcal{S}_{\mathsf{H}}$ simulates the 3-round entity confirmation protocols and the required messaging for $\mathcal{A}$ subject to $\mathcal{A}$'s influence. Therefore $\mathcal{Z}$ distinguishes between simulation and execution in the $(\mathcal{G}_{\mathsf{Constrained\text{-}Ledger}}^{\delta_w,\delta_r,\Lambda,\Sigma_0},\mathcal{G}_{Ticker})$-Global Model with at most negligible probability (due to the security of the signature scheme). □

Lemma 2. *Let ζ_H implement Π_{Hydra} such that HydraSM implements the state machine given in Figure 3. Assuming the underlying signature scheme is EU-CMA, then $\mathcal{Z}$ can distinguish with at most negligible probability by receiving messages from $\mathcal{G}_{\mathsf{Constrained\text{-}Ledger}}^{\delta_w,\delta_r,\Lambda,\Sigma_0}$ and $\mathcal{G}_{Ticker}$.*

Proof. This can be proven with the same argumentation as for Lemma 1 as (1) $\mathcal{F}_{CH}^{\text{HydraSM}}$ does not impede communication between the simulated parties and the global functionalities; (2) $\mathcal{F}_{CH}^{\text{HydraSM}}$ does neither create new transactions for $\mathcal{G}_{\mathsf{Constrained\text{-}Ledger}}^{\delta_w,\delta_r,\Lambda,\Sigma_0}$ or send additional activation messages to $\mathcal{G}_{Ticker}$; and (3) $\mathcal{S}_{\mathsf{H}}$ faithfully simulates Π_{Hydra} as shown in Lemma 1. The messages sent to the ledger $\mathcal{G}_{\mathsf{Constrained\text{-}Ledger}}^{\delta_w,\delta_r,\Lambda,\Sigma_0}$ and $\mathcal{G}_{Ticker}$ and their timings are the same in the ideal world as it would be in the real protocol, with one exception: If $\mathcal{Z}$ can create a transaction that requires a signature of a honest party which it would not sign, but forges a signature to have this transaction be submitted successfully to $\mathcal{G}_{\mathsf{Constrained\text{-}Ledger}}^{\delta_w,\delta_r,\Lambda,\Sigma_0}$, then $\mathcal{Z}$ can observe different outputs from $\mathcal{G}_{\mathsf{Constrained\text{-}Ledger}}^{\delta_w,\delta_r,\Lambda,\Sigma_0}$. However, since the underlying signature scheme is EU-CMA, $\mathcal{Z}$ succeeds with at most negligible probability. □

Before we look into the outputs of the parties and their relation with the functionality $\mathcal{F}_{CH}^{\text{HydraSM}}$, we prove the earlier Claim regarding the protocol Π_{Hydra} and its established security with respect to its known state machine. For completeness we restate the earlier Claim.

Claim. Let Π_{Hydra} be the Hydra Protocol as in [10], and consider the state machine presented in Figure 3, therefore HydraSM does not return $\perp$ for transactions from Π_{Hydra}.

Proof. Consider the security of Π_{Hydra} as presented in [10] and the construction of HydraSM. Note that HydraSM an exhaustive cover of all possible transitions of the state machine introduced in [10] as given by Figure 3. Hence we conclude that, assuming Π_{Hydra} implements the state machine of Figure 3 (as proved in [10]), then, along with the constructions of $\mathcal{F}_{CH}^{\text{HydraSM}}$ and $\mathcal{S}_{\mathsf{H}}$, the procedure HydraSM does not output $\perp$ as HydraSM covers all possible transitions.

Note that our validation procedure HydraSM maps all the known state transitions for the Hydra state machine in its internal if clauses, guaranteeing that not invalid transition could be validated.

Lemma 3. *Let ζ_H implement Π_{Hydra} such that HydraSM implements the state machine given in Figure 3. Assuming the early Claim, and that the underlying signature scheme is EU-CMA, then $\mathcal{Z}$ distinguishes with at most negligible probability by observing the parties' outputs in the simulation provided by $\mathcal{S}_{\mathsf{H}}$.*

Proof. In order to concretely verify the parties outputs of the protocol simulation, it is necessary to thoroughly look into the success probability of the simulation with respect to $\mathcal{F}_{CH}^{\text{HydraSM}}$.

The success probability of $\mathcal{S}_{\mathsf{H}}$ is equal to the probability of the event that any $\mathsf{Assert}\ (\cdot)$ clause receives $\perp$ (with the exception of the one in Figure 6 later explained). This is because the $\mathcal{S}_{\mathsf{H}}$ construction performs the execution of the real Π_{Hydra}, *i.e.*, Figures 12 and 13, as Π_{Hydra} is described in Appendix A. In such situations, *i.e.*, $\mathsf{Assert}\ (\perp)$, the functionality $\mathcal{F}_{CH}^{\text{HydraSM}}$ halts, thus $\mathcal{Z}$ distinguishes from the protocol simulation offered by $\mathcal{S}_{\mathsf{H}}$. Hence it is crucial to look on each clause as they are defined in Figures 6, 7, 8, 9, and 10:

- Figure 6: This clause triggers only when an earlier event of $\mathsf{Assert}\ (\perp)$ already occurred, thus its probability does not interfere with the success of $\mathcal{S}_{\mathsf{H}}$;
- Figure 7: No assert clause exists;
- Figure 8: In general, the following assert verifications would only cause halt of the functionality, if the internal lists of the functionality are inconsistent. In other words, if the channel is not opened, if the some honest part is set to confirmed transactions/states without actually doing it. Both cases are considered negligible given that $\mathcal{S}_{\mathsf{H}}$ provides a correct simulation and an inconsistent state would indicate the adversary $\mathcal{A}$ could generate a forged signature for an honest party, which is negligible given the EU-CMA security of the signature scheme. More concretely

- Acknowledge TX: Either of the conditions $\gamma_S = \mathsf{OPEN} \vee (t \leq t_e + \delta_r)$ (a currently opened channel), b (the proposed transaction is valid), and $\tau \notin S^{\mathsf{TX}}_{\mathsf{Reject},i}$ (the transaction is consistent with the p_i's view) does not return $\top$. It only results in $\bot$ in the case a confirmation, *i.e.*, a signature, from an honest part was not actually signed by the party which can be considered negligible due to the EU-CMA of the signature scheme;
- Confirm Tx: In addition to the conditions in the early case, $\mathcal{F}_{CH}^{\text{HydraSM}}$ also verifies if $\forall p_i \in \mathcal{H} : \tau \in L^{\mathsf{ST}}_{\mathsf{seen},i} || L^{\mathsf{TX}}_{\mathsf{seen},i}$. In case of $\top$, all the honest parties have proposed transaction in their views. This condition is broken only if the adversary impersonated one of the honest parties, which happens with negligible probability given the EU-CMA security of the signature scheme;
- Acknowledge ST: Again, in addition to the open channel and transaction validity condition, $\mathcal{F}_{CH}$ verifies if (1) $T \subseteq S^{\mathsf{TX}}_{\mathsf{conf}}$, *i.e.*, the transaction set for the new state are consistent with the (honest) jointly confirmed set of transactions, (2) $L^{\mathsf{ST}}_{\mathsf{seen},i} = \epsilon$, *i.e.*, the ordered transactions for the state is decided (and flushed) by the p_i, meaning an honest party had already confirmed the state, and (3) $(T \cap (S^{\mathsf{TX}}_{\mathsf{Reject},i} \cup R)) = \varnothing$, *i.e.*, the list (*i.e.* ordered) of transactions for the, to be acknowledged, state are consistent;
- Confirm ST: In addition to the channel opening verification as in the earlier interfaces, $\mathcal{F}_{CH}^{\text{HydraSM}}$ also confirms the correct serialization of the states, one state at the time, (by checking $L^{\mathsf{ST}}_{\mathsf{seen},i} \neq \epsilon$ as before), and then if the first honest party or all late honest parties confirm that the state is consistent by verifying $L^{\mathsf{ST}}_{\mathsf{seen},i} = L^{\mathsf{ST}}_{\mathsf{pending}} \vee \exists L_{\mathsf{prefix}} : L^{\mathsf{ST}}_{\mathsf{conf}} = L_{\mathsf{prefix}} || L^{\mathsf{ST}}_{\mathsf{seen},i}$, given the $\mathcal{S}_{\mathsf{H}}$ construction and $\mathcal{F}_{CH}$ is happens if $\mathcal{A}$ impersonates an honest party by generating a forged signature, thus this event is considered negligible due to the EU-CMA security of the signature scheme.

– Figure 10: There are four conditions that $\mathcal{F}_{CH}$ fails: (1) an invalid state machine transition occurs, (2) the ledger consistency condition of $\mathcal{F}_{CH}^{\text{HydraSM}}$ fails, (3) during the the abort situation the right conditions are not met, and (4) the two time bound violations, respectively, closing and end states. Namely,
  - Invalid transition (Line 2): assuming the early Claim, thus this assert clause does not receive $\bot$;
  - Ledger consistency condition (Line 4): This condition is preserved given it relies on the correct initialization of the channel, as given by $\mathcal{F}_{CH}^{\text{HydraSM}}$ design, and the validity of the transaction τ, assuming that honest players follow Π_{Hydra} as in the simulation of $\mathcal{S}_{\mathsf{H}}$, and $\mathcal{A}$ cannot disrupt it by forging confirmation/acknowledgment due to the signature EU-CMA security, thus this event happens with negligible probability.
  - Abort condition (Line 6): the value of $\mathsf{b}_{AbortCond}$ is set in Figure 9 and it receives $\bot$, however this means the protocol cannot achieve a valid closing state, since either UTxO and account based (as discussed in Section 4.7)

allow achieving it, and given Hydra security, thus we consider this event negligible;

- Close (Line 10), End time bound (Line 12) and Channel Summary (Line 13): Given $\mathcal{S}_{\mathsf{H}}$ construction and the $\mathcal{F}_{CH}^{\text{HydraSM}}$ design in case of all corrupted players, the assert clause is disable, *i.e.*, it does not halt functionality even if it receives $\perp$, and in the case of at least one honest player and given the design of $\mathcal{F}_{CH}^{\text{HydraSM}}$, the limits are preserved, thus not triggering the assert clauses.

Thereby we conclude $\mathcal{Z}$ distinguishes with at most negligible probability via a failure in the protocol parties output simulation provided by $\mathcal{S}_{\mathsf{H}}$. □

The proof of the main security claim. We now prove the security of multi-party isomorphic state channel protocol Π_{Hydra} with respect to the $\mathcal{F}_{CH}$ Functionality in the $(\mathcal{G}_{\mathsf{Constrained\text{-}Ledger}}^{\delta_w,\delta_r,\Lambda,\Sigma_0},\mathcal{G}_{Ticker})$-Global Model.

