Perun: Virtual Payment Hubs over Cryptocurrencies

Stefan Dziembowski
University of Warsaw

Lisa Eckey
TU Darmstadt

Sebastian Faust
TU Darmstadt

Daniel Malinowski
University of Warsaw

***Abstract*—Payment channels emerged recently as an efficient method for performing cheap *micropayments* in cryptocurrencies. In contrast to traditional on-chain transactions, payment channels have the advantage that they allow for nearly unlimited number of transactions between parties without involving the blockchain. In this work, we introduce *Perun*, an off-chain channel system that offers a new method for connecting channels that is more efficient than the existing technique of "routing transactions" over multiple channels. To this end, Perun introduces a technique called "virtual payment channels" that avoids involvement of the intermediary for each individual payment. In this paper we formally model and prove security of this technique in the case of one intermediary, who can be viewed as a "payment hub" that has direct channels with several parties. Our scheme works over any cryptocurrency that provides Turing-complete smart contracts. As a proof of concept, we implemented Perun's smart contracts in *Ethereum*.**

I. Introduction

Decentralized cryptocurrencies have gained great popularity over the last 10 years, and provide a payment infrastructure without any central authority regulating transactions. In addition, cryptocurrencies have helped to accelerate deployment of disruptive technology such as smart contracts, which use program code to enforce complex agreements. The core innovation empowering decentralized cryptocurrencies is a consensus mechanism for maintaining a distributed ledger – the so-called *blockchain*. Because the entire state of the blockchain is replicated among thousands of users, the number of transactions and the speed at which they are processed is limited when compared to centralized systems. For instance, the most prominent blockchain-based cryptocurrency Bitcoin processes up to 7 transactions per second and requires on average 10 minutes to confirm new transactions.

The scalability problems of blockchain-based cryptocurrencies are drastically amplified with the emergence of microtransactions that require users to transfer small amounts of money between each other. Typically, such microtransactions have to be executed instantaneously, which is a problem in cryptocurrencies, where confirmation can take up to several minutes. Moreover, posting transactions on the ledger results into fees, which are much higher than the value of a microtransaction. Therefore, it seems unlikely that current cryptocurrencies can support microtransactions, and the many applications they offer.

An exciting proposal to address the above challenges is a technology called *payment channels* (see also https://en.bitcoin.it/wiki/Payment_channels and the work of Decker and Wattenhofer [3]), which allows two parties, Alice and Bob, say, to rapidly exchange money between each other via so-called "off-chain" transactions. In contrast to the on-chain transactions, the off-chain ones enable users to exchange coins without directly interacting with the ledger. This works as follows. First, a channel is *opened* and both Alice and Bob deposit x_a and x_b (respectively) coins into it (denote this channel with β). Then, the channel mechanics let the parties freely change the distribution of $x_A + x_B$ between each other, thereby enabling payments between Alice and Bob (see Sect. III-A1 for more on this). At any moment in time, each party can decide to *close* the channel β and get her cash transferred to the ledger. In particular, she can do it if she enters into a dispute with the other party. Only opening and closing require interaction with the ledger. On the other hand channel updates are preformed *without* interacting with the ledger, and hence they can be executed any number of times, very quickly and at no cost.

The concept of payment channels has been extended to so-called payment *networks*, which enable users to route transactions via intermediary hubs. An example of such a system is the Lightning network [11] originally designed for Bitcoin. In this system payments are routed over the network in the following way. Suppose two parties, Alice and an *intermediary* called Ingrid, established a channel denoted "β_A", and Ingrid also has a channel with Bob, denoted "β_B" (but Alice and Bob do not have a payment channel between each other). Then Alice can perform a micropayment for y coins to Bob via Ingrid. In [11] each such money transfer requires explicit confirmation by Ingrid. This has the disadvantage of introducing latency and adding costs for fees paid to Ingrid.

A. Summary of our contribution and its applications.

The main contribution of this work is to address the aforementioned shortcoming with a concept that we call *virtual channels*. Again, suppose Alice and Bob are both connected by a channel created over the blockchain with an intermediary payment hub Ingrid (we call such channels that are built directly over the ledger the *ledger channels*). Given these ledger channels, we can establish a virtual channel that establishes a direct (virtual) link between Alice and Bob, where the intermediary Ingrid does not need to get involved in each payment. This significantly reduces latency and costs, and moreover is beneficial for privacy, because Ingrid cannot observe the individual money transfers between Alice and Bob.

We call our system *Perun*[1], which we fully specify in this work, formalize its security properties using ideal/real world paradigm in the UC framework of Canetti [1] and finally prove its security according to our definition.

While our system works over any cryptocurrency which allows Turing complete smart contracts (we give a short introduction to this concept in Sect. II-1), we demonstrate the feasibility of our proposal by providing a prototype implementation of the contracts underlying the Perun channel system in *Solidity* (see Sec. VI), one of the main languages supported by the Ethereum cryptocurrency.

A natural application of Perun is to provide a very fast way to stream tiny payments. For example, consider the situation when a client Alice pays for using WiFi to some Internet provider Bob (and they both have ledger channels with an intermediary Ingrid). By using our approach Ingrid is involved in the communication between Alice and Bob only when the session starts and when it ends. Another natural application of our technique is the Internet of Things. Due to the cost pressure to reduce the power consumption these devices will often only be connected via some short-range communication technology, and will minimize the interaction with remote devices. Hence, routing every payment via a third party server may not be an option in such situations. Our technique removes the need for such interaction. Another related scenario where our solution can be applied is when the payment intermediary cannot be assumed to be always available (for example in the vehicular ad hoc networks).

Organization of the paper. The basic notation and terminology is explained in Sect. II. Then, in Sect. III we describe a "simplified" version of our system, which differs from the full version (described in Sect. V and in Appx. A) in several ways. Most importantly, it is "non-concurrent" (i.e., it does not allow simultaneous operations on the same ledger channel). We present this "simplified" version because it explains the main ideas used in the full construction without going into too many technical details. The formal security definition is presented in Sect. IV. The implementation details are discussed in Sect. VI. Possible extensions and future work are described in Sect. VII. Security analysis is provided in Appx. C.

Further related works. Other channel network systems with different features such as privacy and channel re-balancing have been constructed in [8, 9, 12, 6, 5]. The most widely discussed recent proposals for the channel networks are *Lightning* and *Raiden*. Both of them are routing payments using the interactive mechanism based on the hashlocked transactions. A very interesting construction for creating chains of ledger channels has recently been proposed in [10]. They focus on different aspects of channel networks than we do, namely they do not aim to remove the interaction with the intermediaries, but on making the pessimistic time of channel closing constant.

[1]Perun is the god of thunder and lightning in the Slavic mythology. This choice of a name reflects the fact that one of our main inspirations is the Lightning system.

II. Preliminaries

In this section we introduce some basic notation and describe the terminology used in the area of smart contracts. We sometimes define a function $f : \{x_1, \ldots, x_n\} \to \mathcal{Y}$ by providing its function table as $f = [x_1 \mapsto y_1, \ldots, x_n \mapsto y_n]$ (meaning that for every i it holds that $f(x_i) = y_i$). Let Δ be a constant denoting the maximal blockchain reaction time, i.e. real time needed to post messages on the blockchain (we formalize the notion of "real time" using the concept of "rounds" in Sect. IV-A2). We assume that the parties have access to some fixed signature scheme $(\mathsf{KGen}, \mathsf{Sig}, \mathsf{Vf})$ that is existentially unforgeable against adaptive chosen-message attacks and know each other's public keys (see Sect. V for more on how we model this formally).

1) Smart contracts. Our construction uses smart contracts (see, e.g, https://en.bitcoin.it/wiki/Contract or [7]), which, informally speaking, are agreements written on the ledger, that can accept coins from parties, and distribute these coins between them depending on some well-specified conditions. We distinguish between contract *code* (typically denoted with $\mathcal{C}$) – a static object in which the above conditions are written, and a contract *instance* (typically denoted with C) – a dynamic object executing the code. On an intuitive level one can think of a contract instance as an independent entity (with public internal state) that receives coins and messages from the parties and sends coins and messages back to them. A contract instance is deployed on the ledger by one of the users, who also preloads it with some coins. We assume that deploying contract instances and sending messages to them takes time at most Δ. For more details on modeling smart contracts see Sect. IV-A1.

2) Uniquely attributable faults. When designing our scheme we distinguish between *uniquely* and *non-uniquely attributable faults* (see, e.g., https://github.com/ethereum/wiki/wiki/Glossary). Let us now briefly explain these terms. Suppose a malicious party P does not follow the protocol. In some cases, the other participants will end up in a situation when they are able to convince a contract instance C that P is malicious. We call such a fault *uniquely attributable*. This occurs, e.g, when P signed two contradictory statements, or when P did not send a message to C within certain time. A fault is *non*-uniquely attributable if some participants of the protocol know that P is dishonest, but they are *not* able to prove it to C. A standard example is a situation when P does not send a message m to some party P' when the protocol instructs it to send it. In this case P' *knows* that P is dishonest, but P' *cannot prove it* (since the fact that P did not send m does not have a digital evidence).

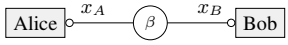
Uniquely attributable faults are easy to handle, since we can instruct the contract to punish the cheating P financially (e.g.: we let P lose all coins that it deposited in the contract). Non-uniquely attributable faults are harder to deal with, since it is not clear who should be punished, as the contract has no way to determine which party is telling the truth.

III. Simplified Perun — an informal description

This section presents an informal description of our system. First, in Sect. III-A, we describe Perun's functionality and discuss what properties it provides. We then give an overview of its security features (in Sect. III-B), and finally (in Sect. III-C) we provide the main ideas of our construction. Compared to the full formal description (see Sect. IV-V and Appx. A) we make several simplifying assumptions in this section. In particular, we present our scheme in a non-concurrent setting, i.e., we assume that the channels are *not* opened or updated in parallel, and that there is at most one virtual channel built over every ledger channel at any given time.

A. System's functionality

1) Ledger channels. We start with a description of the ledger payment channels, which are created by interacting with the blockchain, and allow two parties to instantaneously carry out payments between each other. These channels are essentially the same as those described in prior work (see, e.g., [11]). A ledger payment channel β between two parties, Alice and Bob, is created in an *opening procedure*, where Alice deposits x_A coins into the channel and Bob deposits x_B coins into it (for some $x_A, x_B \in \mathbb{R}_{\geq 0}$). Hence, initially, the *balance* of the channel can be described by a function defined as $[\text{Alice} \mapsto x_A, \text{Bob} \mapsto x_B]$, meaning that Alice has x_A coins in her *account in* β, Bob has x_B coins in his account, and the *value* of the channel is $x_A + x_B$. This can be depicted as:

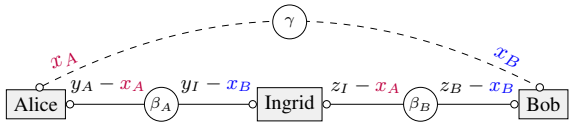


Until the channel β is closed, these coins remain "blocked", i.e., the parties cannot use them for any other purpose. After this set-up is completed, Alice and Bob can *update* the distribution of the funds in the channel multiple times without interacting with the blockchain. The update mechanism is used for performing payments between Alice and Bob. If, for example, Alice is willing to pay some amount $q \leq x_A$ of coins to Bob, then the parties perform an update that changes the balance of β to $[\text{Alice} \mapsto x'_A, \text{Bob} \mapsto x'_B]$, where $x'_A := x_A - q$ and $x'_B := x_B + q$. Such updates can be performed multiple times, but the total value of the channel never changes. At some point one of the parties that opened the channel can decide to *close* it. She then commits the current balance $[\text{Alice} \mapsto x''_A, \text{Bob} \mapsto x''_B]$ of the channel to the blockchain and Alice and Bob receive x''_A and x''_B coins, respectively.

2) Virtual channels. As described in Sect. I-A, the main novelty of Perun are virtual channels that minimize the need for interaction with the intermediaries in the channel chains, and in particular do not require that intermediaries confirm individual payments routed via them. The basic idea of virtual channels is to apply the channel technique recursively, by building a virtual payment channel "on top of" the ledger channels. To better illustrate the concept of virtual channels, consider three parties, Alice, Bob and Ingrid, and suppose that there exist ledger channels: β_A between Alice and Ingrid, and β_B between Ingrid and Bob. Let $[\text{Alice} \mapsto y_A, \text{Ingrid} \mapsto y_I]$ be the balance of β_A, and $[\text{Ingrid} \mapsto z_I, \text{Bob} \mapsto z_B]$ be the balance of β_B. This initial situation is shown below:

(1)

a) Opening and updating a virtual channel. Starting from this, Alice and Bob can now establish a *virtual* payment channel γ with initial balance $[\text{Alice} \mapsto x_A, \text{Bob} \mapsto x_B]$. This is done using channels β_A and β_B, but *without* touching the ledger. By opening γ some coins from the parties' accounts in the underlying ledger channels β_A and β_B will be temporarily removed (we also say that these coins are "blocked" in β_A and β_B, respectively). More precisely, after opening γ the balances of β_A and β_B change as follows: in channel β_A Alice will have x_A coins removed from her account, and Ingrid will have x_B coins removed. Symmetrically, in channel β_B Bob will have x_B coins removed, and Ingrid will have x_A coins removed. This is represented pictorially below.

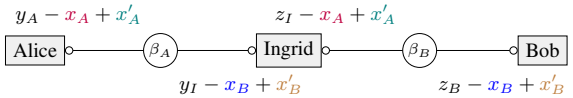


(2)

Opening of γ is possible only if all the values above are non-negative, i.e., $x_A \leq \min(y_A, z_I)$ and $x_B \leq \min(y_I, z_B)$. In other words, Alice, Bob and Ingrid need to have enough coins in their corresponding ledger channels to open γ. The coins x_A and x_B remain removed from parties' accounts in β_A and β_B for as long as the virtual channel is open. For Alice and Bob this is similar to the situation when the coins are blocked on the ledger in a newly created ledger channel.

Once a virtual channel is opened, it can be *updated* multiple times, exactly in the same way as the ledger channel, i.e., transferring q coins from Alice to Bob results into a new balance of γ as before. As long as everybody is honest, Alice and Bob need to interact with Ingrid only when the channel is opened and when it is closed, and in particular each update of γ does *not* require interacting with Ingrid.

b) Closing a virtual channel. The "financial consequences" of closing a virtual channel appear on the ledger channels β_A and β_B, and do not directly affect the parties' accounts on the blockchain. Let $[\text{Alice} \mapsto x'_A, \text{Bob} \mapsto x'_B]$ be the last balance of γ and suppose that the balances of β_A and β_B did not change from Eq. 2. Then closing γ results in β_A having balance $[\text{Alice} \mapsto (y_A - x_A + x'_A), \text{Ingrid} \mapsto (y_I - x_B + x'_B)]$ and β_B having balance $[\text{Ingrid} \mapsto (z_I - x_A + x'_A), \text{Bob} \mapsto (z_B - x_B + x'_B)]$. Pictorially this is presented below.



(3)

Note that for Alice the net financial result is that she gains $x'_A - x_A$ coins in her account in the ledger channel β_A (where "gaining x" coins means loosing $-x$, if x is negative). A similar guarantee holds for Bob (i.e. he gains $x'_B - x_B$ coins in β_B). On the other hand, the consequences for Ingrid are "neutral", i.e., if she gains z coins in β_A then she loses the

same amount in β_B (and vice versa). Suppose, for example, that the final balance of γ is more beneficial for Alice than γ's initial balance (i.e.: $x'_A > x_A$). In some sense, by agreeing to open a virtual channel, Ingrid accepts that she will cover (in β_A) transfers that Bob made to Alice in γ. The security properties of our scheme guarantee for Ingrid that in the process of closing γ she can claim back from Bob (in β_B) all the coins that she has transferred to Alice in β_A.

As long as γ is open, our system prevents the ledger channels β_A and β_B from being closed. In other words, the parties that opened these ledger channels have to wait with closing them until the financial consequences from closing of channel γ are known. One subtlety with this is that Ingrid should be ensured that her coins do not get blocked in β_P's for a very long period (or: forever). This is different from the ledger channels, where the role of the "intermediary" is played by the blockchain, which does not have "its own coins" invested in the protocol. (In particular: it is completely ok "from the point of view of the ledger" if a ledger channel is never closed.) For this reason the virtual channels come with a special real-time value called *validity* that Alice, Bob and Ingrid agree on when the virtual channel was opened. A virtual channel is closed when its validity expires (note that again this is different from the ledger channels, where closing is initiated by Alice or Bob). Thanks to this solution Ingrid can be sure that she gets her coins back after some period. Another (slightly more complicated) option would be to allow Ingrid to request virtual channel closing at any time.

B. Security and efficiency properties.

Let us now informally discuss the security and efficiency properties of our system (they are presented formally, in form of an ideal functionality, in Sect. IV-B). We also provide information on how long our procedures take in the "normal case" (i.e. when every party involved in the procedure is honest), and in the "pessimistic case" (i.e. when some dishonest participants delay protocol's execution). This is done in terms of "real time" (for more on our modeling of time see Sect. IV-A2). We emphasize that our scheme is secure against arbitrary corruptions of Alice, Ingrid, and Bob, and in particular, no assumption about the honesty of Ingrid is needed.

Consensus on channel opening. A ledger/virtual channel δ can only be open if all the parties involved in it agree. In particular, Ingrid has to confirm the creation of a virtual channel (and agree on this channel's validity). Let us emphasize that our protocols guarantees that there is always a consensus among the honest parties whether a ledger/virtual channel has been successfully open. This requirement is easily satisfied for the ledger channels (as they are "visible" on the blockchain), but it is less trivial to achieve for virtual channels. Consensus among the honest parties is needed, since a disagreement on the status of γ may lead to misunderstandings. For instance, if Alice thinks that γ has been opened, while Bob believes the opposite, then he will not respond to Alice's requests to update γ. Opening the ledger channels takes always time $O(\Delta)$. Opening virtual channels takes constant time (i.e. time independent on Δ).

Consensus on channel update. For ledger/virtual channel δ Alice and Bob need to confirm every update. Channel updates always take constant time.

Guaranteed channel closing. Let β be a ledger channel. Both Alice and Bob can request closing of β at any time (provided there is no virtual channel open over β). Once such a request is made, the channel is closed in time $O(\Delta)$. Let γ be a virtual channel, and let v denote its validity. Channel γ is closed in time $v + O(1)$ in the normal case, and $v + O(\Delta)$ in the pessimistic case.

Guaranteed balance payout for end users. The end users of a ledger/virtual channel are guaranteed that the channel's latest balance is paid out. Concretely, this means for a ledger channel β that coins are transferred back to the accounts of the end users on the ledger, and for virtual channel γ it means that the latest balance of the channel is transferred back to the respective ledger channels.

Balance neutrality for intermediary Ingrid. Virtual channels are always "financially neutral" for the intermediary Ingrid. More precisely: suppose γ is a virtual channel built over ledger channels β_A and β_B. Once γ is closed the following holds: if Ingrid loses x coins in β_A, then she gains x coins in β_B (and vice versa).

C. Main construction ideas

In this section we informally describe the main ideas of our protocol (for its formal description see Sect. V and Appx. A). The construction consists of the instructions for Alice, Bob and Ingrid, and a contract code $\mathcal{C}$ that is executed by contract instances deployed on the ledger. For the purpose of this informal description we will think of a contract instance C as yet another party and hence we will simply specify its actions (without formally defining its code).[2] In addition, we will make the following two assumptions about the behavior of the parties.

Assumption 1: If a contract instance C detects a fault that is uniquely attributable to a party P (see Sect. II-2) then it gives all its coins to the honest party and terminates. We also say that C *punishes party* P.

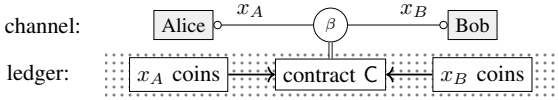
Assumption 2: If a party P detects that some other party P' is dishonest (e.g. it does not send some message in time), then P does not engage in any new protocol with P'. For example, suppose that Alice learns that Bob (with whom she has a channel β) is cheating. Then she does not perform any updates of this channel, and does not open new virtual channels over β. Sometimes, we will simply let Alice immediately close the channel β.

The first assumption clearly causes no harm to the honest parties, as they will never cause uniquely attributable faults. The second one also makes sense, as there is no reason to start

[2] In this context we note that we allow C to take actions by *itself* (e.g. when some time t comes). In our formal description in Sect. A we do *not* make such an assumption, and require that the parties explicitly trigger every action of C (see also Sect. V-0c).

new off-chain procedures if it is likely that they will lead to conflicts that will need to be resolved on the blockchain.

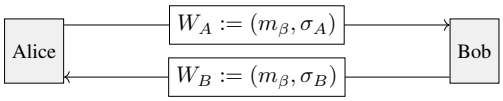
1) Ledger channels. We start with describing a procedure in which Alice and Bob open a ledger channel β with the *initial balance* as in Sect. III-A1 (i.e.: $[\text{Alice} \mapsto x_a, \text{Bob} \mapsto x_B]$). The procedure starts with Alice deploying a contract instance C together with x_A coins. After C appears on the ledger, Bob confirms that he agrees on the opening of β by sending x_B coins to C. If C receives this message within time Δ then the channel is open. Otherwise, Alice is refunded x_a coins. The relation between the channel β and the contract instance C on the ledger can be presented as follows.



Let us now describe the channel update procedure for channel β. We use a standard technique (see, e.g, Sec. 3.3 in [11]) for updating the balance in a payment channel that is based on counters called "version numbers". The parties that opened a channel maintain the *version number* $w \in \mathbb{N}$. Initially w is set to 1, and it is incremented after each update of β. The update procedure is initiated by one party called *initiator* (the other party is called *confirmer*). Suppose Alice is the initiator. In order to propose an update of a channel β to a new balance $[\text{Alice} \mapsto x'_A, \text{Bob} \mapsto x'_B]$ Alice sends to Bob an *update message* $W_A := (m_\beta, \sigma_A)$, where

$$m_\beta = \text{“}\texttt{update}\ \beta\ \texttt{to}\ [\text{Alice} \mapsto x'_A, \text{Bob} \mapsto x'_B],\ \texttt{version number}\ w\text{”} \tag{4}$$

and σ_A is Alice's signature on m_β. If Bob agrees on this update then he replies with $W_B := (m_\beta, \sigma_B)$ where σ_B is Bob's signature on m_β. At this point the channel is updated to its new balance, and w is incremented by 1. Pictorially, this message flow can be represented as follows:



(5)

If one of the parties, Alice, say, wants to close the channel β, then she sends to C the latest update message W_B that she has received from Bob (if no update has been performed then she sends W_B equal to the initial channel balance with version number 0).

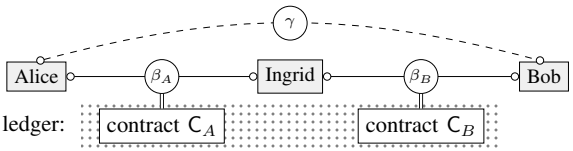
Upon receiving W_B, the contract instance C notifies Bob about Alice's request, and waits time Δ for Bob to reply with the latest update message W_A that he received from Alice (if both Alice and Bob are honest and always agreed on the proposed updates then W_A and W_B contain the same message). Upon receiving both messages C checks which of W_A and W_B has a higher version number, and distributes the money according to the balance that is provided in this message (if the version numbers are the same then C decides arbitrarily). Suppose, for example, W_A has a higher version number and it is equal to

$$\text{“}\texttt{update}\ \beta\ \texttt{to}\ [\text{Alice} \mapsto x''_A, \text{Bob} \mapsto x''_B],\ \texttt{version number}\ w_A\text{”},\ \sigma_B. \tag{6}$$

Then C gives x''_A coins to Alice, and x''_B coins to B. If Bob did not reply within time Δ to C's message, then C distributes the money according to the balance provided in W_B.

2) Virtual channels. We now describe the protocol for the virtual channels. Recall that a virtual channel γ is built with the help of Ingrid using ledger channels β_A and β_B in a similar way as the ledger channels are built with the help of the ledger. There are, however, some important differences between the functionality that the ledger channels and the blockchain provide. Firstly, the ledger channels described in Sect. III-C1 are used only for performing payments between two parties, and they do not allow to execute smart contracts "inside of the channel" (while the ledger allows it). The second problem is that the ledger channels give us a "virtual ledger" for only 2 parties. In other words: what happens in a ledger channel between Alice and Ingrid is "invisible" for Bob (which is different from the global ledger that is used to build the ledger channels). We solve these problems by extending the functionality that the ledger payment channels provide. Concretely, this is done by letting the parties exchange additional signed information (see "opening certificates" below) and by allowing the end users to add more data to the ledger channel update messages that the parties exchange (such a signed update message will be called a "closing certificate").

Let C_A and C_B be the contract instances corresponding to the ledger channels β_A and β_B (respectively). Pictorially, the relation between the ledger channels and the contract instances can be presented as follows:



Compared the the contract instance C described in the previous section, the contract instances C_A and C_B will have additional functionality, corresponding to handling the virtual channels (and, in particular, interpreting the opening and closing certificates), and hence they will be more complex than C.

a) Virtual channel opening. We start with the opening procedure. Suppose the initial balance of γ is $[\text{Alice} \mapsto x_A; \text{Bob} \mapsto x_B]$, and its validity is v. Assume that the channels β_A and β_B have balances as on Eq. (1) (p. 3). Recall that as a result of the opening of γ the balances of β_A and β_B will change as is illustrated on Eq. (2). Let us now discuss how this channel opening is realized at the protocol level.

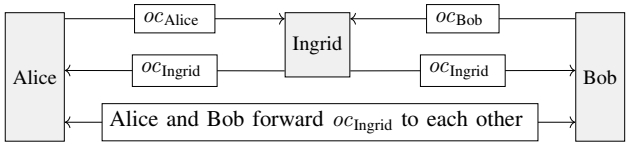
Informally, opening γ is done by letting the parties exchanging "opening certificates for γ". An *opening certificate of* $P \in \{\text{Alice}, \text{Ingrid}, \text{Bob}\}$ *for* γ has the following form:

$$oc_P := (\text{“}\texttt{open virtual channel}\ \gamma\ \texttt{with initial balance}\ [\text{Alice} \mapsto x_A; \text{Bob} \mapsto x_B]\ \texttt{and validity}\ v\text{”},\ \sigma_P),$$

where σ_P is a signature of oc_P by party P. The role of this certificate is to guarantee that a party P cannot deny that she agreed to open γ. They will be used when the parties interact with the contract instances. For example, if Ingrid denies that she every agreed to open γ then Alice can use these certificates

to prove her wrong (see, e.g., Sect. III-C2c "Virtual channel closing" below).

We first describe the process of opening a virtual channel in case all parties are honest. First, Alice and Bob send their opening certificates for γ to Ingrid. If Ingrid receives *both* of these certificates, then she replies to Alice and Bob with her opening certificate for γ and considers the channel open. Parties Alice and Bob upon receiving these certificates forward them to each other (we will explain in a moment the role of this forwarding). Each of them considers the channel open if she/he received Ingrid's opening certificate (either directly from Ingrid, or forwarded in the last step). Pictorially, the message flow looks in this case as follows:



Note that the ledger channels β_A and β_A are *not* updated in this procedure. Therefore, technically, virtual channel opening does not result in immediate direct removal of coins from parties' accounts in the ledger channels (they will be removed later, when the virtual channel is closed, see below). Such "delayed coin removal" is ok, since the parties can locally keep track on how many coins are still not blocked in their ledger channels. Hence, in some sense, these coins are removed "virtually" from the ledger channels at the moment when the virtual channel is opened (and the "real removal" is done when the virtual channel is closed).

Now consider what happens if some parties are misbehaving. In this case the execution of the protocol can result in *not* opening channel γ. Informally, the main properties that our protocol needs to have are: (1) even if some parties cheat, no honest party loses coins, and (2) there is a consensus between the honest parties on whether the channel has been open or not (see Sect. III-B). Let us first discuss how our protocol provides security guarantee (1). The result of the protocol execution is that the parties end up holding opening certificates for γ. Since such a certificate can later be used to claim coins from a party $P \in \{\text{Alice}, \text{Ingrid}, \text{Bob}\}$ that signed it, thus the main security risk for P is that P signs a certificate that will later be used to claim coins from P, while P cannot claim coins from other parties since P itself did not receive an opening certificate for γ. It is easy to see that this problem does not occur for Alice and Bob. This is because these parties will not consider the channel open if they do not receive an opening certificate from Ingrid, and in this case they will never perform any update to γ. Therefore, even if a malicious Ingrid does not send an opening certificate for γ to Alice or Bob, (and then requests to close γ when γ's validity time comes), then the result of her behavior will be "neutral" for both Alice and Bob (as the "default" state of γ is that both parties get the same amount of coins as they deposited).