Theorem 1. *Let HydraSM be the state machine verifier for Π_{Hydra}, the function ζ_H to implement Π_{Hydra}. Given the earlier Claim, and assuming the EU-CMA security of the signature scheme, then Π_{Hydra} UC-realizes $\mathcal{F}_{CH}^{HydraSM}$ in the $(\mathcal{G}_{\mathsf{Constrained\text{-}Ledger}}^{\delta_w,\delta_r,\Lambda,\Sigma_0},\mathcal{G}_{Ticker})$-Global Model.*

Sketch-Proof (Theorem 1).
Overview of the Proof. In order to prove Theorem 1, we need to show that $\mathcal{Z}$ can distinguish between the real execution of Π_{Hydra} in the $(\mathcal{G}_{\mathsf{Constrained\text{-}Ledger}}^{\delta_w,\delta_r,\Lambda,\Sigma_0},\mathcal{G}_{Ticker})$-Global Model and the ideal world with at most negligible probability. The environment $\mathcal{Z}$ has three means to attempt to distinguish both executions:

1. The environment $\mathcal{Z}$ is assisted with an adversary $\mathcal{A}$ who can corrupt parties and observe the protocol execution of Π_{Hydra} in either the real world in the $(\mathcal{G}_{\mathsf{Constrained\text{-}Ledger}}^{\delta_w,\delta_r,\Lambda,\Sigma_0},\mathcal{G}_{Ticker})$-Global Model or it is wrapped by $\mathcal{S}_{\mathsf{H}}$ who simulates the protocol execution for $\mathcal{A}$. We address this in **Lemma 1**;
2. Lastly, $\mathcal{Z}$ has access to the global functionalities $\mathcal{G}_{\mathsf{Constrained\text{-}Ledger}}^{\delta_w,\delta_r,\Lambda,\Sigma_0}$ and $\mathcal{G}_{Ticker}$. Most notably, $\mathcal{Z}$ reads $\mathcal{G}_{\mathsf{Constrained\text{-}Ledger}}^{\delta_w,\delta_r,\Lambda,\Sigma_0}$ state and transactions submitted to it through $\mathcal{A}$. We address $\mathcal{Z}$'s view of the ledger in **Lemma 2**; and
3. The direct way for $\mathcal{Z}$ to distinguish is to observe the outputs from both executions to itself. We address this in **Lemma 3**.

If $\mathcal{Z}$ distinguishes the ideal from the real world through the above means with at most negligible probability, then Theorem 1 follows.

Thereby the theorem follows from Lemmas 1 through 3. □

6 Extension for Hanging Transactions

The Functionality $\mathcal{F}_{CH}$ is agnostic of the underlying ledger paradigm. We observe that, if we additionally assume the UTxO Model (as in Section 2.6) we can create an extension $\mathcal{F}_{CH\text{-}HT}^{SM}$ of $\mathcal{F}_{CH}$ that provides an additional desirable property named *Early Finality* of transactions as in the following Definition 3.

Definition 3 (Early Finality). *A channel γ has early finality if any transaction affects the state of γ at the point at which it is confirmed by at least one honest party in γ and its effect on γ's state is independent on the order in which parties in γ confirm it.*

The intuition behind *Early Finality* is that a transaction effectively is added to the channel state as soon as it is confirmed by at least one honest party, even if it is never included in a channel state change.

Before we describe the extended functionality with the *Early Finality* property, we briefly describe the case of transaction ordering and UTxO based smart contracts. Then we (1) introduce our extended functionality $\mathcal{F}_{CH\text{-}HT}$; (2) formally define the extended Π^{HT}_{Hydra} Hydra Protocol, which was only outlined in [10]; and (3) prove its security with respect to $\mathcal{F}_{CH\text{-}HT}$.

6.1 Transaction Ordering and the Case of UTxO Smart Contracts

In the $\mathcal{F}_{CH}$ setting, transactions are first confirmed and afterwards ordered through confirmation of a state update. This is commonly done in consensus systems where transactions are processed in batches such as blockchains or layer-2 protocols, such as Sidechains [19, 20] and Rollups [25]. In general, the effect of a transaction on the channel's state is only known after its order is determined. For instance, consider a smart contract, *i.e.*, a program executed on a ledger. Such a smart contract has an internal state. Transactions can be created that interact with this smart contract through call of a function and the effect of it, will depend on the state of the smart contract. A different order of function calls can potentially result in a different state.

As a simple example, imagine a smart contract holding some coins. Function calls can be sent to the smart contract to withdraw coins from it. Upon such a function call, if the smart contract received beforehand an even amount of function calls 2 coins are sent to the caller of the function and 1 coin otherwise. While all of the transactions are valid at the moment they are submitted and thus they can be confirmed by the channel, the resulting balance distribution of coins received by parties that interact with this smart contract entirely depends on the order in which these transactions are received.

However, smart contracts implemented using the UTxO Model do not behave as described above. There, smart contracts exist in the form of UTxOs themselves. Assume a UTxO u represents a smart contract. At the moment a transaction interacts with the smart contract, u is spent and a new UTxO is created representing the new state of that smart contract. Another transaction that attempted to spent u cannot be confirmed as it would be double spending u. Thus, whether a transaction can be confirmed in the channel now depends on the order in which they were received by the parties in the channel. If there are conflicting views about which transaction is received first by the channel and thus confirmed, the state update function resolves this conflict. However, if a transaction is confirmed, the resulting state is independent of the specific ordering of all confirmed transactions. To sum up, transactions have a higher bar to

pass to be confirmed, but as a tradeoff can be considered final the moment they are confirmed. Channel state updates now, instead of more generally ordering transactions, specifically are used to resolve conflicting transactions and reduce the memory overhead of parties as transactions do not need to be stored if they were included to compute a new state.

6.2 The Extended $\mathcal{F}_{CH\text{-}HT}$ Functionality

Functionality $\mathcal{F}_{CH\text{-}HT}$ extends $\mathcal{F}_{CH}$ by utilizing the above observation to provide *Early Finality* through the use of a *hanging transactions* mechanism. Hanging transactions ensure that all confirmed transactions are included when creating the channel's final state. For this, we require that ζ_{HT} in addition to its original definition, also contains algorithm HANGING which is used to create an additional type of transactions to implement hanging transactions. $\mathcal{F}^{SM}_{CH\text{-}HT}$ is shown in Figures 14, 15 and 16 where individual changes are highlighted in blue.

$\mathcal{F}^{SM}_{CH\text{-}HT}$ Extended Functionality

Additional Variables: Store the following parameters within the functionality's state:

- Duration of contestation period for hanging transactions $t_h \leftarrow 0$
- Set of submitted hanging transactions $S^{\mathsf{TX}}_{\mathsf{HT}} \leftarrow \varnothing$

Setup: Upon message $(\mathsf{SETUP}, sid, \gamma'_p, t'_c, t'_h, i)$ from a party p, do

1: $p_i \leftarrow p$
2: **if** $\gamma_p = \varnothing$ **then**
3: $\quad \gamma_p \leftarrow \gamma_{p'}$, $t_c \leftarrow t'_c$, $t_h \leftarrow t'_h$
4: $\quad$ Send $(\mathsf{FUNC_SETUP}, sid)$ to $\mathcal{S}$, receiving $(\zeta', \theta', \delta'_r, \mathcal{C}')$
5: $\quad \mathcal{C} \leftarrow \mathcal{C}'$, $\mathcal{H} \leftarrow \gamma_p \setminus \mathcal{C}'$, $\zeta \leftarrow \zeta'$, $\theta \leftarrow \theta'$
6: $\quad$ Send $(\mathsf{REGISTER})$ to $\mathcal{G}_{Ticker}$
7: $\quad$ **if** $\mathcal{C} = \gamma_p$ **then** $\mathsf{Call}(\mathsf{FullAdvControl})$
8: **if** $b_i = \bot \wedge \gamma_p = \gamma'_p \wedge t_c = t'_c$, $t_h = t'_h$ **then**
9: $\quad$ Send $(\mathsf{PARTY_SETUP}, sid, p_i, \gamma_{p'}, t'_c, t'_h, i)$ to $\mathcal{S}$
10: $\quad b_i \leftarrow \top$
11: $\quad$ **if** $p_i \in \mathcal{H}$ **then**
12: $\quad\quad$ Send $(\mathsf{REGISTER}, p_i)$ to $\mathcal{G}_{Ticker}$ on behalf of p_i receiving reply m
13: $\quad\quad$ Send m on behalf of $\mathcal{G}_{Ticker}$ to p_i
14: **if** $\forall p_i \in \gamma_p : b_i = \top$ **then** $\gamma_S \leftarrow$ NotYetOpen

Fig. 14. The functionality $\mathcal{F}_{CH\text{-}HT}$ replicates variables and behavior of $\mathcal{F}_{CH}$ with the following changes. In blue there are the differences between $\mathcal{F}_{CH}$ and $\mathcal{F}_{CH\text{-}HT}$ such as extra time parameter and state, respectively t_h and $S^{\mathsf{TX}}_{\mathsf{HT}}$.