Therefore, what remains is to show that (1) is satisfied for Ingrid (this property was called "balance neutrality" in Sect. III-B). Here, the problem could potentially be larger, as Ingrid could lose coins if she sends her certificate to Alice (say) without getting the certificate from Bob (as during the channel closing she would be forced to pay coins to Alice *without* being guaranteed that she gets the same amount of coins from Bob). This problem is precisely the reason why in our protocol Ingrid signs the opening certificates *only* if she received the opening certificates for γ from *both* Alice and Bob. In other words: she only agrees to cover Bob's commitments in front of Alice if she is guaranteed that Bob can be held responsible for these commitments (and symmetrically for Alice's commitments).

Now let us discuss (2). If Ingrid is honest, then clearly there is a consensus among all honest parties on whether the channel γ was open or not (since either Ingrid sends her opening certificate to both Alice and Bob, or to none of them). If Ingrid is dishonest, then the only situation when there is disagreement between the *honest* Alice and Bob is if the malicious Ingrid sends her opening certificate to one of them, and not to the other one. To avoid this problem we let the parties forward to each other the opening certificate from Ingrid. This guarantees that if at least one of them considers the channel open, then the other one considers it open as well.

Finally, let us comment on the behavior of the parties when the opening procedure successfully ends. One thing that would obviously be illegal is if one of the parties starts the ledger channel closing procedure for β_A or β_B when γ is still open (i.e. before its validity time comes). Therefore after every successful opening of a virtual channel γ, each party $P \in \{\text{Alice}, \text{Ingrid}, \text{Bob}\}$ monitors the situation in the ledger channels, and reacts to it. Suppose, for example, that a malicious Ingrid contacts C_A with a request to close channel β_A while γ is still open. As described above, C_A informs Alice about this fact. Alice then has a chance to stop the closing of β_A by sending to C_A the opening certificate of Ingrid for γ.

b) Virtual channel updating. Virtual channel updates are done exactly as in case of the ledger channels, i.e., the parties maintain a version of the channel, and exchange their signatures on new channel versions. Let V_A and V_B denote the exchanged signed messages (in case of the ledger channels these messages were denoted W_A an W_B, respectively, see Eq. 5).

c) Virtual channel closing. The channel closing procedure is started automatically when the validity of γ expires. The main idea of this procedure is that it is Ingrid who is responsible for closing γ and taking care that the channels β_A and β_B are updated in the correct way (i.e. according the the latest balance of γ). Therefore, in some sense, Ingrid plays a role similar to the role of C for the ledger channel closing. Of course, the situation is much more complicated now, since (unlike C), Ingrid cannot be assumed to be trusted.

Our closing protocol is constructed in such a way that it is guaranteed that an honest Ingrid will always manage to close a virtual channel within some fixed time T_{max} (or at least convince the contract that she correctly started the closing procedure). "Ingrid not closing γ on time" is a uniquely attributable fault, i.e, a contract instance C_A (say) can always determine if it was indeed Ingrid who did not close the channel

γ, or if Alice is falsely accusing Ingrid. This is because (1) the fact that a channel γ (with validity v) has been open can be proven using an opening certificate oc_{Ingrid}, and (2) proving that a channel has been closed is possible thanks to the "closing certificates" (that we define in a moment). Therefore, what remains is to describe the protocol in which Ingrid can close γ in bounded time. If this does not happen, then Alice and Bob complain to the contract instances C_A and C_B respectively, and these instances will punish Ingrid by transferring all of Ingrid's coins to the complaining party.

If everybody is honest then the procedure works in a straightforward way. Let us start by explaining it, and ignoring for a moment some details that are needed for preventing cheating by dishonest parties. First, Alice sends to Ingrid the latest update message V_B that she received from Bob (if no update has been performed then she lets V_B be the initial channel balance of γ with version number 0, and no signature). In parallel, Bob does a symmetric thing with the latest update message V_A that he received from Alice. Then party Ingrid decides what is the latest balance of γ by checking which version has a higher number (this is done according to the same rules as the ones used by C in the ledger channel closing procedure). She then proposes to update the ledger channels accordingly. That is: if the latest balance of γ is $[\text{Alice} \mapsto x'_A; \text{Bob} \mapsto x'_B]$ then the balance of the ledger channel β_A is changed to by adding $-x_A + x'_A$ coins to Alice's account and $-x_B + x'_B$ coins to Ingrid's account in β_A (note that these two values sum up to 0), and, symmetrically: adding $-x_A + x'_A$ coins to Ingrid's account and $-x_B + x'_B$ coins to Alice's account in β_B (see also Eq. (3), p. 3). The "$-x_A$" and "$-x_B$" terms come from fact that we use the "delayed coin removal" approach, i.e., we do not remove these coins from the ledger channel accounts during the opening procedure. Alice and Bob confirm the update, and channel γ is closed.

One problem with the above procedure is that the parties end up with no proof that the virtual channel has been closed. In particular, this means that a dishonest party could later try to close γ again, or Alice and Bob could accuse Ingrid of not closing γ on time (see above). To fix this, we make the following change in the ledger channel update procedure. Instead of exchanging signatures on message m_{β_A} of a form as in Eq. (4), Alice and Ingrid exchange messages on "annotated m_{β_A}" defined as

$$m^*_{\beta_A} = \text{“}\texttt{update}\ \beta_A\ \texttt{to}\ [\text{Alice} \mapsto x'_A, \mathsf{Ingrid} \mapsto x'_B] \\ \underline{\texttt{because of closing}}\ \gamma, \texttt{version number}\ w_A\text{”}, \tag{7}$$

where w_A is the current version number used for updating channel β_A. Symmetrically, Ingrid and Bob exchange signatures on $m^*_{\beta_A}$:= "annotated m_{β_B}" (defined analogously). Hence, a successful closing procedure of γ results in each party holding a signed string that can serve as a proof that γ was correctly closed. We call such signed strings *closing certificates* (cc).

Another problem is that Ingrid has no proof that one of the parties, Alice, say, indeed sent to her the message V_B. In particular, since this message does not contain Alice's signature, it can be easily fabricated by malicious Ingrid collaborating with malicious Bob. Hence, it cannot be later used in Ingrid's interaction with the contract instance C_A. We solve this problem by requiring that this message has to come with Alice's signature (and, symmetrically V_B sent by Bob has to come with Bob's signature). Let msg^i_P (for $i = 1, 2, 3$ and $P \in \{A, B\}$) denote the consecutive messages that should be exchanged between the parties (if all of them are honest), i.e. msg^1_P := "$V_{P'}$ signed by P", and msg^2_P := "$m^*_{\beta_P}$ signed by I", and msg^3_P := "$m^*_{\beta_P}$ signed by P". To summarize, the message flow in the closing procedure (in case everybody behaves honestly) looks as depicted on Fig. 1. Consider now what happens when the parties are malicious.

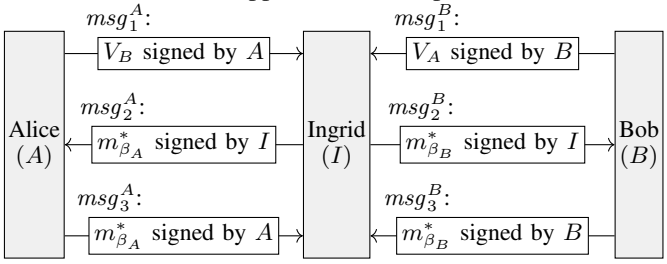


Fig. 1: Closing the virtual channel γ.

Look, e.g, at the interaction between Alice and Ingrid (the interaction between Ingrid and Bob is handled analogously). First, suppose Alice is dishonest and does not send a message msg^1_A or msg^3_A to Ingrid. In this case Ingrid has to resolve this issue by contacting C_A. Note that "not sending a message" is a *non*-uniquely attributable fault (i.e. C_A has no way to determine if Alice indeed did not send this message), and hence Ingrid cannot expect C_A to punish Alice (see Assumption 1 in Sect. III-C for the definition of "punishment"). The procedure works as follows. In both cases ("msg^1_A not sent" and "msg^3_A not sent") Ingrid initiates her conversation with C_A by providing evidence that Alice should send a message to her. This evidence is different in each case.

- *"msg^1_A not sent"*: in this case it is enough that Ingrid sends to C_A Alice's opening certificate oc_A for γ. Contract C_A then checks γ's validity and rejects the complaint if this is not the right moment to close γ. Otherwise C_A informs Alice about Ingrid's complaint. If γ has already been closed then Alice proves it to C_A by replying with a closing certificate on γ signed by Ingrid (in which case C_A punishes Ingrid). Otherwise, Alice sends msg^1_A to C_A who forwards it to Ingrid (if she does not send it within time Δ, then it is a uniquely attributable fault, and C_A punishes Alice). We say that Ingrid received msg^1_A "via the contract".
- *"msg^3_A not sent"*: in this case Ingrid sends to C_A Alice's opening certificate oc_{Alice} for γ, plus messages msg^A_1 and msg^B_1 that Ingrid received earlier (either directly from the Alice and Bob, or via the contract C_B). Note that these messages consist of versions of γ signed by Alice and Bob, and hence C_A can determine the final balance $[\text{Alice} \mapsto x'_A; \text{Bob} \mapsto x'_B]$ of γ. Since the opening certificate contains the initial balance $[\text{Alice} \mapsto x_A; \text{Bob} \mapsto x_B]$ of γ, thus C can compute the value $x := -x_B + x'_B$ coins that should be

transferred from Alice to Ingrid (note that x can be negative, in which case $-x$ coins are transferred from Ingrid to Alice). Ingrid then starts the following emergency closing procedure of channel β_A (recall that by Assumption 2 in Sect. III-C the channel β_A will anyway not be used anymore, and hence it is ok to close it):

Closing of β_A with simultaneous transfer of x coins from Alice to Bob: The channel is closed exactly as described in Sect. III-C1 except that the amounts of coins that the parties get are "corrected" to take into account the transfer x. To be more concrete, suppose Ingrid played the role of Bob in channel β_A (and Alice played the role of Alice). Let the message $m_P^{\beta_A}$ with the higher version number be as on Eq. (6). Then the amount of coins that Alice gets is $x''_A - x$ and Bob (who is Ingrid in our case) gets $x''_B + x$.

The case when Ingrid does not send msg_A^2 to Alice, or send a wrong message msg_A^2 (e.g. a message that proposes to Alice in fewer coins than what Alice is supposed to receive from closing γ) is simpler. In this situation Alice simply does nothing until time T_{max} comes, or until she gets some message from C_A triggered by Ingrid's action (see above). This is ok, since we assumed that the burden to close γ before time T_{max} is on Ingrid.

IV. Formal security definition of the "full" Perun

We now describe our system more formally. We assume a fixed set of parties $\mathcal{P} = \{P_1, \ldots, P_n\}$ that use the channel system. All values are encoded as bit strings. We present tuples by identifying their individual values with keywords called *attributes*: $\mathsf{attr1}, \mathsf{attr2}, \ldots$. Formally, an *attribute tuple* is a function from its set of attributes to $\{0,1\}^*$. To improve readability, the *value of an attribute* attr *in a tuple* T (i.e. $T(\mathsf{attr})$) is referred to as $T.\mathsf{attr}$.

Channel syntax. We start by describing more formally the syntax used for defining ledger and virtual channels. To this end, we first introduce two types of functions for specifying the current balance of a channel and for handling transfers between parties. We say that π is a *balance function for parties P and P'* if its type is $\pi : \{P, P'\} \to \mathbb{R}_{\geq 0}$, and its purpose is to describe the channel's current balance. We say that θ is a *transfer function for parties P and P'* if its type is $\{P, P'\} \to \mathbb{R}$ and $\theta(P) + \theta(P') = 0$. These functions can be added in a natural way, i.e., if f and g are transfer or balance functions for P and P', then $h = f + g$ is a function $h : \{P, P'\} \to \mathbb{R}$ defined as $h(P) := f(P) + g(P)$ and $h(P') := f(P') + g(P')$. In addition, we will use the following conventions. If $\pi : \{P, P'\} \to \mathbb{R}_{\geq 0}$ is a balance function, then *adding $x \in \mathbb{R}$ coins to the account of P in π* results in a function $\pi' : \{P, P'\} \to \mathbb{R}_{\geq 0}$ equal to $\pi(P')$ on input P', and to $\pi(P) + x$ on input P. *Removing x coins* is a shorthand for "adding $-x$ coins" to g.

We define a *ledger channel over a set of parties $\mathcal{P}$* as an attribute tuple β of the form: $\beta = (\beta.\mathsf{id}, \beta.\mathsf{Alice}, \beta.\mathsf{Bob}, \beta.\mathsf{balance})$, and a *virtual payment channel γ over a set of players $\mathcal{P}$* as an attribute tuple of the form: $\gamma = (\gamma.\mathsf{id}, \gamma.\mathsf{Alice}, \gamma.\mathsf{Ingrid}, \gamma.\mathsf{Bob}, \gamma.\mathsf{balance}, \gamma.\mathsf{subchan}, \gamma.\mathsf{validity})$. The meaning of the attributes for a ledger/virtual channel δ is follows. The value $\delta.\mathsf{id} \in \{0,1\}^*$ is called the *identifier of* δ, and $\delta.\mathsf{Alice}, \delta.\mathsf{Bob}$ are two distinct elements of $\mathcal{P}$. If δ is a virtual channel, then $\delta.\mathsf{Ingrid}$ is also an element of $\mathcal{P}$ (distinct from $\delta.\mathsf{Alice}$ and $\delta.\mathsf{Bob}$) and it is sometimes called the *intermediary*. We define the set of *end-users of* δ as $\delta.\mathsf{end\text{-}users} = \{\delta.\mathsf{Alice}, \delta.\mathsf{Bob}\}$ (note that when δ is a virtual channel, then this set does not contain $\delta.\mathsf{Ingrid}$). We also define the shortcut $\delta.\mathsf{other\text{-}party} : \delta.\mathsf{end\text{-}users} \to \delta.\mathsf{end\text{-}users}$ as $\delta.\mathsf{other\text{-}party}(\delta.\mathsf{Alice}) = \delta.\mathsf{Bob}$ and $\delta.\mathsf{other\text{-}party}(\delta.\mathsf{Bob}) = \delta.\mathsf{Alice}$, respectively. If δ is a virtual channel then $\delta.\mathsf{all\text{-}users}$ denotes the set $\{\delta.\mathsf{Alice}, \delta.\mathsf{Bob}, \delta.\mathsf{Ingrid}\}$, and if δ is a ledger channel then simply $\delta.\mathsf{all\text{-}users} = \delta.\mathsf{end\text{-}users}$. The attribute $\delta.\mathsf{balance}$ is a balance function for the parties $\delta.\mathsf{end\text{-}users}$.

In addition to the above, a virtual channel γ has the following attributes. First, the function $\mathsf{subchan} : \gamma.\mathsf{end\text{-}users} \to \{0,1\}^*$, where the values $\gamma.\mathsf{subchan}(\gamma.\mathsf{Alice})$ and $\gamma.\mathsf{subchan}(\gamma.\mathsf{Bob})$ are the identifiers of the *sub-channels* over which γ is *constructed*. Second, $\gamma.\mathsf{validity} \in \mathbb{N}$ denotes the *channel validity*, i.e., the round until which the virtual payment channel stays open. We provide further details on how to model rounds in Sec. IV-A2.

A. The security model

To formally model the security guarantees, we use the UC framework introduced in the seminal work of Canetti [1]. In the UC model security is defined by comparing the execution of a protocol in the *real world* with an idealized protocol – often referred to as an *ideal functionality* – in the *ideal world*. A protocol is said to be *UC secure* if the real-world execution of the protocol cannot be distinguished from the idealized protocol execution. Informally, this means that if the ideal protocol satisfies strong security properties, then these guarantees are inherited by the real world protocol.

1) Real world and ideal world protocols. In the real world our protocol (denoted "`channels`") is run among a group of parties $\mathcal{P}$, which are connected by authentic communication channels. In addition, in the real world an *adversary* $\mathcal{A}$ may corrupt parties, where corruption means that the adversary takes full control over the party's actions. For simplicity, we consider a static adversary, where corruption only takes place at the beginning of the protocol. Our protocol `channels` is designed in the $\mathcal{C}$-hybrid world (i.e., the parties have access to a functionality $\mathcal{C}$, see [1]), where $\mathcal{C}$ is the *contract functionality* that maintains the set of active *contract instances*. Each contract instance has a unique *identifier*. We refer to a contract instance with identifier id as $\mathcal{C}(id)$. In our case each contract instance corresponds to a ledger channel, and, for simplicity, has the same identifier. In other words, a contract instance $\mathcal{C}(\beta.\mathsf{id})$ corresponds to a ledger channel β. When a channel is closed then the corresponding contract instance terminates (i.e. it is removed from the set of contract instances of $\mathcal{C}$).

Contract instances cannot directly refer to each other's variables, and therefore each of them can be easily implemented as a separate contract on the Ethereum ledger, or even as contracts on different ledgers, provided the exchange rate between the

coins on these ledgers is fixed. A new contract instance $\mathcal{C}(\beta.\mathsf{id})$ is created when $\mathcal{C}$ receives a *constructor message*. We also say that a message m is "sent to $\mathcal{C}(id)$" or "sent by $\mathcal{C}(id)$" to denote interaction with this specific contract instance. One can also think about it in the following way: every message (other than the constructor message) that is sent to $\mathcal{C}$ contains the identifier id that specifies to which particular contract instance it is addressed, and a similar rule applies to messages sent by $\mathcal{C}$. All entities of the protocol are operated by a special party $\mathcal{Z}$ – the so-called environment, which provide the inputs for the parties and receives their outputs. In the ideal world we consider a dummy protocol where the parties from set $\mathcal{P}$ just forward their inputs to an ideal functionality that we call Channels and which we will described in detail in Sec. IV-B. The ideal world protocol is run in the presence of an ideal world adversary $\mathcal{S}$ – often called the simulator. All the messages start with a keyword in a typewriter font (e.g. `lc-open`). The messages exchanged between the parties and the environment are underlined (e.g.: `lc-open`).

We assume that before the protocol starts a public-key infrastructure setup phase is executed by some trusted party. The signature of $P \in \mathcal{P}$ on m will be denoted $\mathsf{Sign}_P(m)$. We say that a tuple $(x_1, \ldots, x_n, \sigma)$ is *signed by* P if σ is a valid signature of P on $(x_1, \ldots, x_n)$. We emphasize that the use of a PKI is only an abstraction that helps to describe our protocols. In practice, the trusted setup can, e.g., easily be realized using the blockchain. To keep the model as simple as possible we do not include the transaction fees in our modeling.

2) Synchronous communication model. In this work, we consider a synchronous communication network, where the execution of the protocol happens in rounds. Hence, "rounds" are the measure of real time (e.g. one can think of a "round" as a "second"). Let us specify the number of rounds it takes for parties to communicate with each other (for simplicity, we assume that computation takes no time). If some party P_i sends a message in round t to another party (or an ideal functionality), it arrives to this functionality at the beginning of round $t+1$. The adversary can decide about the order in which the messages arrive in a given round, but we assume that he cannot change the order of messages sent between two honest parties (this can be easily achieved by using, e.g., message counters). The communication between all other parties including communication with the environment $\mathcal{Z}$ is instantaneous. In the description of our protocols and ideal functionalities we often write that some action is executed *within time* τ. This means that the exact round until when this action is completed is up to the adversary to decide but τ is an upper bound.

3) Modeling coins via the global ledger functionality. We model coins and the money mechanics via a global ideal functionality ledger $\mathcal{L}$ and the global UC (GUC) model [2]. The state of the ledger functionality is public, and it maintains a non-negative vector of reals $(x_1, \ldots, x_n)$, where x_i corresponds to the current amount of coins in party P_i's account on $\mathcal{L}$. The parties cannot directly access $\mathcal{L}$. Instead their accounts are maintained via the ideal functionality $\mathcal{C}$ (in the real world) or via Channels (in the ideal world). The ledger functionality $\mathcal{L}$ looks as follows:

Initialization: The functionality is initialized by a message $(x_1, \ldots, x_n) \in \mathbb{R}^n_{\geq 0}$ that describes the initial coin distribution and comes from $\mathcal{Z}$. The functionality stores this tuple.
Adding coins: Upon receiving a message $(\underline{\texttt{add}}, P_i, y)$ (for $P_i \in \mathcal{P}$ and $y \in \mathbb{R}_{\geq 0}$) let $x_i := x_i + y$.
Removing coins: Upon receiving a message $(\underline{\texttt{remove}}, P_i, y)$ (for $P_i \in \mathcal{P}$ and $y \in \mathbb{R}_{\geq 0}$): if $x_i < y$ then do nothing, otherwise let $x_i := x_i - y$.

To maintain the accounts of the users, $\mathcal{L}$ can process messages `add` and `remove`, which allow to *add y coins to* (or *remove y coins from*) P's account on the ledger. To simplify notation, we will sometimes say that a party P sends a message m to an ideal functionality $\mathcal{I}$ *together with x coins*. By this we mean that when m arrives to $\mathcal{I}$, the functionality $\mathcal{I}$ removes x coins from P's account in $\mathcal{L}$ (if P does not have sufficient coins then message m is ignored). Similarly $\mathcal{I}$ sends a message m' to P *together with x' coins* means that x' coins are added to P's account in $\mathcal{L}$. In the ideal world we allow the simulator $\mathcal{S}$ to freely remove money from the accounts of corrupt parties and to add them (with an arbitrary delay) to the accounts of other (corrupt or honest) parties. This corresponds to the fact that we are not interested in preventing the corrupt parties from "acting irrationally" and loosing money.

4) The security definition. To formally define security we consider two random variables. Let λ denote the security parameter (which is given as input to the environment and to the parties). First, $\text{EXEC}^{\mathcal{Z},\mathcal{A}}_{\texttt{channels},\mathcal{C}}(\lambda)$ is the output of $\mathcal{Z}$ running the real world protocol `channels` in the $\mathcal{C}$-hybrid world with adversary $\mathcal{A}$. Second, $\text{IDEAL}^{\mathcal{Z},\mathcal{S}}_{\mathsf{Channels},\mathcal{S}}(\lambda)$, which denotes the output of $\mathcal{Z}$ running in the ideal world with the Channels ideal functionality and the simulator $\mathcal{S}$. In both cases to simplify exposition, we will assume that $\mathcal{Z}$ is from a class of "restricted" environments, i.e. we will make some explicit assumptions about $\mathcal{Z}$'s behavior. Most of them are very natural, and they can be informally captured as "the environment never asks the honest users to do something obviously wrong", e.g., open two different channels with the same identifier, or open a channel without having sufficient funds. These restrictions could be eliminated at the cost of a more complex protocol description, and we defer a full list of these restrictions to Appx. B. We are now ready to state our main security definition.

Definition. We say that protocol `channels` running in the $\mathcal{C}$-hybrid world emulates an ideal functionality Channels with respect to a global ledger $\mathcal{L}$ and with blockchain delay Δ, if for any PPT adversary $\mathcal{A}$ there exists a simulator $\mathcal{S}$ such that for all restricted environments $\mathcal{Z}$ (see Appx. B), we have: $\text{EXEC}^{\mathcal{Z},\mathcal{A}}_{\texttt{channels},\mathcal{C}}(\lambda) \approx \text{IDEAL}^{\mathcal{Z},\mathcal{S}}_{\mathsf{Channels},\mathcal{S}}(\lambda)$, where $\approx$ denotes computational indistinguishability.

B. The ideal functionality Channels

As discussed in the previous section, in the ideal world the parties do not execute any protocol. Instead, they simply

receive messages from the environment and forward them to the ideal functionality Channels. The functionality Channels, shown in Fig. 2, maintains a *channel space*, which is a set Σ that consists of some ledger and virtual channel tuples. We assume that for every id there exists at most one channel $\delta \in \Sigma$ such that $\delta.\mathsf{id} = id$ (we will also refer to such channel as $\Sigma(id)$). We require that for every virtual channel $\gamma \in \Sigma$ there exist ledger channels $\beta_A, \beta_B \in \Sigma$ such that $\gamma.\mathsf{subchan}(\gamma.\mathsf{Alice}) = \beta_A.\mathsf{id}$ and $\gamma.\mathsf{subchan}(\gamma.\mathsf{Bob}) = \beta_B.\mathsf{id}$, i.e., the channels β_A and β_B that were used to construct γ also belong to Σ. Initially Σ is empty.

The Channels functionality offers the following interface for the parties (messages concerning the ledger channels start with "lc", and those concerning the virtual ones start with "vc"). (A) *Opening a ledger channel* β between $\beta.\mathsf{Alice}$ and $\beta.\mathsf{Bob}$, which is triggered via a message $(\underline{\texttt{lc-open}}, \beta)$ from $\beta.\mathsf{Alice}$ with $\beta.\mathsf{balance}(\beta.\mathsf{Alice})$ coins. (B) *Opening and closing of a virtual channel* γ, which is triggered by a message $(\underline{\texttt{vc-open}}, \gamma)$, and also handles the closing of γ when time $\gamma.\mathsf{validity}$ comes. (C) *Ledger/virtual channel update* which is triggered via a message $(\underline{\texttt{update}}, id, \theta, \alpha)$, where id refers to the channel that shall be updated according to the transfer function θ, and $\alpha \in \{0,1\}^*$ is an *update annotation*. As explained in Sect. III-C, this parameter is used to guarantee that the parties agree on why a given update happen (see also Eq. (7) on p. 7). Since channel updating is a 2-phase process, the "confirmer" P' asks the environment if it agrees for an update. This is handled by messages $\underline{\texttt{update-requested}}$ and $\underline{\texttt{update-ok}}$. Finally, (D) *Ledger channel closing* is initiated via a message $(\underline{\texttt{lc-close}}, id)$. Note that the parties can play different roles in the virtual channels, e.g., it may happen that virtual channels γ and γ' are open over β, and $\beta.\mathsf{Alice}$ plays the roles of $\gamma.\mathsf{Alice}$ and $\gamma'.\mathsf{Ingrid}$ while $\beta.\mathsf{Bob}$ plays the roles of $\gamma.\mathsf{Ingrid}$ and $\gamma'.\mathsf{Alice}$, say. We emphasize that the description of the ideal functionality is significantly simplified due to the restrictions on the environment that we make, and which are described in full detail in Appx. B. Note that (unlike the "simplified" protocol in Sect. III) our functionality is fully concurrent, and in particular several channel updates can be performed simultaneously, and multiple virtual channels can be open over the same ledger channel β. The timing requirements in the ideal functionality are a consequence of our implementation choices and should become clearer after we present the technical details in Sect. V and in Appx. A (e.g. the fact that in Point (C) we wait up to 3 rounds comes from the way we handle the parallel updates, see Sect. Va).

Let us now discuss why the ideal functionality Channels from Fig. 2 satisfies the security requirements from Sec. III-B. *Consensus on channel opening and on channel update.* It is easy to see that Channels guarantees that there is always an agreement among the honest parties on whether a channel has been created or updated. This is achieved by the ideal functionality notifying the parties via messages $\underline{\texttt{lc-opened}}/\underline{\texttt{lc-not-opened}}$ or $\underline{\texttt{vc-opened}}/\underline{\texttt{vc-not-opened}}$ on whether a ledger/virtual channel has successfully been created. The same holds for updates executed via Channels. Similarly, by inspection of the ideal functionality it is easy to see that ledger channel opening is completed in $O(\Delta)$ rounds, and all other operations take constant time.

Guaranteed channel closing. A ledger channel β can be closed by any of the parties $P \in \beta.\mathsf{end\text{-}users}$ (if there does not exist a virtual channel that is currently active over β). This closing is completed within time at most 3Δ. If a virtual channel γ, that uses β as a sub-channel, is open, then the parties in $\beta.\mathsf{end\text{-}users}$ have to wait until it is closed. A virtual channel γ is closed "automatically" (see Step (2) Fig. 2 (B)) when time $\gamma.\mathsf{validity}$ comes and closing is completed within round $\gamma.\mathsf{validity} + 7\Delta + 5$ (in the pessimistic case), and within round $\gamma.\mathsf{validity} + 5$ (in the optimistic case).

Guaranteed balance payout for end users. When a ledger channel β is closed its latest balance of coins is added to users' account in the ledger (see Fig. 2 (D)). The same holds for a virtual channel γ, except that the coins are distributed back to the sub-channels (see Step (2a), Fig. 2 (B)).