$\mathcal{F}_{CH\text{-}HT}^{SM}$ Extended Functionality (Interpreting transactions)

Procedure **OnTx**(τ, p_i)**:**

1: Check conditions for hanging transactions
$\mathsf{b}_{min} \leftarrow \bot$; $\mathsf{b}_{max} \leftarrow \bot$
Send $(\mathsf{ORDER}, sid, S_{\mathsf{HT}}^{\mathsf{TX}})$ to $\mathcal{S}$, receiving L_{HT}
Assert (L_{HT} is a permutation of $S_{\mathsf{HT}}^{\mathsf{TX}}$)
If $S_{\mathsf{Conf}}^{\mathsf{TX}} \subseteq S_{\mathsf{HT}}^{\mathsf{TX}} \subseteq S_{\mathsf{Conf}}^{\mathsf{TX}} \cup (\bigcap_{p_j \in \mathcal{H}} L_{\mathsf{Seen},j}^{\mathsf{TX}})$ **then**
$\mathsf{b}_{min} \leftarrow \mathrm{S_EQ}(\Omega_i || L_{\mathsf{conf}}^{\mathsf{ST}} || L_{\mathsf{HT}}, \Omega_i || \tau)$
If
$S_{\mathsf{Conf}}^{\mathsf{TX}} \setminus S_{\mathsf{reject,pending}}^{\mathsf{TX}} \subseteq S_{\mathsf{HT}}^{\mathsf{TX}} \subseteq (S_{\mathsf{Conf}}^{\mathsf{TX}} \cup (\bigcap_{p_j \in \mathcal{H}} L_{\mathsf{Seen},j}^{\mathsf{TX}}) \cup L_{\mathsf{pending}}^{\mathsf{ST}}) \setminus S_{\mathsf{reject,pending}}^{\mathsf{TX}}$
then $\mathsf{b}_{max} \leftarrow \mathrm{S_EQ}(\Omega_i || L_{\mathsf{conf}}^{\mathsf{ST}} || L_{\mathsf{pending}}^{\mathsf{ST}} || L_{\mathsf{HT}}, \Omega_i || \tau)$
2: Set inputs (about the functionality) for the state machine representation
$\mathsf{Fdata} \leftarrow (p_i, \mathsf{param}, S_\sigma, t_c, t_h, \Sigma_0^\gamma, \Sigma_i, S_{\mathsf{open,i}}^{\mathsf{TX}}, S_{\mathsf{HT}}^{\mathsf{TX}}, \mathsf{b}_{max}, \mathsf{b}_{min})$
3: Set bit for condition to safe state machine abortion
// Is there a chance of wrong abort condition, i.e., different state?
$L_{\mathsf{INV}} \leftarrow$ any permutation of $\{\Lambda.INV(\tau) \mid \tau \in S_{\mathsf{open,i}}^{\mathsf{TX}}\}$
$\mathsf{b}_{AbortCond} \leftarrow \mathbf{S_EQ}(\Omega_i || L_{\mathsf{INV}}, \Omega_i || \tau)$
4: Send $(\mathsf{TIMESTAMP}, \tau)$ to $\mathcal{G}_{\mathsf{Constrained\text{-}Ledger}}^{\delta_w, \delta_r, \Lambda, \Sigma_0}$ receiving t
5: **if** $\exists x, \mathsf{param}$: $\tau = \zeta(x, p_i, \mathsf{param}, S_\sigma)$ **then**
6: **for** $\mathsf{tr} \in \theta(\tau)$ **do** ▷ Process state transitions τ triggers 1-by-1
// Single transition in the state machine for tr
7: Let $\mathsf{output} \leftarrow SM(\tau, \mathsf{tr}, \mathsf{Fdata})$ ▷ Transitioning SM
8: **HandleTransitions**(output)
9: $\Omega_i \leftarrow \Omega_i || \tau$

Procedure **S_EQ**(Ω_1, Ω_2)**:**

1: **if** $\mathcal{G}_{\mathsf{Constrained\text{-}Ledger}}^{\delta_w, \delta_r, \Lambda, \Sigma_0}.\Lambda.\mathsf{STATE}(\Sigma_0, \Omega_1) = \mathcal{G}_{\mathsf{Constrained\text{-}Ledger}}^{\delta_w, \delta_r, \Lambda, \Sigma_0}.\Lambda.\mathsf{STATE}(\Sigma_0, \Omega_2)$ **then return** $\top$. Otherwise, **return** $\bot$

Fig. 15. The **OnTX Procedure** captures, as it was early defined for $\mathcal{F}_{CH}$ (*i.e.*, Figure 9) via the internal "for" loop, all the state transitions in the SM. The main loop is added with extra time information, transaction set, and acceptable final states.

$\mathcal{F}^{SM}_{CH\text{-}HT}$ Extended Functionality (Handling State Transitions)

Procedure **HandleTransitions**(output):

1: Parse $(\mathsf{msgs}, \mathsf{newFST}, \mathsf{newST}, \mathsf{HSet}, \mathsf{b}, \mathsf{b}_{Open}, \mathsf{b}_{Abt}, \mathsf{b}_{Dpt}) \leftarrow \mathsf{output}$
2: Assert (b) ▷ $\mathcal{F}_{CH}$ fails on invalid state transition in case of $\mathsf{b} = \bot$
3: **if** $\gamma_S = \text{NotYetOpen} \wedge (\mathsf{b}_{Abt} = \bot) \wedge \mathsf{b}_{Open}$ **then**
$\gamma_S \leftarrow \text{Open}$
4: Assert $(\Sigma_0^\gamma = \bigcup_{i=1}^n \Sigma_i)$ ▷ *All parties have commitment in the initial channel state?*
// In state machine abort, necessary to check conditions
5: **if** $\gamma_S = \text{NotYetOpen} \wedge \mathsf{b}_{Abt}$ **then**
6: Assert $(\mathsf{b}_{AbortCond})$ and $\gamma_S \leftarrow \text{End}$
7: **if** $\gamma_S = \text{Open} \wedge \mathsf{b}_{Dpt} = \top$ **then**
8: $t_e \leftarrow t$ and $\gamma_S \leftarrow \text{NotYetEnd}$ ▷ *Set dispute start time.*
9: Assert $(t \leq t_e + \Delta_{Close})$ ▷ *Hardwired CLOSE time bound.*
//Does τ sum up the channel state?
10: Assert $(\mathbf{S_EQ}(\Omega_i || L^{\mathsf{ST}}_{\mathsf{conf}}, \Omega_i || \tau) \vee \mathbf{S_EQ}(\Omega_i || L^{\mathsf{ST}}_{\mathsf{conf}} || L^{\mathsf{ST}}_{\mathsf{pending}}, \Omega_i || \tau))$
11: **if** $\gamma_S = \text{NotYetEnd}$ **then** $\gamma_S \leftarrow \text{End}$ ▷ *Only way valid tr.*
12: Assert $(t \geq t_e + t_c \wedge t \leq t_e + t_c + \Delta_{End})$ ▷ *Hardwired END time bound.*
13: Set new states for functionality and party view
$\Sigma_0^\gamma \leftarrow \mathsf{newFST}$, $S^{\mathsf{TX}}_{\mathsf{open,i}} \leftarrow \mathsf{newST}$ and $S^{\mathsf{TX}}_{\mathsf{HT}} \leftarrow \mathsf{HSet}$
14: **for** $\mathsf{msg} \in \mathsf{msgs}$ **do** ▷ Some state transitions result in messaging
15: Parse $(\mathsf{type}, \mathsf{attr}, \mathsf{receiver}) \leftarrow \mathsf{msg}$ ▷ *Not sanitized msg's.*
16: **if** $\mathsf{attr} = \cdot$ **then** send (type, sid) to $\mathsf{receiver}$
17: **else** send $(\mathsf{type}, sid, \mathsf{attr})$ to $\mathsf{receiver}$

Fig. 16. The procedure receives the set of hanging transactions updated during the state transitions which are used to update the internal set of the functionality.

In addition to the changes in the Functionality $\mathcal{F}_{CH\text{-}HT}$, it is also necessary to update the state machine verifier in order to support hanging transactions. Thus we introduce the Hydra-HT State Machine Verifier **HydraSM-HT** in Figure 17.

Hydra-HT State Machine Verifier **HydraSM-HT**

HydraSM-HT(τ, tr, Fdata):

Let ζ_H be the state transaction function for Hydra
Set $\mathsf{b} \leftarrow \bot$ ▷ Set to $\top$ if τ performs any valid state transition
Set $\mathsf{b}_{Abt} \leftarrow \bot$ ▷ Set to $\top$ if the valid transition was for abort
Set $\mathsf{b}_{Dpt} \leftarrow \bot$ ▷ Set to $\top$ if in dispute
Set $\mathsf{b}_{Open} \leftarrow \bot$ ▷ Set to $\top$ if channel is open
Define $\mathsf{newFST} = \mathsf{newST} = \cdot$ ▷ Initially empty variables
Define $\mathsf{FState} = \mathsf{SETUP}$ ▷ Internal State initially in 'Setup'
Define empty list msgs ▷ To allow correct messaging later
Parse Fdata as
$(\mathsf{party}, \mathsf{param}, S_\sigma, t_c, t_h, \mathsf{ChInitState}, \mathsf{OpenCommits}, \mathsf{OpenTXs}, \mathsf{HSet}, \mathsf{b}_{max}, \mathsf{b}_{min})$

// From here, it looks for the correct state transition according to tr
if $\mathbf{SM}(\mathsf{SETUP}, \mathsf{INIT}, \mathsf{INIT}, \mathsf{tr}) = \top$ **then** $b \leftarrow \top$
if $\mathbf{SM}(\mathsf{INIT}, \mathsf{COMM}, \mathsf{INIT}, \mathsf{tr}) = \top \wedge \tau = \zeta_H(\mathsf{COMM}, \mathsf{party}, \mathsf{param}, S_\sigma)$ **then**
  Assert $(\Sigma_i \notin \Sigma_0^\gamma$, for $\Sigma_i \in \mathsf{param})$ ▷ *Is i-th party's only commitment?*
  $\mathsf{b} \leftarrow \top$, $\mathsf{newST} \leftarrow \mathsf{OpenTXs} \cup \{\tau\}$,
  $\mathsf{newFST} \leftarrow \mathsf{ChInitState} \cup \mathsf{OpenCommits}$
if $\mathbf{SM}(\mathsf{INIT}, \mathsf{COLL}, \mathsf{OPEN}, \mathsf{tr}) = \top$ **then**
  Set $\mathsf{b} \leftarrow \top$ and $\mathsf{b}_{Open} \leftarrow \top$, and
  add $(\mathsf{OPEN}, \mathsf{ChInitState}, \mathsf{party})$ to msgs
if $\mathbf{SM}(\mathsf{OPEN}, \mathsf{CLOSE}, \mathsf{DISPUTE}, \mathsf{tr}) = \top$ **then**
  Set $\mathsf{b} \leftarrow \top$, $\mathsf{b}_{Dpt} \leftarrow \top$ and $\mathsf{b}_{Open} \leftarrow \bot$,
  add $(\mathsf{CLOSING}, \cdot, \mathsf{party})$ to msgs
if $\mathbf{SM}(\mathsf{DISPUTE}, \mathsf{CONTEST}, \mathsf{DISPUTE}, \mathsf{tr}) = \top \wedge t \leq t_e + t_c$ **then** $\mathsf{b} \leftarrow \top$
if $\mathbf{SM}(\mathsf{DISPUTE}, \mathsf{FANOUT}, \mathsf{END}, \mathsf{tr}) = \top \wedge t > t_e + t_c$ **then**
  Set $b \leftarrow \top$, $\mathsf{b}_{Dpt} \leftarrow \bot$ and $\mathsf{b}_{Open} \leftarrow \bot$,
  add $(\mathsf{CLOSED}, \cdot, \mathsf{party})$ to msgs
if $\mathbf{SM}(\mathsf{INIT}, \mathsf{ABORT}, \mathsf{END}, \mathsf{tr}) = \top$ **then**
  Set $\mathsf{b} \leftarrow \top$ and $\mathsf{b}_{Abt} \leftarrow \top$ ▷ *Channel not even opened.*
  Add $(\mathsf{CLOSED}, \cdot, \mathsf{party})$ to msgs
if $\mathsf{SM}(\mathsf{DISPUTE}, \mathsf{HANG}, \mathsf{DISPUTE}, \mathsf{tr}) = \top$ **then**
  if $t_e + t_c < t \leq t_e + t_c + t_h$ **then** $b \leftarrow \top$; $\mathsf{HSet} \leftarrow \mathsf{HSet} \cup \mathsf{param}.S_{\mathsf{TH}}^{\mathsf{TX}}$
if $\mathsf{SM}(\mathsf{DISPUTE}, \mathsf{FANOUT}, \mathsf{END}, \mathsf{tr}) = \top \wedge t > t_e + t_c + t_h$ **then**
  if $b_{\mathsf{min}} \vee b_{\mathsf{max}}$ **then**
    $b \leftarrow \top$ and add $(\mathsf{CLOSED}, \cdot, \mathsf{party})$ to msgs