Balance neutrality for intermediary Ingrid. Consider a virtual channel γ. Let $\beta_A := \Sigma(\gamma.\mathsf{subchan}(\gamma.\mathsf{Alice}))$ and $\beta_B := \Sigma(\gamma.\mathsf{subchan}(\gamma.\mathsf{Bob}))$. Observe that as a result of the virtual channel opening $\gamma.\mathsf{Ingrid}$ had $x_A := \gamma.\mathsf{balance}(\gamma.\mathsf{Bob})$ coins removed from her account of in β_A, and $x_B := \gamma.\mathsf{balance}(\gamma.\mathsf{Alice})$ coins from removed from her account in β_B (see Step (1b) Fig. 2 (B)). Later, in the closing procedure (see Step (2b), Fig. 2 (B)) she had $x'_A := \widehat{\gamma}.\mathsf{balance}(\gamma.\mathsf{Bob})$ coins added to her account of in β_A, and $x'_B := \widehat{\gamma}.\mathsf{balance}(\gamma.\mathsf{Alice})$ coins added to her account in β_B. Since channel updates cannot change the value of channel γ (as θ in the update requests has to be a transfer function), thus $x_A + x_B = x'_A + x'_B$. Therefore $(x_A - x'_A) + (x_B - x'_B) = 0$. Hence the balance neutrality holds.

V. An overview of the technical details

The details of the protocol and the contract appear in Appx. A. In this section we explain the main differences between our actual protocol and its informal presentation in Sect. III-C. Most of these issues are consequences of the fact that in our full model (unlike in the informal description) we consider fully parallel settings.

a) Concurrent channel updates. Let δ be a ledger or virtual channel. As mentioned above, one of the issues that needs to be handled is that it may happen that some updates are initiated by $\delta.\mathsf{Alice}$ and some by $\delta.\mathsf{Bob}$ in the same round. Note that in this case the protocol described in Sect. III-C would run into problems, since the parties would use the same value of w for two different updates (see Eq. (4)). To avoid this we define *P's update rounds for δ* as the rounds when P can send the first message in the update procedure (if the update has been requested by the environment in some other round then P waits for his update round to start). More precisely, for a channel δ round τ is called a *$\delta.\mathsf{Alice}$'s update round* if $\tau = 0 \pmod 4$ and it is called a *$\delta.\mathsf{Bob}$'s update round* if $\tau = 2 \pmod 4$. Since a channel update takes 2 rounds, therefore having only 1 in 4 rounds as an "update round" for P guarantees that the entire update procedure proposed by P

(A) **Opening a ledger channel**:

Upon receiving a message $(\underline{\texttt{lc-open}}, \beta)$ from $\beta.\mathsf{Alice}$ with $\beta.\mathsf{balance}(\beta.\mathsf{Alice})$ coins in round τ, where β is a ledger channel, proceed as follows:

1) Within round $\tau + \Delta$ remove $x_A := \beta.\mathsf{balance}(\beta.\mathsf{Alice})$ coins from $\beta.\mathsf{Alice}$'s account on the ledger $\mathcal{L}$.
2) If within time Δ after Step 1 was completed you receive a message $(\underline{\texttt{lc-open}}, \beta)$ from party $\beta.\mathsf{Bob}$, then remove $x_B := \beta.\mathsf{balance}(\beta.\mathsf{Bob})$ coins from $\beta.\mathsf{Bob}$'s account on the ledger $\mathcal{L}$, and add β to Σ. Output $(\underline{\texttt{lc-opened}})$ to parties in $\beta.\mathsf{end\text{-}users}$ and to the simulator $\mathcal{S}$, and stop.
   Otherwise, within time 2Δ after Step 1 was completed add x_A coins to the account of $\beta.\mathsf{Alice}$ on the ledger $\mathcal{L}$ and output $(\underline{\texttt{lc-not-opened}})$ to $\beta.\mathsf{Alice}$.

(B) **Opening and closing a virtual channel**:

1) Upon receiving a message $m = (\underline{\texttt{vc-open}}, \gamma)$ (where γ is a virtual channel) from *all* the parties in $\gamma.\mathsf{all\text{-}users}$ (within 2 rounds), do the following:
   a) for $P \in \gamma.\mathsf{end\text{-}users}$ remove $\gamma.\mathsf{balance}(P)$ coins from P's account in $\Sigma(\gamma.\mathsf{subchan}(P))$.
   b) remove $\gamma.\mathsf{balance}(\gamma.\mathsf{Bob})$ coins from the account of $\gamma.\mathsf{Ingrid}$ in $\Sigma(\gamma.\mathsf{subchan}(\gamma.\mathsf{Alice}))$, and $\gamma.\mathsf{balance}(\gamma.\mathsf{Alice})$ coins from the account of $\gamma.\mathsf{Ingrid}$ in $\Sigma(\gamma.\mathsf{subchan}(\gamma.\mathsf{Bob}))$.

   Then add γ to Σ, output $(\underline{\texttt{vc-opened}})$ to the parties in $\gamma.\mathsf{all\text{-}users}$, and go to Step 2.
   If within 2 rounds (from receiving m for the first time) you do not receive m from *all* the parties in $\gamma.\mathsf{all\text{-}users}$ then output $\underline{\texttt{vc-not-opened}}$ to them and stop.
2) Wait until round $\gamma.\mathsf{validity}$ (in the meanwhile accepting the "channel update" requests that concern γ, see below). When $\gamma.\mathsf{validity}$ comes let $\widehat{\gamma} := \Sigma(\gamma.\mathsf{id})$ be the current version of γ, and execute the following operations within round $\gamma.\mathsf{validity} + 7\Delta + 5$ (this is reduced to $\gamma.\mathsf{validity} + 5$ in the optimistic case, i.e., when all parties in $\gamma.\mathsf{end\text{-}users}$ are honest):
   a) for $P \in \gamma.\mathsf{end\text{-}users}$ add $\widehat{\gamma}.\mathsf{balance}(P)$ coins to P's account in $\Sigma(\gamma.\mathsf{subchan}(P))$.
   b) add $\widehat{\gamma}.\mathsf{balance}(\gamma.\mathsf{Bob})$ coins to the account of $\gamma.\mathsf{Ingrid}$ in $\Sigma(\gamma.\mathsf{subchan}(\gamma.\mathsf{Alice}))$, and $\widehat{\gamma}.\mathsf{balance}(\gamma.\mathsf{Alice})$ coins to the account of $\gamma.\mathsf{Ingrid}$ in $\Sigma(\gamma.\mathsf{subchan}(\gamma.\mathsf{Bob}))$.

   Output $(\underline{\texttt{vc-closed}})$ to the parties $\gamma.\mathsf{all\text{-}users}$ and erase $\widehat{\gamma}$ from Σ.

(C) **Ledger/virtual channel update**:

Upon receiving a message $m := (\underline{\texttt{update}}, id, \theta, \alpha)$ (such that there exists a channel $\delta \in \Sigma$ with identifier id) from a party $P \in \delta.\mathsf{end\text{-}users}$, where θ is a transfer function. If for some $P \in \delta.\mathsf{end\text{-}users}$, we have that $\delta.\mathsf{balance}(P) + \theta(P) < 0$ then ignore this message. Otherwise, within 3 rounds send a message $(\underline{\texttt{update-requested}}, id, \theta, \alpha)$ to $P' := \delta.\mathsf{other\text{-}party}(P)$.
If in the next round P' replies with a message $(\underline{\texttt{update-ok}})$ replace δ in Σ with a channel $\widehat{\delta}$ that is equal to δ, except that $\widehat{\delta}.\mathsf{balance} := \delta.\mathsf{balance} + \theta$ and send $(\underline{\texttt{updated}})$ to P.

(D) **Ledger channel closing**:

Upon receiving a message $(\underline{\texttt{lc-close}}, id)$ (such that there exists a ledger channel $\beta \in \Sigma$ with identifier id and there is no open virtual channel built over β) from a party $P \in \beta.\mathsf{end\text{-}users}$ do the following: within time 3Δ (this is reduced to 2Δ in the optimistic case, i.e, when both $\beta.\mathsf{end\text{-}users}$ are honest) add $\beta.\mathsf{balance}(\beta.\mathsf{Alice})$ coins to $\beta.\mathsf{Alice}$'s account on $\mathcal{L}$, and $\beta.\mathsf{balance}(\beta.\mathsf{Bob})$ coins to $\beta.\mathsf{Bob}$'s account on $\mathcal{L}$, respectively. Erase β from Σ and send $(\underline{\texttt{lc-closed}})$ to the parties in $\beta.\mathsf{end\text{-}users}$ and to the adversary.

Fig. 2: Functionality Channels. It maintains a channel space Σ that is initially empty.

will end before P' starts any procedure containing her update proposal. Note also that, since we assumed that the adversary cannot reorder messages sent from P to P' (see Sect. IV-A2), the version number w will remain synchronized between the parties.

b) Multiple simultaneous virtual channels over the same ledger channel. Another problem comes from the fact that our protocol allows the parties to open several virtual channels simultaneously over the same ledger channel. Recall that in the "*Virtual channel closing*" procedure in our informal description (see Sect. III-C2c) once Ingrid detected cheating of Alice or Bob, then it was ok for her to close the corresponding ledger channel. This was used in the procedure called "*Closing of β_A with simultaneous transfer of x coins from Alice to Bob*" (p. 8), where the channel β_A was closed and during this closing it was taken into account that x coins should be transferred from Alice to Bob. Unfortunately, this cannot be done in the parallel settings, since there may be other virtual channels that are still open in β_P, and hence their final balance is not yet known. We solve it in the following way. Firstly, we do *not* instruct Ingrid to request channel closing in this case. Instead, we let the contract instance $\mathcal{C}(\beta_P.\mathsf{id})$ (that corresponds to β_P) simply record the information about x in its memory. Observe that there may be multiple such x's that need to be stored in $\mathcal{C}(\beta_P.\mathsf{id})$ during the lifetime of β_P (each x coming from closing a different virtual channel that is constructed over β_P). To save space in the contract's storage we simply accumulate all of them by adding them together. Technically, this is done by defining a transfer function *transfer* : $\beta_P.\mathsf{end\text{-}users} \to \mathbb{R}$ that is initially equal to 0 on both inputs. This function keeps track on the amount of coins that needs to be transfer between the parties. That is: each time x coins are transferred from $\beta_P.\mathsf{Alice}$ to $\beta_P.\mathsf{Bob}$, the function is updated by letting *transfer*$(\beta_P.\mathsf{Alice}) :=$ *transfer*$(\beta_P.\mathsf{Alice}) - x$ and

transfer(β_P.Bob) := *transfer*(β_P.Bob)$+x$. This function will be kept in contract's storage until the contract is closed. During the channel closing it will be used to "correct" the amounts of coins that the parties receive. Suppose, for example, that the last balance of β_P on which that parties exchanged the signatures is $[\beta_P.\mathsf{Alice} \mapsto y_A, \beta_P.\mathsf{Bob} \mapsto y_B]$. Then as a result of closing the channel β_P.Alice will get $y_A +$ *transfer*(β_P.Alice) coins, and β_P.Bob will get $y_B +$ *transfer*(β_P.Bob) coins.

c) Triggering smart contract actions. In our informal description we assumed that the contract instances can act by themselves (see Sect. III-C and footnote 2). Unfortunately, the Ethereum contracts do not work this way as every action of an Ethereum contract needs to be triggered by a user. To see why this is a problem imagine a scenario where a party P should react to a message m that another party P' sent of a contract instance C, and this should happen within time Δ. If P does not react then C should execute some procedure[3].

Hence, C needs to be "woken up" by a message from P'. Our protocol takes care of such triggering. For example if time t comes then one of the parties (typically: the one that is financially interested in C taking the given action) sends a special message to C. Since this situation appears frequently in our protocol we introduce the following convention (that can be viewed as a "macro" for writing protocols). We say that P' *sends to* $\mathcal{C}(id)$ *a* Δ*-forced reply message* m if: (1) P' immediately sends a message m to $\mathcal{C}(id)$ (let τ be the time when $\mathcal{C}(id)$ receives m), (2) if P' does not receive a reply to m from $\mathcal{C}(id)$ within time $\tau + \Delta$ then she sends a message (`timeout`) to $\mathcal{C}(id)$. The "(`timeout`)" message serves precisely the purpose of "waking the contract up". Typically, after receiving it, the contract will check if time Δ indeed passed and if P has not replied to m'. If yes, then the contract instance C concludes that it found a fault that is uniquely attributable to P and "punishes P" (see below).

d) Punishing for uniquely attribute faults. When the contract instance $\mathsf{C} = \mathcal{C}(\beta.\mathsf{id})$ detects that one of the users P of channel β is cheating (e.g. P does not reply to a "forced reply" message) then it "punishes" P. Recall (see Assumption 1 in Sect. III-C) that in the informal description "punishing" meant that C transfers all coins to the honest party and terminates. In our real protocol we need to be slightly more careful, since it may happen that there are some virtual channels that are still open over β (note that this is similar to the problem described in paragraph (b) in this section). In this case if C disappears from the set of contract instances in $\mathcal{C}$ then in the virtual channel closing procedures the parties would be sending their messages to a contract instance that does not exist.

This problem is purely technical and it can be solved in several ways. We choose the following straightforward solution: "punishing for a uniquely attributable fault" will always concern coins in the channel that is involved in the procedure where the fault occurred. More precisely, if the fault happened when a channel γ is closed, then all the coins *deposited in* γ will be transferred to the honest party (this transfer will happen in the ledger channel) (technically this will be done by updating appropriately the *transfer* function in the contract, see subroutine (C) on Fig. 5, Appx. III-C2c).

We are now ready to state our main security theorem, whose proof sketch appears in Appx. C (its complete proof will be provided in the extended version of this paper).

Theorem 1: Assume the underlying signature scheme is existentially unforgeable against adaptive chosen-message attacks. Then the protocol `channels` running in the $\mathcal{C}$-hybrid world emulates an ideal functionality $\mathsf{Channels}$ with respect to a global ledger $\mathcal{L}$ and with blockchain delay Δ.

VI. Implementation and Evaluation

We created a simple proof of concept implementation of our particular ledger channel contract in Ethereum using the programming language Solidity. The source code is publicly available on https://github.com/PERUNnetwork/Perun. Our main goal was to illustrate the feasibility of our protocols and the underlying smart contracts. To this end, our implementation follows closely the protocol from Fig. 5 given in the appendix. More precisely, the `LedgerChannel` contract from the implementation corresponds to the ledger channel and function calls correspond to messages to the functionality $\mathcal{C}$ from Fig. 5. The contract uses as subroutine an external contact called `LibSignatures`. The role of this contract is that it provides a simple interface for the verification of signatures needed for our `LedgerChannel` contract. In addition to the contract code, we provide unit tests for all functions of the contracts written in Python. For the implementation of the unit test evaluation we use the *populus development framework* for Ethereum smart contracts.