// Report on the state transition, including new state, messages, etc
Return $(\mathsf{msgs}, \mathsf{newFST}, \mathsf{newST}, \mathsf{HSet}, \mathsf{b}, \mathsf{b}_{Open}, \mathsf{b}_{Abt}, \mathsf{b}_{Dpt})$

// Verifies whether such transition exists
Procedure **SM**(state, edge, next_state, tr): If $\mathsf{FState} = \mathsf{state} \wedge \mathsf{tr} = \mathsf{edge}$ then $\mathsf{FState} \leftarrow \mathsf{next_state}$, and return $\top$. Otherwise, return $\bot$.

Fig. 17. The Hydra State Machine which supports hanging transactions requires new state transitions in addition to the ones in **HydraSM** (Figure 11).

6.3 The Extended Π^{HT}_{Hydra} Protocol

While the Hydra protocol is formally specified [10], handling of hanging transactions in the protocol exists only as a high level description. We close this gap by including handling of hanging transactions through polling upon each activation. Π^{HT}_{Hydra} extends Π_{Hydra} and is specified in Figure 18 where individual changes are highlighted in blue.

Hydra Channel Protocol Π^{HT}_{Hydra} with Hanging Transactions

Additional Variables: Store the following parameters:
- Start time of the contestation phase $t_e = -1$
- Duration of contestation period for hanging transactions $t_h = 0$
- Channel time $t = 0$

Behavior: On activation: Before proceeding, first send (TICKED?) to $\mathcal{G}_{Ticker}$. If reply is (TICKED): $t \leftarrow t + 1$. Then, send (READ) to $\mathcal{G}^{\delta_w,\delta_r,\Lambda,\Sigma_0}_{\text{Constrained-Ledger}}$, receiving reply Ω'. If $\Omega \subseteq \Omega'$ let D where $\Omega' = \Omega||D$ holds, then Call(OnTx, D) followed by Call(Hanging). Moreover, before returning control, first executes procedure ExecWait() and afterwards sends (ACTIVATED, sid) to $\mathcal{G}_{Ticker}$.

Setup: Upon input (SETUP, $sid, \gamma'_p, t'_c, t'_h, \Sigma'_i, i'$)

1: **if** $t'_c \leq |\gamma'_p|(\delta_r + \delta_w) \vee t'_h \leq \delta_r + \delta_w$ or this *SETUP* message has been received before **then** drop message
2: Create key pair $(\mathsf{sk}', \mathsf{vk}) \leftarrow \mathsf{Gen}$ with fresh randomness
3: $K \leftarrow \{\mathsf{vk}\}$, $\mathsf{sk} \leftarrow \mathsf{sk}'$
4: $\gamma_p \leftarrow \gamma'_p$, $t_c \leftarrow t'_c$, $t_h \leftarrow t'_h$, $\Sigma \leftarrow \Sigma'_i$, $i \leftarrow i'$
5: Send (ReqSetup, $sid, \mathsf{vk}, \Sigma, i$) to all parties in γ_p

Procedure ChainClose(τ, param):
  Output (CLOSING, sid) to $\mathcal{Z}$
  $t_e \leftarrow t$
  Call(ChainContest, τ, param)

Procedure **Hanging**:
  if $\gamma_S = \mathsf{CLOSED} \wedge t_e + t_c < t < t_e + t_c + t_h$ **then**
    $\mathsf{param}' \leftarrow (\mathcal{T}^c)$
    Call(Submit, HANGING, param′)

Fig. 18. Changes between Π_{Hydra} and Π^{HT}_{Hydra} are concentrated in the introduction of the new additional variables, t_e, t_h and t, changes in the internal procedure **ChainClose**, and a new procedure Hanging.

6.4 Security of Π^{HT}_{Hydra} with respect to $\mathcal{F}_{CH\text{-}HT}$

Now we prove the security of Π^{HT}_{Hydra} by showing that it implements $\mathcal{F}^{SM}_{CH\text{-}HT}$. The proof is analogous to the early one for Π_{Hydra}. However, in addition to the early proof, it requires the addressing of hanging transactions. Note that for the next theorem we consider a factory of the extended version factory ζ_{HT}.

Claim. Let Π^{HT}_{Hydra} be the Hydra Protocol as in [10] supporting hanging transactions (as with the changes in Figure 18), therefore HydraHTSM (as in Figure 17) does not return $\perp$ for transactions from Π^{HT}_{Hydra}.

The proof is analogous of similar Claim about HydraSM, and this Claim will be used in the proof of the next Theorem on the extended version of Hydra.

Theorem 2. *Let HydraHTSM be the state machine verifier for Π^{HT}_{Hydra}, and the function ζ_{HT} to implement Π^{HT}_{Hydra}. Given the earlier Claim on Π^{HT}_{Hydra}, and assuming the EU-CMA security of the signature scheme, then Π^{HT}_{Hydra} UC-realizes $\mathcal{F}^{HydraHTSM}_{CH}$ in the $(\mathcal{G}^{\delta_w,\delta_r,\Lambda,\Sigma_0}_{\mathsf{Constrained\text{-}Ledger}}, \mathcal{G}_{Ticker})$-Global Model.*

The proof strategy is analogous to the proof of Theorem 1. We proceed with the same proof structure as follows: First, we define a simulator $\mathcal{S}_{\mathsf{HT}}$ this time simulating Π^{HT}_{Hydra}. Then, we show that any environment $\mathcal{Z}$ cannot distinguish with more than negligible probability even when it observes (1) leakage from adversary $\mathcal{A}$; (2) transactions on the ledger; and (3) protocol outputs.

The Simulator $\mathcal{S}_{\mathsf{HT}}$. First, we define a new simulator $\mathcal{S}_{\mathsf{HT}}$ as shown in Figure 19. This simulator behaves as Simulator $\mathcal{S}_{\mathsf{HT}}$ as shown in Figure 12 with the following differences: (1) It simulates parties executing Π^{HT}_{Hydra} instead of Π_{Hydra} and communicates with $\mathcal{F}^{\text{HydraHTSM}}_{CH\text{-}HT}$; (2) it additionally communicates the contestation period for hanging transactions t_h with $\mathcal{F}^{\text{HydraHTSM}}_{CH\text{-}HT}$; (3) it chooses a different transaction functions ζ_{HT} and θ_{HT} to submit to $\mathcal{F}^{\text{HydraHTSM}}_{CH\text{-}HT}$ during setup which now implements a state machine that handles hanging transactions, for instance, the Hydra state machine with hanging transactions [10]. Lastly, (4) it accepts a message of form $(\mathsf{ORDER}, sid, T_h)$ where T_h is a set of hanging transactions which must been acknowledged, thus seen, by all honest parties. It then selects one arbitrary honest party and orders all transactions in T_h according to the order in which that party acknowledged them or drops the message if T_h contains a transaction that was not signed by all honest parties.

Simulator $\mathcal{S}_{\mathsf{HT}}$ (Differences to $\mathcal{S}_{\mathsf{H}}$)

Behavior: At start, initialize simulated parties for all protocol participants. $\mathcal{S}_{\mathsf{HT}}$ simulates Π^{HT}_{Hydra} for all honest parties. Moreover, $\mathcal{S}_{\mathsf{HT}}$ expects messages from $\mathcal{F}^{\text{HydraHTSM}}_{CH\text{-}HT}$ instead of $\mathcal{F}^{\text{HydraSM}}_{CH}$. Whenever $\mathcal{S}_{\mathsf{H}}$ would have sent a message to $\mathcal{F}^{\text{HydraSM}}_{CH}$, $\mathcal{S}_{\mathsf{HT}}$ instead sends a message to $\mathcal{F}^{\text{HydraHTSM}}_{CH\text{-}HT}$.

Setup: Upon message $(\mathsf{SETUP}, sid, p_i, \gamma'_p, t_c, t_h, \Sigma_i, i)$ from $\mathcal{F}_{CH\text{-}HT}$, do

1: **if** $\gamma_p = \varnothing$ **then** $\gamma_p \leftarrow \gamma'_p$
2: **if** $p_i \notin \mathcal{C}$ **then**
3: Simulate p_i receiving message $(\mathsf{SETUP}, sid, \gamma_p, t_c, h_t, \Sigma_i, i)$
4: Leak $p_i.\mathsf{vk}$ to $\mathcal{A}$

Algorithm Setup: Upon message $(\mathsf{ALG_SETUP}, sid)$ from $\mathcal{F}_{CH\text{-}HT}$, do

1: **if** $\gamma_p = \mathcal{C}$ **then**
2: Take control of $\mathcal{F}_{CH\text{-}HT}$. Make it behave normally but prevent it from sending any messages except to $\mathcal{S}_{\mathsf{HT}}$, $\mathcal{G}^{\delta_w,\delta_r,\Lambda,\Sigma_0}_{\mathsf{Constrained\text{-}Ledger}}$ and $\mathcal{G}_{Ticker}$, *i.e.*, $\mathcal{F}_{CH\text{-}HT}$ does not send messages to $\mathcal{Z}$, or any party, and only forwards messages between $\mathcal{S}_{\mathsf{HT}}$ and the global functionalities
3: Select ζ_{HT} such that it implements the Hydra on-chain state machine [10] with hanging transactions on $\mathcal{G}^{\delta_w,\delta_r,\Lambda,\Sigma_0}_{\mathsf{Constrained\text{-}Ledger}}$
4: Select θ_{HT} such that for a transaction $\tau = \zeta_{\mathsf{HT}}(x, \dots)$, $\theta_{\mathsf{HT}}(\tau) = \{x\}$ holds
5: $\delta \leftarrow \delta_r + \delta_w$
6: **return** $(\zeta_{\mathsf{HT}}, \theta, \delta, \mathcal{C})$ to $\mathcal{F}_{CH\text{-}HT}$

Order: Upon receiving input $(\mathsf{ORDER}, sid, T_h)$ from $\mathcal{F}_{CH\text{-}HT}$, do

1: **if** $\neg(\forall p \in (\gamma_p \setminus \mathcal{C}) : T_h \subseteq p.S^{\mathsf{TX}}_{\mathsf{CONF}})$ **then return** $\perp$
2: $p \leftarrow p' \in (\gamma_p \setminus \mathcal{C})$ uniformly chosen at random
3: $L_h \leftarrow$ Transactions in T_h in order in which they were acknowledged by p
4: **return** L_h

Fig. 19. How the simulator $\mathcal{S}_{\mathsf{HT}}$ handles messages from $\mathcal{F}^{\text{HydraHTSM}}_{CH\text{-}HT}$. The main changes are in the **Behavior** interface of $\mathcal{S}_{\mathsf{H}}$ and the introduction of the new interface Order.