An important evaluation criteria for smart contracts over Ethereum are costs due to fees, which in Ethereum are calculated via an internal currency called *gas*, which is paid by the users of the system to the miners. The amount of gas for each transaction heavily depends on the amount of data it sends and the complexity of the computation that follows. In our case, transactions provide input to and trigger the execution of the smart contract's functions. Moreover, the final price in Ether (Ethereum's currency) depends on the exchange rate between gas and Ether. This exchange rate is chosen by the sender of a transaction, and typically lies between $2 \cdot 10^{-9}$ Ether (in this case the transaction waiting time is about 5 minutes) and $2 \cdot 10^{-8}$ Ether (in this case the waiting time is about 30s) for 1 gas. In our calculation we use the exchange rate 1 gas = $4 \cdot 10^{-9}$ Ether corresponding to an approximate waiting time of about 50s. Using an Ether exchange rate of 500 USD deployment of the `LedgerChannel` contract costs about 0.011 Ether (2757111 gas, approx. 5.51 USD). We emphasize that in our prototype implementation we were not aiming at optimizing gas costs. A straightforward approach to lower the fees significantly would be to exclude all functionality of the `LedgerChannel` contract into an external library, like it was done for the signature evaluation. This results into much lower fess for creating the channel as the main parts of the contract

[3]This happens, e.g., in point *"msg_A^1 not sent"* of our informal description (p. 7): the contract instance C_A expects Alice to send msg_A^1 to it within time Δ, and if Alice does not send it, then C_A punishes her.

are deployed only once in form of a library and users can then re-use this functions for their ledger channel instance. For this reason we analyze the costs of running our protocol without considering deployment costs.

	# on-chain trans-actions	cost			# off/ on chain msg		# signa-tures
		Gas	ETH	USD	end users	Ingrid	
Open LC	2	62337	0.00025	0.12	0/2	0/0	2
Update LC	0	0	0	0	2/0	0/0	2
Open VC	0	0	0	0	4/0	2/0	4
Update VC	0	0	0	0	2/0	0/0	2
VC Closing:							
optimistic	0	0	0	0	4/0	2/0	6
pessimistic	3	418318	0.00167	0.84	4/2	2/1	6
LC Closing:							
optimistic	2	147788	0.00099	0.50	2/2	0/0	2
pessimistic	2	275049	0.00110	0.55	2/2	0/0	2

TABLE I: The costs for executing a single ledger channel contract as well as the message and computational complexity of the Perun protocol. Above, LC denotes a "ledger channel" and VC denotes a "virtual channel". The column "# messages" counts the number of messages both end users (accumulated) or the intermediate Ingrid have to send and the column"# signatures" corresponds to pairs of operations: sign and verify.

Table I displays the execution costs and message complexity of running the Perun protocol using the `LedgerChannel` contract. For the execution costs we focused on the costs of computing digital signatures, which is the dominating computational cost. By message complexity of *EndUsers* we count the total off-chain messages and on-chain transactions that Alice and Bob send for each function call in a ledger or virtual channel. To make the overhead of the intermediary explicit, we count each message that Ingrid sends in an individual column of the table. If both Alice and Bob jointly agree to open, update and close the channel (the optimistic case) they need to execute four on-chain transactions and pay less than 0.00124 Ether (excluding the deployment cost for the code libraries). Specifically, both parties have to send one transaction each for the opening and closing. To measure the costs for disagreement we always consider the worst possible case with most on-chain transactions and highest gas costs (pessimistic case). If either Alice or Bob tries to close the ledger channel with an outdated state while a virtual channel is still active (LC close pessimistic) the other party sends a proof of a (newer) version with an open virtual channel to the smart contract. Settling this disagreement in the smart contract raises the costs for both parties to 0.00135 Ether. If the parties go to the smart contract in order to dispute over the virtual channels, they need to additionally pay 0.00167 Ether for every open virtual channel. Note that in this case, Ingrid needs to participate in the on-chain dispute on behalf of one of the parties. In the most costly scenario, she needs to request the closing of the virtual channels, then wait for the other party (e.g. Alice) to make a move, and send another transaction to the blockchain to finalize the dispute. This worst case scenario limits the fees Ingrid can be forced to pay in the most unfortunate outcome.

Let us now take a look at the message complexity of our protocol, i.e., the number of messages sent between the parties involved in the protocol. Notice that most of such messages consist of a subset of two Ethereum addresses, eight integers (three channel ids, the cash distribution, the validity and a version number) and two signatures over all of these values. As signatures are the dominating factor for both message length and computation complexity, in each step we highlight how many signatures need to be generated and verified. For the ledger channel opening and closing procedures the message complexity is similar to that of existing payment network systems like Lightning [11] and Sprites [10]. The main advantage of Perun is the fact that the virtual payment channel can be updated instantaneously by the two parties without sending messages to the intermediaries. This means that after a virtual channel is set up, it can be updated without additional delays by sending only two update messages. The new version, new balances and a signature is send by the sender and the receiver responds with a single signature. Sending the same transaction through one relay in hashlocked-based systems requires the computation of at least six signatures and the intermediary has to receive, compute and send at least two messages. In other systems this is even higher. The message complexity limits the effective throughput of how many transactions can be sent over such a system per second.

VII. Conclusion

We introduced an off-chain payment channel system called *Perun*. Its main advantage over the existing solutions is that it allows to create the *virtual channels*, which are channels of length 2 that do not require interacting with the intermediary for every payment. The security of our protocol is defined in the UC framework and is formally proven. Our work can be generalized in many directions. Longer state channels are described in subsequent work [4]. One can also ask if it is possible to create a scheme in which the intermediaries do not need to block the coins that are used for constructing virtual channels. This can be done by slightly relaxing the security guarantees. Namely, one can replace the full cheating-resilience (that has been assumed in this work), by a weaker notion of "cheating-*evidence*". We leave formalizing this as an interesting future direction.

References

[1] R. Canetti. "Universally Composable Security: A New Paradigm for Cryptographic Protocols". In: *FOCS*. 2001.

[2] R. Canetti et al. "Universally Composable Security with Global Setup". In: *TCC*. 2007.

[3] C. Decker and R. Wattenhofer. "A Fast and Scalable Payment Network with Bitcoin Duplex Micropayment Channels". In: *SSS 2015*. 2015.

[4] S. Dziembowski, S. Faust, and K. Hostakova. *Foundations of State Channel Networks*. manuscript. 2017.

[5] M. Green and I. Miers. "Bolt: Anonymous Payment Channels for Decentralized Currencies". In: *CCS*. 2017.

[6] R. Khalil and A. Gervais. "Revive: Rebalancing Off-Blockchain Payment Networks". In: *CCS*. 2017.
[7] A. E. Kosba et al. "Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts". In: *IEEE S&P*. 2016.
[8] J. Lind et al. "Teechan: Payment Channels Using Trusted Execution Environments". In: *CoRR* (2016).
[9] G. Malavolta et al. "Concurrency and Privacy with Payment-Channel Networks". In: *CCS*. 2017.
[10] A. Miller et al. "Sprites: Payment Channels that Go Faster than Lightning". In: *CoRR* (2017).
[11] J. Poon and T. Dryja. *The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments*. 2016.
[12] S. Roos et al. "Settling Payments Fast and Private: Efficient Decentralized Routing for Path-Based Transactions". In: *NDSS*. 2018.

APPENDIX

A. *The* channels *protocol*

The protocol channels is presented on Figs. 3 and 4, and the contract functionality $\mathcal{C}$ appears on Fig. 5. The contract consists of the following parts: (A) the part used for constructing a given contract instance, (B) the main execution part of the contract, and (C) a subroutine for punishing users of a virtual channel for uniquely attributable faults (see Sect. V-0d, p. 12). The assumption that for every channel δ each party P can send at most one message of a given type that concerns δ (see point (B)) is a technical restriction that simplifies the presentation (by "message type" we mean the keyword that starts the message). It essentially means that, e.g., no party can ask to close the same channel twice.

Recall that in Sect. III-C in order to update a channel δ the parties exchange signed messages V (if δ was a ledger channel) and W (if δ was virtual). In the formal description we use the following terminology to describe such tuples. Let $w \in \mathbb{N}$ be a natural number called a *version number*, and $\alpha \in \{0,1\}^*$ be an update annotation (see Sect. IV-B). Then $(\widehat{\delta}, w, \alpha)$ is called *a version of* δ if $\widehat{\delta}$ is equal to δ on all attributes except of δ.balance, and the value of $\widehat{\delta}$ is equal to the value of δ. Moreover, $(\widehat{\delta}, w, \alpha, \sigma)$ is called *a version of* δ *signed by* P if $(\widehat{\delta}, w, \alpha, \sigma)$ is a tuple signed by P. If $w = 0$ then we call $(\widehat{\delta}, w, \alpha)$ the *initial* version of δ, and in the "signed" tuple $(\delta, w, \alpha, \sigma)$ we allow $\sigma = \perp$. The *winner selection procedure* Win serves to determine which version of a channel is newer. It was already implicitly defined for V's and W's in Sect. IV-B. Formally it is defined as follows. Let δ be a ledger or virtual channel. Win takes as input a pair $((\delta^0, w^0, \alpha^0, \sigma^0), (\delta^1, w^1, \alpha^1, \sigma^1))$ of signed versions of δ, and returns as output a cash function $\theta : \delta.\mathsf{end\text{-}users} \to \mathbb{R}_{\geq 0}$ defined as follows: let i be such that $w^i > w^{1-i}$ (if no such i exists then choose $i := 0$) and then let $\theta := \delta^i.\mathsf{balance}$. Since the protocol has already been informally presented in Sections III-C and V), we only focus on the concrete details that were not explained there.

Let us start with the procedures from Fig. 3. To open a channel β, in Step 1 (Fig. 3, part (A)) party β.Alice sends to $\mathcal{C}$ a contract constructor message for $\mathcal{C}(\beta.\mathsf{id})$ together with $\beta.\mathsf{balance}(\beta.\mathsf{Alice})$ coins. This is a "Δ-forced reply message" meaning that β.Alice sends a (timeout) message if she does not receive a reply from $\mathcal{C}(\beta.\mathsf{id})$ within time Δ after the contract instance $\mathcal{C}(\beta.\mathsf{id})$ appeared on the ledger. The part of the contract that handles the ledger channel opening appears on Fig. 5 (A). The contract defines a transfer function *transfers* : $\beta.\mathsf{end\text{-}users} \to \mathbb{R}$ initially equal to 0 on both inputs. As explained in Sect. V.b this function will keeps track on the sum of the transfers between β.Alice and β.Bob that were communicated to the contract. In our case these transfers will come only from the closing of virtual channels, see Steps (4a) and (4c) on Fig. 5 (B)). The contract also stores information about virtual channels (built on top of β) that were closed "via the contract". Technically, we say that some channel γ is *marked as closed* if it is added to the list of such closed channels. The contract sends a message (lc-opening, β) to β.Bob informing him about the fact that β.Alice initiated ledger channel opening. Once the contract gets the confirmations message lc-open from β.Bob (together with Bob's coins) then the channel is opened. If this message does not arrive to the contract within time Δ then β.Alice "automatically" sends a (timeout) message, and she gets her coins back. It is easy to see that the timing conditions from Fig. 2 (A) are satisfied.

Channel updating (Fig. 3) is done in the way described in Sect. III-C, with messages "updating" and "update-ok" playing the roles of messages W_A and W_B on Eq. 5 (p. 5). Note that channel updating can take up to 4 rounds due to the restriction from Sect. Va.

The ledger channel closing procedure appears on Fig. 3 (C), and the corresponding part of the contract is described in Step 4 of Fig. 5 (B). It is performed as described in Sect. III-C, with P and P' playing roles of Alice and Bob, and messages (lc-close, W) and (lc-close, W') corresponding to W_A and W_B. Message "lc-closing" is used by $\mathcal{C}(\beta.\mathsf{id})$ to communicate to party P' that P requested channel closing. Message "vc-active" is used to communicate to the contract (in Step (2a)) that there is a virtual channel still open over the ledger channel β (see Sect. III-C2). This is handled by the contract in Step (4b), Fig. 5 (B). Note that in the optimistic case this procedure takes time 2Δ (one Δ for proposing the closing, and one for confirming). In the pessimistic case it takes 3Δ since P sends the (timeout) message the latest in time 2Δ, and it up to one additional Δ for the contract to process it.

The virtual channel opening procedure is presented on Fig. 4 (A). It follows the outline from Sect. III-C. The virtual channel closing procedure is also done as in Sect. III-C, with the following naming conventions. The "vc-close-init" messages sent by $P \in \gamma.\mathsf{end\text{-}users}$ correspond to messages msg_P^1 on Fig. 1. The updates proposed by γ.Ingrid in Step 3 correspond to messages msg_2^A and msg_2^B. These updates are annotated with strings "channel γ.id closed" since they correspond to messages of a form as on Eq. (7) (p. 7). Recall that in Sect. III-C on p. 7 we considered two cases of malicious

(A) **Opening ledger channel** β:

1) Upon receiving a message $(\underline{\texttt{lc-open}}, \beta)$ from the environment, party $\beta.\mathsf{Alice}$ starts a new thread, sends to $\mathcal{C}$ a Δ-forced reply contract constructor message $(\texttt{lc-open}, \beta)$ together with $\beta.\mathsf{balance}(\beta.\mathsf{Alice})$ coins, and goes to Step 3.
2) Upon receiving a message $(\underline{\texttt{lc-open}}, \beta)$ from the environment, party $\beta.\mathsf{Bob}$ waits time at most Δ to receive a message $(\texttt{lc-opening}, \beta)$ from $\mathcal{C}(\beta.\mathsf{id})$. Consider the following cases.
   a) He receives this message — then he replies to $\mathcal{C}(\beta.\mathsf{id})$ with a message $(\texttt{lc-open})$ together with $\beta.\mathsf{balance}(\beta.\mathsf{Bob})$. He then waits to receive a message $(\texttt{lc-opened})$ from $\mathcal{C}(\beta.\mathsf{id})$. Once received, he outputs $(\underline{\texttt{lc-opened}})$ and goes to Step 4.
   b) He does not receive this message — then he outputs $(\underline{\texttt{lc-not-opened}})$ and goes idle.
3) Party $\beta.\mathsf{Alice}$ waits for one of the following:
   a) She receives a message $(\texttt{lc-not-opened})$ from $\mathcal{C}(\beta.\mathsf{id})$ — then she outputs $(\underline{\texttt{lc-not-opened}})$ and goes idle.
   b) She receives a message $(\texttt{lc-opened})$ from $\mathcal{C}(\beta.\mathsf{id})$ — then she outputs $(\underline{\texttt{lc-opened}})$ and goes to Step 4.
4) Party $P \in \beta.\mathsf{end\text{-}users}$ goes in the idle state waiting for messages that concern β and come from the environment, $\beta.\mathsf{other\text{-}party}(P)$ or from the contract. These messages are handled in Points (B) and (C) below.