Auxiliary Lemmas. The following auxiliary lemmas are analogous to the early Lemmas 1, 2 and 3, but adapted to the extended to Π^{HT}_{Hydra} and $\mathcal{F}^{\text{HydraHTSM}}_{CH\text{-}HT}$.

Lemma 4. *Let ζ_{HT} implement Π^{HT}_{Hydra} and assuming the underlying signature scheme is EU-CMA, then $\mathcal{S}_{\mathsf{HT}}$ simulates Π^{HT}_{Hydra} such that $\mathcal{Z}$ can distinguish through the view of $\mathcal{A}$ with at most negligible probability.*

Proof. This proof is analogous to the proof of Lemma 1: As $\mathcal{F}^{\text{HydraHTSM}}_{CH\text{-}HT}$ does not impede communication between $\mathcal{Z}$, parties in the simulation and the global functionalities, and because $\mathcal{S}_{\mathsf{HT}}$ simulates Π^{HT}_{Hydra} the arguments for proving Lemma 1 hold here and Lemma 4 follows. □

Lemma 5. *Let ζ_{HT} implement Π^{HT}_{Hydra}, and assuming the underlying signature scheme is EU-CMA, then $\mathcal{Z}$ can distinguish with at most negligible probability by receiving messages from $\mathcal{G}^{\delta_w,\delta_r,\Lambda,\Sigma_0}_{\mathsf{Constrained\text{-}Ledger}}$ and $\mathcal{G}_{Ticker}$.*

Proof. This proof is analogous to the proof of Lemma 2: As $\mathcal{F}^{\text{HydraHTSM}}_{CH\text{-}HT}$ does not impede communication between $\mathcal{Z}$, parties in the simulation and the global functionalities, and because $\mathcal{S}_{\mathsf{HT}}$ simulates Π^{HT}_{Hydra} correctly due to Lemma 4, the arguments for proving Lemma 2 hold here, and Lemma 5 follows. □

Lemma 6. *Let ζ_{HT} implement Π^{HT}_{Hydra} such that HydraHTSM implements the state machine given in Figure 3 extended with handing transactions support. Assuming the early Claim, and that the underlying signature scheme is EU-CMA, then $\mathcal{Z}$ distinguishes with at most negligible probability by observing the parties' outputs in the simulation provided by $\mathcal{S}_{\mathsf{HT}}$.*

Proof. The proof is based on the one of Lemma 3. The extension to support hanging transactions introduces points in $\mathcal{F}_{CH\text{-}HT}$ (*i.e.* in Figures 15 and 16) and HydraHTSM (*i.e.* in Figure 17) which can potentially make $\mathcal{S}_{\mathsf{HT}}$ fail the simulation, therefore it is necessary to estimate the probability of these events:

- $\mathcal{F}^{\text{HydraHTSM}}_{CH\text{-}HT}$ (Figures 15/Line 1): given the construction $\mathcal{S}_{\mathsf{HT}}$, we can argue that the assert clause receives $\perp$ with negligible probability;
- HydraHTSM (Figure 17/Lines 29 and 32): in these lines the transitions are tested, assuming the Claim on the security of Π^{HT}_{Hydra}, we can argue that the transitions are correct, thus the assert clause in Figure 16 /Line 2 receives $\perp$ if negligible probability.

In addition to the probability analysis from the proof of Lemma 3, thus we have the proof of this lemma. □

The proof of the security claim for Π^{HT}_{Hydra}. We now prove the security of extended version of the multiparty isomorphic state channel protocol Π^{HT}_{Hydra} extended with hanging transaction handling with respect to the $\mathcal{F}^{\text{HydraHTSM}}_{CH\text{-}HT}$ Functionality in the ($\mathcal{G}^{\delta_w,\delta_r,\Lambda,\Sigma_0}_{\mathsf{Constrained\text{-}Ledger}}$,$\mathcal{G}_{Ticker}$)-Global Model.

Theorem 3. *Let HydraHTSM be the state machine verifier for Π^{HT}_{Hydra}, and the function ζ_{HT} to implement Π^{HT}_{Hydra}. Given the earlier Claim on Π^{HT}_{Hydra}, and assuming the EU-CMA security of the signature scheme, then Π^{HT}_{Hydra} UC-realizes $\mathcal{F}^{HydraHTSM}_{CH}$ in the $(\mathcal{G}^{\delta_w,\delta_r,\Lambda,\Sigma_0}_{\mathsf{Constrained\text{-}Ledger}}, \mathcal{G}_{Ticker})$-Global Model.*

Proof. In order to prove Theorem 2, we need to show that $\mathcal{Z}$ can distinguish between the real execution of Π^{HT}_{Hydra} in the $(\mathcal{G}^{\delta_w,\delta_r,\Lambda,\Sigma_0}_{\mathsf{Constrained\text{-}Ledger}},\mathcal{G}_{Ticker})$-Global Model and the ideal world with at most negligible probability. The environment $\mathcal{Z}$ has three means to attempt to distinguish both executions:

1. The environment $\mathcal{Z}$ is assisted with an adversary $\mathcal{A}$ who can corrupt parties and observe the protocol execution of Π^{HT}_{Hydra} in either the real world in the $(\mathcal{G}^{\delta_w,\delta_r,\Lambda,\Sigma_0}_{\mathsf{Constrained\text{-}Ledger}},\mathcal{G}_{Ticker})$-Global Model or it is wrapped by $\mathcal{S}_{\mathsf{HT}}$ who simulates the protocol execution for $\mathcal{A}$. We address this in **Lemma 4**;
2. Lastly, $\mathcal{Z}$ has access to the global functionalities $\mathcal{G}^{\delta_w,\delta_r,\Lambda,\Sigma_0}_{\mathsf{Constrained\text{-}Ledger}}$ and $\mathcal{G}_{Ticker}$. Most notably, $\mathcal{Z}$ reads $\mathcal{G}^{\delta_w,\delta_r,\Lambda,\Sigma_0}_{\mathsf{Constrained\text{-}Ledger}}$ state and transactions submitted to it through $\mathcal{A}$. We address $\mathcal{Z}$'s view of the ledger in **Lemma 5**; and
3. The direct way for $\mathcal{Z}$ to distinguish is to observe the outputs from both executions to itself. We address this in **Lemma 6**.

If $\mathcal{Z}$ distinguishes the ideal from the real world through the above means with at most negligible probability, then Theorem 2 follows.

Thereby the theorem follows from Lemmas 4 through 6. □

7 Conclusion

In this work we proposed a general framework for layer-2 multi-party channel protocols with realistic time constraints abstracting away from concrete ledger models. Concretely, among other contributions, we introduced a universally composable general framework to model multi-party isomorphic state channels .

The first crucial technical challenge was to properly model time constrains as they are crucial to most channel constructions. With this goal, we introduced a global ledger functionality, named $\mathcal{G}^{\delta_w,\delta_r,\Lambda,\Sigma_0}_{\mathsf{Constrained\text{-}Ledger}}$, which constrains the known global functionality $\mathcal{G}^{\delta}_{\mathsf{Ledger}}$ [26], making it more realistic than previous ledger functionalities, combined with a tick global functionality $\mathcal{G}_{Ticker}$ as a clock. The combination of $\mathcal{G}^{\delta_w,\delta_r,\Lambda,\Sigma_0}_{\mathsf{Constrained\text{-}Ledger}}$ and $\mathcal{G}_{Ticker}$ provides a solid notion of time very suitable for layer-2 protocols given their dependency on timely honest user action (for dispute phases, for example). Our proposed constrained ledger functionality introduces two different parameters for modeling time delays, *i.e.*, δ_w and δ_r, respectively for write and read transactions (while $\mathcal{G}^{\delta}_{\mathsf{Ledger}}$ only has one, the time delay parameter δ for both read and write of the transactions).

Given the early described more realistic setting, with $\mathcal{G}^{\delta_w,\delta_r,\Lambda,\Sigma_0}_{\mathsf{Constrained\text{-}Ledger}}$ and $\mathcal{G}_{Ticker}$, we introduced a second major technical contribution of this work: the general multi-party isomorphic state channel functionality $\mathcal{F}^{SM}_{CH}$. As a core, and driving, intuition in the design of $\mathcal{F}^{SM}_{CH}$ is its natural abstraction from the underpinnings of the concrete layer-2 channel protocols. This desirable feature is

accomplished by keeping tracking only of the acknowledged and confirmed states of the channel, which offers plenty of room for novel concrete protocols yet to be proposed. In particular, the functionality is purposely designed to be both state and ledger agnostic (*i.e.*, independent of UTxO or account based ledgers). In order to show the practicability of our framework, we proved the security of an existing protocol, namely Hydra Protocol Π_{Hydra}, by showing that it UC-realizes $\mathcal{F}_{CH}^{\text{HydraSM}}$ in the $\mathcal{G}_{\mathsf{Constrained\text{-}Ledger}}^{\delta_w,\delta_r,\Lambda,\Sigma_0}$ and $\mathcal{G}_{Ticker}$ global models. Demonstrating the composability of Π_{Hydra} and the practicality of $\mathcal{F}_{CH}$.

As an unexpected feature of our functionality that demonstrates its flexibility is that we showed it can be further tailored, albeit making it specialized for the UTXO Model, to support an extended version of Π_{Hydra}, named Π_{Hydra}^{HT}. This extension includes the settlement of *hanging transactions*, *i.e.* transactions which were proposed but not yet confirmed in the setting of the layer-2 protocol, and was only outlined by the authors of Π_{Hydra}. Until our study, it was not concretely presented to the best of our knowledge. As an extra contribution of this work, we fully and formally introduced Π_{Hydra}^{HT}, and proposed a UTXO tailored version of $\mathcal{F}_{CH}^{\text{HydraSM}}$ named Π_{Hydra}^{HT}. Hence we showed Π_{Hydra}^{HT} UC-realizes $\mathcal{F}_{CH\text{-}HT}^{\text{HydraHTSM}}$ (in a similar fashion of the earlier described case of Π_{Hydra} and $\mathcal{F}_{CH}^{\text{HydraSM}}$).