(B) **Update of (ledger or virtual) channel** δ:

1) Upon receiving message $(\underline{\texttt{update}}, id, \theta, \alpha)$ (where id is an identifier of some channel δ) from the environment party P (the "initiator") waits for the next P's update round of δ. When this round comes P lets $(\widehat{\delta}_P, w_P)$ denote the last version of δ that P is aware of, and lets $\widetilde{\delta}$ be equal to $\widehat{\delta}_P$ except that $\widetilde{\delta}.\mathsf{balance} := \widehat{\delta}_P.\mathsf{balance} + \theta$. Then she sends a tuple $(\texttt{updating}, (\widetilde{\delta}, w_P + 1, \alpha, \sigma))$ (where σ is P's signature on $(\widetilde{\delta}, w_P + 1, \alpha)$) to P', waits 1 round, and goes to Step 3
2) This step starts when $P' \in \delta.\mathsf{end\text{-}users}$ (the "confirmer") receives a correctly signed message $(\texttt{updating}, (\widetilde{\delta}, w, \alpha, \sigma))$. Then P' lets $(\widehat{\delta}_{P'}, w_{P'})$ denote the last version of δ that P' is aware of. If $w \neq w_{P'} + 1$ then P' ignores this message. Otherwise P' computes $\theta' := \widetilde{\delta}.\mathsf{balance} - \widehat{\delta}_{P'}.\mathsf{balance}$ and outputs $(\underline{\texttt{update-requested}}, \beta.\mathsf{id}, \theta', \alpha)$ to the environment.
   If the environment replies with $(\underline{\texttt{update-ok}})$ then P' computes her signature σ' on $(\widetilde{\delta}, w, \alpha)$, sends a message $(\texttt{update-ok}, (\widetilde{\delta}, w, \alpha, \sigma'))$ to P' and stops this procedure. Otherwise P' stops this procedure.
3) If P receives a message $(\texttt{update-ok}, (\widetilde{\delta}_P, w_P + 1, \alpha, \sigma'))$ (where the tuple is signed by P') then she outputs $(\underline{\texttt{updated}})$ and stops this procedure. Otherwise P outputs $(\underline{\texttt{not-updated}})$ and stops this procedure.

(C) **Closing the ledger channel with identifier** id:

1) Upon receiving a message $(\underline{\texttt{lc-close}}, id)$ (where id is an identifier of some ledger channel β) from the environment party P lets $(\beta, 0)$ be the initial version of the channel with identifier id and lets V be the last signed version of β which P received from $\beta.\mathsf{other\text{-}party}(P)$ (if P has never received such a version then she lets $V = (\beta, 0, \varepsilon, \bot)$). She sends to $\mathcal{C}(id)$ a Δ-forced reply message $(\texttt{lc-close}, W)$.
2) Upon receiving (in some round τ) a message $(\texttt{lc-closing})$ from $\mathcal{C}(\beta.\mathsf{id})$ party P' does the following:
   a) If earlier she received an opening certificate z of $P := \beta.\mathsf{other\text{-}party}(P)$ on some virtual channel γ that is constructed over virtual channel β and $\gamma.\mathsf{validity} + 7\Delta + 5 > \tau$ — then she sends to $\mathcal{C}(\beta.\mathsf{id})$ a message $(\texttt{vc-active}, z)$. Then she continues waiting.
   b) Otherwise she sends to $\mathcal{C}(\beta.\mathsf{id})$ a message $(\texttt{lc-close}, W')$, where W' is the last signed version of β that she received from P (if she have never received such a version then she lets $V' = (\beta, 0, \varepsilon, \bot)$). Upon receiving a message $(\texttt{lc-closed})$ from $\mathcal{C}(\beta.\mathsf{id})$ she outputs $(\underline{\texttt{lc-closed}})$ and stops goes idle.
3) Upon receiving a message $(\texttt{lc-closed})$ from $\mathcal{C}(\beta.\mathsf{id})$ party P outputs $(\underline{\texttt{lc-closed}})$ and goes idle.

Fig. 3: The procedures: (A) "opening ledger channel", (B) "update of channel δ", and (C) "closing the ledger channel".

behavior of the parties. The actions of $\gamma.\mathsf{Ingrid}$ in the first case (*"msg_P^1 not sent"*) are described in Step (2b) (Fig. 1 (B)), where $\gamma.\mathsf{Ingrid}$ sends a vc-close-init message to the contract. The contract receives this message in Step (1a) (Fig. 5 (B)) and acts as described in Sect. III-C The second case (*"msg_P^3 not sent"*) is handled in Step 5 (Fig. 1 (B)). Recall that in our informal description $\gamma.\mathsf{Ingrid}$ had to send P's opening certificate on γ and messages msg_A^1 and msg_B^1 to the contract. In the formal description these values correspond to oc_P and pairs $(V_{\gamma.\mathsf{Bob}}, S_{\gamma.\mathsf{Alice}})$ and $(V_{\gamma.\mathsf{Alice}}, S_{\gamma.\mathsf{Bob}})$) (respectively), which are sent to $\mathcal{C}(\beta_P.\mathsf{id})$ in the "vc-close-final" message.

Note also that in Step (4a) (on Fig. 4 (B)) in case P in the past did not receive a confirmation on her last update message (that contained a channel tuple $\widehat{\gamma}$) then she accepts that $\gamma.\mathsf{Ingrid}$ transfers to her the amount of coins that she should get from $\widehat{\gamma}$ (and not from $\gamma_{P'}$). This is needed since $\gamma.\mathsf{Ingrid}$ has no way to find out what happened between P and P' when they were updating γ (more concretely: she does not know if indeed P' did not confirm the update).

Let us now look at how much time, pessimistically, $\gamma.\mathsf{Ingrid}$ needs to close γ (i.e. what is the value of T_{max} discussed in Sect. III-C). First, she needs to wait 1 round to receive the "vc-close" messages from both Alice and Bob. If she does not receives any of them, then she needs to let the contract know about it by sending a "vc-close-init" message to the contract. Receiving this message takes time Δ. Then, the end-party has

to respond to the contract (which takes another Δ time), and if she does not respond then γ.Ingrid sends a (timeout) message (receiving this takes another Δ time). Then γ.Ingrid initiates a channel update procedure (that takes at most 4 rounds). If this is unsuccessful then she sends a message "vc-close-final" to the contract (which takes one more Δ time). This is received by the contract in time at most Δ. Hence within time $\gamma + T_{\text{max}}$, where $T_{\text{max}} = 4\Delta + 5$: either γ.Ingrid closed the channel γ, or the contract received a message "vc-close-final". The end-party either responds to "vc-close-final" with a "vc-already-closed" message, or another (timeout) message is needed (which in total takes time 2Δ). If γ.Ingrid did not close γ within time $\gamma + T_{\text{max}}$ and then the end-parties have to close it. It is easy to see that it takes time at most 3Δ (one Δ for the "vc-close-timeout" message, another one for waiting for γ.Ingrid's response, and yet another one of the (timeout) message). Thus, pessimistically, the virtual channel closing takes time $\gamma+T_{\text{max}}+3\Delta = 7\Delta+5$. Optimistically, the virtual channel closing procedure takes 5 rounds (1 round for the "vc-close" messages, and 4 rounds for channel update).

B. Restrictions on the environment

Below we list the restrictions on the environment that were mentioned in Sect. IV-A4. (1) The environment never asks the parties to open a channel δ such that δ.id already exists, or when the parties do not have enough funds. (2) If the environment asks the parties to open a virtual channel γ then the channel with identifiers specified in γ.subchan exists in Σ, and no closing procedure for them has been initiated. (3) The environment never asks to close a ledger channel in time earlier than γ.validity $+ 7\Delta + 5$ where γ is a virtual channel whose opening has been initiated by the environment (even if this opening was unsuccessful). (4) If the environment asks one of the parties $P \in \delta$.all-users to open a channel δ, then it asks all the other parties in δ.all-users to do the same (in the same round). (5) The environment does not perform (or confirm) any update procedures for channels whose closing has been initiated. (6) If a previous update of a channel δ failed, then the environment will not request a new update of δ. (7) The environment always confirms an update that it initiated, and never confirms an update which she did not initiate. A consequence of these restrictions is that in our protocol we can assume that all the honest parties have the same view on what channels should be open. For example: β.Alice knows that if she received a $(\underline{\texttt{vc-open}}, \beta)$ message from the environment then β.Bob also received such a message (in the same round). This, in particular, means that if β.Bob refuses to participate in the procedure of opening channel β then she must be corrupt.

C. Security analysis: proof sketch of Thm. 1

We have already informally argued about the security of our scheme while presenting it in the previous sections. Here we focus on describing the simulator $\mathcal{S}$ for some fixed adversary $\mathcal{A}$. Recall that $\mathcal{S}$ interacts with the environment $\mathcal{Z}$ and the ideal functionality Channels (via the so-called "dummy" parties, see [1]), and its goal is to "emulate" the behavior of $\mathcal{A}$ for the environment. At the beginning the simulator $\mathcal{S}$ starts the adversary $\mathcal{A}$ and corrupts the parties that $\mathcal{A}$ corrupts. The simulator also generates the (public key, private key) pairs for all the users. He passes the public keys of all the users to the corrupt users, and to each corrupt P_i he also sends his private key sk_i. Then $\mathcal{S}$ simulates the behavior of $\mathcal{A}$, and watches the instructions of $\mathcal{A}$ to the corrupt parties. Depending on the behavior of the simulated $\mathcal{A}$ the simulator sends inputs of his choice to the Channels functionality. To make it impossible to distinguish between the simulated and the real execution, the simulator needs to emulate the messages sent to the corrupt parties by the $\mathcal{C}$ functionality and by the other (honest) parties. Recall also that the adversary $\mathcal{A}$ "controls the network" meaning that he decides when the messages are delivered, subject to some timing constraints. In particular, we assumed that sending message to $\mathcal{C}$ takes time at most Δ. In our protocol the honest parties always send messages to $\mathcal{C}$ early enough so that they reach $\mathcal{C}$ before its "too late" (e.g. an honest β.Bob always sends the lc-opening message to $\mathcal{C}$ immediately after receiving message lc-open from β.Alice in Step (2a) on Fig. 3 (A)). On the other hand, the corrupt parties, may send such messages at any time they want. Therefore, our simulator has to observe the network and watch how much delay $\mathcal{A}$ introduces when delivering a given message m to $\mathcal{C}$, and based on this decide whether m was delivered "on time" or not. Below we describe how different parts of the channels protocol are handled by $\mathcal{S}$. It is easy to see that the only non-trivial cases are when some of the parties participating in a given part of the protocol (e.g.: δ.all-users) are corrupt and some are honest.

a) Ledger channel opening. This part starts when $\mathcal{Z}$ sends an $(\underline{\texttt{lc-open}}, \beta)$ message to both β.end-users in some round τ. Simulating it is straightforward: $\mathcal{S}$ simply simulates the contract functionality, plays the role of the honest party to the corrupt one, and removes the coins from the ledger when the parties send messages with coins to the contract (and refunds this money to β.Alice if the channel is not open).

b) Channel updating. The part for updating a ledger or virtual channel δ starts when $\mathcal{Z}$ sends an $m = (\underline{\texttt{update}}, id, \theta, \alpha)$ message to the update initiator $P \in \delta$.end-users. If P is corrupt then $\mathcal{S}$ sends m to P. If in the next round P sends the updating message to $P' := \delta.\mathsf{other\text{-}party}(P)$ (with all the parameters computed correctly) then $\mathcal{S}$ sends m (in the name of P) to the ideal functionality Channels. Then in the next round $\mathcal{S}$ sends to P the confirmation message from P' (recall that by Restriction (7) in Appx. B we assumed that the environment always confirms such updates). Note that this requires signing messages with P' private key, but $\mathcal{S}$ can do it, since he knows the private keys of all the parties.

Simulating the update procedure is a bit more tricky in case when the initiator P is honest and the confirmer P' is corrupt. This is because the confirmer may not send his signature on the updated channel state back to P, but, since he already knows P's signature on it, he can use it when the channel is closed. We distinguish two cases. The first case is when

(A) **Opening virtual channel** γ:

1) Upon receiving a message $(\underline{\texttt{vc-open}}, \gamma)$ from the environment each party $P \in \gamma.\mathsf{end\text{-}users}$ sends to $\gamma.\mathsf{Ingrid}$ her opening certificate on γ, waits one round and goes to Step 3
2) Upon receiving a message $(\underline{\texttt{vc-open}}, \gamma)$ from the environment party $\gamma.\mathsf{Ingrid}$ waits one round to receive opening certificates on γ of both $P \in \gamma.\mathsf{end\text{-}users}$. Consider the following cases.
   a) She receives both opening certificates: then she replies to each $P \in \gamma.\mathsf{end\text{-}users}$ with her opening certificate on γ. Then she outputs $(\underline{\texttt{vc-opened}})$, waits until round $\gamma.\mathsf{validity}$ and then goes to the "(B) *Virtual channel closing*" procedure.
   b) Otherwise: she outputs $(\underline{\texttt{vc-not-opened}})$ and stops.
3) If a party $P \in \gamma.\mathsf{end\text{-}users}$ receives an opening certificate of $\gamma.\mathsf{Ingrid}$ on γ from $\gamma.\mathsf{Ingrid}$ then she forwards this certificate to $\gamma.\mathsf{other\text{-}party}(P)$, outputs $(\underline{\texttt{vc-opened}})$ and goes to Step 4 below.
4) Party $P \in \gamma.\mathsf{all\text{-}users}$ goes in the idle state waiting for the "channel update" messages that concern γ and come from the environment or $\gamma.\mathsf{other\text{-}party}(P)$ or from the contract. These messages are handled by the procedure described of Fig. 3 (B). When time $\gamma.\mathsf{validity}$ comes P goes to Point (B) below.

(B) **Virtual channel closing**:

For $P \in \gamma.\mathsf{end\text{-}users}$ let β_P denote the channel with identifier $\gamma.\mathsf{subchan}(P)$, and let $(\gamma_0, 0)$ be the initial version of channel γ. For $P \in \gamma.\mathsf{all\text{-}users}$ let oc_P denote the opening certificate of P on γ. If $P \in \gamma.\mathsf{end\text{-}users}$ then P' denotes $\gamma.\mathsf{other\text{-}party}(P)$.