We understand as potential venues for future work can include (1) further demonstrating versatility of $\mathcal{F}_{CH}^{SM}$ by demonstrating that other common channel constructions can implement $\mathcal{F}_{CH}^{SM}$, notably [32]; or (2) utilize $\mathcal{F}_{CH}^{SM}$ as a starting point to create functionalities for protocols built on top of channel construction, *e.g.*, virtual channel constructions [22].

References

1. Raiden network. raiden.network. Accessed: 2018-09-03.
2. Speedymurmurs. https://crysp.uwaterloo.ca/software/speedymurmurs. Accessed: 2018-10-17.
3. Lukas Aumayr, Matteo Maffei, Oguzhan Ersoy, Andreas Erwig, Sebastian Faust, Siavash Riahi, Kristina Hostáková, and Pedro Moreno-Sanchez. Bitcoin-compatible virtual channels. In *2021 IEEE Symposium on Security and Privacy*, pages 901–918. IEEE Computer Society Press, May 2021.
4. Georgia Avarikioti, Felix Laufenberg, Jakub Sliwinski, Yuyi Wang, and Roger Wattenhofer. Incentivizing payment channel watchtowers. In *Scaling Bitcoin*, 2018.
5. Carsten Baum, Bernardo David, Rafael Dowsley, Ravi Kishore, Jesper Buus Nielsen, and Sabine Oechsner. CRAFT: Composable randomness beacons and output-independent abort MPC from time. In Alexandra Boldyreva and Vladimir Kolesnikov, editors, *PKC 2023, Part I*, volume 13940 of *LNCS*, pages 439–470. Springer, Cham, May 2023.
6. Carsten Baum, Bernardo David, Rafael Dowsley, Jesper Buus Nielsen, and Sabine Oechsner. TARDIS: A foundation of time-lock puzzles in UC. In Anne Canteaut and François-Xavier Standaert, editors, *EUROCRYPT 2021, Part III*, volume 12698 of *LNCS*, pages 429–459. Springer, Cham, October 2021.
7. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.
8. Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. Universally composable security with global setup. In Salil P. Vadhan, editor, *TCC 2007*, volume 4392 of *LNCS*, pages 61–85. Springer, Berlin, Heidelberg, February 2007.
9. Manuel M. T. Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Michael Peyton Jones, and Philip Wadler. The extended UTXO model. In Matthew Bernhard, Andrea Bracciali, L. Jean Camp, Shin'ichiro Matsuo, Alana Maurushat, Peter B. Rønne, and Massimiliano Sala, editors, *FC 2020 Workshops*, volume 12063 of *LNCS*, pages 525–539. Springer, Cham, February 2020.
10. Manuel M. T. Chakravarty, Sandro Coretti, Matthias Fitzi, Peter Gazi, Philipp Kant, Aggelos Kiayias, and Alexander Russell. Fast isomorphic state channels. In Nikita Borisov and Claudia Díaz, editors, *FC 2021, Part II*, volume 12675 of *LNCS*, pages 339–358. Springer, Berlin, Heidelberg, March 2021.
11. Stefan Dziembowski, Lisa Eckey, Sebastian Faust, Julia Hesse, and Kristina Hostáková. Multi-party virtual state channels. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part I*, volume 11476 of *LNCS*, pages 625–656. Springer, Cham, May 2019.
12. Stefan Dziembowski, Lisa Eckey, Sebastian Faust, and Daniel Malinowski. Perun: Virtual payment hubs over cryptocurrencies. In *2019 IEEE Symposium on Security and Privacy*, pages 106–123. IEEE Computer Society Press, May 2019.
13. Stefan Dziembowski, Sebastian Faust, and Kristina Hostáková. General state channel networks. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 949–966. ACM Press, October 2018.
14. Karim Eldefrawy, Benjamin Terner, and Moti Yung. Composing timed cryptographic protocols: Foundations and applications. Cryptology ePrint Archive, Paper 2024/676, 2024. https://eprint.iacr.org/2024/676.
15. Roberto Canonico Giovanni Di Stasi, Stefano Avallone and Giorgio Ventre. Routing payments on the lightning network.

http://wpage.unina.it/giovanni.distasi/pub/blockchain2018-main.pdf, 2018. Accessed: 2019-03-28.
16. Oded Goldreich. Foundations of cryptography: Basic tools, vol. 1, 2001.
17. Shafi Goldwasser, Silvio Micali, and Chales Rackoff. The knowledge complexity of interactive proof-systems. In *Providing sound foundations for cryptography: On the work of shafi goldwasser and silvio micali*, pages 203–225. 2019.
18. Lewis Gudgeon, Pedro Moreno-Sanchez, Stefanie Roos, Patrick McCorry, and Arthur Gervais. SoK: Layer-two blockchain protocols. In Joseph Bonneau and Nadia Heninger, editors, *FC 2020*, volume 12059 of *LNCS*, pages 201–226. Springer, Cham, February 2020.
19. Nick Jonas, Poelstra Andrew, and Sanders Gregory. Plasma: Scalable autonomous smart contracts. https://blockstream.com/assets/downloads/pdf/liquid-whitepaper.pdf, 2020.
20. Poon Joseph and Buterin Vitalik. Plasma: Scalable autonomous smart contracts. https://plasma.io/plasma.pdf, 2017.
21. Maxim Jourenko, Kanta Kurazumi, Mario Larangeira, and Keisuke Tanaka. Sok: A taxonomy for layer-2 scalability related protocols for cryptocurrencies. Cryptology ePrint Archive, Report 2019/352, 2019. https://eprint.iacr.org/2019/352.
22. Maxim Jourenko and Mario Larangeira. State machines across isomorphic layer 2 ledgers. In Foteini Baldimtsi and Christian Cachin, editors, *FC 2023, Part II*, volume 13951 of *LNCS*, pages 75–91. Springer, Cham, May 2023.
23. Maxim Jourenko, Mario Larangeira, and Keisuke Tanaka. Lightweight virtual payment channels. In Stephan Krenn, Haya Shulman, and Serge Vaudenay, editors, *CANS 20*, volume 12579 of *LNCS*, pages 365–384. Springer, Cham, December 2020.
24. Maxim Jourenko, Mario Larangeira, and Keisuke Tanaka. Payment trees: Low collateral payments for payment channel networks. In Nikita Borisov and Claudia Díaz, editors, *FC 2021, Part II*, volume 12675 of *LNCS*, pages 189–208. Springer, Berlin, Heidelberg, March 2021.
25. Harry Kalodner, Steven Goldfeder, Xiaoqi Chen, S Matthew Weinberg, and Edward W Felten. Arbitrum: Scalable, private smart contracts. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1353–1370, 2018.
26. Thomas Kerber, Aggelos Kiayias, and Markulf Kohlweiss. Kachina–foundations of private smart contracts. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*, pages 1–16. IEEE, 2021.
27. Rami Khalil and Arthur Gervais. Revive: Rebalancing off-blockchain payment networks. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 439–453. ACM, 2017.
28. Giulio Malavolta, Pedro Moreno-Sanchez, Aniket Kate, and Matteo Maffei. Silentwhispers: Enforcing security and privacy in credit networks. NDSS, 2017.
29. Andrew Miller, Iddo Bentov, Surya Bakshi, Ranjit Kumaresan, and Patrick McCorry. Sprites and state channels: Payment networks that go faster than lightning. In Ian Goldberg and Tyler Moore, editors, *FC 2019*, volume 11598 of *LNCS*, pages 508–526. Springer, Cham, February 2019.
30. Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.
31. Olaoluwa Osuntokun. [lightning-dev] amp: Atomic multi-path payments over lightning. https://lists.linuxfoundation.org/pipermail/lightning-dev/2018-February/000993.html. Accessed: 2018-10-11.
32. Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments. *See https://lightning. network/lightning-network-paper. pdf*, 2016.

33. Pavel Prihodko, Slava Zhigulin, Mykola Sahno, Aleksei Ostrovskiy, and Olaoluwa Osuntokun. Flare: An approach to routing in lightning network. 2016.
34. Srinivas Vutukury and Jose Joaquin Garcia-Luna-Aceves. Mdva: A distance-vector multipath routing protocol. In *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 1, pages 557–564. IEEE, 2001.
35. Douglas Wikström. Simplified universal composability framework. In *Theory of Cryptography Conference*, pages 566–595. Springer, 2015.

A Multiparty Isomorphic State Channel Protocol Π_{Hydra}

The Hydra Protocol [10] is a multiparty state channel executed between $n \in \mathbb{N}$ parties and built upon a ledger that operates under the UTxO paradigm. The Hydra Protocol consists of two parts, (1) an on-chain state machine and (2) the off-chain head protocol. $\mathcal{G}_{\mathsf{Constrained\text{-}Ledger}}^{\delta_w,\delta_\tau,\Lambda,\Sigma_0}$, using the EUTxO Model, acts as a trusted third party to correctly execute the on-chain state machine, whereas the off-chain head protocol is executed by a set of parties that can be corrupted by an adversary. Let h be a cryptographic hash function *s.t.*, $y = h(X)$ is the hash value of a preimage x used as input to h.

Hydra utilizes snapshot objects $\mathcal{U} = (s, U, h_U, T, S, \sigma)$ where $s \in \mathbb{N}$ is a snapshot sequence number, U is a set of UTxO, $h_U = h(U)$, T is a list of transactions, S is an array of signatures where $S[i]$ contains either a signature of p_i or $\perp$ and σ is an aggregate signature of all signatures in S or $\perp$. Let $\mathcal{U}_i$ be the snapshot where $\mathcal{U}.s = i$. Then it must hold that $\mathcal{U}_{i+1}.U$ results from applying $\mathcal{U}_{i+1}.T$ on $\mathcal{U}_i.U$, *i.e.*, T specifies how the UTxO set in a snapshot relates to the UTxO set from its predecessor. A snapshot object can be created using the function $\mathsf{snObj}(s, U, T) = \mathcal{U}_s = (s, U, h(U), T, S_\perp, \perp)$ where $S_\perp$ is an array with n values and each entry being $\perp$. Moreover, Hydra utilizes transaction objects $\mathsf{tx} = (i, \tau, h_\tau, S, \sigma)$ where $i \in \mathbb{N}, 1 \leq i \leq n$ is the index of p_i, τ is a transaction, $h_\tau = h(\tau)$ and S and σ are analogous to the respective entries in snapshot objects. A transaction object is initialized with $\mathsf{txObj}(i, \tau) = (i, \tau, h(\tau), S_\perp, \perp)$.

From now we briefly review the Hydra Head Protocol without hanging transactions framed within the UC framework in the next figures, namely Figures 20, 21, 22, 23 and 24. For a full description of the protocol we refer the reader to [10]. Note that, Hydra is fully specified only without handling of hanging transactions, and hanging transactions have been only described abstractly. The formal specification of the Hydra Protocol with hanging transactions is provided in Section 6.3 instead.

Leader Selection Function. The protocol utilizes a leader selection function $\mathsf{LEADER} : \mathbb{N} \to \gamma_p$ which on input of a sequence number $s \in \mathbb{N}$ outputs a party $p \in \gamma_p$ which is permitted to propose the s-th state update. The Leader selection function for the Hydra Protocol [10] is picked such that parties become leader in a round-robin manner.

Hydra Channel Protocol Π_{Hydra} (Part 1)

Variables and Initialization:

- State machine transaction function ζ initialized to implement the Hydra State machine on $\mathcal{G}^{\delta_w,\delta_r,\Lambda,\Sigma_0}_{\mathsf{Constrained\text{-}Ledger}}$
- $s^s = s^c = 0$: Seen and confirmed snapshot number
- $\mathcal{U}^s = \mathcal{U}^c = \bot$: Seen and confirmed snapshot objects
- $\mathcal{L}^s = \mathcal{L}^c = \varnothing$: Seen and confirmed set of UTxO
- $\mathcal{T}^s = \mathcal{T}^c = \varnothing$: Seen and confirmed set of transaction objects
- $\mathcal{R}^s = \mathcal{R}^c = \varnothing$: Seen and confirmed sets of rejected transaction objects
- Internal channel state $\gamma_S = \mathsf{NEW}$
- List of signature verification keys $\mathcal{K} = \epsilon$ where $\mathcal{K}[i]$ is key of p_i
- Secret Key $\mathsf{sk} = \bot$
- Channel participants $\gamma_p = \varnothing$
- Duration of the contestation phase $t_c = 0$
- This party's index $i_P = 0$
- Initial contribution to the channel $\Sigma = \bot$
- List of pending instructions to wait for $W = \epsilon$
- Set of party indexes $I = \varnothing$
- Signature message pairs $S = \varnothing$ used for Hydra setup
- View of the ledger $\Omega = \epsilon$
- Initial snapshot $\mathcal{U}_0 = \varnothing$ and committed party states $\Sigma_{\mathsf{comm}} = \varnothing$
- Ordered list of transactions $L^{\mathsf{TX}} = \epsilon$ included in confirmed snapshots

Behavior: On activation: Before proceeding, first send (READ) to $\mathcal{G}^{\delta_w,\delta_r,\Lambda,\Sigma_0}_{\mathsf{Constrained\text{-}Ledger}}$, receiving reply Ω'. If $\Omega \subseteq \Omega'$ let D where $\Omega' = \Omega||D$ holds, then $\mathsf{Call}(\mathsf{OnTx}, D)$. Moreover, before returning control, first executes procedure ExecWait() and afterwards send (ACTIVATED) to $\mathcal{G}_{Ticker}$.

Procedure **Wait**(instr, param)

1: $W \leftarrow W||(\mathsf{instr}, \mathsf{param})$

Procedure **ExecWait()**

1: $W' \leftarrow W$, $W \leftarrow \epsilon$
2: **for** (instr, param) $\in W'$ **do**
3: $\quad \mathsf{Call}(\mathsf{instr}, \mathsf{param})$

Procedure **Submit** (type, param)

1: $\tau \leftarrow \zeta(type, p_i, \mathsf{param}, \bot)$
2: **if** $\tau = \bot$ **then return**
3: $\sigma \leftarrow \mathsf{Sign}(\mathsf{sk}, \tau)$
4: $\tau \leftarrow \zeta(type, p_i, \mathsf{param}, \sigma)$
5: Send (SUBMIT, τ) to $\mathcal{G}^{\delta_w,\delta_r,\Lambda,\Sigma_0}_{\mathsf{Constrained\text{-}Ledger}}$

Fig. 20. The Hydra Protocol allows n parties to exchange transactions directly without interaction with the ledger, which significantly speeds up the transaction settlement in comparison to a on chain design. Here we denote parameters, initialization and some auxiliary procedures. Furthermore, we assume the parties p_i are equipped with a signature scheme $\mathsf{DS} = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ as given by Definition 1.

Hydra Channel Protocol Π_{Hydra} (Part 2)

Setup: Upon input $(\mathsf{SETUP}, sid, \gamma_p', t_c', \Sigma_i', i')$

1: **if** $t_c' \leq |\gamma_p'|(\delta_r + \delta_w)$ or this *SETUP* message has been received before **then** drop message
2: Create a fresh key pair $(\mathsf{sk}', \mathsf{vk})$ using algorithm GEN
3: $\mathcal{K}[i'] \leftarrow \mathsf{vk}$, $\mathsf{sk} \leftarrow \mathsf{sk}'$
4: $\gamma_p \leftarrow \gamma_p'$, $t_c \leftarrow t_c'$, $\Sigma \leftarrow \Sigma_i'$, $i_P \leftarrow i'$
5: **for all** $p' \in \gamma_p$ **do**
6: Send $(\mathsf{ReqSetup}, sid, \mathsf{vk}, \Sigma, i)$ to p'

Request Setup: Upon input $(\mathsf{ReqSetup}, sid, \mathsf{vk}, \Sigma, j)$ from party p

1: **if** $\gamma_p = \varnothing$ **then** $\mathsf{Call}(\mathsf{Wait}, (\mathsf{ReqSetup}, \mathsf{param}))$, where $\mathsf{param} = (\mathsf{vk}, \Sigma, j)$ **return**
2: **if** $j \in I$ **then return** ▷ $j \in I$ if $\mathsf{ReqSetup}$ already received from p_j
3: $\mathcal{K}[j] \leftarrow \mathsf{vk}$, $I \leftarrow I \cup \{j\}$, $p_j \leftarrow p$
4: **if** $|\gamma_p| = |I|$ **then** ▷ Received $\mathsf{ReqSetup}$ from all parties
5: $m \leftarrow (\bigcup_{k \in I} \Sigma_k, t_c, K)$
6: $\sigma \leftarrow \mathsf{SIGN}(m, \mathsf{sk})$
7: **for all** $p' \in \gamma_p$ **do**
8: Send $(\mathsf{AckSetup}, sid, \sigma)$ to p'

Acknowledge Setup: Upon input $(\mathsf{AckSetup}, sid, \sigma_j)$ from party p_j

1: $M \leftarrow M \cup \{(p_j, \sigma_j)\}$
2: $m \leftarrow (\bigcup_{k \in I} \Sigma_k, t_c, K)$
3: **if** $\mathsf{VERIFY}(\mathsf{vk}_j, m, \sigma_j) = 1$ **then** $S \leftarrow S \cup \{(m, \sigma_j)\}$
4: **if** $|S| = |\gamma_p| \wedge \nexists i' \in I : i' < i_P$ **then** ▷ Executed if p_{i_P} is canonically first
5: $\mathcal{U}_0 \leftarrow \bigcup_{i \in I} \Sigma_i$; $\mathcal{L}^c \leftarrow \mathcal{U}_0$
6: $\mathsf{param} \leftarrow \{m, S_\sigma\}$
7: $\mathsf{Call}(\mathsf{Submit}, \mathsf{INIT}, \mathsf{param})$

New: Upon input $(\mathsf{NEW}, sid, \tau)$ from $\mathcal{Z}$

1: **if** $\gamma_S \neq \mathsf{OPEN} \wedge \tau \notin \mathcal{R}^c$ **then return**
2: Send $(\mathsf{EVAL}, \mathcal{L}^c, \epsilon, \tau)$ to $\mathcal{G}^{\delta_w, \delta_r, \Lambda, \Sigma_0}_{\mathsf{Constrained\text{-}Ledger}}$ receiving b
3: **if** $b = \top$ **then** Send $(\mathsf{ReqTx}, sid, \tau)$ to $p \in \gamma_p$

NewSt: Upon input $(\mathsf{NEW_ST}, sid, T, R)$ from $\mathcal{Z}$

1: **if** $\gamma_S \neq \mathsf{OPEN} \vee \mathcal{U}^s \neq \varnothing$ **then return**
2: **if** $((R \cap \mathcal{T}^c) \neq \varnothing \wedge T \not\subseteq \mathcal{T}^c)$ **then return**
3: Send $(\mathsf{ReqSn}, sid, s^c + 1, T, R)$ to each $p \in \gamma_p$

Close: Upon input (CLOSE, sid) from $\mathcal{Z}$

1: $\gamma_S \leftarrow \mathsf{CLOSED}$
2: $\mathsf{param}' \leftarrow (s^c, \mathcal{U}_C)$
3: $\mathsf{Call}(\mathsf{Submit}, \mathsf{CONTEST}, \mathsf{param}')$

Fig. 21. Here we define protocol setup procedures and the protocol interface, that rule new snapshot (**Sn**), *i.e.*, close, abort, and other specific Hydra operations. p_j indicates another party executing protocol Π_{Hydra}.

Hydra Channel Protocol Π_{Hydra} (Part 3)

Abort Upon input (ABORT, sid) from $\mathcal{Z}$

1: $\gamma_S \leftarrow \mathsf{CLOSED}$
2: $\mathsf{Call}(\mathsf{Submit}, \mathsf{ABORT}, \varnothing)$

Fanout Upon input (FANOUT, sid) from $\mathcal{Z}$, run $\mathsf{Call}(\mathsf{Submit}, \mathsf{FANOUT}, \varnothing)$.