1) In round $\gamma.\mathsf{validity}$ each $P \in \gamma.\mathsf{end\text{-}users}$ lets $V_{P'} := (\gamma_{P'}, w_{P'}, \alpha_{P'}, \sigma_{P'})$ be the latest signed version of γ that P received from P'. If P never received a signed version of γ from P' (which means that no updates of γ have been performed) then P lets $V_{P'} := (\gamma_0, 0, \varepsilon, \bot)$. Then P sends to $\gamma.\mathsf{Ingrid}$ a tuple $(\texttt{vc-close}, V_{P'}, \mathsf{Sign}_P(V_{P'}))$ and goes to Step 4.
2) In round $\gamma.\mathsf{validity} + 1$ party $\gamma.\mathsf{Ingrid}$ does the following for each $P \in \gamma.\mathsf{end\text{-}users}$:
   a) If she receives a correctly formated $(\texttt{vc-close}, V_{P'}, S_P)$ message from P then she goes to Step 3.
   b) Otherwise she sends a Δ-forced reply message $(\texttt{vc-close-init}, oc_P)$ to $\mathcal{C}(\beta_P.\mathsf{id})$. If she then receives a message $(\texttt{vc-close}, V_{P'}, S_P)$ from $\mathcal{C}(\beta_P.\mathsf{id})$ then she goes to Step 3. Otherwise she receives a message $(\texttt{vc-closed})$ — in this case she sets $V_{P'} := (\gamma_0, 0, \varepsilon, \bot)$ and $S_P := \bot$ and goes to Step 3.
3) Party $\gamma.\mathsf{Ingrid}$ waits to learn $(V_{P'}, S_P)$ for *both* $P \in \gamma.\mathsf{end\text{-}users}$ (either by getting $(V_{P'}, S_P)$ directly from a party, or "via the contract" in Step (2b)). She then lets $\theta := \mathsf{Win}(V_{\gamma.\mathsf{Alice}}, V_{\gamma.\mathsf{Bob}})$. Then for each $P \in \gamma.\mathsf{end\text{-}users}$ she proposes an update of β_P that adds $x := \theta(P) - \gamma_0.\mathsf{balance}(P)$ coins to P's account and $-x$ coins to $\gamma.\mathsf{Ingrid}$'s account and is annotated with a string "`channel` $\gamma.\mathsf{id}$ `closed`", and goes to Step 5.
4) Party $P \in \gamma.\mathsf{end\text{-}users}$ waits for one of the following events to happen:
   a) Party $\gamma.\mathsf{Ingrid}$ proposes an update to ledger channel β_P that adds $\gamma_{P'}.\mathsf{balance}(P) - \gamma_0.\mathsf{balance}(P)$ coins to P's account and is annotated with a string "`channel` $\gamma.\mathsf{id}$ `closed`": P confirms this update, outputs $(\underline{\texttt{vc-closed}})$ and goes to Step 6. (In case P in the past did not receive a confirmation on her last update message $(\texttt{updating}, (\widehat{\gamma}, \widehat{w}, \widehat{\alpha}, \widehat{\sigma}))$ she also accepts updates that add $\widehat{\gamma}.\mathsf{balance}(P) - \gamma_0.\mathsf{balance}(P)$ coins to her account.)
   b) Party P receives a message $(\texttt{vc-closing}, \gamma.\mathsf{id})$: $\mathcal{C}(\beta_P.\mathsf{id})$: party P replies $(\texttt{vc-closing}, V_{P'}, \mathsf{Sign}_P(V_P))$ and continues waiting.
   c) Within round $\gamma.\mathsf{value}+4\Delta+5$ none of the above happens: P sends a Δ-forced reply message $(\texttt{vc-close-timeout}, oc_{\gamma.\mathsf{Ingrid}})$ to $\mathcal{C}(\beta_P.\mathsf{id})$, outputs $(\underline{\texttt{vc-closed}})$ and and continues waiting.
   d) Party P receives a message $(\texttt{vc-closed})$ from $\mathcal{C}(id)$: party P outputs $(\texttt{vc-closed})$ and goes to Step 6.
5) For each of the update procedures proposed by her in Step (3) $\gamma.\mathsf{Ingrid}$ does the following:
   a) If the update procedure is successful then she outputs $(\underline{\texttt{vc-closed}})$ and goes to Step 6.
   b) Otherwise she sends to $\mathcal{C}(\beta_P.\mathsf{id})$ a Δ-forced reply message $(\texttt{vc-close-final}, oc_P, (V_{\gamma.\mathsf{Bob}}, S_{\gamma.\mathsf{Alice}}), (V_{\gamma.\mathsf{Alice}}, S_{\gamma.\mathsf{Bob}}))$. Once she receives a message $(\texttt{vc-closed})$ from $\mathcal{C}(\beta_P.\mathsf{id})$ she outputs $(\underline{\texttt{vc-closed}})$ and stops this procedure.
6) A party $P \in \gamma.\mathsf{all\text{-}users}$ goes in to an idle state. If at any point later P receives a message from $\mathcal{C}$ that concerns channel γ then P answers with $(\texttt{vc-already-closed}, cc)$, where cc is the closing certificate on γ (see Sect. III-C2c).

Fig. 4: The procedures: (A) "opening virtual channel" and (B) "virtual channel closing".

the transfer is beneficial to P', i.e., $\theta(P') > 0$. In this case we will assume that even if P' does not immediately send her signature on the updated state to P then the update did happen. Intuitively, this is because any "rational" P' will use this new updated state during the channel closing. Hence $\mathcal{S}$ sends $(\underline{\texttt{update-ok}})$ in the name of P' to the ideal functionality $\mathsf{Channels}$, no matter if P' sends his signature on the new state to P or not. Of course, an "irrational" P' can still use the old channel state when the channel is closed, and get less coins than he could get by posting the newest version of the state. This is not a problem, since $\mathcal{S}$ can always remove these extra coins from the account of P' and move them to the account of P immediately after channel closing has been performed. The case when $\theta(P') \leq 0$ is symmetric. In this situation, if the simulated corrupt P' does not send his confirmation on the update immediately to P, then $\mathcal{S}$ does not send (in the name

(A) **The contract for channel β opening**:

Upon receiving a contract constructor message $(\underline{\texttt{lc-open}}, \beta)$ with $\beta.\mathsf{balance}(\beta.\mathsf{Alice}))$ coins from $\beta.\mathsf{Alice}$ start a new contract instance with identifier $\beta.\mathsf{id}$. More precisely: send a message $(\texttt{lc-opening}, \beta)$ to $\beta.\mathsf{Bob}$ and wait for one of the following messages:

1) $(\texttt{lc-open})$ together with $\beta.\mathsf{balance}(\beta.\mathsf{Bob})$ coins from $\beta.\mathsf{Bob}$: then let *transfers* : $\beta.\mathsf{end\text{-}users} \to \mathbb{R}$ be a transfer function initially equal to 0 on both inputs; Send a message $(\texttt{lc-opened})$ to $\beta.\mathsf{end\text{-}users}$ and go to point (B) below.
2) $(\texttt{timeout})$ from $\beta.\mathsf{Alice}$ in time at least Δ after you sent the message to $\beta.\mathsf{Bob}$: then close the contract and send a message $(\texttt{lc-not-opened})$ together with $\beta.\mathsf{balance}(\beta.\mathsf{Alice})$ coins to $\beta.\mathsf{Alice}$.

(B) **The contract $\mathcal{C}(id)$ execution**:

Assumption: for every channel δ each party P can send at most one message of a given type that concerns δ.

Wait for one of the following messages from parties $\beta.\mathsf{end\text{-}users}$:

1) $(\texttt{vc-close-init}, (\gamma, \sigma))$ from $\gamma.\mathsf{lngrid}$ in time at least $\gamma.\mathsf{validity} + 2$ (where (γ, σ) is an opening certificate of $P := \beta.\mathsf{other\text{-}party}(\gamma.\mathsf{lngrid})$ on γ) and γ has not been marked as closed: then send a message $(\texttt{vc-close-init}, \gamma.\mathsf{id})$ to P and wait for one of the following messages:
   a) $(\texttt{vc-already-closed}, z)$ from P, where z is a closing certificate of $\beta.\mathsf{other\text{-}party}(P)$ on γ: then mark γ as closed.
   b) $m := (\texttt{vc-close}, W, \mathsf{Sign}_P(W))$ from P (where W is a version of γ signed by $\gamma.\mathsf{other\text{-}party}(P)$): then send m to $\gamma.\mathsf{lngrid}$.
   c) $(\texttt{timeout})$ from $\gamma.\mathsf{lngrid}$ in time at least Δ after you sent the message $(\texttt{vc-close-init}, \gamma.\mathsf{id})$: then go to subroutine (C) below.
2) $m := (\texttt{vc-close-final}, oc_P, (V_{\gamma.\mathsf{Bob}}, S_{\gamma.\mathsf{Alice}}), (V_{\gamma.\mathsf{Alice}}, S_{\gamma.\mathsf{Bob}}))$ from $\gamma.\mathsf{lngrid}$ where oc_P is an opening certificate of $P := \beta.\mathsf{other\text{-}party}(\gamma.\mathsf{lngrid})$ on γ, each V_P is a version of γ signed by $\gamma.\mathsf{other\text{-}party}(P)$, and S_P is a signature of P on W (or is equal to $\perp$ if W is the initial version of γ), and γ has not been marked as closed: then send message m to P and wait for one of the following messages:
   a) $(\texttt{vc-already-closed}, z)$ from P, where z is a signed tuple containing a string "$\texttt{channel}\ \gamma.\mathsf{id}\ \texttt{closed}$": then do nothing.
   b) $(\texttt{timeout})$ from $\gamma.\mathsf{lngrid}$ in time $\Delta + 1$ after you sent m to P: then go to subroutine (C) below.
3) $(\texttt{vc-close-timeout}, (\gamma, \sigma))$ from $P \in \gamma.\mathsf{end\text{-}users}$ in time at least $\gamma.\mathsf{validity} + 4\Delta + 5$ where (γ, σ) is an opening certificate of $\gamma.\mathsf{lngrid}$ on γ and γ has not been marked as closed): send a message $(\texttt{vc-closing}, \gamma.\mathsf{id})$ to $\gamma.\mathsf{lngrid}$ and wait for one of the following messages:
   a) $(\texttt{vc-already-closed}, z)$ from $\gamma.\mathsf{lngrid}$, where z is a closing certificate of $\beta.\mathsf{other\text{-}party}(P)$ on γ: then do nothing.
   b) $(\texttt{timeout})$ from P in time at least Δ after you sent the $(\texttt{vc-closing}, \gamma.\mathsf{id})$ message to $\gamma.\mathsf{lngrid}$: then in this case go to subroutine (C) below.
4) $(\texttt{lc-close}, W)$ from P, where $W = (\gamma_P, w_P, \varepsilon, \sigma)$ is a version of β signed by $P' = \beta.\mathsf{other\text{-}party}(P)$: send a message $(\texttt{lc-closing})$ to P' and wait for one of the following to happen:
   a) P' replies with $(\texttt{lc-close}, W')$ where W' is a version of β signed by $P' = \beta.\mathsf{other\text{-}party}(P)$: let $\mathit{balance} := \mathsf{Win}(W, W') +$ *transfers*. For $\widehat{P} \in \beta.\mathsf{end\text{-}users}$ send $\mathit{balance}(\widehat{P})$ coins to $\widehat{P}$'s account on the ledger together with a message $(\texttt{lc-closed})$, and close the contract.
   b) In time τ party P' replies with a message $(\texttt{vc-active}, z)$, where z is an opening certificate of P on some channel γ constructed over β and $\tau \leq \gamma.\mathsf{validity} + 7\Delta + 5$: then do nothing.
   c) $(\texttt{timeout})$ from P in time Δ after you sent the $(\texttt{lc-closing})$ message to P': then let $\mathit{balance} := \gamma_P.\mathsf{balance} +$ *transfers*. For $\widehat{P} \in \beta.\mathsf{end\text{-}users}$ send $\mathit{balance}(\widehat{P})$ coins to $\widehat{P}$'s account on the ledger together with a message $(\texttt{lc-closed})$, and close this contract instance.

(C) **Subroutine for closing a virtual channel when cheating by party P is detected**:

Let $x := \gamma.\mathsf{balance}(\gamma.\mathsf{Alice}) + \gamma.\mathsf{balance}(\gamma.\mathsf{Bob})$. Remove x coins from P's account in *transfer* and add x coins to $\beta.\mathsf{other\text{-}party}(P)$'s account in *transfer*. Mark γ as closed. Send a message $(\texttt{vc-closed})$ to both $\beta.\mathsf{end\text{-}users}$.

Fig. 5: The contract functionality $\mathcal{C}$.

of P') the message $(\underline{\texttt{update-ok}})$ to the ideal functionality Channels. In other words, we make P conclude that P' did not accept a transfer that was beneficial to P. Again, it can happen during the channel closing that P' will use the newer version. This again can be corrected by $\mathcal{S}$ transferring the coins from P' to P after channel closing.

c) Ledger channel closing. This part starts when $\mathcal{Z}$ sends to P a message $(\underline{\texttt{lc-close}}, id)$ or it is started by a corrupt user. As in the case of channel opening the simulation is also straightforward: the simulator simply simulates the other parties and the contract functionality for the corrupt party, and sends the lc-close message Channels once P successfully closes a channel. The only thing that we need to remember is that a corrupt $\widehat{P} \in \beta.\mathsf{end\text{-}users}$ may submit a version of a channel that is less beneficial for him, but newer than the latest version that the other user of β submits (see above). As described above, in this case $\mathcal{S}$ simply moves the appropriate amount of coins from $\widehat{P}$'s account in the ledger to the account of $\beta.\mathsf{other\text{-}party}(\widehat{P})$ to "correct" this difference.

d) Virtual channel opening. The simulation proceeds as in the previous cases. When the simulated honest parties output a $(\underline{\texttt{vc-opened}})$ message then $\mathcal{S}$ sends the $(\underline{\texttt{vc-open}}, \gamma)$ message in the name of corrupted $P \in \gamma.\mathsf{all\text{-}users}$ to the ideal functionality Channels and lets the ideal functionality immediately

output (vc-opened) to all the users.

e) Virtual channel closing. Closing of virtual channel starts automatically when time $\gamma.\mathsf{validity}$ comes. As argued in Sect. III-C2 the virtual channel is always closed, as long as at least one party on $\gamma.\mathsf{all\text{-}users}$ is honest. Again, $\mathcal{S}$ simulates the corrupt parties, and, depending on their behavior instructs the ideal functionality $\mathsf{Channels}$ to send the (vc-closed) message to the honest parties (in time at most $\gamma.\mathsf{validity} + 7\Delta + 5$). □