Procedure **OnTx**(D) For each $\tau \in D$ in order, do

1: **if** $\tau = \zeta(\mathsf{INIT}, \mathsf{param}, \dots)$ **then** $\mathsf{Call}(\mathsf{ChainInitial}, \tau, \mathsf{param})$
2: **if** $\tau = \zeta(\mathsf{COMM}, \mathsf{param}, \dots)$ **then** $\mathsf{Call}(\mathsf{ChainCommit}, \tau, \mathsf{param})$
3: **if** $\tau = \zeta(\mathsf{COLL}, \mathsf{param}, \dots)$ **then** $\mathsf{Call}(\mathsf{ChainCollectCom}, \tau, \mathsf{param})$
4: **if** $\tau = \zeta(\mathsf{CLOSE}, \mathsf{param}, \dots)$ **then** $\mathsf{Call}(\mathsf{ChainClose}, \tau, \mathsf{param})$
5: **if** $\tau = \zeta(\mathsf{CONTEST}, \mathsf{param}, \dots)$ **then** $\mathsf{Call}(\mathsf{ChainContest}, \tau, \mathsf{param})$
6: **if** $\tau = \zeta(\mathsf{FANOUT}, \mathsf{param}, \dots)$ **then** $\mathsf{Call}(\mathsf{ChainClosed}, \tau, \mathsf{param})$
7: **if** $\tau = \zeta(\mathsf{ABORT}, \mathsf{param}, \dots)$ **then** $\mathsf{Call}(\mathsf{ChainClosed}, \tau, \mathsf{param})$

Procedure $\mathsf{ChainInitial}(\tau, \mathsf{param})$

1: $(m, L_{\mathsf{vk}}, L_\sigma) \leftarrow \Lambda.\mathsf{MS}(\Sigma_0', \Omega', \tau)$
2: **if** $L_{\mathsf{vk}} \neq \mathcal{K}$ **then return**
3: $\mathsf{param}' \leftarrow \{\Sigma\}$
4: $\mathsf{Call}(\mathsf{Submit}, \mathsf{COMM}, \mathsf{param}')$

Procedure $\mathsf{ChainCommit}(\tau, \mathsf{param})$

1: **if** $\Sigma \in \mathsf{param}$ **then**
2: $\quad \Sigma_{\mathsf{comm}} \leftarrow \Sigma_{\mathsf{comm}} \cup \Sigma$
3: $\quad$ **if** $\Sigma_{\mathsf{comm}} = \mathcal{U}_0$ **then** $\mathsf{Call}(\mathsf{Submit}, \mathsf{COLL}, \mathsf{param}')$

Procedure $\mathsf{ChainCollect}(\tau, \mathsf{param})$

1: $\gamma_S \leftarrow \mathsf{OPEN}$
2: Output $(\mathsf{OPEN}, sid, \mathcal{U}_0)$ to $\mathcal{Z}$

Procedure $\mathsf{ChainClose}(\tau, \mathsf{param})$

1: $\mathsf{Call}(\mathsf{ChainContest}, \tau, \mathsf{param})$
2: Output $(\mathsf{CLOSING}, sid)$ to $\mathcal{Z}$

Procedure $\mathsf{ChainContest}(\tau, \mathsf{param})$

1: $\gamma_S \leftarrow \mathsf{CLOSING}$
2: $(s, \mathcal{U}) \leftarrow \mathsf{param}$
3: **if** $s < s^c$ **then**
4: $\quad \mathsf{param}' \leftarrow (s^c, \mathcal{U}_C)$
5: $\quad \mathsf{Call}(\mathsf{Submit}, \mathsf{CONTEST}, \mathsf{param}')$

Procedure $\mathsf{ChainClosed}(\tau, \mathsf{param})$

1: $\gamma_S \leftarrow \mathsf{CLOSED}$
2: Output (CLOSED, sid) to $\mathcal{Z}$

Fig. 22. The above procedures are used to interpret the channel's state on $\mathcal{G}^{\delta_w, \delta_r, \Lambda, \Sigma_0}_{\mathsf{Constrained\text{-}Ledger}}$ and react to changes of it. Note that the chain related procedures, *i.e.*, **ChainInitial**, **ChainCommit**, **ChainClose**, and **ChainContest**, rely on the procedure **Submit** described in Fig. 20.

Hydra Channel Protocol Π_{Hydra} (Part 4)

ReqTx: Upon receipt of $(\mathsf{ReqTx}, sid, \tau)$ from p_j

if $\gamma_S \neq \mathsf{OPEN} \wedge \tau \notin \mathcal{R}^c$ **then return**
if $p_j \notin \gamma_p$ **then return**
Send $(\mathsf{EVAL}, \mathcal{L}^s, \epsilon, \tau)$ to $\mathcal{G}^{\delta_w,\delta_r,\Lambda,\Sigma_0}_{\mathsf{Constrained\text{-}Ledger}}$ receiving b
if $b = \top$ **then**
    $h \leftarrow H(\tau)$ ▷ H is a cryptographic hash function
    $\mathcal{T}^s[h] \leftarrow \mathsf{txObj}(j, \tau)$
    $\mathcal{L}^s \leftarrow \Lambda.\mathsf{STATE}(\mathcal{L}^s, \epsilon||\tau)$
    $\mathsf{Call}(\mathsf{SeeTx}, \tau, p_j)$

Procedure $\mathsf{SeeTx}(\tau,\ p)$:

if $\gamma_S \neq \mathsf{OPEN} \wedge \tau \notin \mathcal{R}^c$ **then return**
if $\Lambda.\mathsf{EVAL}(\mathcal{L}^c, \epsilon, \tau) = \top$ **then**
    $\sigma_i \leftarrow \mathsf{SIGN}(\mathsf{sk}, \tau)$
    Send $(\mathsf{AckTx}, sid, \tau, \sigma_i)$ to p
else
    $\mathsf{Call}(\mathsf{WAIT}, \mathsf{SeeTx}, (\tau, p_j))$ **return**

AckTx: Upon receipt of $(\mathsf{AckTx}, sid, \tau, \sigma_j)$ from p_j

if $\gamma_S \neq \mathsf{OPEN} \wedge \tau \notin \mathcal{R}^c$ **then return**
if $p_j \notin \gamma_p$ **then return**
if $\mathcal{T}[H(\tau)].i \neq i_P$ **then return** ▷ Recipient must have issued τ
if $\mathsf{VERIFY}(\mathsf{vk}_j, \tau, \sigma_j) \neq 1$ **then return**
$\mathcal{T}[H(\tau)].S[j] \leftarrow \sigma_j$
if $\forall i' \in I : \mathcal{T}[H(\tau)].S[i'] \neq \perp$ **then**
    Send $(\mathsf{ConfTx}, sid, \tau, \mathcal{T}[H(\tau)].S)$ to each $p \in \gamma_p$

ConfTx: Upon receiving $(\mathsf{ConfTx}, sid, \tau, \mathcal{T}[H(\tau)].S)$ from p_j

if $\gamma_S \neq \mathsf{OPEN} \wedge \tau \notin \mathcal{R}^c$ **then return**
if $p_j \notin \gamma_p$ **then return**
$(m, L_{\mathsf{vk}}, L_\sigma) \leftarrow \Lambda.\mathsf{MS}(\mathcal{L}^c, \epsilon, \tau)$
$\mathcal{T}^c[H(\tau)] \leftarrow \mathcal{T}^s[H(\tau)]$
$\mathcal{T}^s \leftarrow \mathcal{T}^s \setminus \mathcal{T}^s[H(\tau)]$

Fig. 23. The transaction (**Tx**) 3-round-confirmation (**Req**, **Ack** and **Conf**) protocol as run by the p_j. The protocol relies on the procedure SeeTx, which relies on the interface WAIT from Fig. 20.

Hydra Channel Protocol Π_{Hydra} (Part 5)

ReqSn: Upon receipt of $(\mathsf{ReqSn}, sid, s, T, R)$ from p_j

1: **if** $p_j \neq \mathsf{leader}(s)$ **then return**
2: **if** $\gamma_S \neq \mathsf{OPEN} \vee s < s^c \vee ((R \cap \mathcal{T}^c) \neq \varnothing)$ **then return**
3: **if** $s \neq s^c + 1 \vee s^s \neq s^c \vee T \not\subseteq \mathcal{T}^c$ **then** $\mathsf{Call}(\mathsf{WAIT}, (\mathsf{ReqSn}, s, T, R))$ **return**
4: $\mathcal{R}^s \leftarrow \mathcal{R}^s \cup R$
5: $\mathcal{T}^s \leftarrow (\mathcal{T}^s \setminus \mathcal{R}^s)$
6: $\mathcal{L}^s \leftarrow \Lambda.\mathsf{STATE}(\mathcal{L}^c, \mathcal{T}^s)$
7: $s^s \leftarrow s^s + 1$
8: $\mathcal{U}^s \leftarrow \mathsf{snObj}(s^s, \mathcal{U}^c.U, T, R)$
9: $\sigma \leftarrow \mathsf{SIGN}(\mathsf{sk}, \mathcal{U}^s)$
10: Send $(\mathsf{AckSn}, sid, s, \sigma)$ to p_j

AckSn: Upon receipt of $(\mathsf{AckSn}, sid, s, \mathcal{T})$ from p_j

1: **if** $\gamma_S \neq \mathsf{OPEN}$ **then return**
2: **if** $s \neq s^s$ **then return**
3: **if** $\mathcal{U}.S[j] \neq \perp$ **then return**
4: **if** $\mathsf{VERIFY}(\mathsf{vk}_j, \mathcal{U}^s, \sigma) \neq 1$ **then return**
5: $\mathcal{U}.S[j] \leftarrow \sigma$
6: **if** $\forall j \in I : \mathcal{U}.S[j] \neq \perp$ **then**
7: Send $(\mathsf{ConfSn}, sid, s, \mathcal{U})$

ConfSn: Upon receipt of $(\mathsf{ConfSn}, sid, s, \mathcal{U})$ from p_j

1: **if** $\gamma_S \neq \mathsf{OPEN}$ **then return**
2: **if** $s \neq s^s \vee s^s = s^c$ **then return**
3: **for** $\mathsf{vk} \in \mathcal{K}$ **do**
4: **if** $\nexists \sigma \in \mathcal{U}.S : \mathsf{VERIFY}(\mathsf{vk}, \mathcal{U}, \sigma)$ **then return**
5: $s^c \leftarrow s^s$; $\mathcal{U}^c \leftarrow \mathcal{U}^s$; $\mathcal{U}^s \leftarrow \varnothing$; $\mathcal{L}^c \leftarrow \mathcal{L}^s$
6: $\mathcal{T}^c \leftarrow \mathcal{T}^s$; $\mathcal{R}^c \leftarrow \mathcal{R}^s$
7: $L^{\mathsf{TX}} \leftarrow L^{\mathsf{TX}} || \mathcal{U}.T$

State Request: Upon receiving input (STATE, sid) from $\mathcal{Z}$ do

1: Send $(\mathsf{STATE}, sid, L^{\mathsf{TX}}, \mathcal{U}^s.T, \mathcal{R}^c \cup \mathcal{R}^s, \mathcal{T}^c, \mathcal{T}^s)$ to $\mathcal{Z}$

Fig. 24. The snapshot (**Sn**) 3-round-confirmation (**Req**, **Ack** and **Conf**) protocol as run by p_j, similarly to Fig. 23 for transactions. In the end, it outputs the confirmation of the new snapshop.